Laboratorio Virtual de Robótica: Entorno integrado de simulación, teleoperación y realidad aumentada.

Proyecto Final de Estudios - Ing. Mecatrónica

2020

UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

Autores:
Ariel Salassa
Francisco Tinelli

Directora:
Dra. Carolina Diaz

Agradecimientos

Agradecemos muy especialmente a Samir Gattás y Rodrigo Torres por su constante cooperación y por haber hecho un excelente proyecto final titulado "*Desarrollo de Sistema Robótico Educativo*", dejando las bases sólidas para seguir construyendo sobre él múltiples funcionalidades que permiten enriquecer tanto su trabajo como el nuestro.

No queremos dejar de mencionar a nuestros profesores, Dra. Carolina Díaz e Ing. Eric Sánchez, por confiar en nosotros al aceptar ser nuestros directores, por su constate apoyo, por su gran vocación y dedicación, y por acompañarnos y apadrinarnos en el cierre de este tramo de nuestras vidas.

A nuestra casa de estudios, Facultad de Ingeniería de la Universidad Nacional de Cuyo, queremos dar las gracias por alojarnos sin nunca pedir nada a cambio y darnos las herramientas necesarias para desenvolvernos en nuestra vida adulta y profesional, así como también a cada uno de los profesores de la carrera de Ingeniería Mecatrónica por la enseñanza y la formación que hemos recibido de ellos.

Finalmente, el agradecimiento más sincero hacia nuestras familias, nuestros amigos y nuestros compañeros, por haber estado a nuestro lado en todos estos años y por ser grandes responsables de que la culminación de esta etapa sea una realidad.

Resumen

Este proyecto expone el trabajo realizado durante seis meses de ardua labor en el marco de la robótica industrial, los laboratorios virtuales y el control a distancia. Se ha tomado como banco físico de pruebas el robot industrial “MOVEO” desarrollado por los Ing. Samir Gattas y Rodrigo Torres y se ha construido sobre él múltiples escenarios de simulación independientes en su funcionamiento, pero complementarios, que permiten abordar la robótica desde varias ópticas como ser: robótica educativa y recreativa, robótica de supervisión y control de procesos, diseño robótico, cinemática y dinámica de robots, etc.

Las simulaciones, que en este caso serán denominadas “nodos”, abarcan cuatro frentes distintos y para ello se han utilizado distintas tecnologías. Primeramente, se ha desarrollado un driver que permite exponer información relevante del robot a su entorno. En segundo lugar, se ha desarrollado en Matlab un programa que simula ser un *teach pendant* simplificado y que está acompañado de una simulación utilizada y desarrollada por el personal de la cátedra de Robótica I. Adicionalmente, se ha decidido utilizar V-Rep/CoppeliaSim como el software ideal para simular y supervisar procesos industriales. Finalmente, se ha desarrollado una aplicación móvil con Unity y Vuforia, que supervisan el movimiento del robot con módulos de realidad aumentada.

La integración de todas las partes antes mencionadas es posible gracias a ROS (*Robot Operative System*), el cual pone a disposición un ambiente ideal para la comunicación de las distintas partes del sistema. Para gestionar la comunicación entre las partes, así como para determinar qué nodo que tomará el control (*Publisher*) ha sido necesario desarrollar un orquestador de conexiones. Dicho orquestador ha sido desarrollado con VueJS y es la pieza fundamental que da visibilidad del ambiente integrado a cada una de las partes que a él se conectan.

Como resultado final, se ha obtenido un producto que es perfectamente extrapolable a cualquier otro proyecto de robots serie, pues cada elemento componente se ha pensado y desarrollado de forma modular y genérica para su futura reutilización.

Índice

1. Introducción

En el presente informe se desarrolla el resultado del Proyecto Final de Estudios (en adelante PFE) de la carrera de Ingeniería en Mecatrónica, titulado “Entorno Integrado de Simulación y Control a distancia de Robots Industriales”. Aquí se describe el desarrollo de las distintas partes funcionales del sistema, qué tecnología se utilizó para cada caso, cómo funcionan de manera independiente y cómo funcionan cuando hay otros actores en el entorno. Primeramente, se hará una breve mención a la idea inicial y lo que sirvió de base para llevar a cabo este desarrollo.

a. Motivación

Este proyecto surge de la motivación de realizar un PFE desafiante que contribuya tanto a la cátedra de Robótica I como al “Laboratorio de Robótica de Servicio, Teleoperación y Tecnologías Aplicadas” (ROSETTA Lab.) del Instituto de Automática y Electrónica Industrial, que contenga un aporte original donde se apliquen algunas de las tecnologías de punta que se utilizan en el ámbito industrial.

En el mismo, se han volcado e integrado los conocimientos adquiridos durante toda la carrera, con el fin de producir un sistema funcional y de calidad.

Esta plataforma, no sólo servirá como soporte de estudio y práctica de la cátedra de Robótica I, sino también como base y complemento para proyectos existentes y futuros, tanto de la misma cátedra, como de otras. E incluso podría ser complementario con cualquier otro PFE dentro del campo de la robótica que quiera tener un valor agregado en su trabajo.

b. Justificativos

La principal virtud del trabajo realizado es que permite establecer las bases para el desarrollo de proyectos que responden a los lineamientos que pretende promulgar el "Laboratorio de Robótica de Servicio, Teleoperación y Tecnologías Aplicadas" (ROSETTA Lab.) y que son perfectamente acoplables a cualquier otro desarrollo dentro del marco de la robótica.

El resultado que se ha alcanzado tiene un gran contenido didáctico ya que plantea poner a disposición herramientas visuales y de fácil comprensión que puedan ser utilizadas para el dictado de las clases de la cátedra “Robótica I”.

Finalmente, este proyecto es un ejemplo de integración de tecnologías de punta aplicadas al campo de la robótica, que incluyen, entre otras cosas:

- Realidad Aumentada
- Simulación de Procesos Industriales
- Simulación en Tiempo Real
- Framework de desarrollo para proyectos robóticos robustos y multiplataforma.

El seguimiento del mismo puede motivar a sus lectores a embarcarse en proyectos similares o en la búsqueda de nuevas soluciones que respondan a requerimientos semejantes.

c. Objetivos

Al comenzar este Proyecto se plantearon los siguientes objetivos:

- Desarrollar una plataforma que comprenda:
  - Un motor de Realidad Aumentada (AR) que proporcione una interfaz para la supervisión del robot a cuanto usuario se conecte.
  - Un módulo de Simulación de un proceso industrial dentro del cual el robot analizado participa. Se utilizará un software específico de la industria: V-Rep.
  - Un driver de comunicación para comandar y supervisar el robot real conectándolo al sistema y sincronizando las simulaciones. En primera instancia, dicho driver apuntará a hacer un puente de comunicación entre nuestro sistema y LinuxCNC (sistema operativo concebido para controlar máquinas CNC y que corre sobre el microprocesador que comanda al robot real).
  - Un módulo que simule el *teach-pendant* de un robot, desarrollado en Matlab y que incorpora las herramientas didácticas de programación y simulación utilizadas por la cátedra para el dictado de sus clases.
- Integrar los cuatro módulos antes mencionados en una unidad de trabajo, dentro de un entorno sólido y confiable como el que ofrece ROS, y que respondan a un sistema que de gestión de conexiones u orquestador.
- Extender el desarrollo de este proyecto para otros proyectos ya realizados para la cátedra y/o el laboratorio.
- Realizar una prueba de concepto de las simulaciones, la Teleoperación y los cambios de mandos, utilizando como banco de prueba físico al robot MOVEO robot serie 5GDL, tipo industrial de código abierto, diseñado por BCN3D [1] y [2].
- Desarrollar sistemas de compatibilidad para volver la plataforma lo más genérica posible y facilitar su acoplamiento con otros proyectos/robots.
- Verificar la compatibilidad del sistema y la adaptabilidad del código desarrollado realizando pruebas sobre un robot industrial distinto, tal como el robot UR3 de la empresa Universal Robots.
- Asentar las bases para poder crear Laboratorios Virtuales con otras instituciones.

Cada módulo ha sido concebido de forma tal de tener autonomía en su funcionamiento, por lo que el estado de activación o disponibilidad de alguno de ellos, no debería impactar en la ejecución de los otros.

d. Estudio de antecedentes y estado del arte.

En cada una de las principales ramas de este proyecto se han realizado numerosos trabajos que servirían de antecedentes al informe aquí presentado.

La **robótica** va teniendo una presencia cada vez mayor tanto en entornos industriales manufactureros, como así también en otros entornos, como el caso de robots quirúrgicos, robots de servicio, etc.

Por su parte la **realidad aumentada** está alcanzando cada vez más inmersión en los servicios de video juegos, en servicios de compra venta, etc. Algunas empresas de tecnologías importantes avalan que "la realidad aumentada se convertirá en el nuevo lente para ver el mundo" [3].

Cuando se habla de realidad aumentada vinculada específicamente a la robótica se hace referencia al control y/o supervisión de un robot mediante un dispositivo móvil remoto que puede tener la capacidad de configurar la máquina y de obtener y visualizar información que a simple vista no se podría consultar en la máquina física (Figura 1).

Figura 1. Aplicación de robótica y realidad virtual.

Un antecedente es el estudio realizado por la Universidad Nacional Técnica de Atenas que plantea el desarrollo de un sistema de formación de operarios dentro de un entorno virtual inmersivo e interactivo para la colaboración entre humanos y robots en un ambiente de manufactura de telas utilizadas en la industria aeroespacial. La aplicación involucra tareas altamente colaborativas tales como la remoción de fibras adhesivas y la colocación de telas en un molde [4]. (Figura 2)

Figura 2. Trabajo de inmersión virtual para capacitar a los operarios de robots en una industria.

Otro caso de estudio interesante es el planteado por la universidad de Malasia, en el departamento de ingeniería mecánica, donde se evalúa el uso de la realidad aumentada para planear, programar y simular una célula de trabajo robótica. En este caso, se utilizan los beneficios de la realidad aumentada, principalmente el incremento de *feedback* a los operarios y la reducción de errores, para simular un entorno de realidad aumentada para su aplicación en una celda robótica, utilizando detectores de colisión e información transmitida a través de un visualizador *heads-up* (HUD) para incrementar los sentidos perceptivos del usuario final. En este proyecto se abordó la generación y planificación de trayectorias de forma que el robot siguiera el movimiento de la mano del usuario, y se planteó su aplicación en estaciones robóticas reales [5].

Finalmente, el estudio que se tomó como base para realizar este proyecto es el publicado por la Universidad de Alicante, proveniente de la maestría en automática y robótica. En el mismo se detalla el desarrollo de una interfaz de realidad aumentada (Figura 4) que se utiliza para supervisar el estado de un robot industrial UR3 de Universal Robots a lo largo de un proceso industrial simulado (Figura 3). El objetivo final consta de la visualización de los valores de las variables más importantes de cada articulación (posición, torque motor, temperatura) [6].

Este estudio y la continuidad de este proyecto surgen de la permanente cooperación y colaboración que existe entre el grupo de investigacion HURO (*Humans and Robots*) de la Universidad de Alicante y el laboratorio ROSETTA de esta facultad. Este trabajo se tomo como base para la creación de un laboratorio virtual entre ambas instituciones. Buscando tabajar en forma conjunta potenciando los recursos humanos y materiales de ambos grupos

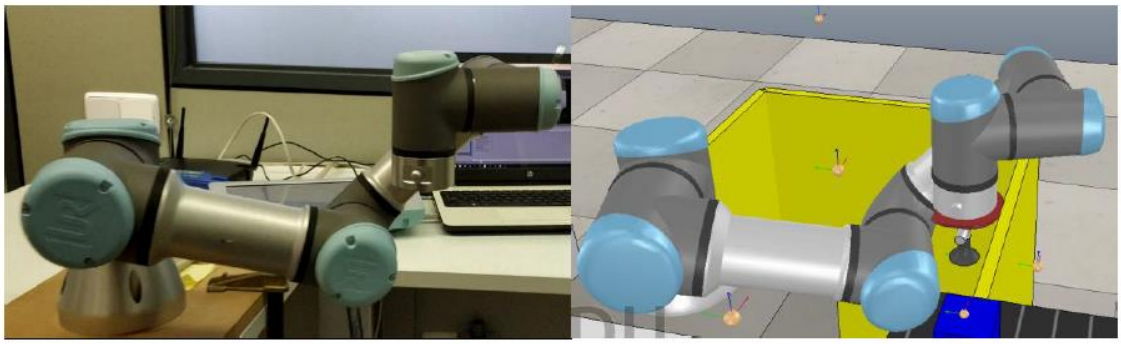

Figura 3. Interfaz de realidad virtual desarrollada por la Universidad de Alicante para simular y supervisar un robot UR3 en un proceso industrial.

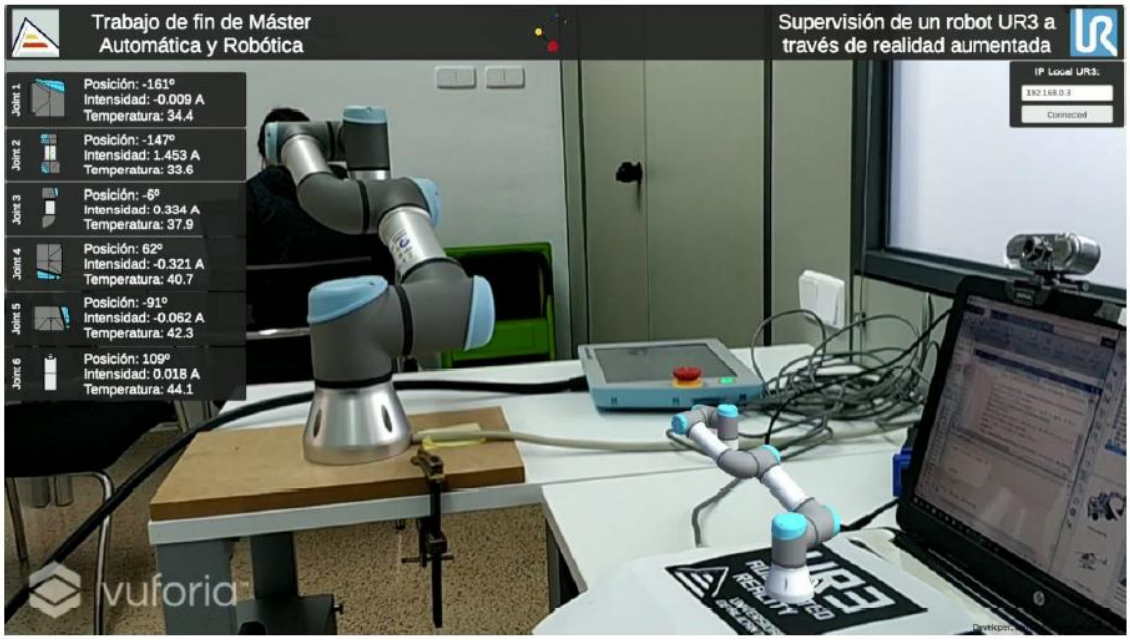

Figura 4. Interfaz de realidad aumentada desarrollada por la Universidad de Alicante para supervisar un robot UR3, donde se ilustran las principales variables de estado de las articulaciones del robot.

e. Banco de prueba físico

Como banco de prueba físico, se contó con el robot MOVEO mejorado realizado como PFE por el Ing. Samir Gattás y el Ing. Rodrigo Torres, y puesto a disposición de Rosetta Lab para sus mejoras incrementales [1]. En dicho trabajo se desarrolló un sistema robótico integral al partir del proyecto abierto denominado MOVEO de *BCN3D Technologies* [2]. Al modelo disponible se le realizaron mejoras mecánicas a sus piezas componentes y se ensambló nuevamente. Sumado a esto se diseñaron placas PCB para proteger drivers y la placa controladora. Una vez montados todos los componentes electrónicos y la etapa de potencia, se programó y configuró el sistema controlador basado en LinuxCNC que corre sobre una placa *BeagleBone Black*. Finalmente, se desarrolló una pequeña interfaz de usuario con Matlab para generar movimientos y realizar trayectorias.

f. Metodología de trabajo

Cada integrante de este PFE ha dedicado alrededor de 550 horas de trabajo a la realización del mismo, lo que hace un total de 1100 horas de trabajo aproximadamente.

De esta forma, cada integrante del equipo ha destinado en promedio unas 25 horas semanales, lo que implica que cada semana se vuelquen 50 horas de trabajo al proyecto.

Se ha trabajado siguiendo una metodología *Agile* en su forma más elemental. Cada *Sprint* (incremento temporal del proyecto) estará compuesto por 2 semanas de trabajo, equivalente a 100hs [7] [8]. Las tareas a realizar en cada *sprint* fueron visibles en todo momento a los directores y tutores del proyecto para su correspondiente supervisión (Figura 5). Como herramienta de control de versiones y de gestión de proyecto se utilizó GitLab [9].

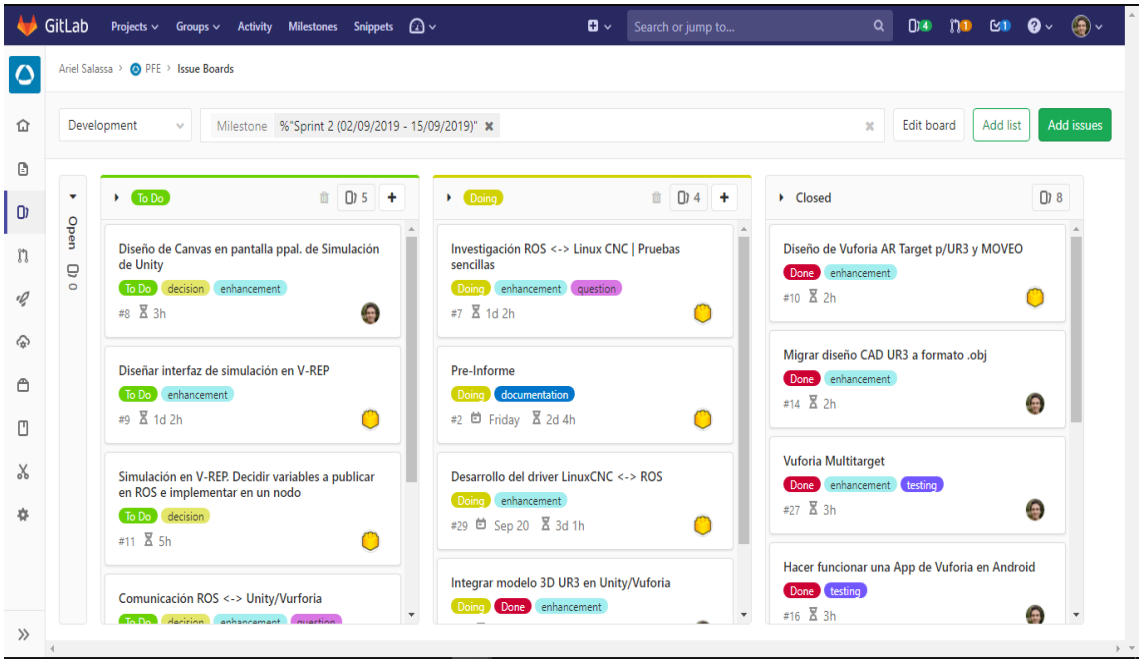

Figura 5. Planificación y visualización de avance en el Sprint 3 del proyecto.

A lo largo del desarrollo del mismo, se dieron dos momentos de éxito remarcables. En una primera instancia, a los tres meses de comenzado el proyecto, se obtuvo un hito de "sincronización exitosa", esto es, ya teniendo una buena parte de la funcionalidad de los nodos desarrollada, las instancias de simulación eran capaces de copiar el movimiento del robot en tiempo real.

La segunda instancia, alcanzada a final del proyecto, se culminó al completar el desarrollo de los nodos de forma tal que de que sean completamente compatibles con ROS y al verificar el correcto funcionamiento de un orquestador de conexiones que permite que cualquiera de las instancias del sistema pueda tomar control del robot real de manera ordenada, y que todas las demás instancias imiten su movimiento.

2. Tecnologías utilizadas

a. ROS

Figura 6. Logo de ROS.

ROS (Figura 6) es el acrónimo de Sistema Operativo Robótico (o *Robot Operating System*, por sus siglas en inglés) [10] [11]. Es un framework para el desarrollo de software para varios tipos de robots que provee la funcionalidad de un sistema operativo (OS) en un clúster homogéneo. ROS provee los servicios estándar de un OS tales como abstracción de hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos, que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. Este sistema está diseñado para correr sobre sistemas UNIX (sistemas operativos Linux principalmente), aunque tiene adaptaciones consideradas "experimentales" para correr sobre otros sistemas.

ROS tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente; y *ros-pkg*, un conjunto de paquetes de desarrollo aportado por la contribución de la comunidad y que implementan funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc.

La distribución utilizada en este proyecto es "*ROS Melodic Morenia*", última versión estable lanzada en 2018.

b. V-Rep

Figura 7. Logo de V-Rep (versión educativa).

V-Rep (Figura 7) es un software simulador de robótica [12] [13]. Permite programar, emular y validar un robot sin necesidad de su presencia física, ahorrando así costos y tiempos. Posee un entorno gráfico propio para crear objetos complejos que interaccionan entre sí y pone a disposición la posibilidad de construir un mundo virtual donde sea posible validar los comportamientos programados para luego transferirlos al robot real. Es multiplataforma y dispone de un IDE propio para construir los escenarios virtuales y controlar cada robot mediante scripts internos escritos en LUA [14], nodos ROS o programas externos escritos en lenguajes como C++, Matlab, Java y Python (entre otros). La interfaz ofrece modelos de robots u otros elementos prediseñados, así como todas las funcionalidades necesarias para crear un modelo propio adaptado a las necesidades del usuario. Es posible importar, si se desea, el modelo de robot que se quiere validar y se pueden diseñar robots casi tan complejos como se desee.

c. Vue JS

Figura 8. Logo de Vue JS.

Vue (Figura 8) es un framework *open-source* de JavaScript diseñado para construir interfaces de usuarios [15]. Vue está diseñado desde cero para ser utilizado incrementalmente. La librería central está enfocada solo en la capa de visualización, y es fácil de usar e integrar con otras librerías o proyectos existentes. Por otro lado, Vue es perfectamente capaz de impulsar sofisticadas *Single-Page Applications* cuando se utiliza en combinación de otras herramientas y librerías complementarias.

Una de las características más importantes es la modularización, es decir, el trabajo a nivel de componentes. Un componente Vue es, esencialmente, un elemento que encapsula código reutilizable. Dentro del mismo podemos encontrar maquetado en HTML, estilos de CSS, y código JavaScript. Los componentes permiten desarrollar proyectos modularizados y fáciles de escalar.

Otra de las principales características de Vue es que es un framework reactivo. Cuando una instancia Vue es creada, agrega todas las propiedades encontradas en su objeto *data* al sistema de reactividad de Vue. Cuando los valores de estas propiedades cambian, la vista “reaccionará”, actualizándose para coincidir con los nuevos valores y volviendo a renderizar la vista.

d. Unity

Figura 9. Logo de Unity.

Unity (Figura 9) es uno de los frameworks de desarrollo de videojuegos más famosos del mundo. Es una herramienta que engloba motores para el renderizado de imágenes, de cuerpos dinámicos 2D/3D, de audio, de animaciones y más. Además, permite importar paquetes para hacer desarrollos en red, incorporar funcionalidades de inteligencia artificial y de Realidad Virtual [16].

Una de las características más importantes y más cómodas de Unity es que soporta la exportación a una gran cantidad de plataformas. No solo se puede elegir la plataforma con la que se va a trabajar creando y editando un juego, cuyo editor en este momento soporta Windows, MacOS y Linux (éste último de forma experimental), sino que además se puede crear dicho juego para más de 25 plataformas.

Unity tiene una muy buena compatibilidad e integración con los principales softwares de diseño, tales como Blender o 3ds Max, lo que permite que los cambios realizados a los objetos creados con ellos se actualicen automáticamente en todas las instancias de ese objeto sin necesidad de volver a realizar la importación manualmente.

El desarrollo de scripts se basa en "Mono"[1]. Los programadores pueden utilizar UnityScript (un lenguaje personalizado inspirado en la sintaxis ECMAScript), C# [17] o Boo (que tiene una sintaxis inspirada en Python), aunque la mayoría de la comunidad elige C# por ser éste el más popular de los anteriormente mostrados [18].

La comunidad de Unity es muy grande y está brindando constantemente nuevos desarrollos (pagos o gratuitos) así como permanente soporte [19] [20].

[1] MonoDevelop es un entorno de desarrollo integrado libre y gratuito, diseñado primordialmente para C# y otros lenguajes .NET framework.

e. Vuforia

Figura 10. Logo de Vuforia.

Vuforia (Figura 10) es un SDK (*Software Development Kit*) de realidad aumentada para dispositivos móviles tales como *smart-phones*, *tablets* y lentes de realidad virtual. Utiliza tecnología de visión artificial para seguir imágenes planas y objetos tridimensionales en tiempo real. Esta propiedad permite a los desarrolladores posicionar y orientar objetos virtuales, en relación con objetos del mundo real cuando se ven a través de la cámara de un dispositivo móvil. El objeto virtual sigue la posición y orientación de la imagen en tiempo real para que la perspectiva del espectador sobre el objeto corresponda con la perspectiva del objeto, de forma tal que el objeto virtual pareciera formar parte del mundo real percibido a través de la ventana de nuestro dispositivo móvil.

Vuforia proporciona interfaces de programación de aplicaciones (API) en C ++, Java, Objective-C ++ y los lenguajes .NET a través de una extensión del motor de juego Unity [21] [22]. De esta manera, el SDK admite el desarrollo nativo para iOS, Android y UWP, mientras que también permite el desarrollo de aplicaciones con realidad aumentada en Unity que son fácilmente portables en dichas plataformas.

f. Matlab

Figura 11. Logo de Matlab - MathWorks.

Matlab (abreviatura de *Matrix Laboratory* – Laboratiorio de Matrices) (Figura 11) es un entorno de desarrollo (IDE) optimizado para el análisis iterativo y el manejo de matrices, que cuenta con un lenguaje de programación propio, el lenguaje M [23]. Este IDE incluye dos herramientas principales que expanden sus prestaciones:

- Simulink: plataforma de simulación la cual permite la simulación de sistemas programándolas en forma gráfica con la utilización de bloques.
- Guide: un editor de interfaces de usuario (GUI).

Matlab es muy utilizado por universidades y centros de desarrollo e investigación.

Además, es posible ampliar las capacidades de Matlab con el agregado de diversos *toolbox* (por ejemplo, para estadística, tratamiento de imágenes, inteligencia artificial, etc).

g. Machine Kit

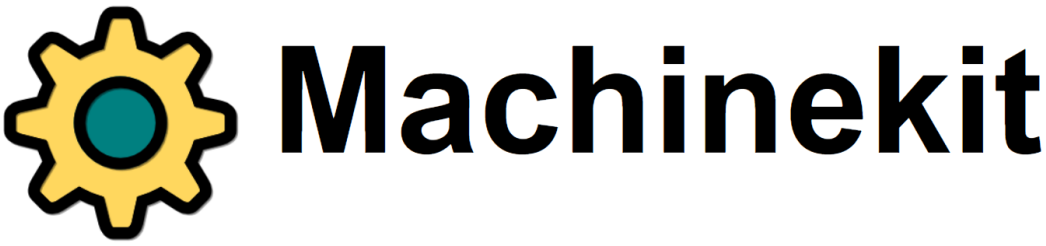

Figura 12. Logo de Machinekit.

Machinekit (Figura 12) es una plataforma *open-source* para aplicaciones de control de maquinaria en tiempo real [24]. El proyecto surgió como una expansión de LinuxCNC, el cual se centra en el control local de máquinas herramientas, o máquinas a control numérico (ej. fresadoras, tornos, limadoras, etc.) [25].

Esta plataforma utiliza la arquitectura de componentes HAL[2], la cual permite diseñar modelos y sistemas de forma fácil e intuitiva mediante la utilización de bloques con diferentes funciones, entre ellas lógica digital, planificación de trayectoria, bucles de control, procesamiento de señales, drivers para hardware, etc. Además, amplía las posibilidades de LinuxCNC brindando soporte para más plataformas de hardware y brindando posibilidades de control remoto.

Vale la pena aclarar que Machinekit fue utilizado principalmente en el proyecto final de estudios de nuestros colegas quienes construyeron el robot MOVEO. Aquí se menciona debido a que, para la incorporación del robot al sistema, se desarrolló un driver el cual hace uso de una librería de Machinekit, como se verá más adelante.

[2] La capa de abstracción de hardware (en inglés, *Hardware Abstraction Layer* o HAL) es un elemento del sistema operativo que funciona como una interfaz entre el software y el hardware del sistema que provee una plataforma de hardware consistente sobre la cual corren las aplicaciones. Cuando se emplea una HAL, las aplicaciones no acceden directamente al hardware, sino que lo hacen a la capa abstracta provista por la HAL. Las HAL permiten que las aplicaciones sean independientes del hardware porque abstraen información acerca de tales sistemas, como lo son las cachés, los buses de E/S y las interrupciones, y usan estos datos para darle al software una forma de interactuar con los requerimientos específicos del hardware sobre el que deba correr.

3. Descripción del sistema global

El sistema global se ha diseñado con cinco nodos preparados para trabajar dentro de un entorno de ROS[3]. Un nodo es, básicamente, un proceso que ejecuta código, escrito usando de alguna las "librerías cliente" nativas de ROS, *roscpp* o *rospy* [26]. Los nodos se combinan en grafos y pueden comunicarse mediante tópicos (*topics*), servicios (*services*) o el servidor de parámetros (*parameter server*). El uso de nodos en ROS proporciona varios beneficios al sistema general. Hay una mejor tolerancia a fallas ya que los eventuales bloqueos están aislados en nodos individuales, sin afectar necesariamente al resto. La complejidad del código se reduce en comparación con los sistemas monolíticos. Los detalles de implementación también están bien ocultos ya que los nodos exponen una API mínima al resto del grafo y las implementaciones alternativas, incluso en otros lenguajes de programación, pueden sustituirse fácilmente.

En este proyecto, se han desarrollado cinco nodos:

- *R.R. Node*: es, básicamente, un driver que permite exponer y consumir información relevante del robot físico real a y desde ROS.
- *Pro.Sim. Node*: es el encargado correr la simulación de los procesos industriales en V-Rep.
- *Control GUI Node*: simula ser un *teach-pendant* del robot y provee la interfaz necesaria para conectar Matlab con ROS.
- *A.R. Engine Node*: es una aplicación móvil que utiliza realidad aumentada para representar la postura del robot y, eventualmente, controlarlo.
- *Orchestrator Node*: gestiona la comunicación entre las partes y determina el nodo que tomará el control sobre el robot real, si éste estuviera presente en el sistema.

Los nodos se comunican entre sí utilizando tópicos y servicios. La Figura 13 ilustra la topología del sistema global.

[3] Para la ejecución de este proyecto se ha optado por trabajar con la última distribución estable de ROS, llamada *Melodic Moreira.*

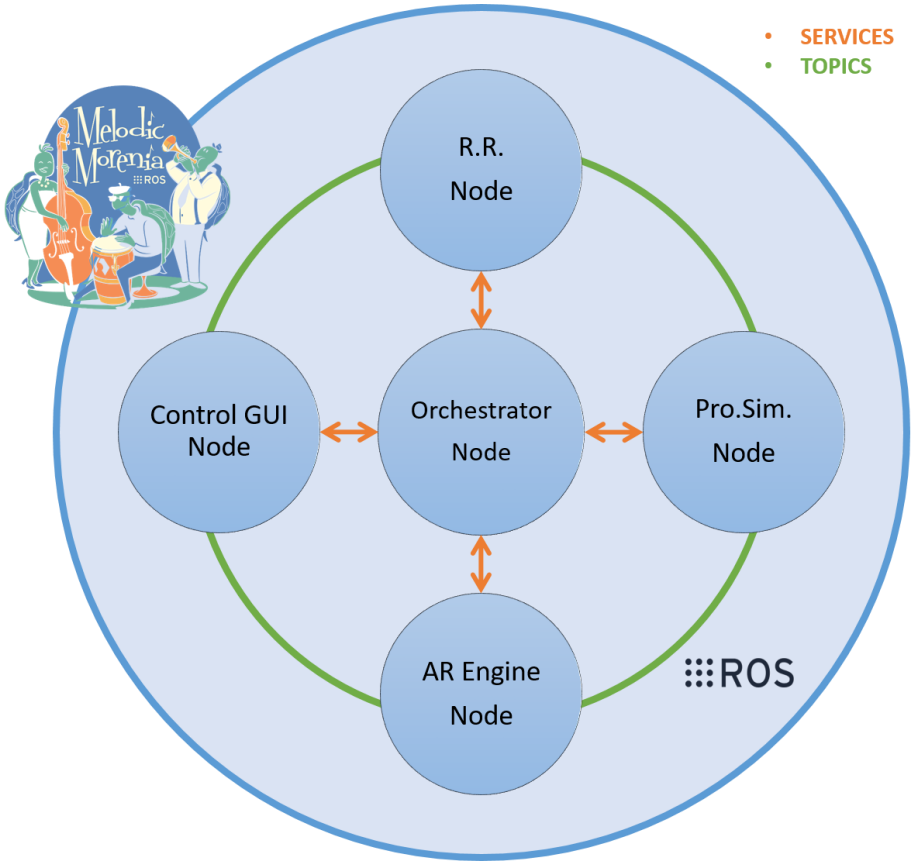

Figura 13. Topología general de funcionamiento del sistema desarrollado.

Un tópico es un canal identificado con un nombre único que funciona como *endpoint* y sobre el cual los nodos intercambian mensajes de manera unidireccional. Hablando en el lenguaje de ROS, un mensaje es una estructura simple de datos compuesta de varios campos. Un campo que generalmente está presente se denomina *header* y contiene metadatos adicionales: *seq*, *timestamp* y *frameID*). El resto de los campos contiene los datos de interés expuestos de una forma fuertemente tipada para su consistente publicación y/o consumo. Volviendo a los tópicos, se debe destacar que los mismos tienen una semántica anónima de publicación y subscripción, que desacopla la producción de información de su consumo. En este caso, al igual que en la mayoría de los casos, los nodos no son conscientes de con quién se están comunicando, sino que consumen y publican información relevante sobre un tópico determinado. El framework de ROS ha sido diseñado para que haya múltiples *publishers* y *subscribers* al mismo tiempo sobre un mismo tópico. ROS actualmente admite el transporte de mensajes basado en TCP/IP y UDP. El transporte basado en TCP/IP se conoce como TCPROS y transmite datos de mensajes a través de conexiones TCP/IP persistentes. TCPROS es el transporte predeterminado utilizado en ROS. El transporte basado en UDP, que se conoce como UDPROS y actualmente solo se admite en *roscpp* (nodos desarrollados en el lenguaje C++), separa los mensajes en paquetes UDP. UDPROS es un transporte con

baja latencia y pérdida. Los nodos ROS "negocian" el transporte deseado en tiempo de ejecución. Por ejemplo, si un nodo prefiere el transporte UDPROS, pero el otro nodo no lo admite, puede recurrir al transporte TCPROS. Este modelo de negociación permite agregar nuevos transportes con el tiempo a medida que surgen casos de uso convincentes.

En el sistema presentado, los nodos periféricos utilizan un único tópico denominado *"/joints_states*" para realizar simulaciones de forma sincronizada cuando están conectados al sistema. El mismo fue ideado para describir el estado de un set de articulaciones controladas. La definición programática del mismo es la siguiente:

```
std_msgs/Header header[4]
string[] name
float64[] position
float64[] velocity
float64[] effort
```

Cada articulación está inequívocamente identificada por su nombre y, lógicamente, cada uno de los arreglos debe tener la misma cantidad de elementos o debe estar vacío para que cada parámetro del estado esté asociado a su nombre correspondiente.

En la presente aplicación, el robot no tiene ningún tipo de control a lazo cerrado, por lo que el único campo de interés es *position*. Aquí se vuelcan y se consumen los valores de posición en grados de cada una de las posiciones articulaciones del robot. El resto de los campos de datos quedan como arreglos vacíos o vectores de ceros, pero pueden ser utilizados perfectamente en el caso de que se acople un robot con la capacidad de sensar más variables.

La comunicación entre cualquiera de los nodos periféricos y el nodo orquestador se realiza mediante servicios. El modelo productor-consumidor comentado anteriormente es un paradigma de comunicación muy flexible pero no es apropiado cuando se necesita un esquema de solicitud-respuesta. Este esquema en ROS se realiza mediante la implementación de servicios, cada uno de los cuales es definido mediante un par de mensajes: uno para la solicitud y otro para la respuesta. Un nodo de ROS puede anunciar (*advertise*) un servicio de nombre único, y un cliente puede llamar a dicho servicio enviando un mensaje de solicitud y esperando un mensaje de respuesta.

[4] Definición de un mensaje del tipo *Header*:

```
uint32 seq
time stamp
string frame_id
```

En el desarrollo de este proyecto se trabajó con servicios que, aunque presenten distintos nombres, en todos los casos envían una cadena de texto en un campo denominado *data,* y devuelven como resultado un booleano que indica el éxito de la petición dentro de un campo denominado *success*, junto con otra cadena de texto presente en un campo denominado *message*. La definición del servicio sigue la siguiente estructura:

```
string data
---
bool success
string message
```

Cada uno de los nodos de simulación funciona como una máquina de estados, y los cambios de estado provienen del intercambio de información a través de servicios. Cuando el orquestador comienza su ejecución, pone a disposición cuatros servicios que pueden ser llamados por los nodos:

- */orchestrator/request_connection*
- */orchestrator/release_connection*
- */orchestrator/request_control*
- */orchestrator/release_control*

Cuando un nodo quiera conectarse al sistema tiene que notificar al orquestador, llamando al servicio */orchestrator/request_connection*, enviando dentro de *data* su IP, así como el nombre asociado a la clase del nodo (*AR Engine, Pro.Sim, R.R.* o *Control GUI*) separados por el carácter '|'. El orquestador, luego de validar exitosamente los datos, devolverá en su mensaje un identificador denominado "OrchID".

Cuando un nodo quiera desconectarse del sistema debe llamar al servicio */orchestrator/release_connection*, enviando dentro del campo *data* su OrchID. El orquestador devolverá automáticamente un *True* dentro del campo *success* y un mensaje de aviso en el campo *message*.

Cuando un nodo quiera publicar sobre el tópico que los demás nodos del sistema consumen, debe llamar al servicio */orchestrator/request_control*, enviando dentro del campo *data* su OrchID. El orquestador devolverá un *True* dentro del campo *success*, si los datos enviados son válidos y si la petición es aceptada. En caso contrario devolverá un False. En el campo *message* se volcará un mensaje de aviso descriptivo, ya sea que la petición haya sido exitosa o no.

Cuando un nodo quiera ser consumidor sobre el tópico, debe llamar al servicio */orchestrator/release_control*, enviando dentro del campo *data* su OrchID. El orquestador devolverá un *True* dentro del campo *success*, si los datos enviados son válidos y si la petición es aceptada. En caso contrario devolverá un *False*. En el campo

message se volcará un mensaje de aviso descriptivo, ya sea que la petición haya sido exitosa o no.

Una vez que el nodo periférico conoce su OrchID, anuncia el servicio */[OrchID]/orch_com*. Mediante la información enviada, el orquestador puede obligar al nodo en cuestión a pasar directamente de *subscriber* a *publisher* o viceversa, según sea el caso.

Entendiendo el porqué de los servicios y cómo funcionan los mismos se pueden entender bien las máquinas de estados ilustradas en las Figuras 14 y 15.

Vale la pena aclarar que el estado de inicio siempre será *Disconnected* y que se han definido estados intermedios transitorios, *ConnectingState* y *ChangingState*, necesarios para actualizar las interfaces de simulación y mantenerlas en un estado determinado, hasta tanto las peticiones críticas no hayan sido resueltas.

Por otro lado, nótese que para el nodo R.R., una vez que se ha conectado al sistema, entra como *Publisher*. Esto se ha decidido así por cuestiones de seguridad: si un robot real comienza a recibir consignas de posición instantáneamente al momento inicial puede resultar en una catástrofe. Por otro lado, para el resto de los nodos, la entrada al sistema se hace en modo *Subscriber*. Esto es debido a que originalmente el sistema se ideó para que en un primer momento los nodos de simulación copien el movimiento del robot real.

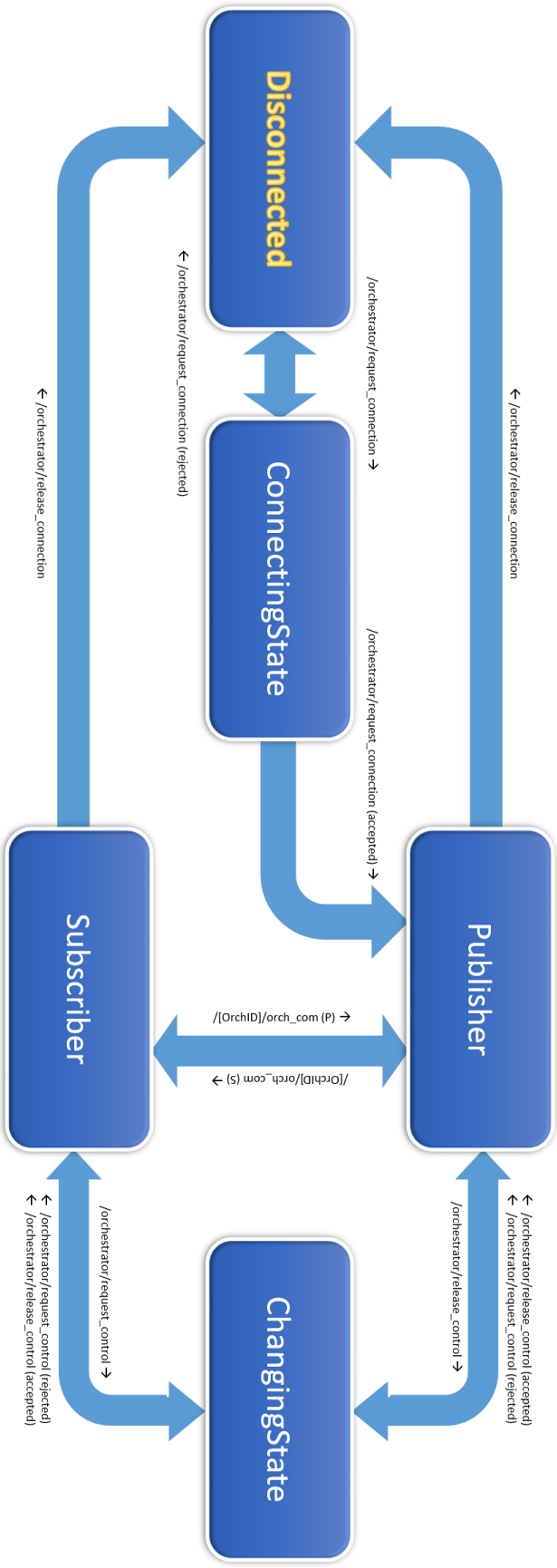

Figura 14. Máquina de estados para el nodo "Real Robot".

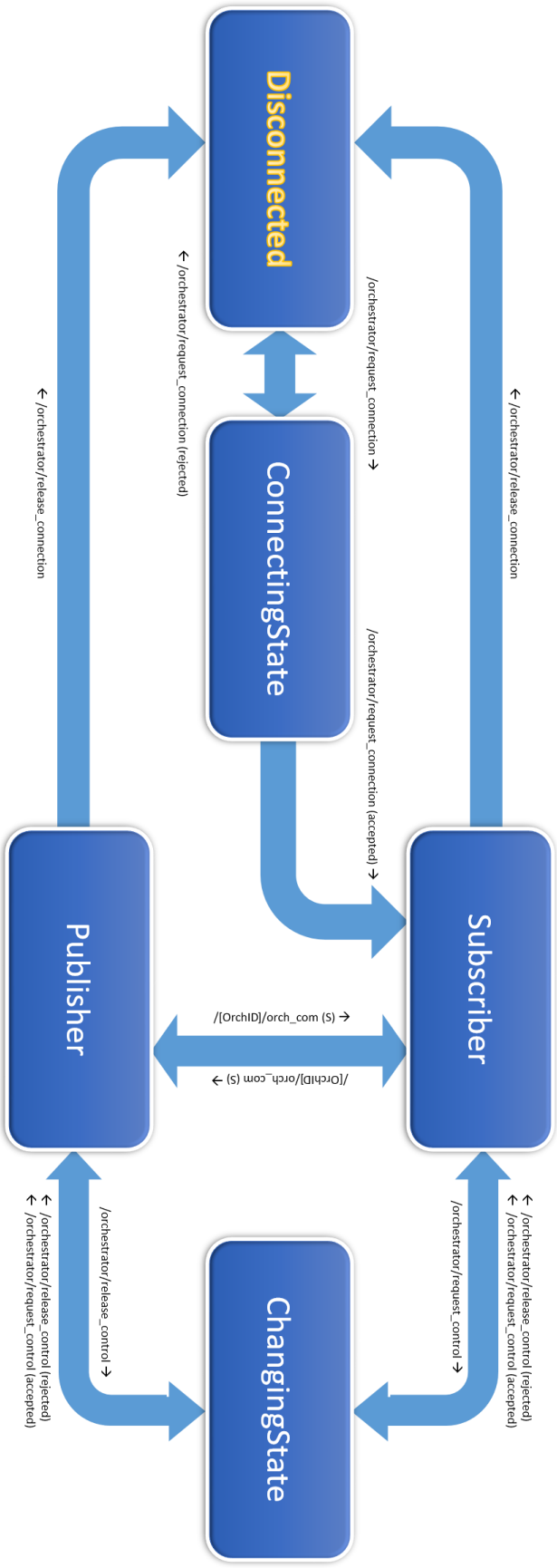

Figura 15. Máquina de estados para los nodos Pro.Sim., AR Engine y Control GUI.

a. Rosbridge

El esquema de la Figura 13, es en realidad, una simplificación del sistema desarrollado. Los nodos *A.R. Engine, Control GUI* y *Orchestrator*, al estar desarrollados con C#, Matlab y JavaScript respectivamente, son invisibles como tal al entorno de ROS, pues no pueden utilizar las librerías nativas anteriormente mencionadas *roscpp* y *rospy*. Para salvar este problema se ha hecho uso de un paquete de ROS denominado *Rosbridge* [27]. Éste pone una disposición una API que permite tratar datos serializados en formato JSON a mensajes de ROS volcados a algún canal de comunicación y viceversa, para aquellos programas que no tienen una compatibilidad de base con ROS.

Rosbridge está compuesto de dos partes principales: el protocolo y la implementación.

El protocolo es una especificación para enviar comandos basados en JSON que puedan ser traducidos a ROS (y en teoría, cualquier otro *middleware*[5] de robot). Un ejemplo del protocolo para suscribirse a un tópico sería el siguiente:

```
{
  "op": "subscribe",
  "topic": "/cmd_vel",
  "type": "geometry_msgs/Twist"
}
```

Dicha especificación es agnóstica en cuanto al lenguaje de programación y al transporte. La idea es que cualquiera que pueda trabajar con JSON pueda hablar el protocolo de Rosbridge e interactuar con ROS ya que el protocolo cubre tópicos de suscripción y publicación, llamadas a servicios, obtención y configuración de parámetros, compresión de mensajes y más.

En cuanto a la implementación, Rosbridge es una colección de paquetes que implementan el protocolo antes mencionado y proporciona una capa de transporte WebSocket [28].

Los paquetes incluyen:

- *Rosbridge_library*: es el paquete central de *rosbridge*. *Rosbridge_library* es responsable de tomar la cadena JSON y enviar los comandos a ROS y viceversa.
- *Rosapi*: hace que ciertas acciones ROS sean accesibles a través de llamadas a servicios que normalmente están reservadas para las bibliotecas de clientes ROS. Esto incluye obtener y establecer parámetros, obtener una lista de tópicos y más.
- *Rosbridge_server*: aunque *rosbridge_library* proporciona la conversión bidireccional entre JSON y ROS, deja la capa de transporte a otros. *Rosbridge_server* proporciona una conexión WebSocket para que los navegadores puedan "hablar *rosbridge*". Por ejemplo, *Roslibjs* es una biblioteca

[5] Un *Middleware* (o lógica de intercambio de información entre aplicaciones) es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, redes, hardware o sistemas operativos

de JavaScript para el navegador que puede comunicarse con ROS a través de *rosbridge_server*.

b. Watchdogs

Para asegurar un funcionamiento robusto y fiable del sistema global y cada uno de sus nodos, se implementó un sistema de *watchdog,* los cuales verifican activamente la conectividad de cada uno de los componentes de sistema y tratan las eventuales perdidas de conexión.

Cada nodo periférico del sistema implementa obligadamente un *watchdog* y ejecuta tres funciones de forma cíclica:

- Ciclo de *Heartbeat:* consiste en un *timer* o *thread* que ejecuta una función periódica con una frecuencia global determinada, con un valor de 2Hz. En dicha función, el nodo publica constantemente sobre un *topic* propio con el formato *'/[OrchID]/heartbeat'*. De esta manera, cualquier nodo conectado a ROS es capaz de verificar la conexión activa del nodo suscribiéndose a ese *topic*.
- Ciclo de Escucha*:* se utiliza un *thread* de escucha, en modo *subscriber*, para consultar activamente los mensajes de *heartbeat* emitidos por el orquestador a fin de verificar su presencia en el sistema. Cada vez que un mensaje es escuchado, se desactiva una variable bandera (*flag*): *watchdog_timeout*.
- *Ciclo de Control:* consiste en un *timer* o *thread* que ejecuta una función periódica con una frecuencia global determinada, submúltiplo de la utilizada en el ciclo de *Heartbeat* (actualmente se utiliza 1 Hz lo cual implica una exigencia alta para el sistema). En la función de control, el nodo verifica el estado de la variable bandera *watchdog_timeout.* Si la variable está activada, se considera que el orquestador o el propio nodo se ha desconectado del sistema y se procede con la desconexión inmediata del nodo. Si está desactivada, se activa.

Funcionamiento:

En cada ciclo de control, el *flag* de *timeout* es activado. Por cada un ciclo de control y con una relación de frecuencias de ciclos de dos a uno (2:1), se tienen dos ciclos de *heartbeat* del orquestador.

En cada ciclo de *heartbeat,* el flag es desactivado por lo que, en un funcionamiento normal, el ciclo de control siempre encontrará el *flag* desactivado por más de que lo active en cada vuelta.

Ahora, si en algún momento dado la conexión entre ambos se pierde, los mensajes de *heartbeat* dejan de llegar, el *flag* deja de desactivarse, y el ciclo de control se encuentra con el *flag* desactivado, ejecutando inmediatamente la rutina de desconexión del nodo.

Para ilustrar mejor el funcionamiento, se puede observar el gráfico de la Figura 16:

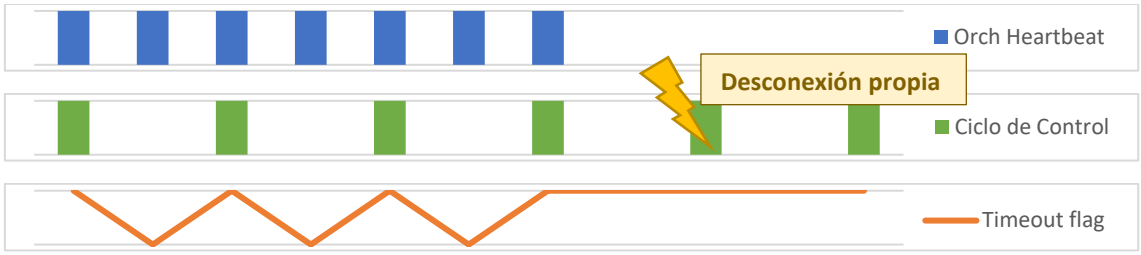

Figura 16. Funcionamiento del watchdog de un nodo periférico escuchando el heartbeat del nodo orquestador.

Suponiendo que el *heartbeat* del orquestador es emitido una milésima de segundo antes del ciclo de control, el *flag* será activado en cada ciclo de control (ilustrado en color verde en el gráfico) y desactivado en cada *heartbeat* del orquestador (ilustrado en color azul en el gráfico). El valor del *flag* oscilará, pero siempre estará desactivado al entrar en el ciclo de control. Si en algún instante ocurre que se desfasan estos ciclos, el funcionamiento sería similar ya que siempre se tendrá al menos 2 ciclos de *heartbeat* por cada ciclo de control.

Si en algún momento la conexión se pierde, el ciclo de control activa el *flag* y, al no actuar más el ciclo de *Orch Heartbeat,* en el próximo ciclo de control, el *flag* seguirá activado (rayo en el gráfico de la Figura 16). Es ahí cuando el nodo considera la conexión perdida y procede a desconectarse.

De una manera análoga, el orquestador también implementa un *watchdog*, aunque esta vez, no es solo un *heartbeat* el escuchado ni un solo *flag*, sino tantos como nodos se encuentren conectados al sistema (N nodos). De esta forma las tres funciones ejecutadas por el *watchdog* del orquestador harán lo siguiente:

- Ciclo de *Heartbeat:* consiste en un *timer* o *thread* que ejecuta una función periódica con una frecuencia global normalizada a 2Hz. En la misma, el orquestador publica constantemente sobre el *topic* '*orchestrator/heartbeat*'. De esta manera, cualquier nodo conectado a ROS es capaz de verificar la conexión activa del orquestador suscribiéndose a ese *topic*.
- Ciclo de Escucha*:* utiliza N *threads* de escucha, en modo *subscriber*, para escuchar activamente los mensajes de *heartbeat* emitidos por cada nodo conectado al sistema para verificar su presencia. Cada vez que un mensaje es escuchado, se desactiva una variable bandera (*flag*) correspondiente al nodo en cuestión: *watchdog_timeout[orchID]*, siendo *orchID* el ID asignado por el orquestador al nodo en el momento de conexión inicial del mismo.
- Ciclo de Control*:* consiste en un *timer* o *thread* que ejecuta una función periódica con una frecuencia global normalizada submúltiplo de la utilizada en el ciclo de

heartbeat (actualmente se utiliza 1 Hz lo cual implica una exigencia alta). En la función de control, el nodo verifica el estado de cada variable bandera *watchdog_timeout[orchID].* Si la variable está activada, se considera que el nodo se ha desconectado del sistema y se procede con la expulsión inmediata del nodo del sistema. Si está desactivada, se activa. Complementariamente, el ciclo de control integra un sistema muy similar para verificar la conexión activa con el proceso principal de ROS.

Como se puede constatar, el funcionamiento es análogo al de los watchdogs de los nodos periféricos, pero escuchando N *heartbeats* diferentes y manejando N *flags*, siendo N la cantidad de nodos conectados al sistema.

A continuación, un ejemplo gráfico para el nodo RR (orchID = R1) y el nodo Control GUI (orchID = M1),

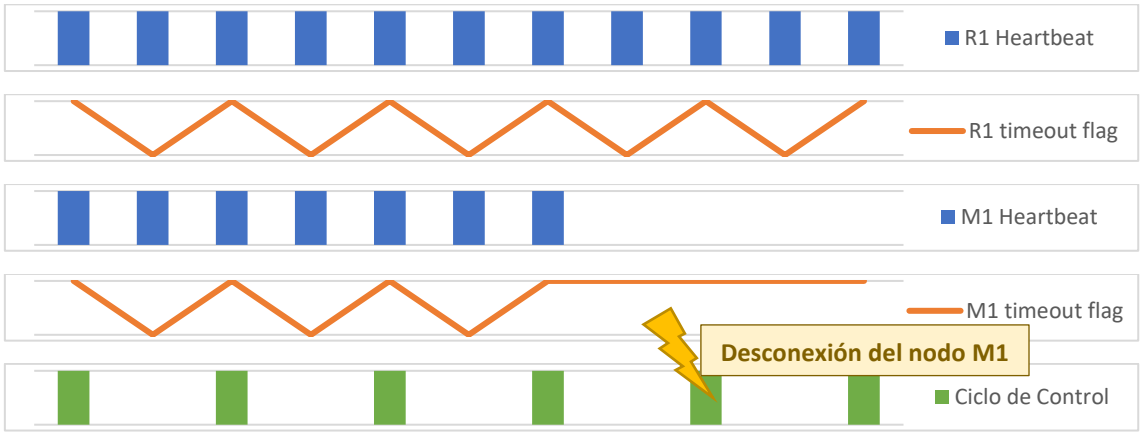

Figura 17. Funcionamiento del watchdog del orquestador cuando están presentes en el sistema dos nodos con IDs R1 y M1.

4. Componentes del sistema

a. Nodo R.R.: Driver de conectividad con robots físicos

Un nodo RR (real-robot) actúa como interfaz/driver entre el sistema y el procesador de un robot real físico. El mismo se ocupa de todos los aspectos específicos de comunicación y funcionamiento del robot brindando una interfaz genérica mediante la cual se comunica con los otros componentes del sistema (Figura 18) (Tabla 1).

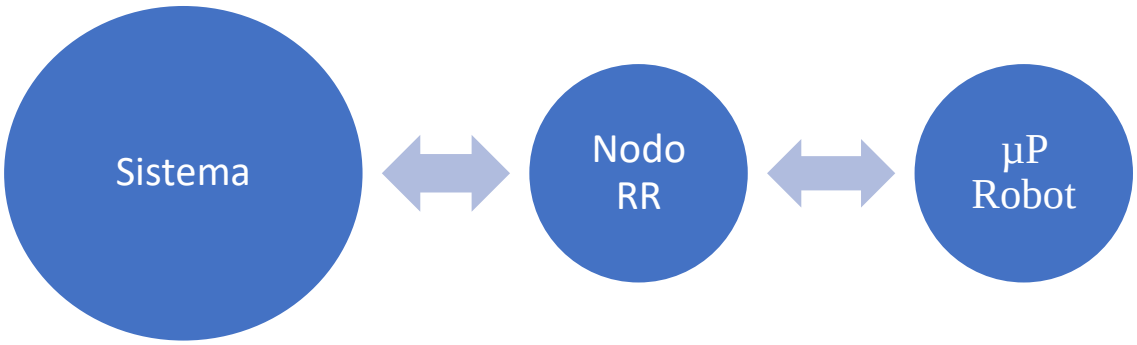

Figura 18. Esquema de conexión del sistema con el robot real.

Genérico	Específico
- Envío y recepción de variables articulares - Funciones de conectividad - Funciones de cambio de modo - Informe de estado - Informe de errores - Máquina de estados	- Comunicación remota con el robot - Inicio, configuración y apagado del robot - Protocolo de comunicación específico

Tabla 1. Tareas genéricas y específicas del driver RR.

Cuando un nodo RR se conecta al sistema, haciéndose visible ante el orquestador, adquiere automáticamente, por una cuestión de seguridad y prioridad, el control del sistema (modo *publisher*). Es decir, que el resto de los nodos, si estuviesen conectados, pasarían a modo *subscriber*, imitando los movimientos realizados por el robot real gracias a la posibilidad de lectura de los valores articulares publicados por el nodo RR sobre el *topic /joints_states*.

Actualmente solo se permite la existencia de un nodo de este tipo por instancia de sistema, lo cual podría ser mejorado en un futuro.

i. LinuxCNC Driver

Al momento de iniciar el proyecto final de estudios, dos colegas se encontraban terminando el suyo, el cuál consistía la fabricación completa de un brazo robótico industrial *open-source*: MOVEO (Figura 19).

Figura 19. Robot MOVEO.

Aprovechando la disponibilidad de este robot, se decidió usarlo como punto de partida para nuestro proyecto.

El microcontrolador central es una placa *Beaglebone* (Figura 20) que corre un sistema operativo en tiempo real, distribución Linux Debian 7.x (Wheezy) y sobre el cual funciona Machinekit, una suite de herramientas para el control de maquinaria en tiempo real. Una característica importante de este microcontrolador, es que integra dos unidades programables de tiempo real (PRU) que otorgan una robustez y sincronía vitales para una aplicación de control real. Un sistema operativo Linux estándar no puede ser usado en estos casos.

Este microcontrolador se comunica a través de sus entradas/salidas digitales con drivers que controlan los motores paso a paso de las articulaciones del robot. La estructura del robot consta de piezas fabricadas con impresoras 3D.

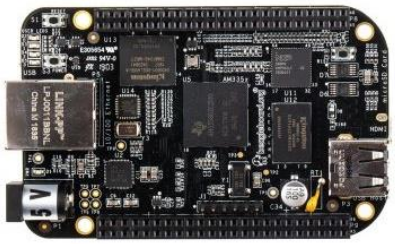

Figura 20. Placa Beaglebone.

Para la conectividad del robot real se agregó un receptor wifi USB a la placa *Beaglebone*.

Para el desarrollo del nodo ROS de este robot se configuró un simulador del mismo utilizando un software de virtualización: *VMWare Workstation Player* [29]. Se creó una máquina virtual con las mismas características y mismo sistema operativo (Debian7) sobre el cual se instaló Machinekit y se montaron los mismos archivos de configuración del robot real. Fue necesario hacer algunas modificaciones para obviar el uso de entradas y salidas digitales ahora inexistentes.

Se configuró la máquina virtual para tener una IP diferente al host. De esta manera, a los ojos del driver ROS, el simulador es equivalente al robot real.

En la siguiente imagen, se muestra la interfaz gráfica que utiliza LinuxCNC por defecto, *AXIS GUI*, la cual constituye una interfaz de control y supervisión de la máquina/robot:

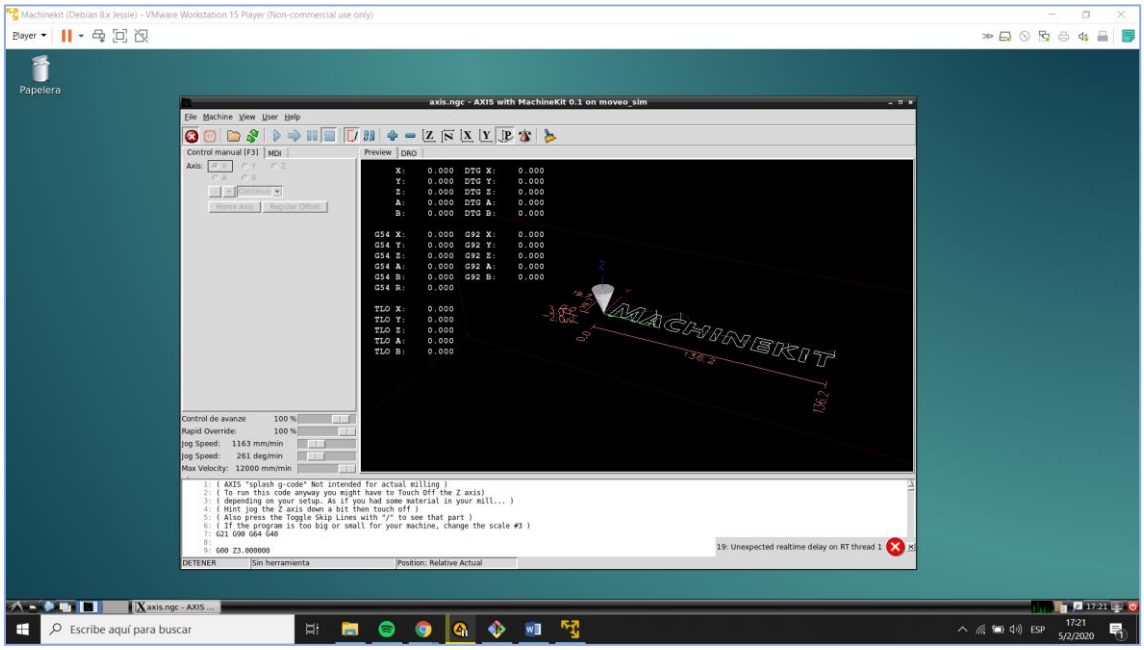

Figura 21. MachineKit corriendo sobre una máquina virtual de LinuxCNC montada sobre VMWare.

La disponibilidad de este simulador, permitió el desarrollo de una manera más ágil y prescindiendo de la limitación de contar con el robot real a disposición para continuar avanzando.

El driver RR se ha desarrollado con el objetivo de producir un código genérico reutilizable y compatible con cualquier robot o máquina herramienta que utilice LinuxCNC.

El driver consiste en 4 scripts de Python [30]:

- ***rri_driver.py***: es el script principal a ejecutar. Se encarga de la conexión inicial del el Robot (vía SSH), con ROS y con el orquestador del sistema. Gestiona los cambios de estado.

- ***modules.py***: se definen diferentes módulos que son lanzados en el *script* principal y que desempeñan diferentes tareas.
    - *JointsStatesSub*: sincroniza el robot remoto con los valores recibidos en el *topic* /joints_states cuando el nodo esta en modo *Publisher*.
    - *StandAloneSubs*: habilita el envío manual de comandos MDI o de código G al robot remoto.
    - *StatusMonitor*: conecta a ROS el canal de estado del LinuxCNC remoto. Vuelca en forma de *topics* variables de estado del robot y su controlador.
    - *ErrorMonitor*: conecta a ROS el canal de error del LinuxCNC remoto. Vuelca en forma de *topics* mensajes de error del robot y su controlador.
    - *WatchDog*: se encarga de verificar continuamente la conexión con el orquestador y tratar su eventual desconexión. Emite pulsos de *heartbeat* para informar al orquestador su presencia.
- ***ssh_functions.py:*** conjunto de funciones para tratar la comunicación vía SSH con el sistema operativo del robot remoto. Para ello hace uso de una librería de SSH para Python: *paramiko* [31]. Incluye funciones para ejecutar comandos remotamente, lanzar Python remotamente, copiar archivos por red, etc.
- ***config.py:*** parámetros de configuración del driver. Entre ellos, frecuencia de comunicación de los módulos, datos de conexión (IP, Puerto, credenciales), etc.

Desde un ordenador se lanza el nodo RR. Este, por un lado, se conecta vía SSH con el microcontrolador del robot, y por el otro, se conecta a ROS. El nodo copia y ejecuta varios scripts en el robot, con los cuales mantiene una comunicación activa. Estos scripts, a su vez, hacen uso del módulo Python LinuxCNC, el cual permite la comunicación con el sistema de control del robot o máquina herramienta, tanto para enviar comandos como para consultar el estado y los eventuales errores.

De esta forma, en modo *subscriber*, las variables articulares leídas en el *topic /joints_states* son transformadas en líneas de código MDI y son enviadas al intérprete de comandos del LinuxCNC. De forma inversa, en modo *publisher*, las variables articulares leídas en el canal de estado LinuxCNC son volcadas continuamente sobre el *topic /joints_states*.

Además de los servicios comunes detallados en la sección del funcionamiento general, este nodo también provee el servicio */orchID/toggle_error_channel.* El mismo puede ser invocado para habilitar el volcado de errores LinuxCNC sobre ROS, por defecto deshabilitado. Estos pueden ser desde errores del intérprete de comandos (ej. mala sintaxis de una línea de comando, consigna fuera del espacio de trabajo del robot) hasta errores de conexión de motores o sensores del robot detectados por LinuxCNC [32].

Es importante aclarar que el canal de errores de LinuxCNC es una pila de tipo FIFO, por lo que, si se habilita el canal de error, los mismos dejan de aparecer en la interfaz gráfica

remota del robot (AXIS GUI). Además, la rutina de *polling* que gestiona el canal genera un consumo de recursos considerable para el microcontrolador del robot.

El envío de comandos manuales (deshabilitado por defecto en el código) debe utilizarse con cuidado ya que corre paralelamente a ROS y puede interferir con el sistema. Al iniciar el módulo ***StandAloneSubs*** el nodo escucha mensajes por los siguientes *topics*:

- *'real_robot_interface/linuxcnc/cmd/mdi_line' – String – Ex: "G1 X0 Y5 F800"*
  Ejecuta la línea de código MDI remotamente. String = línea de código G
- *'real_robot_interface/linuxcnc/cmd/gcode_file' – String – Ex: "test.ngc"*
  Envía al robot por red el archivo de código G con ese nombre y disponible en la carpeta *'gcode_files'* y lo ejecuta.

b. Nodo Control GUI: Integración interfaz Matlab de ROSSETA Lab.

En el dictado de la cátedra Robótica I de la facultad se utiliza para la parte práctica una interfaz y una simulación de Matlab desarrollada por el ingeniero Eric Sánchez. La misma está inspirada en el *toolbox* para robótica desarrollado por Peter Corke y es de gran ayuda para los alumnos a la hora de aprender y poner en prácticas temáticas elementales de la robótica.

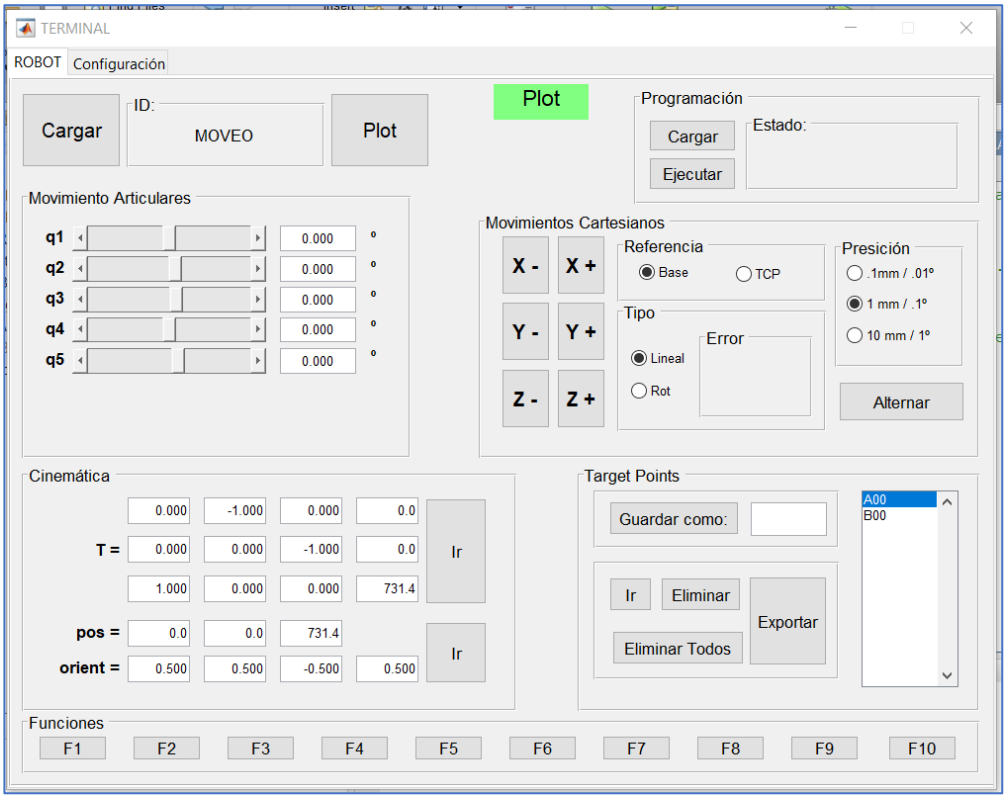

Figura 22. Modelo de teach-pendant desarrollado en Matlab y utilizado por la cátedra de Robótica 1 y por ROSETTA Lab.

Al ejecutar el programa, se inicia una interfaz gráfica (Figura 22) en la cual es posible cargar un robot específico a utilizar cargando un archivo de extensión ".m" que importa

variables propias del robot en cuestión: parámetros de Denavit-Hartenberg, *offsets* articulares, etc. Una vez cargado el robot, al presionar el botón "*Plot*", se inicia una simulación del robot en otra ventana para la cual Matlab recupera archivos STL del modelo del robot (Figura 23).

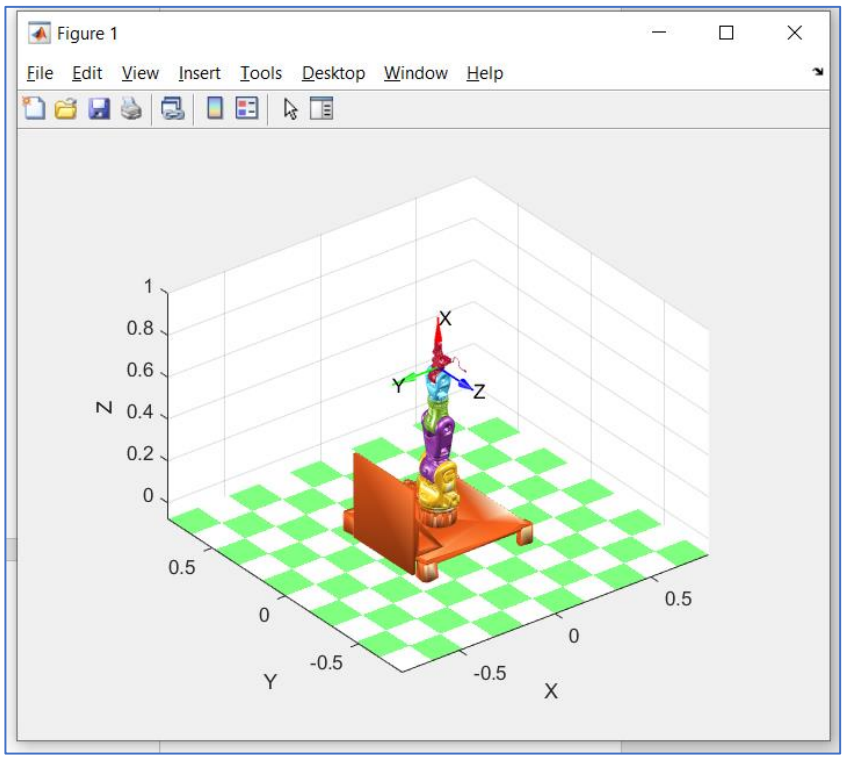

Figura 23. Visualización del robot MOVEO en Matlab.

Es posible entonces manipular el robot de diferentes maneras utilizando la interfaz: variando directamente las variables articulares del robot, variando la posición cartesiana con referencia a la base o al efector final del robot, ingresando la matriz de transformación o valores de posición y orientación del efector final, etc. Además, la interfaz permite exportar puntos e importar rutinas de movimientos interpolados.

Su enfoque es principalmente didáctico para la comprensión y puesta en práctica de conceptos de robótica industrial.

La interfaz fue retomada para controlar un robot presentado como proyecto final de la cátedra. Se desarrolló para ello una comunicación por puerto serie unidireccional para el envío de variables articulares cada vez que un cambio tuviera lugar en la interfaz.

Durante la realización de este proyecto, se incorporó esta interfaz al sistema completo a modo de *teach-pendant* (dispositivo para programar manualmente un robot industrial).

Para ello, se desarrolló un driver que actúa como interfaz entre el sistema global y el programa de Matlab, utilizando como herramienta principal de conexión la librería

roslibpy [33]. El mismo consta de una serie de scripts Python que se describen a continuación:

- ***control_gui.py:*** es el script principal lanzado por Matlab. Se encarga de la conexión con ROS, la comunicación con el orquestador y los eventuales cambios de estado del nodo.
- ***modules.py:*** se definen diferentes módulos que son lanzados en el script principal y que desempeñan diferentes tareas.
    - *GUI:* este módulo, desarrollado con las funcionalidades que provee *TkInter* [34], lanza y gestiona una pequeña interfaz gráfica en la cual se gestiona la conexión con el sistema y el orquestador.
    - *Syncronizer:* este módulo se encarga de sincronizar los valores de las variables articulares bidireccionalmente entre Matlab y el sistema, con la ayuda de la librería *pySerialI* [35].
    - *StatusMonitor:* vuelca sobre la interfaz gráfica los valores leídos sobre el *topic /joints_states*.
    - *Watchdog:* se encarga de verificar continuamente la conexión con el orquestador y tratar su eventual desconexión. Emite pulsos de *heartbeat* para informar al orquestador su presencia.
- ***ssh_functions.py:*** utilizado en el modo *Standalone* del driver desarrollado más adelante. Conjunto de funciones para tratar la comunicación directa con el robot real vía SSH.
- ***config.py:*** parámetros de configuración del driver. Entre ellos, frecuencia de comunicación de los módulos, datos de conexión (IP del puente *Rosbridge*), etc.

La comunicación del driver con el programa de Matlab se realiza a través de un puerto serie virtual o emulado aprovechando que la interfaz ya era capaz de establecer una comunicación unidireccional por puerto serie antes de ser retomada para incorporarla al sistema. Para ello se ha utilizado el software **VSPE (Virtual Serial Ports Emulator)** de *Eterlogic*, el cual permite simular un par de puertos series virtuales conectados entre sí. A uno de ellos se conectará Matlab y al otro, el driver [36].

Una vez instalado el programa, la configuración es realizada por un archivo denominado “sim_COM1.vspe”, el cual es ejecutado al pulsar el botón “RUN VSPE” en la pestaña de configuración de la interfaz de Matlab (Figura 24). Una vez en marcha, si se escanean los puertos en la sección “Puerto” de la misma pestaña, se podrá encontrar el puerto COMx al cual se debe conectar.

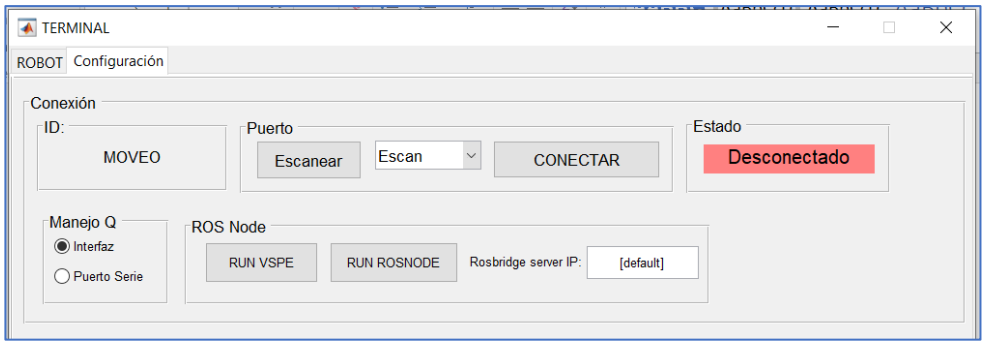

Figura 24. Desarrollo complementario sobre la Interfaz de control previamente desarrollada, para incorporar la posibilidad de conectarse a un entorno de ROS.

Una vez que la conexión por puerto serie ha sido exitosa, es posible entonces iniciar el nodo ROS pulsando sobre el botón "RUN ROSNODE" el cual ejecuta paralelamente el driver lanzando el script ***control_gui.py***.

El script inicializa una pequeña interfaz gráfica sobre la cual se vuelcan mensajes de funcionamiento del nodo a modo de consola y sobre la cual es posible gestionar los cambios de modo *Publisher*/*Subscriber* (Figura 25).

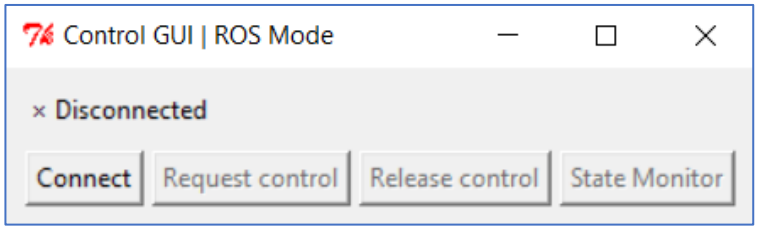

Figura 25. Interfaz gráfica lanzada luego de pulsar RUN ROSNODE.

Para integrarse al sistema global, el driver se conectará a ROS a través de *Rosbridge*, el cual, como describió anteriormente, es un paquete que permite la conexión con ROS a sistemas sobre los cuales la suite de ROS no se encuentra instalada. Se utiliza esta metodología ya que ROS se desarrolló en Linux y, a pesar de que existe una versión lanzada recientemente para Windows, la misma es inestable y compleja de compilar e instalar. Por otro lado, no existe una versión estable de Matlab para Linux.

Una vez que *rosbridge* y el orquestador están en ejecución, pulsando el botón "*Connect*", el driver intentará establecer la comunicación con el sistema y realizará un pedido de conexión al orquestador. Es importante que la IP del ordenador sobre el cual corre el *rosbridge_server* sea correctamente ingresada en el campo correspondiente de la interfaz de Matlab (pestaña Configuración) o, en su defecto, en el script ***config.py***.

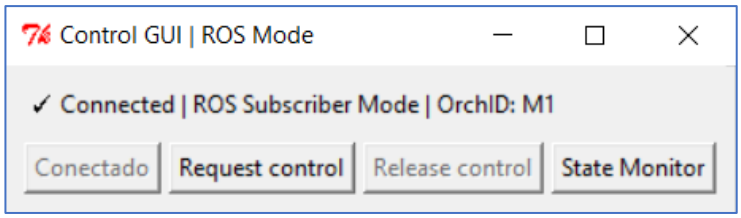

Figura 26. Visualización de la interfaz una vez que ha establecido conexión con el orquestador.

Si la conexión es exitosa, el nodo pasa automáticamente al modo *Subscriber* o de escucha, y se muestra el *OrchID* asignado por el orquestador. Para tomar el control del sistema, es necesario hacer una llamada al nodo orquestador pulsando el botón "*Request Control"*, el cual puede ser aceptado o rechazado según lo desee dicho nodo. El botón "*State Monitor*" permite volcar sobre la interfaz los valores leídos desde el *topic /joints_states.*

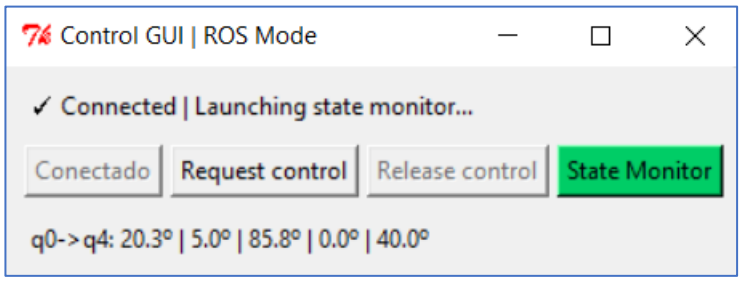

Figura 27. Visualización de información adicional si se ha hecho click sobre "State Monitor".

La comunicación por puerto serie previamente existente en la interfaz de Matlab consistía en el envío de las variables articulares normalizadas y codificadas en valores hexadecimales cada vez que las mismas eran modificadas. Este funcionamiento se conservó y, cuando el nodo está en modo *Publisher*, el driver decodifica los mensajes, recupera los valores de las variables articulares y las publica sobre el *topic /joints_states*.

Sin embargo, fue necesario pedir al ingeniero Eric Sánchez, el desarrollador de la interfaz, mejorar esta comunicación para convertirla en bidireccional y hacer posible no solo el envío, sino también la recepción de mensajes del lado de Matlab. De esta manera, estando el nodo en modo *Subscriber*, el driver envía por puerto serie los valores articulares leídos en */joints_states*, Matlab los recibe y actualiza la interfaz y simulación consecuentemente. Para ello, es importante configurar en la pestaña "Configuración" de la interfaz el ítem "Manejo Q" en la opción "Puerto Serie" (Figura 28).

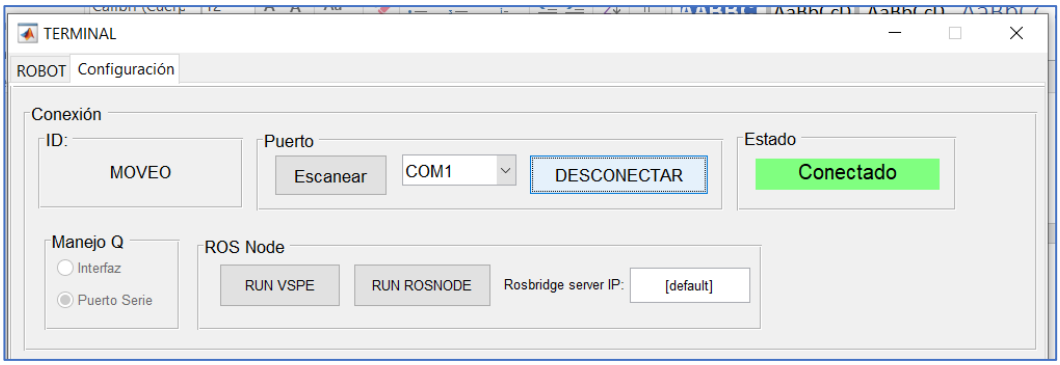

Figura 28 Visualización de la interfaz complementaria cuando el nodo es visible dentro del entorno de ROS.

i. Control GUI – Standalone Mode

Existe un modo de funcionamiento alternativo del nodo Control GUI el cual puede ser configurado en el archivo ***config.py*** del driver. Es el modo independiente o *Standalone*, en el cual el driver prescinde de ROS y del orquestador y se conecta directamente al robot real vía SSH. Para ello, importa el mismo archivo ***ssh_functions.py*** utilizado por el nodo RR – Realrobot, y utiliza las credenciales, IP y puerto definidos en el archivo ***config.py***.

Configurando la variable *'ros_mode = False'* en el archivo ***config.py***, el driver se iniciará en modo *Standalone,* tanto si se lanza desde Matlab con el mismo botón "RUN ROSNODE" (aunque esta vez sin ROS), como si se lanza manualmente desde consola ejecutando el archivo ***control_gui.py***. La interfaz generada ahora es, como se muestra en la Figura 29, ligeramente distinta.

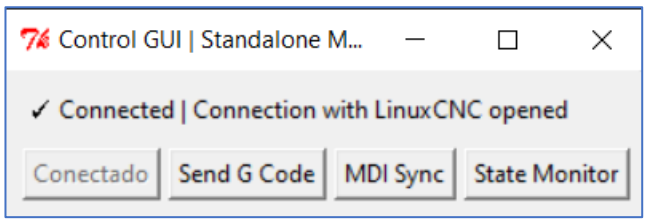

Figura 29. Interfaz en modo "standalone".

Si activamos el monitor de estado (*State Monitor*) podemos obtener información de las posiciones articulares del robot así como del intérprete de comandos de LinuxCNC. En el siguiente ejemplo (Figura30), el botón de parada de emergencia no está liberado y la máquina no está encendida lo cual LinuxCNC informa con los flags ESTOP y ENABLED respectivamente.

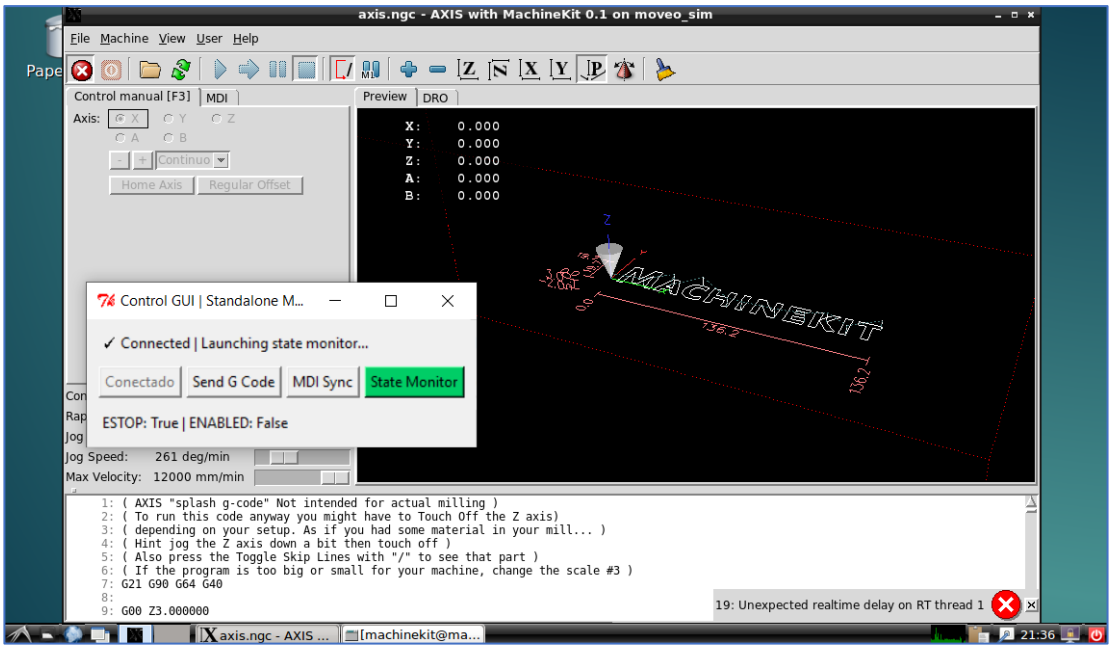

Figura 30. MachineKit comunicándose con Matlab en un estado inicial, cuando la máquina no está encendida.

Si ponemos en funcionamiento la máquina simulada o robot real liberando la parada de emergencia y energizando los motores, el monitor de estado nos comunicará el modo de funcionamiento, el estado del intérprete de comandos y los valores de las variables articulares del robot (Figura 31)

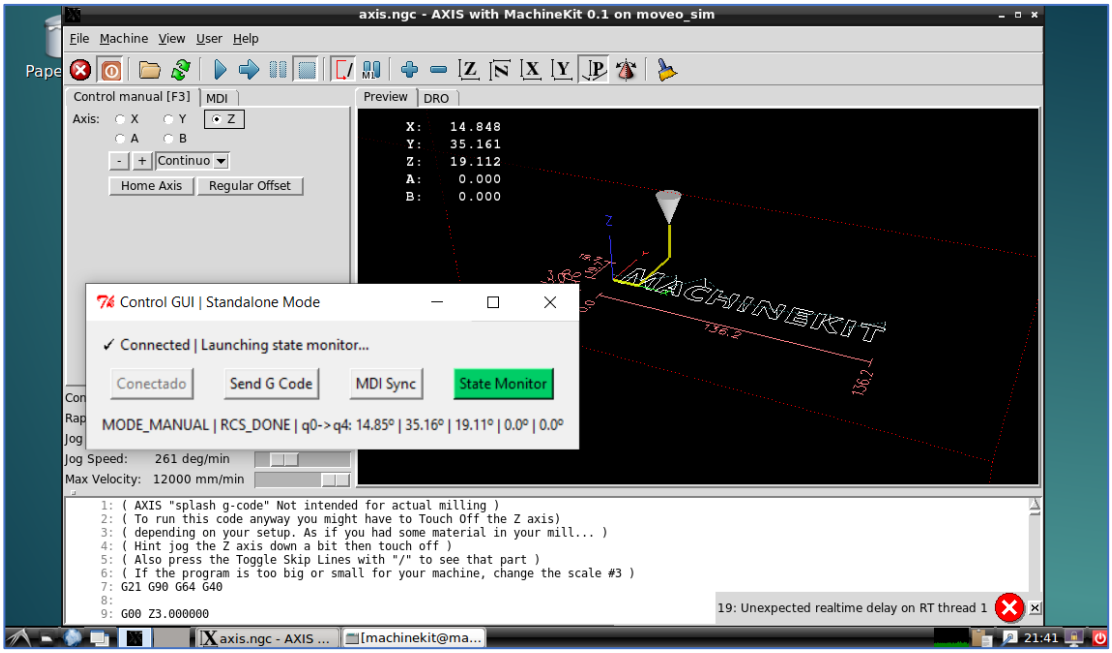

Figura 31. MachineKit comunicándose con Matlab, una vez que la máquina se ha encendido.

El botón “Send G Code” envía por red el archivo definido en el script ***control_gui.py*** al robot y lo ejecuta remotamente.

El botón “MDI Sync” inicializa el sincronizador de movimientos, el cual transmite los valores articulares desde Matlab hacia el robot remoto (Figura 32). Si se lo compara con el modo de funcionamiento descrito anteriormente, Matlab se encontraría en modo *Publisher* y el robot real en modo *Subscriber*.

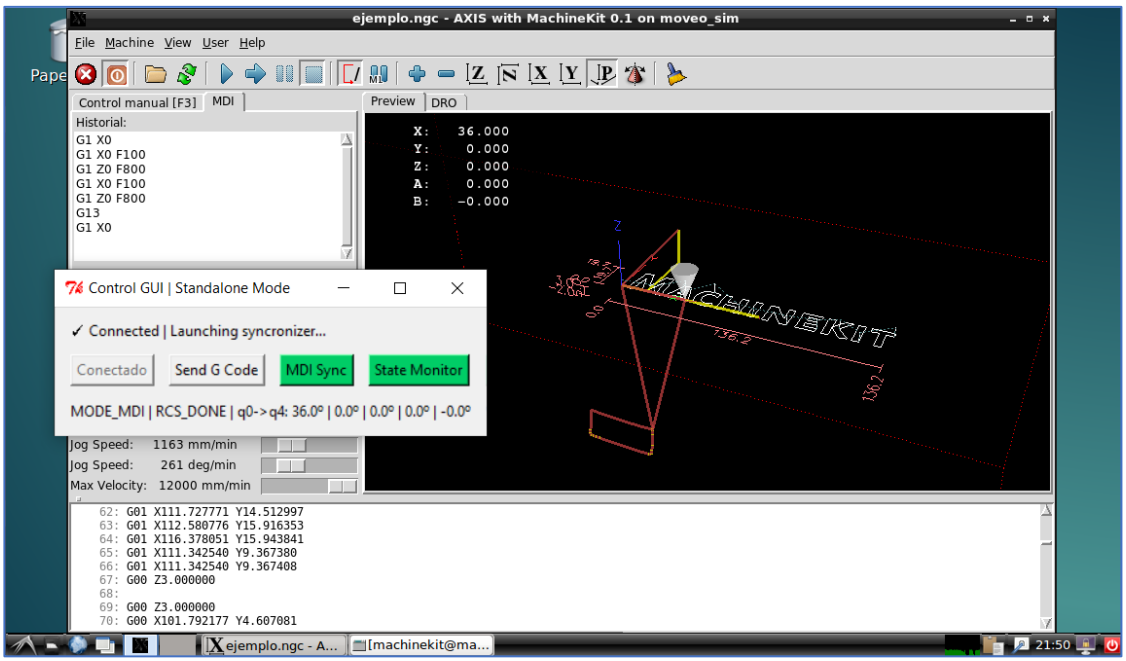

Figura 32. Modo standalone con sincronizador de movimientos activado. Visualización en microprocesador del robot real.

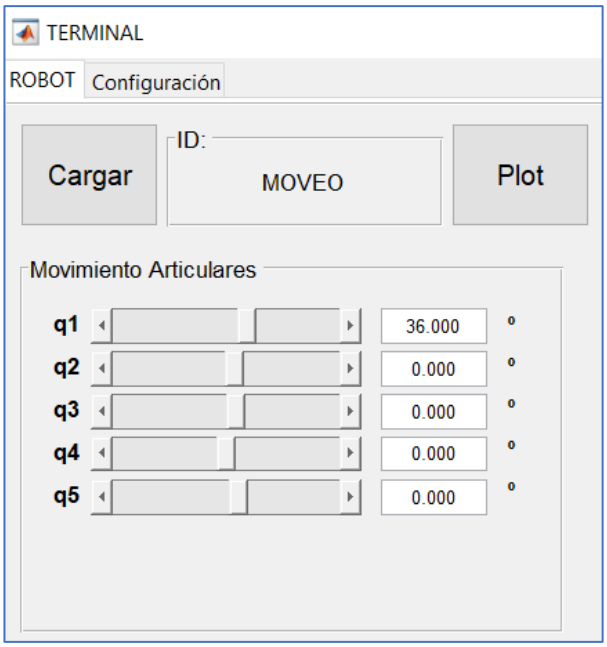

Figura 33. Modo standalone con sincronizador de movimientos activado. Visualización en Matlab.

Este funcionamiento es análogo a un robot real industrial y su *teach-pendant* (Figura 33). Actualmente el funcionamiento inverso no está disponible en modo *Standalone*.

c. Nodo ProSim: Simulador industrial (VRep/CoppeliaSim)

El nodo ProSim integra al sistema un simulador industrial en donde el robot podrá incorporarse en el seno de un proceso industrial simulado en el cual podría interactuar con otros objetos virtuales.

Como simulador industrial se ha adoptado V-Rep, un simulador muy potente utilizado para aplicaciones tales como: prototipado rápido, automatización industrial simulada, monitoreo remoto, educación en robótica, etc. Durante la realización del proyecto una versión nueva del simulador fue lanzada ahora con el nombre de *CoppeliaSim* y se continuó trabajando esta arista del proyecto sobre esta nueva plataforma.

Este simulador corre en sistemas operativos tipo Linux y, entre las librerías disponibles, incluye la *ROS Interface API*, un conjunto de rutinas que permite la comunicación entre el simulador *CoppeliaSim* y ROS. Para su activación es necesario copiar en la carpeta del simulador el archivo *'libsimExtROSInterface.so'* [37]. [6]

Para definir el robot a simular, se adoptó la convención de ROS de utilizar el formato URDF *(Unified Robot Description Format)* [38]. Un fichero de extensión ".urdf" es un archivo de formato XML que define los eslabones y articulaciones de un robot en forma de etiquetas o *"tags"*.

Los eslabones se definen de la siguiente manera:

```
<link name="linkX">
  <inertial>
    <mass value="XX" />
    <inertia ixx="XX" ixy="XX" ixz="XX" iyy="XX" iyz="XX" izz="XX" />
  </inertial>
  <visual>
    <geometry>
      <mesh filename="[stl_path].dae" />
    </geometry>
  </visual>
  <collision>
    <geometry>
      <mesh filename="[stl_path].dae" />
    </geometry>
  </collision>
</link>
```

Cada link posee un nombre como atributo y, dentro, 3 elementos:

- *Inertial:* conjunto de parámetros de inercia, masa y matriz de inercia.
- *Visual:* elementos visuales que serán renderizados por el simulador. Pueden ser elementos sencillos predefinidos (*box, cylinder*, etc.) o archivos de elementos digitales 3D en formato STL, COLLADA, etc.

[6] Para este proyecto en particular, fue necesario recompilar esta librería para incorporar un paquete de servicios no estándar de ROS (cob_srvs) [39].

- *Collision:* elementos que el motor físico del simulador utilizará para evaluar colisiones con otros objetos de la simulación.

Las articulaciones se definen de la siguiente manera:

```
<joint name="linkX_to_linkY" type="revolute">
  <origin xyz="0 0 0.5" rpy="0 0 0"/>
  <parent link="linkX"/>
  <child link="linkY"/>
  <axis xyz="0 0 1" />
  <limit lower="XX" upper="XX" effort="500" velocity="60"/>
</joint>
```

Cada articulación posee un nombre y un tipo de articulación como atributos. Dentro:

- *Origin:* origen del link hijo (n+1) referido al link padre (n).
- *Parent:* nombre del link padre (n).
- *Child:* nombre del link hijo (n+1).
- *Axis:* eje de rotación de la articulación.
- *Limit:* limites articulares, de velocidad y esfuerzo.

Respetando este formato, se escribieron los archivos URDF correspondientes a los robots utilizados en el proyecto. Es posible importar estos archivos en el simulador *CoppeliaSim*, el cual los interpreta y genera el robot simulado. En la Figura 34 puede verse el robot MOVEO importado y el árbol de eslabones y articulaciones a la izquierda:

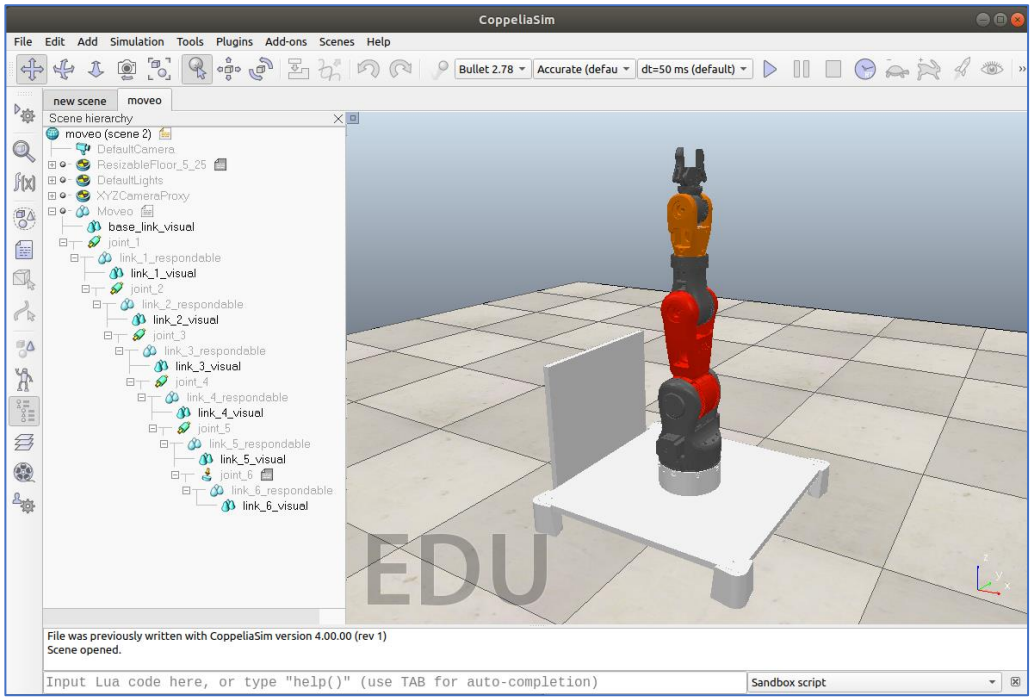

Figura 34. Montaje de la escena con el Robot MOVEO en V-Rep.

En *CoppeliaSim* es posible asociar scripts a los objetos simulados (*child scripts*) que realicen diferentes rutinas síncronas (*non-threaded*) o asíncronas (*threaded*) durante la simulación. Estos scripts se escriben en lenguaje de programación ***Lua***, y tienen la

ventaja de ser portátiles, escalables y optimizados para su ejecución con el bucle de simulación.

Se desarrolló entonces un script síncrono (*non-threaded child script*)[7] en el cual se integró la totalidad del código de gestión del nodo ROS y se asignó el mismo al objeto robot surgido del archivo URDF.

El script desarrollado implementa la máquina de estado anteriormente mencionada. Para el control de la simulación, se desarrolló una pequeña interfaz gráfica dentro del mismo script, que permite manipular las variables articulares a través de *sliders*, conectarse al sistema global y realizar los eventuales cambios de modo de funcionamiento.

[7] Estructura básica de un *non-threaded child script*

```
function sysCall_init()
    -- Código de inicialización (Ejecutado solo una vez al inicio)
end

function sysCall_actuation()
    -- Modificaciones a realizar en cada bucle de simulación
end

function sysCall_sensing()
    -- Lecturas/sensados a realizar en cada bucle de simulación
end

function sysCall_cleanup()
    -- Código de finalización (Ejecutado solo una vez al final)
end
```

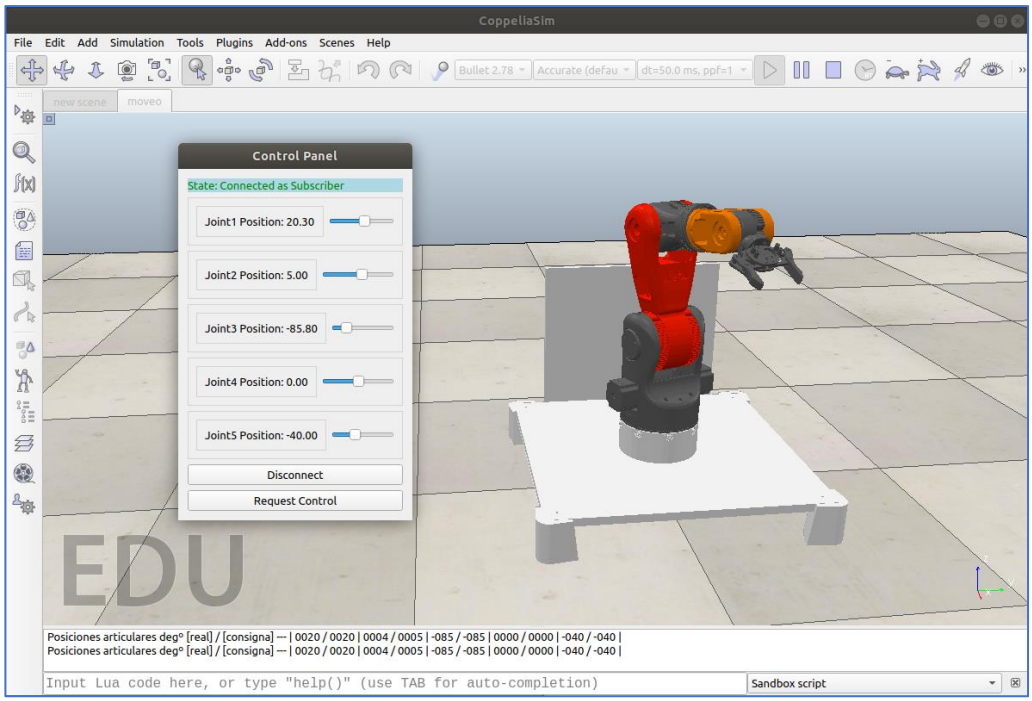

Figura 35. Robot MOVEO en V-Rep en modo Subscriber. El Robot copia la postura que llega por el tópico /joints_states. En modo Disconnected y modo Publisher, la posición articular de cada articulación se controla mediante sliders

En modo *Subscriber*, el script se pone a la escucha de los valores articulares publicados sobre el *topic /joints_states* y pasa esos valores como consignas de posición a los controladores de posición simulados de los motores de las articulaciones. De esta manera, el robot simulado sigue los movimientos del nodo que tenga asignado el control.

En modo *Publisher*, el control fue pedido y otorgado al nodo por el orquestador. El script publica sobre sobre el *topic /joints_states* los valores articulares del robot en cada bucle de simulación.

Es importante aclarar que el simulador *CoppeliaSim* debe ejecutarse en un ordenador con ROS instalado, más allá de si es el ordenador donde está corriendo *roscore* (proceso principal de ROS) o no.[8]

Al iniciar la simulación, se puede variar la postura del robot con sliders. Para conectarlo al sistema global es necesario pulsar el botón "*Connect*" de la GUI. Entonces, el nodo intenta conectarse al orquestador y, si la conexión es exitosa, obtiene del mismo una confirmación acompañada de un ID único para ese nodo, su *OrchId*.

[8] Este último caso se da cuando se corre ROS de manera distribuida en múltiples ordenadores. Un ordenador corre el proceso principal *roscore* y los demás se comunican con este asignando su IP a la variable de entorno global ROS_MASTER_URI antes de iniciar los nodos.

Para cambiar de modo su utiliza el botón "*Request Control*" / *"Release Control"*, el cual realiza una petición al orquestador y cambia consecuentemente de nodo si la misma es exitosa.

d. Nodo AREngine: Motor de Realidad Aumentada (Unity/Vuforia)

Este nodo ha sido desarrollado con el fin sumar contenido didáctico de realidad aumentada y al alcance de cualquier usuario que quiera utilizar el sistema. Como resultado se ha obtenido una aplicación móvil que puede ser utilizada en dispositivos Android. Para llegar a concebir la misma se han utilizado múltiples paquetes disponibles para Unity, cada uno de los cuales aportan una funcionalidad característica de la aplicación.

i. Vuforia

La principal atracción de esta aplicación es el uso de realidad aumentada, el cual es posible gracias al desarrollo realizado por Qualcomm, y conocido con el nombre de Vuforia. Vuforia es un kit de desarrollo de software (SDK, por sus siglas en inglés) de realidad aumentada para dispositivos móviles que permite la creación de aplicaciones que explotan esta nueva tecnología. Utiliza tecnología de visión por computadora ("*computer visión*") para reconocer y seguir imágenes planas y objetos 3D (denominados "*targets*") en tiempo real. Esta capacidad de identificación y seguimiento de imágenes permite a los desarrolladores posicionar y orientar objetos virtuales, en relación con objetos del mundo real cuando se ven a través de la cámara de un dispositivo móvil. El objeto virtual sigue la posición y orientación de la imagen en tiempo real para que la perspectiva del espectador sobre el objeto se corresponda con la perspectiva del objetivo. Por lo tanto, parece que el objeto virtual es parte de la escena del mundo real.

Todas las aplicaciones de Vuforia utilizan una clave única que se obtiene a través del *Vuforia License Manager*. Vuforia ofrece una licencia gratuita con la que se pueden desarrollar aplicaciones con un límite de uso mensual. Además, ofrecen diferentes alternativas de pago para aplicaciones que vayan a ser lanzadas al mercado. En el caso de este proyecto, al tratarse de un trabajo académico, se ha utilizado una licencia gratuita.

Vuforia necesita registrar los *targets* dentro de una base de datos asociados a una clave o licencia de desarrollo para luego ser importados dentro de Unity. Es importante para el correcto funcionamiento del motor de reconocimiento de imágenes y objetos de Vuforia que el target diseñado tenga una importante cantidad de vértices. El siguiente *target* ha sido diseñado con el objetivo de incentivar a los usuarios al reconocimiento de los robots del laboratorio, como así también de los robots más utilizados. (Figura 36)

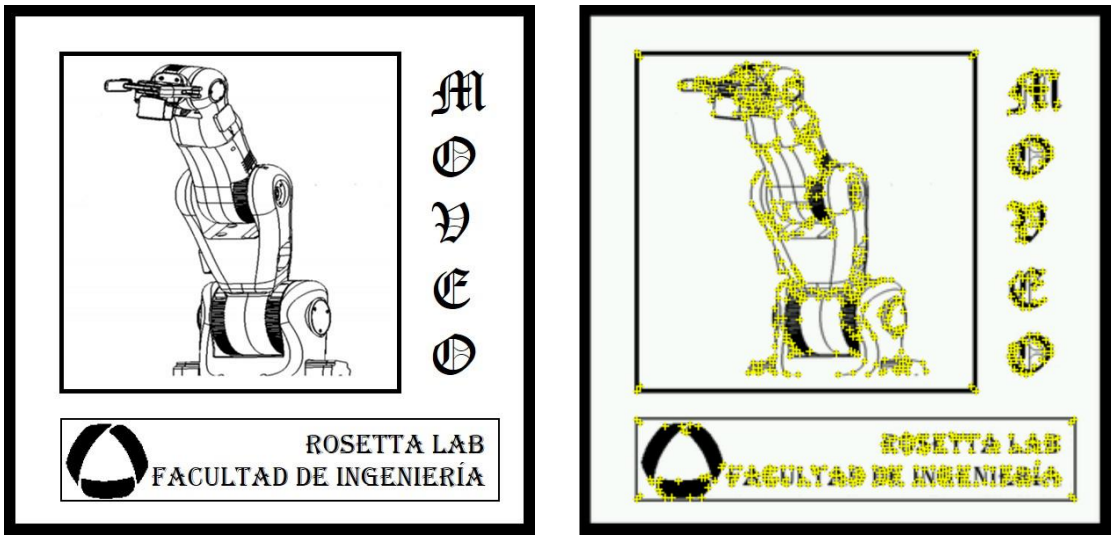

Figura 36. Targets cargados en el portal de desarrollo de Vuforia. A la izquierda se muestra la imagen cruda, y a la derecha el análisis de identificación de vértices realizado por el motor de reconocimiento de imágenes de vuforia.

ii. ROS

ROS# (Figura 37) es un conjunto de librerías y herramientas de software de código abierto desarrolladas en C# para comunicarse con ROS desde aplicaciones .NET, en particular Unity [40].

Figura 37. Logo del paquete ROS Sharp.

ROS# permite, entre otras cosas:

- Comunicarse con ROS desde una aplicación de Windows (subscribirse y publicar tópicos, llamar y anunciar servicios, establecer y obtener parámetros y usar todas las funciones proporcionadas por *rosbridge*).
- Usar modelos URDF de robots como objeto de importación en Unity para crear los *GameObject* correspondientes.
- Controlar un robot real a través de Unity.
- Visualizar el estado actual del robot y la información de sensores en Unity.
- Simular un robot en Unity con los datos proporcionados por el URDF y sin usar una conexión a ROS. Además de los componentes visuales como mallas y texturas, también se importan parámetros de articulaciones, masas, centros de masa, inercia y especificaciones de colisión de cuerpos rígidos y se utilizan para la simulación física en Unity.

iii. Lean Touch

Lean Touch es un paquete que facilita la implementación de gestos táctiles (tales como estirar, comprimir o girar) cuando se quiere desarrollar aplicaciones móviles [41].

Unity hace de esta tarea algo difícil de realizar, ya que solo proporcionan la matriz *Input.touches* a partir de la cual es necesario realizar todos los cálculos de rotación y redimensionamiento. Lean Touch permite abstraerse de esta tarea. Además, permite simular gestos multitáctiles en la aplicación de escritorio, para que no tener que perder tiempo desplegando la aplicación en un dispositivo móvil mientras se configura su entrada.

iv. Descripción de los objetos principales de la aplicación

Luego de la instalar los paquetes de funcionamiento antes mencionados, se comenzó con el desarrollo de la aplicación.

Primeramente, se importaron los targets de Vuforia al mundo virtual de Unity (Figura 38), y luego se importó el robot correspondiente en formato URDF y se colocó en la escena (Figura 39).

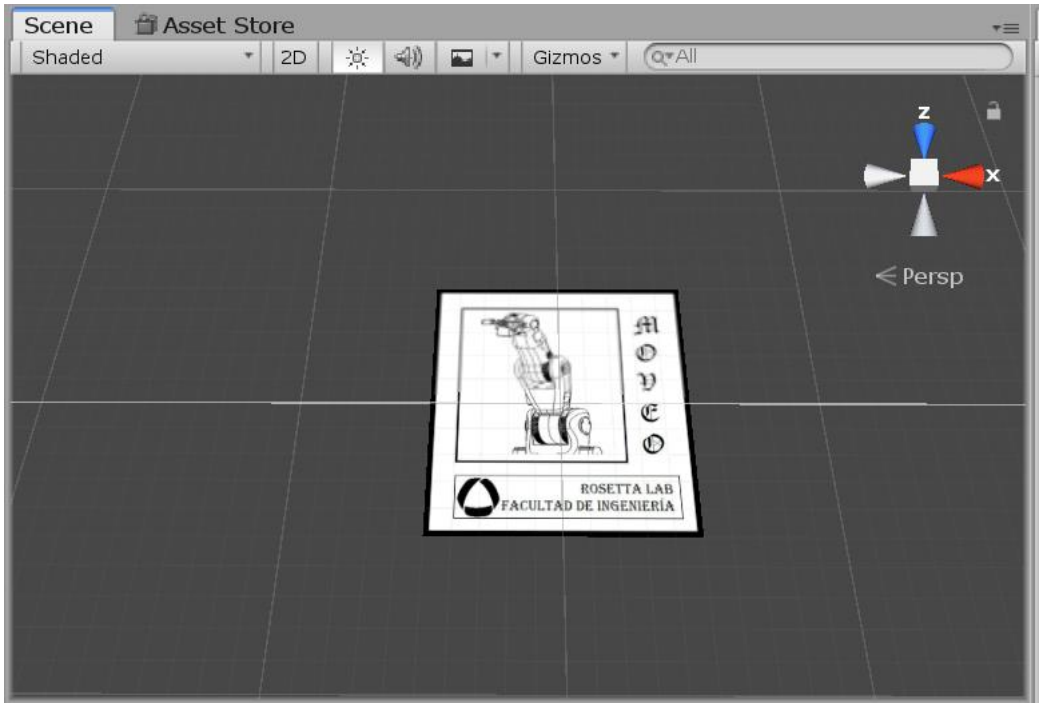

Figura 38. Incorporación del target de la base de datos de Vuforia en la escena de Unity.

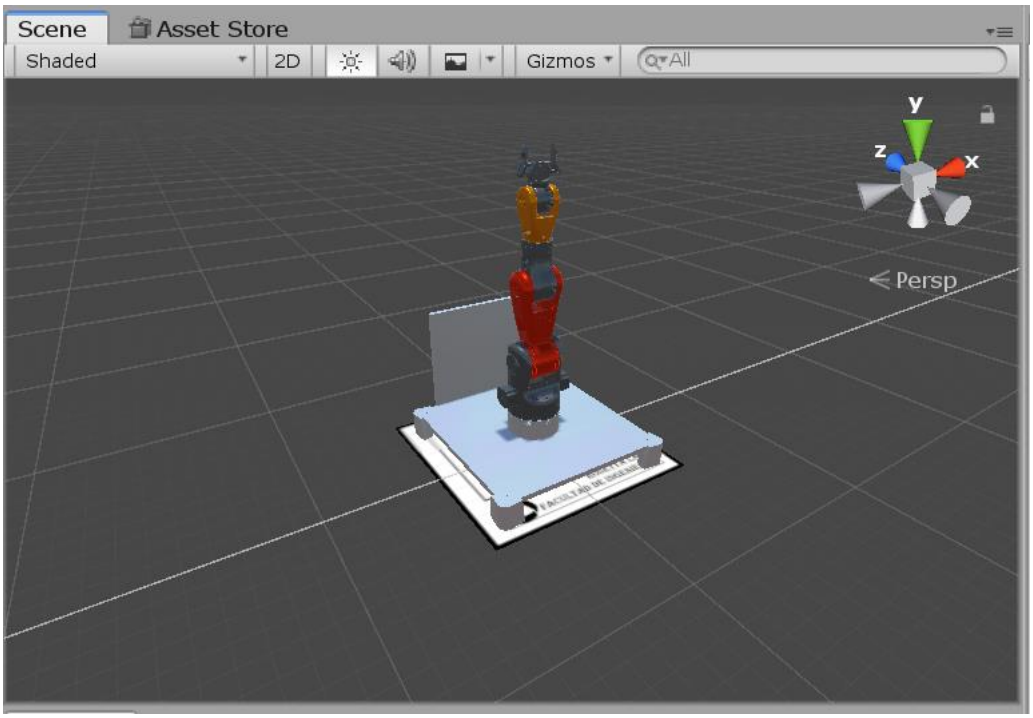

Figura 39. Montaje del robot MOVEO sobre su target.

Como se describió anteriormente, la aplicación funciona con un esquema de máquina de estados. El desafío de implementar dicho esquema en un entorno de desarrollo como el que propone Unity es que debe ser posible que un objeto del mundo virtual modifique el comportamiento de otro objeto, que dicho objeto se actualice o aparezca en la escena, y así sucesivamente, en función de los estados por los que va pasando la aplicación. Para no crear una interdependencia anidada de *GameObjets* imposible de mantener o hacer abuso de las funciones de búsqueda de objetos en la escena activa que propone Unity (y que tienen un alto costo computacional) se optó por utilizar como patrón de diseño un "*Singleton*". El *Singleton* (o instancia única) permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella. El patrón *Singleton* se implementa creando en la clase en cuestión, un método que crea una instancia de sí mismo sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (en este caso se realiza dentro de la función "*Awake*" de Unity, la cual es invocada antes de que la aplicación comience a ejecutar la escena de juego). La instrumentación del patrón puede ser delicada en programas con múltiples hilos de ejecución. Si dos hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía, solo uno de ellos debe lograr crear el objeto. La solución clásica para este problema es utilizar exclusión mutua en el método de creación de la clase que implementa el patrón [42].

El *Singleton* descrito está a cargo del *GameObject* denominado "*PersistentManager*", el cual, además de implementar la máquina de estados, tiene referencia a todos los objetos de interfaz con el usuario y de conexión que se actualizan en función del estado (Figura 40).

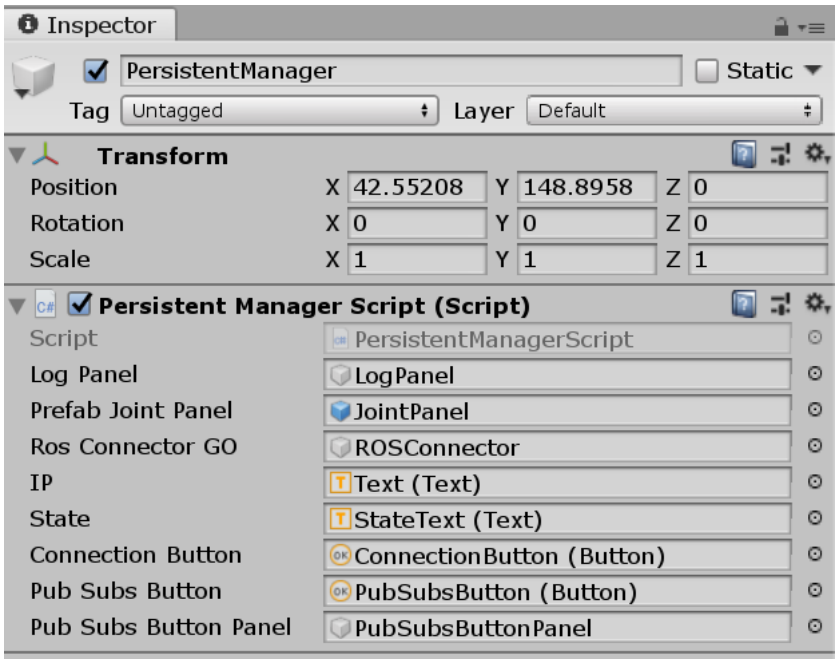

Figura 40. Componentes del GameObject "Persistent Manager".

Dichos objetos están presentes en la escena dentro de un "*Canvas*" dentro del cual se colocan todos los elementos de interfaz de usuario, tales como botones, entradas de texto y otros, y donde se manejan los eventos correspondientes.

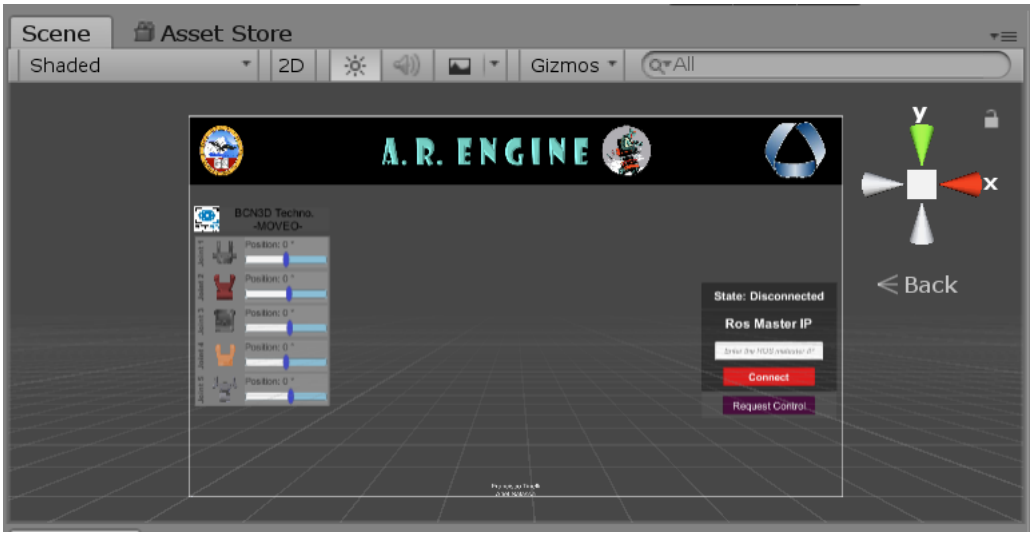

Figura 41. Canvas desarrollado para el nodo AR Engine.

El *Canvas* de la aplicación, tal como se muestra en la figura 41, se divide en distintos paneles de visualización. Tanto en la parte superior como inferior se encuentran los

paneles estáticos. En la parte superior se encuentra la cabecera con el nombre del nodo e imágenes estáticas y en la parte inferior, se ha colocado un pie de página con la firma de los autores. A la izquierda se encuentra el panel de control de las articulaciones, que se genera dinámicamente según sea el *target* detectado. En el mismo se presentan los *sliders* que imprimen la posición articular a la articulación correspondiente del robot, cuando el programa se encuentra en modo *Disconnected* o *Publisher*, o que copian los datos volcados en el tópico correspondiente, cuando el programa se encuentra en modo *Subscriber*. A la derecha se presenta el panel de estado y de conexión. En la parte superior se le indica al usuario en qué estado se encuentra la aplicación, como así también el *OrchID* cuando este nodo se hace visible en ROS. Debajo se ha colocado una entrada de texto para colocar la IP del servidor donde corre *Rosbridge*. Dicha IP será validada por el *Singleton* al requerir una nueva conexión y siempre se utilizará el puerto por defecto, es decir el puerto 9090, para establecer conexión. Finalmente se tienen los botones de conexión y de cambio de modo.

La conexión a ROS, que hace uso de una buena parte de los scripts provistos por ROS#, está a cargo del *GameObject* "*RosConnector*" (Figura 42). En el mismo se gestiona la conexión/desconexión a ROS contemplando la IP, así como la información del estado de cada articulación que se publica/consume sobre un tópico determinado. Dentro de los scripts que componen al mismo están además desarrolladas las funciones que realizan las llamadas a servicios y el tratamiento de sus respuestas, así como el anuncio de otros servicios necesarios para que el orquestador controle al nodo.

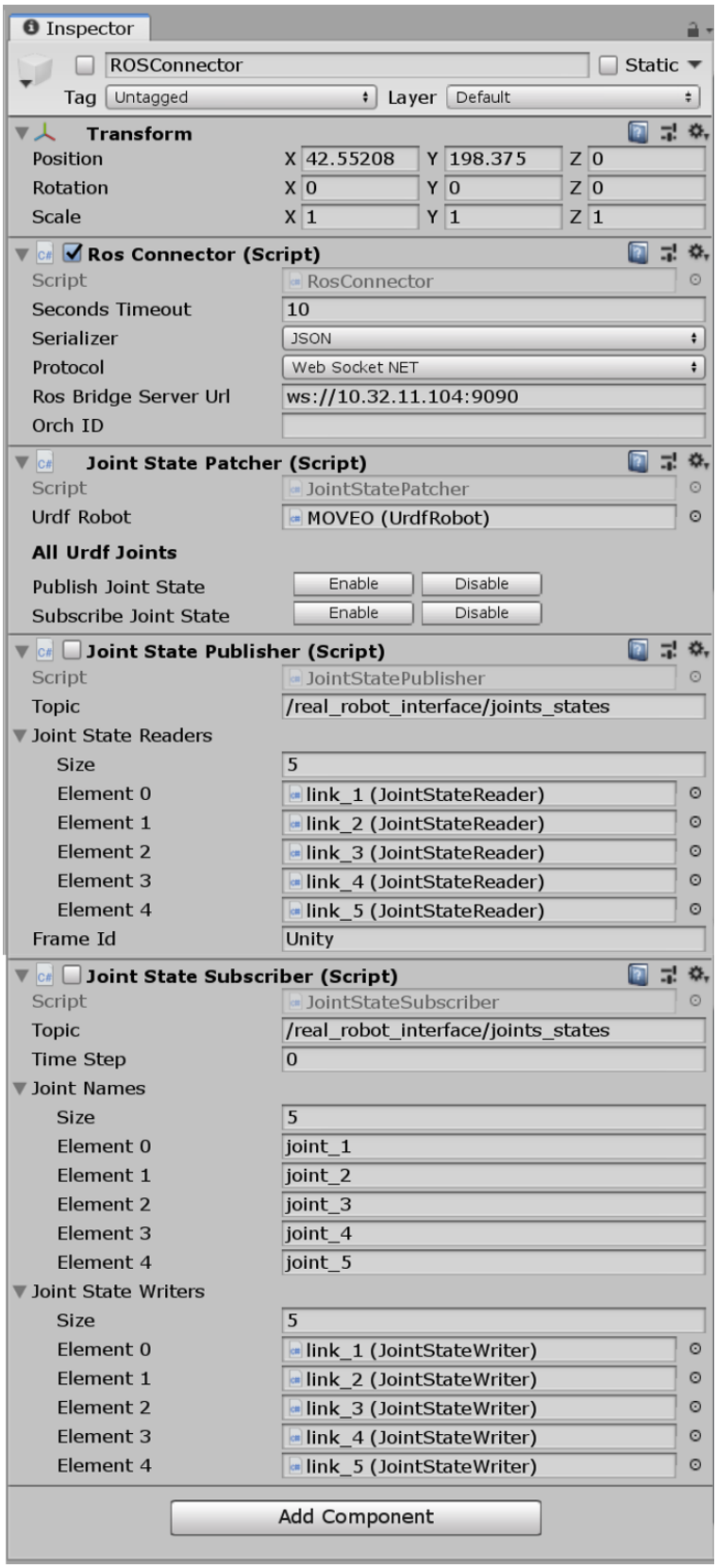

Figura 42. Componentes del GameObject "ROS Connectorr".

v. Happy Path[9]

Apenas se lanza la aplicación, como no ha sido instanciado ningún robot, la misma se encuentra en un estado indeterminado. Los paneles de control de las articulaciones son invisibles, el *GameObject* "*RosConnector*" está inactivo, y el botón de cambio de tipo de conexión ("*Publisher* ↔ *Subscriber*") es también invisible.

Una vez que la cámara detecta un target, la aplicación entra en modo "*Disconnected*", instancia el robot correspondiente, activa el *GameObject* del robot en la escena, y se generan los paneles de control de las articulaciones (Figura 43).

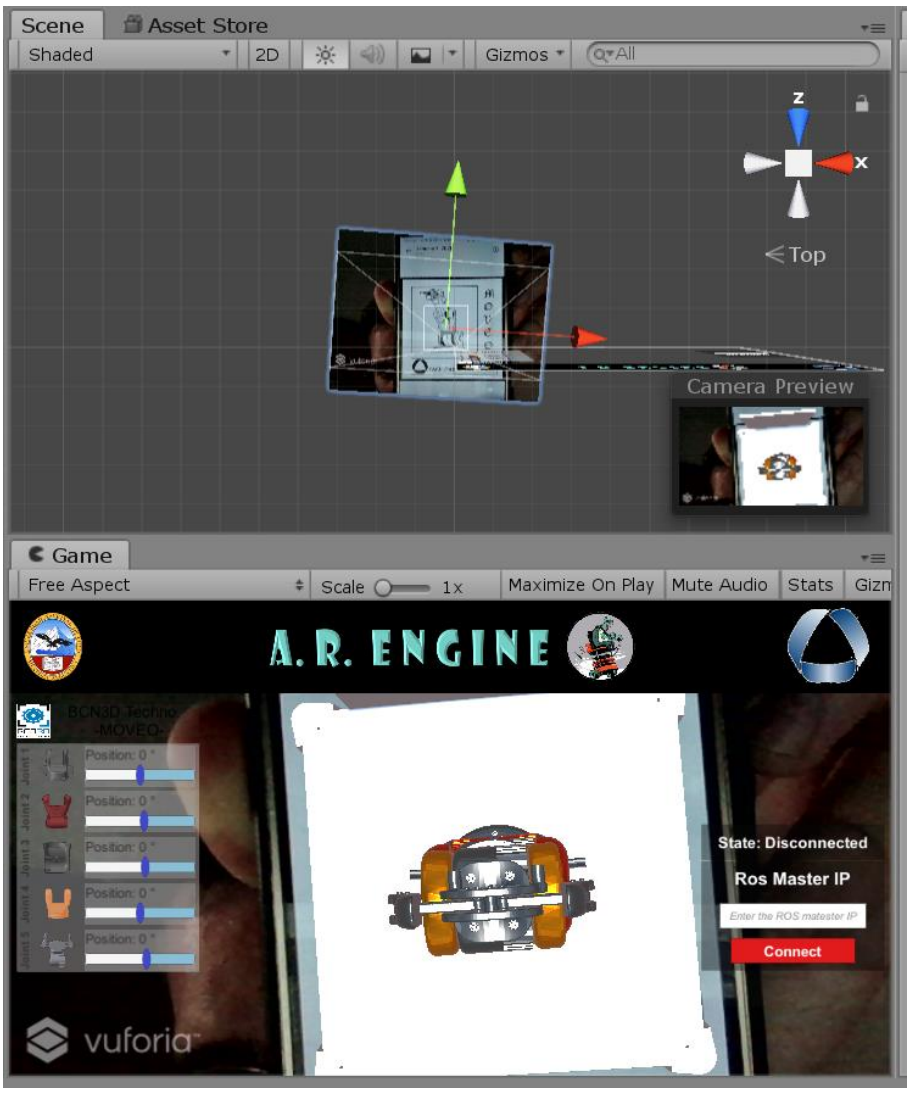

Figura 43. Arriba: Reconocimiento de imágenes de la cámara de realidad aumentada de Vuforia. Aplicación corriendo, en estado desconectado, una vez que se ha identificado el target.

El paso siguiente es escribir la IP de Ros Master URI presente en el panel de conexión y darle click al botón de conexión. En este momento dicho botón pasa a color amarillo y

[9] En el contexto del software o el modelado de información, una ruta feliz (o *Happy Path*) es un escenario predeterminado que no presenta condiciones excepcionales o de error.

sobre el mismo se imprime "*Connecting*", dando indicio del estado al que entra la aplicación (Figura 44). En el mismo se activa *RosConnector*, se le setea la IP ingresada y se lanza un nuevo hilo de conexión hacia el servidor *Rosbridge*. Cuando la conexión se ha establecido se llama al servicio "*/orchestrator/request_connection*" para que el nodo se haga visible para el orquestador. Cuando el Orquestador responde, envía el *OrchID*, con el cual la aplicación anuncia el servicio "*/[OrchID]/orch_com*". Además, se activan los temporizadores que se utilizan en los *watchdogs* del sistema, verificando que el Orquestador esté en servicio y notificando la actividad de este nodo.

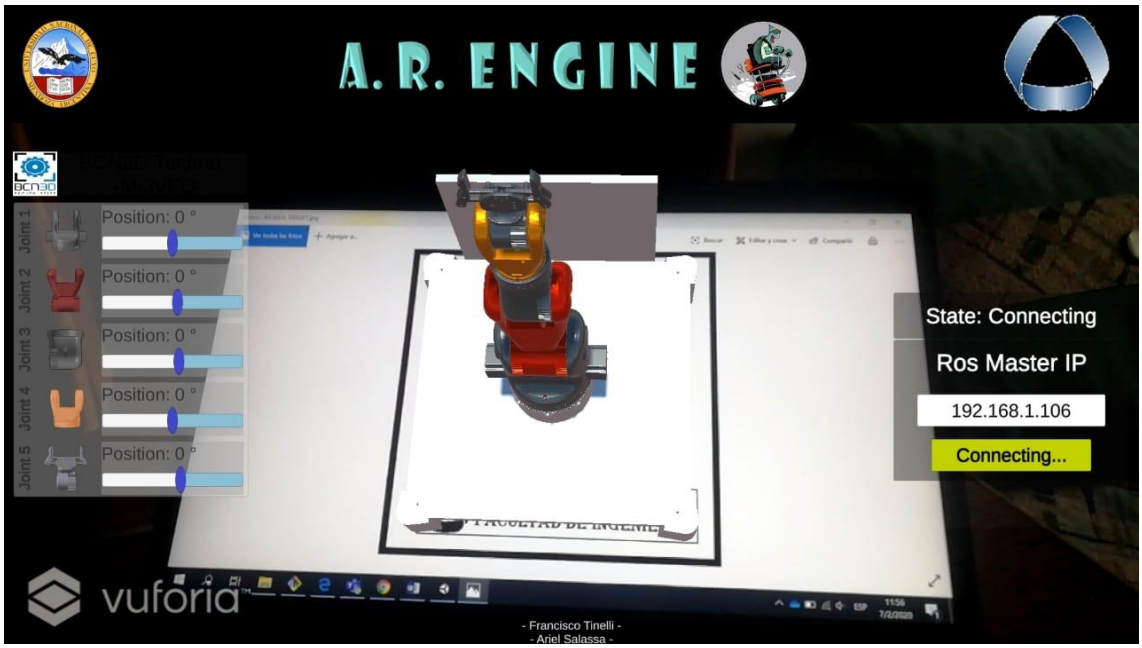

Figura 44. Nodo en el estado de transición "Connecting".

Una vez completado el estado transitorio "*Connecting*" el nodo entra en estado "*ConnectedAsSubscriber*". En este momento se cambia el color del botón de conexión a rojo y se coloca "*Disconnect*". El botón de cambio de tipo de conexión se hace visible y se activa para una eventual petición de publicación (Figura 45). Además, se habilita el componente "*Joint State Subscriber*" del *GameObject RosConnector*, y se deshabilita el componente "*Joint State Publisher*". De esta forma "*Joint State Subscriber*" recoge la información que llega por el tópico y realiza las transformaciones correspondientes a las articulaciones del robot y a los *sliders* de su panel de control.

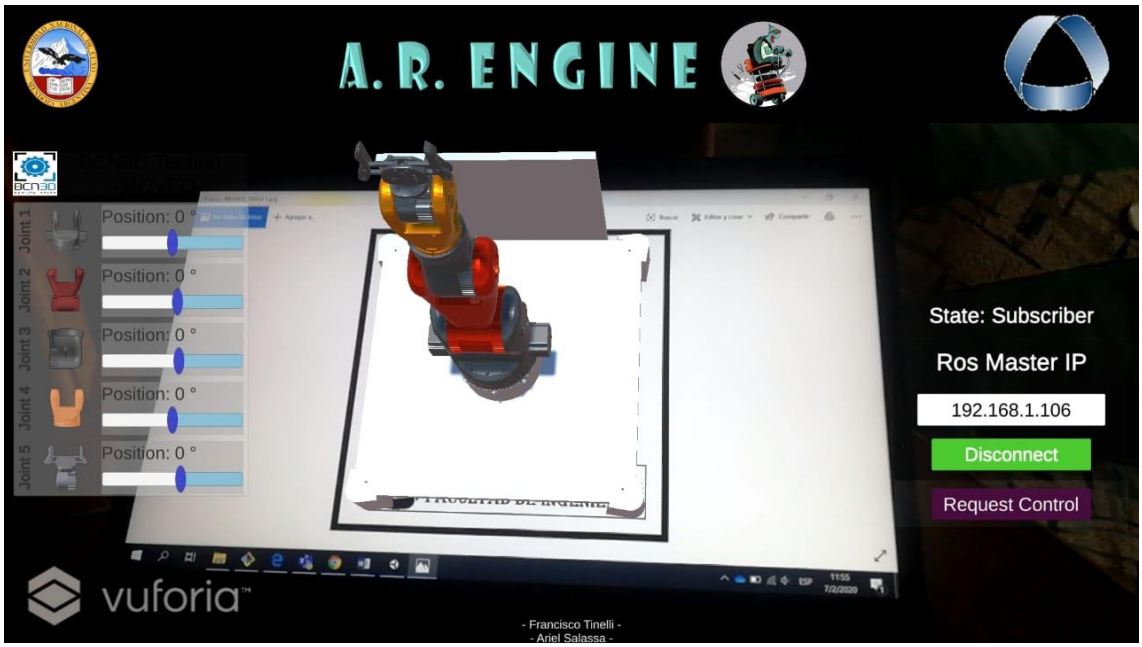

Figura 45. AR Engine en modo Subscriber, una vez que se ha establecido la conexión.

Si se le da click al botón recientemente activado, el mismo cambia de color a amarillo y sobre él se imprime el texto "*Changing P - S*" (Figura 46). La aplicación entra en el estado transitorio "*ChangingStatePS*", donde se llama al servicio del orquestador "*orchestrator/request_control*", enviando como parámetro el *OrchID* de la aplicación. La aplicación persiste en este estado hasta que el Orquestador aprueba la petición y la respuesta de retorno es tratada, pasando al siguiente estado.

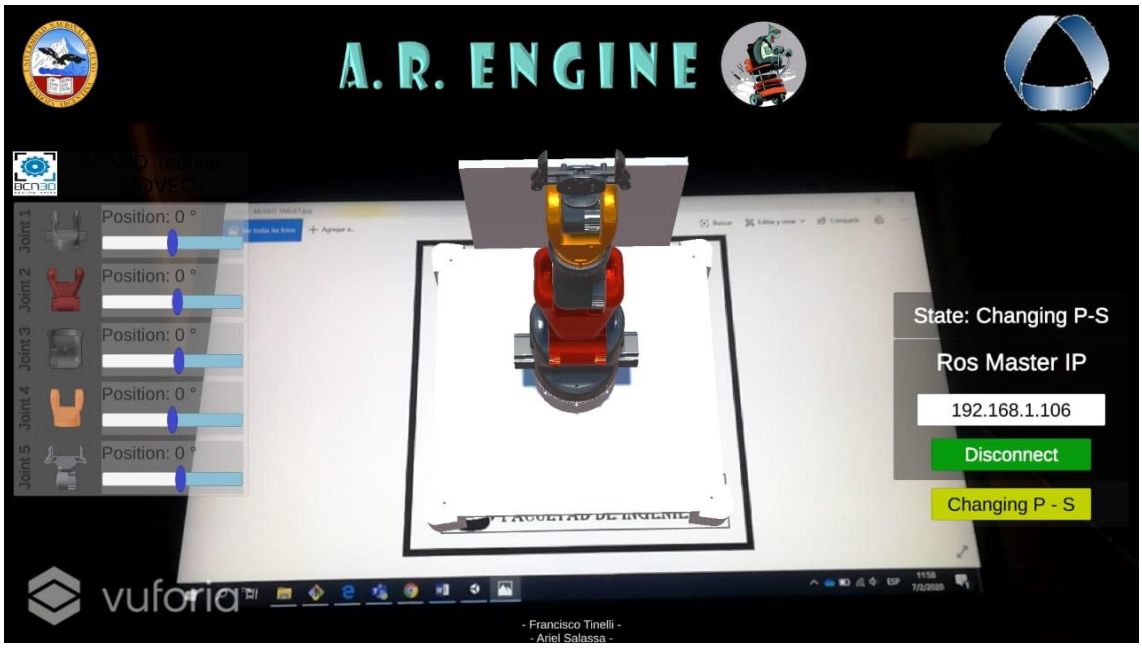

Figura 46. Estado transitorio de la aplicación cuando ha solicitado pasar a modo Publisher.

Una vez en el estado "*ConnectedAsPublisher*" el botón de cambio de tipo de conexión pasa a color naranja y sobre él se imprime el texto "*Realease control*". Además, se habilita el componente "*Joint State Publisher*" del *GameObject RosConnector*, y se deshabilita el componente "*Joint State Subscriber*". De esta forma "*Joint State Publisher*" recoge la información de los *sliders* y la vuelca sobre el tópico.

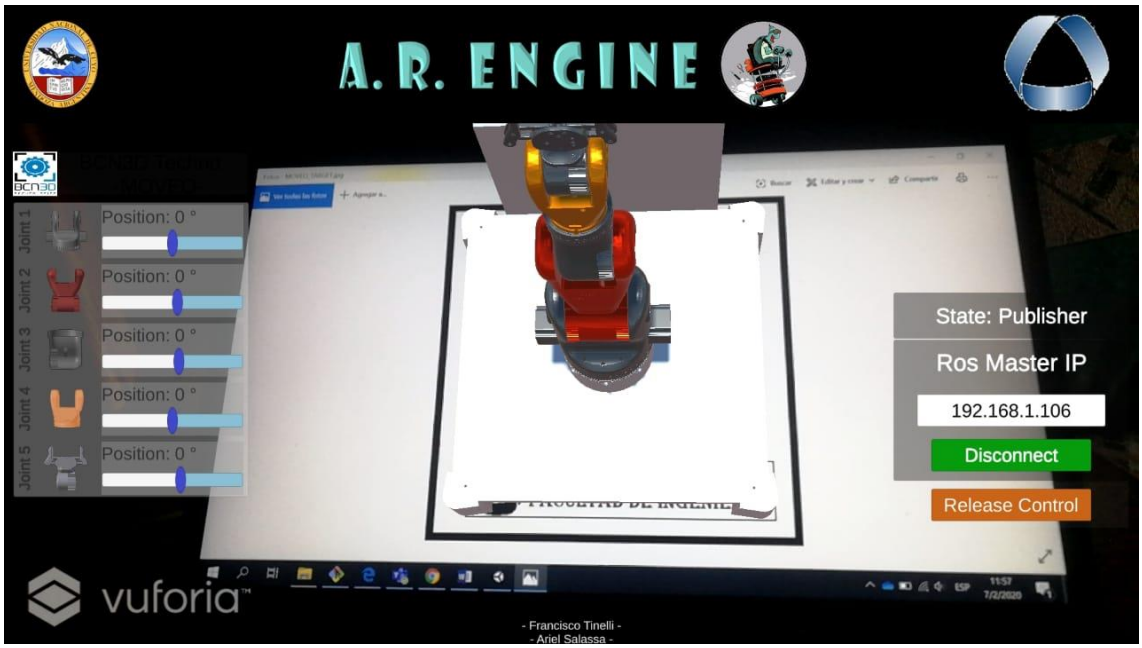

Figura 47. Estado de la aplicación, una vez que el orquestador la ha autorizado a tomar el control sobre el tópico.

Finalmente, si se presiona nuevamente el botón de conexión el nodo pasa nuevamente al estado "*Disconnected*", cerrando todos los hilos de comunicación con ROS y volviendo a hacer invisible el botón de cambio de tipo de conexión.

e. Nodo Orquestador: Gestor de funcionamiento y comunicaciones

El nodo Orquestador ha sido desarrollado con el framework de Vue js [. Para entablar la conexión con Rosbridge, se ha utilizado la librería de JavaScript *roslibjs*. *Roslibjs* es la librería principal de JavaScript para interactuar con ROS desde el navegador. Utiliza WebSockets para conectarse con *Rosbridge* y proporciona publicaciones, suscripciones, llamadas de servicio, *actionlib*, análisis de URDF y otras funciones ROS esenciales [43].

Una aplicación de Vue se divide en componentes, los cuales permiten extender elementos HTML básicos para encapsular código reutilizable. En un nivel alto, los componentes son elementos personalizados a los que el compilador de Vue les añade comportamiento. Los componentes se han organizado de forma tal de que los componentes que tienen como hijos a otros componentes y que renderizan una vista completa en el navegador, se denominan componentes vista (o simplemente vistas).

Ellos se encargan de la parte de presentación y de pasar información hacia los componentes hijos. Para este proyecto se han desarrollado cuatro vistas:

- *Home*: es la vista principal donde se tiene la información de lo que está pasando en el entorno de ROS diseñado para este proyecto (Figura 48)

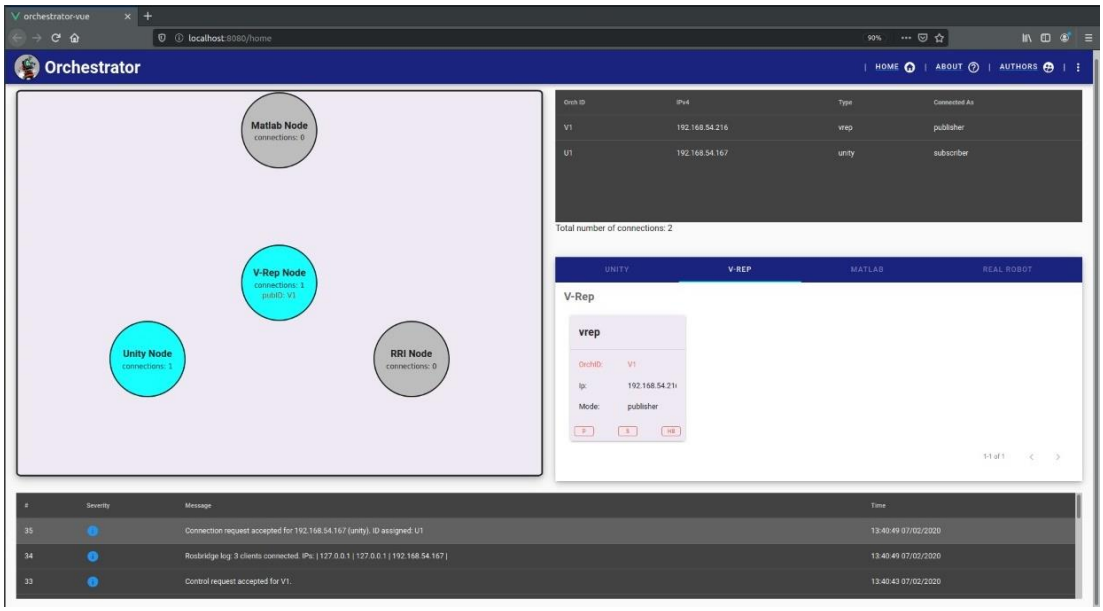

Figura 48. Vista Home.

- *About*: es una vista secundaria que provee links con la información adicional sobre los recursos que se han usado para desarrollar este nodo (Figura 49)

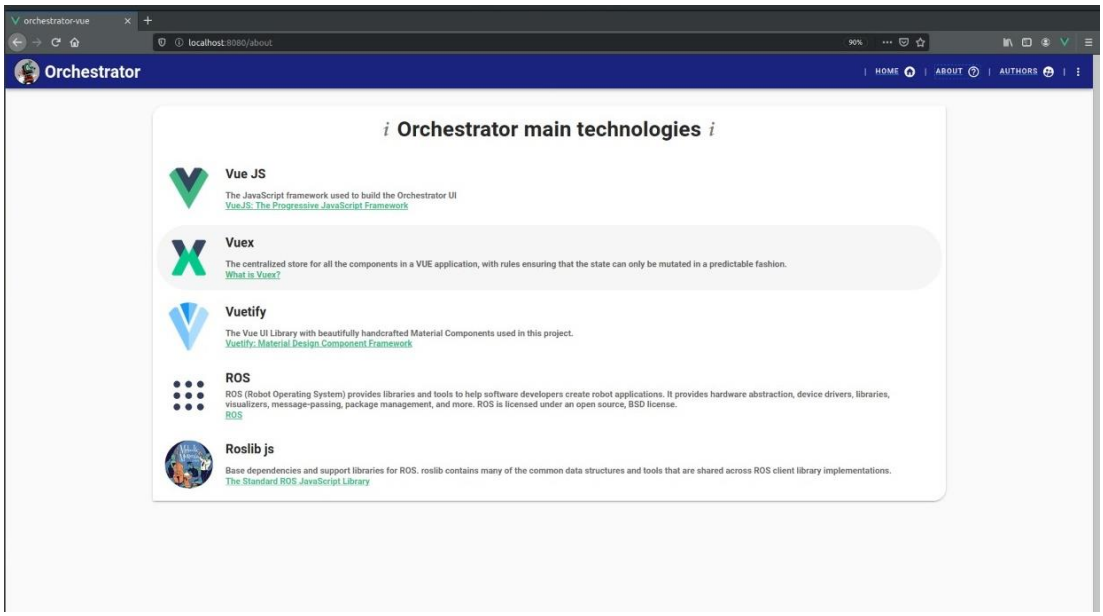

Figura 49. Vista About.

- *Authors*: es una vista secundaria donde se muestra información de contacto sobre los autores y los directores del proyecto (Figura 50).

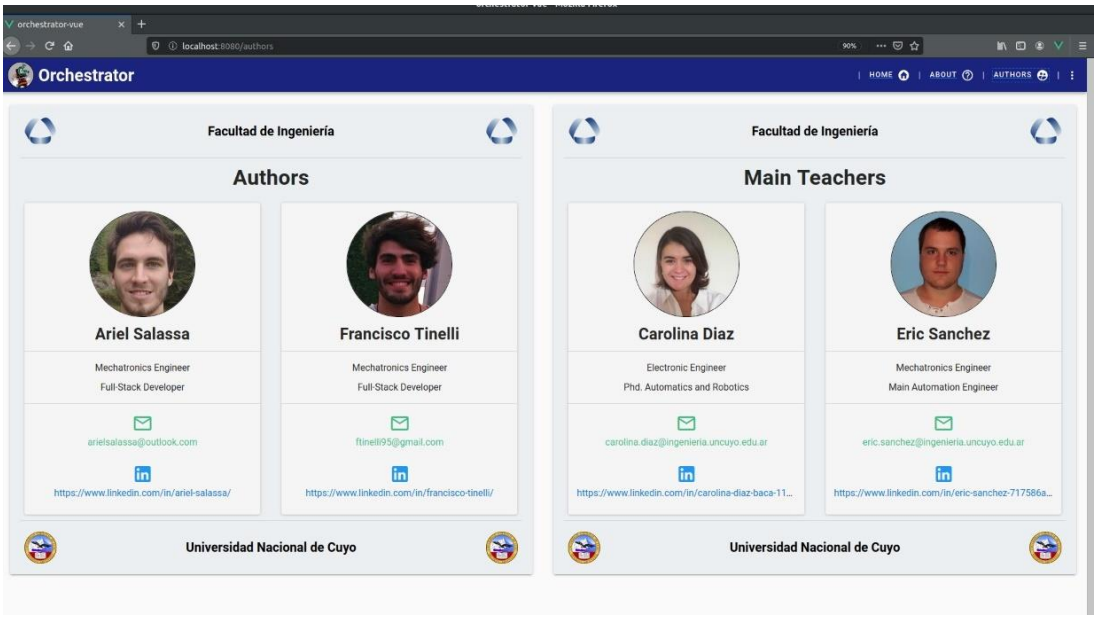

Figura 50. Vista Authors.

- *PageNotFound*: es una vista auxiliar que muestra un mensaje de error en caso de que la URL ingresada por el navegador no coincida con ninguna de las rutas registradas por el enrutador.

Para el diseño de los componentes se ha utilizado Vuetify, el cual proporciona a los desarrolladores todo necesario para crear aplicaciones web ricas y atractivas utilizando las especificaciones de *Material Design* [44].

Para pasar de un componente vista a otro, ya sea mediante la interacción con la interfaz gráfica o bien mediante la escritura de una url en el navegador, se ha utilizado el enrutador oficial de VueJS: Vue-Router [45].

Finalmente, para manejar el estado de la aplicación y ser capaz de comunicar dicho estado a todos los componentes para que éstos puedan reaccionar a él, así como para que los componentes sean capaces de mutar dicho estado, se ha utilizado Vuex. Vuex es un patrón de gestión de estado y una biblioteca para aplicaciones Vue.js. Sirve como un almacén centralizado para todos los componentes de una aplicación, con reglas que aseguran que el estado solo pueda mutarse de manera predecible [46]. Lo escrito anteriormente se resume en la Figura 51.

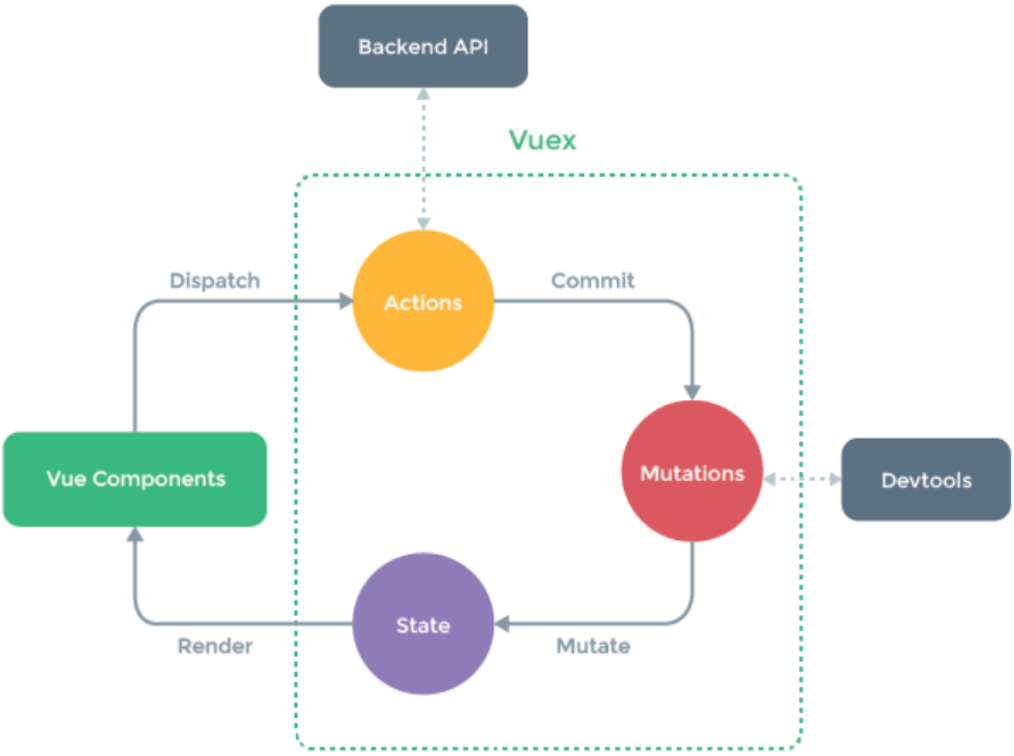

Figura 51. Esquema de funcionamiento general de Vuex.

i. Conectividad con ROS

Para implementar la conectividad con ROS se desarrolló un componente que implementa todas las funciones que pone a disposición *Roslibjs*, en función de las necesidades de este sistema.

Lo primero que realiza este componente al momento de su creación es resetear el estado del "*store*" de Vuex. Acto seguido, se crea el objeto nodo de ROS que va a comunicarse con el servidor de *Rosbridge*. Este objeto tiene "*listeners*" nativos referidos a la conexión, que son: "*on connection*", "*on error*", "*on close*". En el primero, el orquestador asocia al objeto ROS los *handlers* que estarán a cargo de subscribirse a los tópicos de *Rosbridge*, por un lado, y a anunciar los servicios del orquestador y manejar las peticiones entrantes, por el otro. Además, se inician los temporizadores que controlan que la conexión no se haya perdido. Cuando se produce un error en la conexión el *listener* correspondiente emite un error hacia el componente padre que hace que toda la aplicación se bloque. Normalmente este error viene porque no se ha podido establecer conexión con el servidor, ya sea porque la URL no es válida o porque se ha caído la red. Finalmente, cuando se cierra la conexión, se resetean los *timers*.

Los servicios que anuncia el orquestador son cuatro y cada vez que uno de ellos es llamado se produce una modificación en el estado del *store*:

- /orchestrator/request_connection
  Este servicio es utilizado por los nodos para solicitar la conexión al sistema. La solicitud es enviada al orquestador junto con la IP y el tipo de nodo en cuestión. Si la conexión es exitosa, el orquestador agrega un nuevo objeto conexión[10] a la arreglo de conexiones del store, y devuelve un identificador único denominado "OrchID". Este identificador está compuesto por una letra identificadora del tipo de nodo y un número de instancia (ej.: V1). El nodo pasa a ser parte del sistema y puede visualizarse en la interfaz del orquestador. Es importante aclarar que los nodos se conectan en modo "Subscriber" por defecto excepto por el de tipo RealRobot (se conecta en modo "Publisher").
- /orchestrator/release_connection
  Este servicio es llamado por los nodos al momento de desconectarse (voluntaria o involuntariamente) del sistema. En este caso, el nodo envía como dato su OrchID previamente asignado, el orquestador verifica su existencia en el arreglo de conexiones y devuelve una respuesta exitosa eliminando al nodo del sistema.
- /orchestrator/request_control
  Este servicio es invocado cuando un nodo desea tomar el control sobre el tópico donde se publica el estado de las articulaciones. El nodo que desee ser "Publisher" debe llamar al servicio enviando su OrchID previamente asignado. El orquestador verifica que se trate de un nodo conectado (presente en el arreglo de conexiones) y lanza una alerta sobre la interfaz pidiendo confirmación al usuario. En función de la decisión del usuario, se envía al nodo una respuesta (True/False). Si la petición se acepta, entonces al objeto conexión correspondiente se le asigna el modo "Publisher", previamente cambiando a modo "Subscriber" el anterior nodo "Publisher".
- /orchestrator/release_control
  Este servicio es invocado cuando un nodo quiere abandonar el control sobre el tópico donde se publica el estado de las articulaciones. El nodo que desee ser "Subscriber" debe enviar al servicio su OrchID previamente asignado. El orquestador verifica que se trate de una conexión válida y entonces modifica el modo del objeto conexión correspondiente. En ese momento el sistema queda sin ningún nodo volcando información sobre el tópico antes mencionado.

[10] El objeto conexión consta de cuatro propiedades que son:
- OrchID: identificador asignado por el nodo orquestador.
- Type: indica de qué programa surgió la conexión (V-Rep, Unity, Matlab o RealRobot).
- ConnectionMode: indica si el nodo es Publisher o Subscriber.
- IP: IP del nodo conectado.

ii. Comportamiento de la vista principal

La vista principal, "*Home*", está compuesta por varios componentes que responden dinámicamente a los datos que se van actualizando en el estado del *store*. Algunos de ellos también tienen la capacidad de despachar las acciones para mutar también dicho estado.

Primeramente, en la parte superior izquierda se tiene el componente de animación. En la misma se muestra de forma visual cuáles son, según el tipo, los nodos conectados, cuántas conexiones hay por cada tipo de nodo e identifica claramente cuál es el nodo que publica sobre el tópico. De haber al menos un nodo de un determinado tipo conectado, el círculo que representa al nodo en cuestión se torna celeste y debajo se muestra cuántos nodos de ese tipo se han conectado. El nodo que se encuentra en el centro de la animación siempre es el *Publisher*. En color rojizo se muestra además el *OrchID* del Publisher en cuestión. Cuando un nodo pide el control y la petición es aceptada se realiza el movimiento correspondiente para que el nuevo *Publisher* pase al centro y quien estaba en el centro pase a la periferia, como *Subscriber*. De no haber ningún nodo que haya pedido tomar el control, la animación se torna completamente gris tapando el comportamiento del componente.

A su derecha tenemos dos componentes relativos a las conexiones. Cada vez que se modifique la colección de conexiones en el *store*, dichos componentes se actualizarán inmediatamente. La tabla permite visualizar una lista de todas las conexiones presentes en el sistema. Es posible ordenar las entradas de la tabla en forma creciente o decreciente según cualquiera de los campos de la misma. Debajo, las conexiones se presentan más amigablemente en forma de cartas agrupadas en distintas pestañas según sea el tipo de conexión. A través de los botones situados en la parte inferior de las cartas se puede hacer uso de los servicios anunciados por cada uno de los nodos "*/[OrchID]/orch_com*". De esta forma el orquestador puede forzar a cambiar un nodo de modo *Publisher* a modo *Subscriber* y viceversa.

Finalmente, en la parte inferior de la vista se tiene una tabla que va imprimiendo todos los *logs* de eventos que va arrojando el nodo orquestador a medida que se va utilizando.

5. Configuración y Lanzamiento del Sistema (Manual del Usuario)

a. ROS

Para que el sistema funcione correctamente, *ROS Melodic Moreira* debe estar ejecutándose en algún servidor cuyo sistema operativo sea Ubuntu, dentro de una red determinada. A su vez, es condición necesaria que el nodo *Rosbridge* esté instalado y sea lanzado. Para instalar *Rosbridge,* ejecutar en consola el siguiente comando:

```
$ sudo apt-get install ros-melodic-rosbridge-server
```

Para ejecutar nodos pueden utilizarse los comandos *rosrun*[11] o *roslaunch*[12]. La principal diferencia entre ellos es que *rosrun* sólo puede lanzar un nodo a la vez de un *package*, mientras que *roslaunch* puede lanzar múltiples nodos de múltiples *packages* al mismo tiempo. Además, *roslaunch* inicia automáticamente *roscore* (hilo principal de ROS) si este no ha sido lanzado.

Si se desea utilizar *rosrun,* se debe ejecutar:

```
$ rosrun roscore
$ rosrun robridge_server rosbridge_websocket
```

Si se desea utilizar *roslaunch,* se debe ejecutar:

```
$ roslaunch rosbridge_server rosbridge_websocket.launch
```

Para controlar la información pertinente de los *topics* se puede utilizar *rostopic*[13]. A continuación, se muestra una lista de los comandos soportados.

```
$ rostopic bw  <topic-name>
$ rostopic delay <topic-name>
$ rostopic echo <topic-name>
$ rostopic find <msg-type>
$ rostopic hz  <topic-name>
$ rostopic info <topic-name>
$ rostopic list
$ rostopic pub  <topic-name> <topic-type> [data...]
```

[11] Link para más información: http://wiki.ros.org/rosbash#rosrun.
[12] Link para más información: http://wiki.ros.org/roslaunch.
[13] Link para más información: http://wiki.ros.org/rostopic.

```
$ rostopic type <topic-name>
```

Por otro lado, también es de suma utilidad el comando *rosservice*[14]. Este implementa una variedad de opciones de comandos que permiten visualizar qué servicios están actualmente en línea desde qué nodos y profundizar para obtener información específica sobre un servicio, como su tipo y argumentos, entre otros. También puede llamar a un servicio directamente desde la línea de comando. A continuación, se muestra una lista de los comandos soportados:

```
$ rosservice args <service-name>
$ rosservice call <service-name> [service-args]
$ rosservice find <service-type>
$ rosservice info <service-name>
$ rosservice list
$ rosservice node <service-name>
$ rosservice type <service-name>
$ rosservice uri  <service-name>
```

b. Orquestador

El nodo orquestador ha sido desarrollado como una aplicación web. Éste nodo puede ejecutarse en cualquier sistema operativo de *laptops* (Windows, Linux, MacOS). Simplemente es necesario que en el ordenador que lo ejecute esté instalado NodeJS[15], y que dicho ordenador esté conectado a la misma red que *Rosbridge*.

i. Instalación de dependencias

NodeJS maneja de manera automática la instalación de dependencias de los proyectos. Para instalar las dependencias de este proyecto es necesario abrir una consola de comandos, dirigirse al directorio del mismo y ejecutar:

```
$ npm i
```

ii. Lanzamiento

Para ejecutar la aplicación web en modo de desarrollo, se debe ejecutar el siguiente comando:

```
$ npm run serve
```

Esto compilará el proyecto y lo pondrá a disposición en una URL local determinada que se indicará en la consola. (Figura 52)

[14] Link para más información: http://wiki.ros.org/rosservice.
[15] Link oficial de descarga: https://nodejs.org.

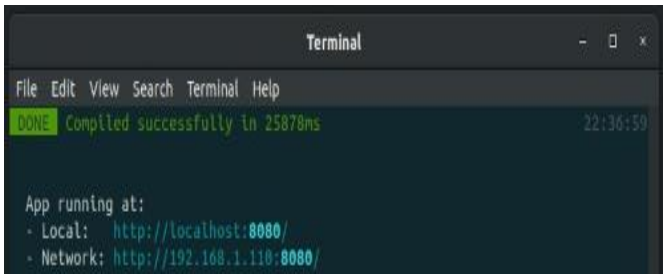

Figura 52. Link de acceso a la aplicación web compilada, una vez que se ha ejecutado el comando "npm run serve"

Si por el contrario se quiere obtener una compilación de producción (minimizada) para evaluar su funcionamiento y eventualmente "*hostear*" en un servidor, se debe ejecutar el siguiente comando:

```
$ npm run build
```

iii. Configuración

El orquestador debe comunicarse con *Rosbridge* y para ello es necesario que se conozca su IP. La aplicación por defecto carga la IP local, considerando que tanto el orquestador como *Rosbridge* están corriendo en el mismo ordenador, pero eso no tiene que ser necesariamente de esta manera. Para modificar la IP para entablar la comunicación con *Rosbridge* se debe presionar sobre el símbolo de ajustes en la esquina superior derecha de la cabecera y escribir la IP que corresponda. (Figura 53)

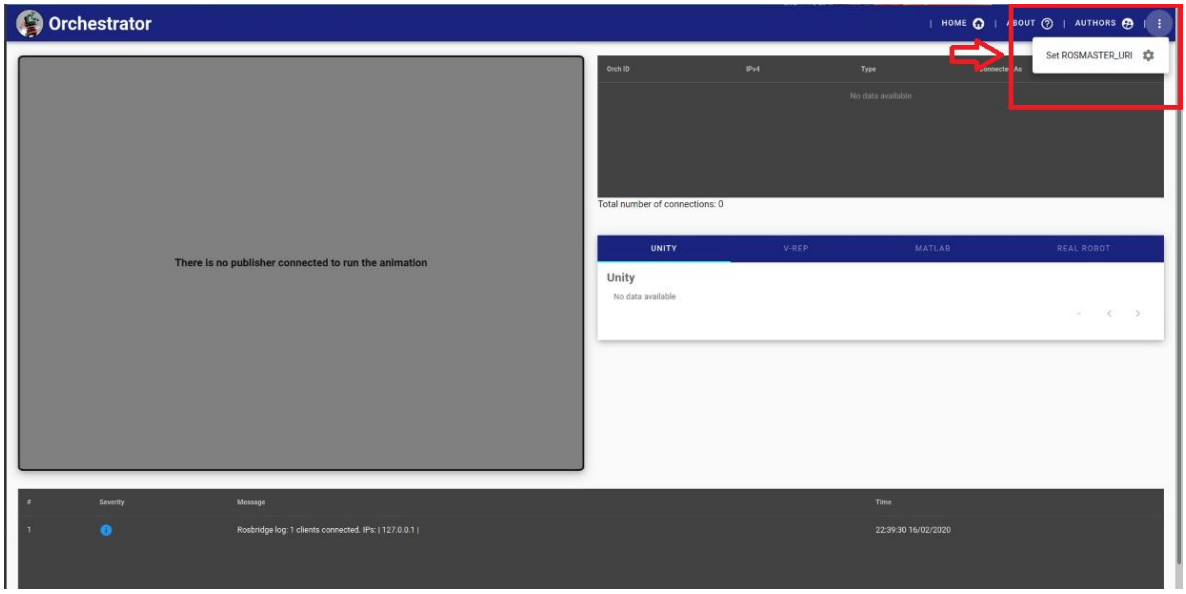

Figura 53. Ventana "Home" del orquestador, configuración de ROS Master URI

c. Nodo RR

Para lanzar el nodo RR LinuxCNC, no es necesario instalar ningún complemento en la *Beaglebone* o cualquiera sea el microcontrolador que maneja la máquina herramienta o robot. Sin embargo, es necesario que la misma esté conectada a la misma red que el sistema.

Entonces, desde un ordenador[16] diferente dentro de la misma red lanzaremos el driver RR. Es necesario que este ordenador cumpla con algunos requisitos y disponga de determinados programas para poder funcionar correctamente, como se detallará a continuación.

i. Dependencias

- Linux: El driver no es compatible con Windows o Mac.
- ROS: El ordenador debe tener instalada al menos la versión de base de *ROS Melodic*.
- Python 2.7.15: El nodo no es compatible con las versiones 3.x de Python.
- Paramiko: SSH para Python. Se puede instalar con el gestor de paquetes de Python ejecutando el siguiente comando en la consola:

```
$ pip install paramiko
```

- Cob_srvs: Mensajes ROS no nativos. Instalador *.deb* en carpeta "Otros".

Para utilizar el simulador de Machinekit, es necesario, además:

- VMWare Workstation Player: Virtualizador de sistemas operativos[17].
- Máquina virtual de Machinekit.

ii. Configuración

Una vez instaladas las dependencias en el ordenador que lanzará el nodo y conectados ambos ordenadores a la misma red, se puede proceder a configurar el nodo.

```
11  ## LinuxCNC SSH Credentials##
12  #  Simulator
13  sim_ssh = {
14      'ip' : '192.168.54.239',
15      'port' : 22,
16      'username' : 'machinekit',
17      'password' : 'machinekit'
18  }
19  #  Robot
20  robot_ssh = {
21      'ip' : '10.32.2.23',
22      'port' : 22,
23      'username' : 'machinekit',
24      'password' : 'machinekit'
25  }
26  used_ssh = sim_ssh
27  ##############################
```

Figura 54. Configuración de las variables más importantes para establecer una conexión ssh entre el nodo RR y el microcontrolador del robot real (o su máquina virtual)

[16] Este ordenador puede ser el mismo que lance el orquestador o el nodo Pro.Sim., por ejemplo.
[17] Link para más información: https://www.vmware.com/ar/products/workstation-player.html

Para ello se debe abrir el archivo “config.py” presente en la carpeta del *driver* para configurar las credenciales de acceso al microcontrolador del robot. (Figura 54)

En la sección correspondiente, se puede modificar la variable “sim_ssh” o “robot_ssh” según si estaremos utilizando el simulador de Machinekit o un robot real que utiliza Machinekit respectivamente.

Es importante al final de la sección asignar la variable “used_ssh” según el caso de uso actual.

Los campos a modificar son los siguientes:

- IP: dirección IP de la placa *Beaglebone* o de la máquina virtual. Si ROS corre en sistema Linux, se puede ejecutar el comando:

  ```
  $ ifconfig
  ```

- *Port*: puerto de conexión SSH. *Default*: 22
- *Username* y *Password*: credenciales de acceso. *Default*: machinekit/machinekit.

Si se está utilizando el simulador con VMWare, es necesario configurarlo para que se conecte directamente la red utilizando una dirección IP diferente a la de la máquina host. Esto se configura en la barra superior de la máquina virtual una vez ejecutada. (Figura 55).

Figura 55. Configuración de IP con VMWare.

Click derecho -> Settings -> Network Connection -> Bridged.

iii. Lanzamiento

1. Encender el robot / simulador. Lanzar Machinekit (Menú -> CNC -> Machinekit) y cargar el archivo correspondiente al robot actual. Una vez iniciado, desactivar el botón de parada de emergencia y accionar el botón de encendido (Figura 56). Verificar la conexión a la red y la dirección IP ejecutando el comando anteriormente mencionado en una terminal (Menú -> Herramientas del Sistema -> XTerm).

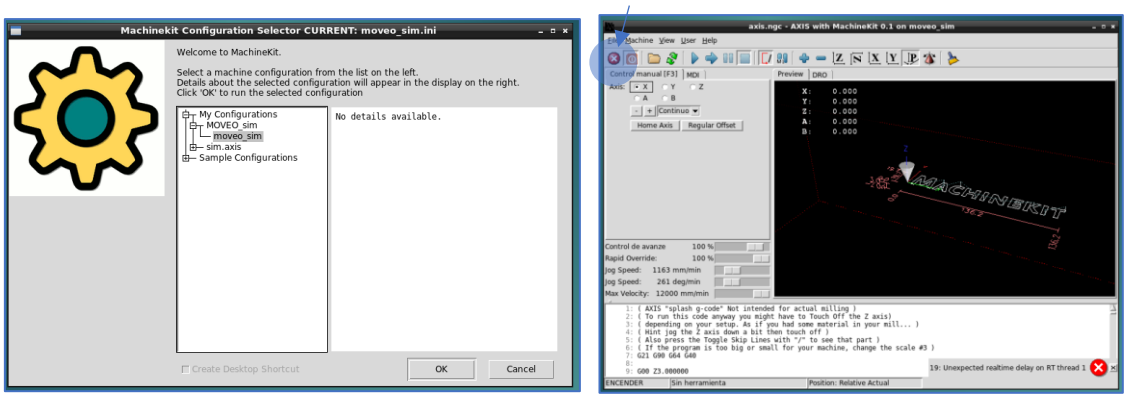

Figura 56. Lanzamiento de Machinekit

2. Asegurarse de que el orquestador del sistema esté corriendo.
3. Desde el segundo ordenador, una vez realizada la configuración en el archivo config.py, abrir una consola en el directorio del driver (Real Robot Interface / LinuxCNC Driver / Driver) y ejecutar el driver con el siguiente comando

```
$ python2 rri_driver.py
```

Si el lanzamiento se ejecuta correctamente deberíamos obtener los mensajes en la consola informando una conexión exitosa, tal como se observa en la Figura 57:

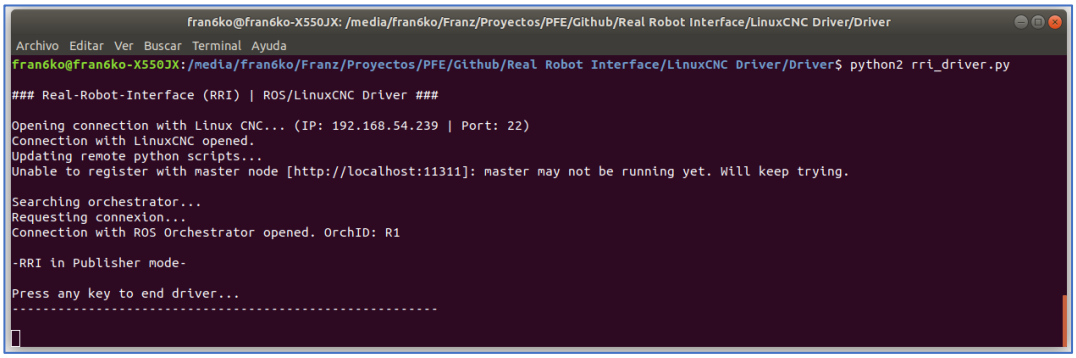

Figura 57. Ejecución del comando "python2 rri_driver.py" por consola y su resultado.

4. Si eventualmente se quiere ceder el control del sistema, se abre otra consola y se llama al servicio ROS correspondiente con el siguiente comando:

```
$ rosservice call /R1/release_control.
```

De igual forma se puede hacer una petición para tomar el control con el comando:

```
$ rosservice call /R1/request_control
```

d. Nodo Control GUI

i. Dependencias

- Windows: El nodo es únicamente compatible con este sistema operativo.
- Matlab: La interfaz fue desarrollada utilizando este software.
- Python 2.7.15: El driver no es compatible con las versiones de Python 3.x.

- Roslibpy: Librería de ROS bridge para Python.

```
$ py -2 –m pip install roslibpy
```

- Pyserial: Librería de comunicación serie para Python.

```
$ py -2 –m pip install pyserial
```

- VSPE: Emulador de puertos series[18].
- Paramiko: SSH para Python (para el modo *Standalone*).

```
$ pip install paramiko
```

ii. Configuración

Normalmente no es necesario realizar ninguna configuración antes del lanzamiento del nodo salvo si se utiliza al nodo en modo independiente.

Existen configuraciones por defecto en el archivo "config.py" que se encuentra dentro de la carpeta de archivos del nodo. Los parámetros principales son 2:

- *Ros_mode*: variable booleana para configurar el modo de funcionamiento del driver.
  - *True* -> funcionamiento normal como nodo ROS del sistema global (*default*).
  - *False* -> funcionamiento independiente o *Standalone* en el cual el driver se conecta directamente al robot real vía SSH.
- *Rosbridge_server_URI*: dirección IP default del ordenador que está corriendo el servidor de *Rosbridge*. Normalmente, debería ser el mismo ordenador que lanzó el orquestador.

En el modo de funcionamiento *Standalone*[19], el parámetro *"rosbridge_server_URI"* es ignorado y es necesario configurar la sección "LinuxCNC SSH Credentials" como si se tratara del nodo RR.

iii. Lanzamiento

1. Iniciar Matlab, posicionar como directorio de trabajo a la carpeta "r1_programación" (Control GUI / Interface Matlab / r1_programación) y ejecutar el *script* "start_PROGRAMACION.m".
2. Pulsando el botón "CARGAR", elegir el archivo de parámetros correspondiente al robot actual. (Ejemplo: ROBOTS / MOVEO / parametros.m). Acto seguido, pulsar el botón "PLOT" el cual importará los archivos STL del robot para su simulación. Se debería tener en pantalla algo similar a lo que se muestra en la Figura 58.

[18] Link para descarga y mayor información: http://www.eterlogic.com/Products.VSPE.html

[19] En este modo no es necesario lanzar el nodo RR. El actual driver actúa como una fusión de los nodos RR y Control GUI independiente.

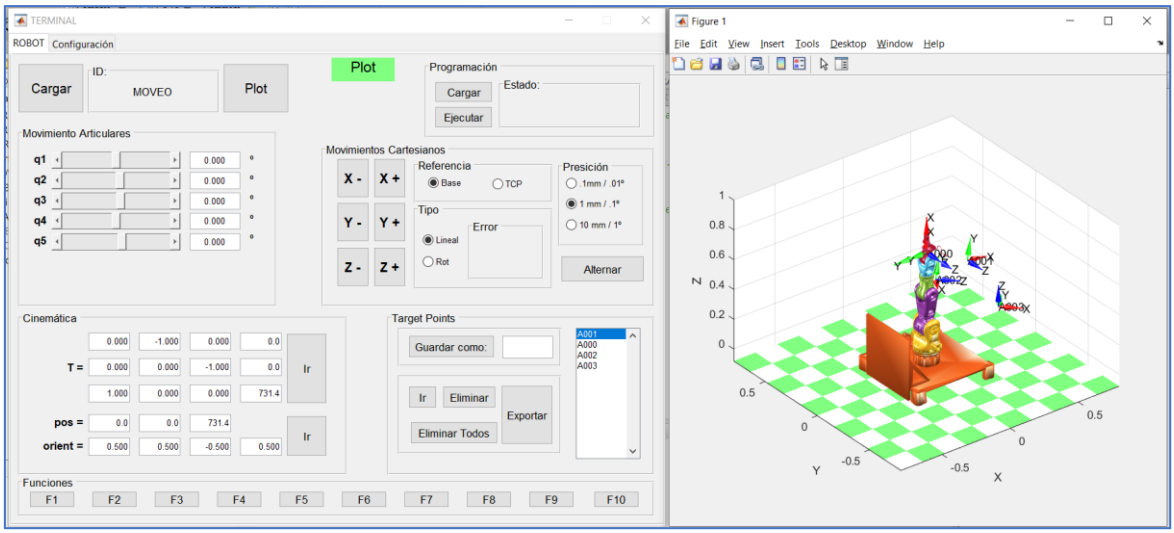

Figura 58. Interfaz gráfica de Matlab con el Robot Moveo cargado y ploteando su comportamiento.

3. En la ventana de la interfaz de control, cambiar a la pestaña “Configuración”. En el cuadro “Manejo Q”, seleccionar la opción “Puerto serie”. (Figura 59 - A).
4. Pulsar el botón “*RUN VSPE*” el cual iniciará el software de emulación de puertos series VSPE emulando un puerto serie doble en “COM1”. (Figura 59 - B).
5. Pulsar el botón “Escanear” (Figura 59 - C). Debería seleccionarse automáticamente el puerto COM1 ahora disponible. Pulsar el botón “CONECTAR”. (Figura 59 - D).
6. En la sección Rosbridge server IP, es posible ingresar una dirección IP la cual sobreescribirá a la definida en el archivo “config.py”. (Figura 59 - E).

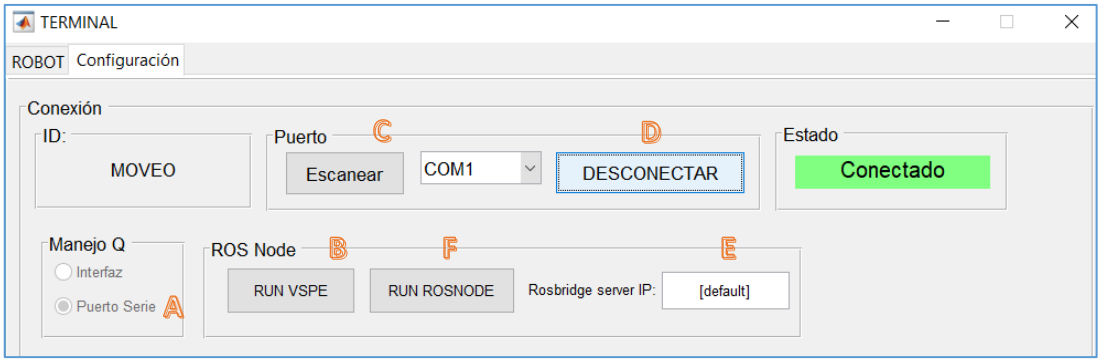

Figura 59. Terminal de control, panel de configuración detallado.

7. Pulsar el botón “*RUN ROSNODE*” (Figura 59 - F) para lanzar el nodo ROS. Debería abrirse una mini-interfaz que gestionará la comunicación con el orquestador del sistema, y una consola para reportes. (Figura 60)

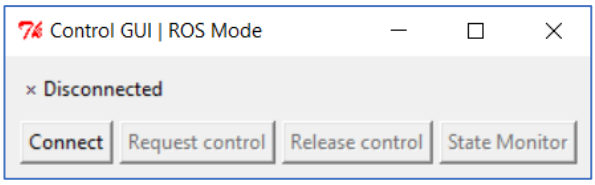

Figura 60. Interfaz resultante que gestiona la comunicación entre el orquestador y el Matlab.

8. Pulsando el botón "*Connect*", el nodo se conectará al sistema en modo escucha (*Subscriber*) y el orquestador le asignará un ID único. En este modo, la simulación imitará al nodo que tenga el control y la interfaz no tendrá efecto alguno.
9. El botón "*Request Control*" enviará una petición al orquestador para tomar el control sobre el sistema. SI la misma es aceptada, la interfaz de control pasa a comandar todos los nodos del sistema.
10. Finalmente, el botón "*State Monitor*" permite la visualización activa de las variables articulares publicadas sobre el sistema sin importar el modo del nodo. (Figura 61).

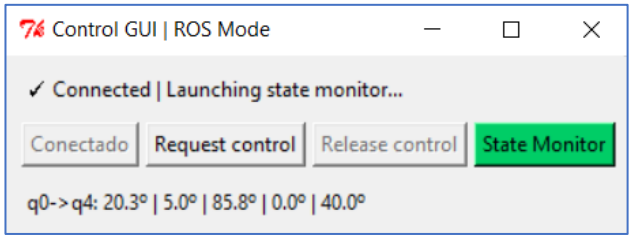

Figura 61. Supervisión de las variables articulares que se vuelcan sobre el tópico.

iv. Lanzamiento en modo Standalone

En el modo *Standalone* o independiente, el driver prescinde de ROS y del sistema y se conecta directamente con el robot real tomando el control del mismo. En este modo, el driver actúa como una fusión de los nodos RRI y Control GUI.

Una vez configurado el modo y las credenciales SSH en el archivo "config.py", se deben seguir prácticamente los mismos pasos del lanzamiento normal hasta el punto 5, con la excepción de que en el punto 3, en la sección "Manejo Q" esta vez se debe seleccionar "Interfaz".

Luego, en lugar de pulsar el botón "*RUN ROSNODE*", es necesario lanzar el driver por consola mediante el intérprete de Python (2.7.15).

La mini-interfaz generada será ligeramente diferente a la anterior (Figura 62). El botón "*Connect*" permitirá al driver conectarse esta vez con el robot real vía SSH usando la IP y credenciales definidas.

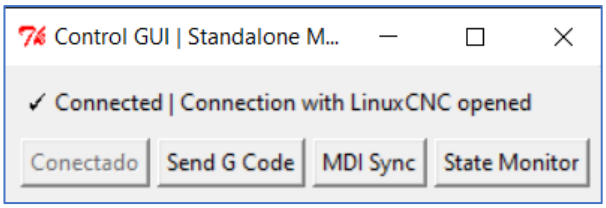

Figura 62. Interfaz en modo standalone.

El botón "*MDI Sync*" permitirá la sincronización unidireccional activa de variables articulares en el sentido MATLAB -> ROBOT. De esta manera, cualquier cambio en la interfaz se verá reflejado en el robot real respetando la dinámica del mismo (Feedrate de comandos MDI configurable en el archivo config.py).

El botón "*State Monitor*" permite la visualización sobre la mini-interfaz de las variables articulares y del estado del intérprete de comandos de LinuxCNC.

Por último, el botón "*Send G Code*" permite el envío y ejecución de un archivo de código G (cuyo nombre actualmente está "hardcodeado") al robot.

e. Nodo Pro.Sim.

i. Dependencias

- Linux: Si bien *CoppeliaSim* puede correr en Windows, MacOS o Linux, ROS funciona establemente en Linux por lo que, para ejecutar correctamente la API de ROS que pone a disposición *CoppeliaSim,* es necesario que el programa corra en Linux.
- ROS: El ordenador debe tener instalada al menos la versión de base de *ROS Melodic*.
- CoppeliaSim (V-Rep): Simulador para robótica desarrollado por *Coppelia Robotics*[20].
- LibsimExtROSInterface.so: *Plugin* de ROS para *CoppeliaSim*[21].
- Cob_srvs: Mensajes ROS no nativos. Instalador .deb en carpeta "Otros".
- Luarocks: Es el gestor de paquetes de Lua [47]. Para instalarlo, se debe ejecutar el siguiente comando:

```
$ sudo apt-get install -y luarocks
```

- F-strings: Librería de interpolación de variables de tipo *string* para Lua [48]. Para instalarlo se debe ejecutar el siguiente comando:

```
$ luarocks install f-strings
```

[20] Link de descarga: https://www.coppeliarobotics.com/downloads
[21] Utilizar el archivo que se encuentra en la carpeta "Otros" ya que fue necesario recompilarlo para incluir los servicios "cob_srvs".

ii. Configuración

Luego de instalar *CoppeliaSim*, es necesario copiar el archivo "libsimExtROSInterface.so" (correspondiente al *plugin* ROS) en el directorio raíz del programa donde se encuentran todos los *plugins* activos.

Si el proceso principal de ROS, *roscore*, no está siendo ejecutado en el mismo ordenador que el simulador, es necesario precisar la dirección IP del ordenador que lo ejecuta. Esto se realiza exportando algunas variables de entorno:

- *ROSMASTER_URI*: dirección de red del ordenador que ejecuta *roscore*.

```
$ export ROS_MASTER_URI=http://192.168.1.10:11311
```

- *ROS_IP*: dirección IP del ordenador local que ejecuta *CoppeliaSim*.

```
$ export ROS_IP=http:// 192.168.1.17:11311
```

Para más detalles sobre la configuración de red de ROS, consultar la página oficial[22].

iii. Lanzamiento

Teniendo en cuenta que *roscore* ya está lanzado en el ordenador o remotamente (configurando adecuadamente las variables de entorno como se explicó en "Configuración"), se puede proceder con los pasos para el lanzamiento del simulador.

1. Lanzar el simulador ejecutando por consola el archivo "CoppeliaSim.sh" presente en el directorio raíz del programa. Verificar que el *plugin* ROS se inició correctamente, como se ilustra en la Figura 63:

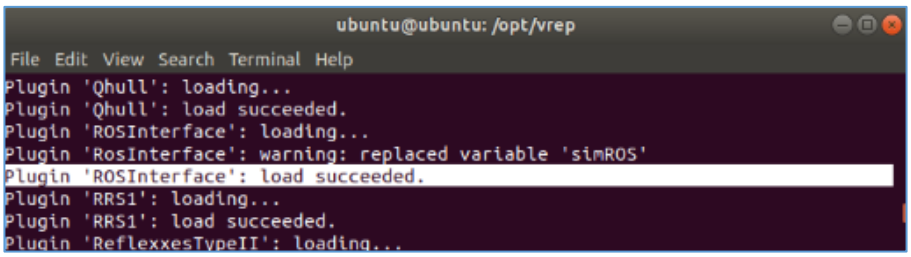

Figura 63. Lanzamiento de V-Rep o CoppeliaSim por consola. Verificación de la correcta ejecución del plug-in de ROS

2. Cargar el robot que se simulará en la escena. Para ello, en la barra de tareas seleccionar *"File -> Open scene…"* y abrir el archivo con formato *".ttt"* correspondiente.
3. Iniciar la simulación con el botón "*Play*" en la barra superior del programa. Aparecerá una interfaz donde es posible controlar las articulaciones del robot localmente. En la parte superior de la misma se puede ver el estado del robot (actualmente desconectado del sistema global) y en la parte inferior, dos botones los cuales permiten la conexión y desconexión al sistema y los eventuales cambios de modo (*Publisher* y *Subscriber*). (Figura 64).

[22] Link de configuración de red de ROS: http://wiki.ros.org/ROS/NetworkSetup

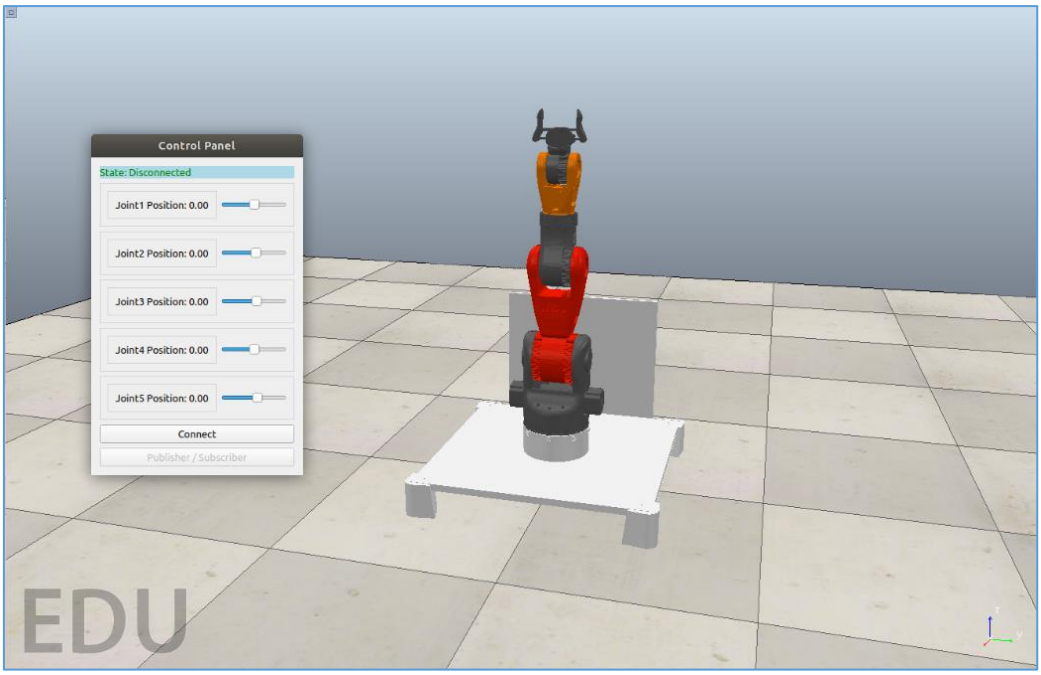

Figura 64. Estado inicial de la simulación en V-Rep/CoppeliaSim luego de correr la escena.

4. Considerando que el orquestador está en ejecución, pulsando el botón "*Connect*" el nodo envía una petición de conexión al mismo. Si la petición es aprobada, un ID único es asignado por el orquestador y el nodo se conecta en modo *Subscriber* al sistema reproduciendo los movimientos del nodo que tenga el control. (Figura 65).

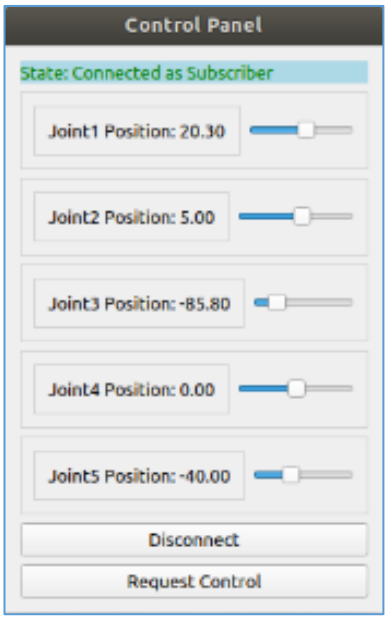

Figura 65. Panel de Control. Nodo en modo Subscriber.

5. Si se desea tomar el control del sistema, pulsando el botón *"Request Control"* el nodo envía un pedido al orquestador. Si el mismo es aceptado, el nodo pasa a

modo *Publisher* y todos los movimientos del robot son imitados por cada uno de los nodos del sistema.

6. De igual manera, es posible dejar el control pulsando el botón "*Release Control*" pasando nuevamente a modo *Subscriber*.

f. Nodo A.R. Engine

i. Dependencias

- Windows: Unity ha sido diseñado para correr en Windows. La versión que corre en sistemas Linux es experimental.
- *Android Software Development Kit*: Es necesario para compilar la aplicación y generar un archivo de instalación que luego pueda ser portable a dispositivos Android.
- *Android Native Development Kit:* Paquete de Android para realizar un proceso de compilación optimizado.

ii. Configuración para utilizar Vuforia

Las versiones más recientes de Unity vienen por defecto con el kit de desarrollo de Vuforia preparado para integrar a cualquier proyecto. Para incorporar este paquete se debe ir a la barra de tareas *Edit -> Project Settings* y dentro de la ventana emergente, se debe seleccionar *Player* sobre la lista de la izquierda, tal como se muestra en la Figura 66.

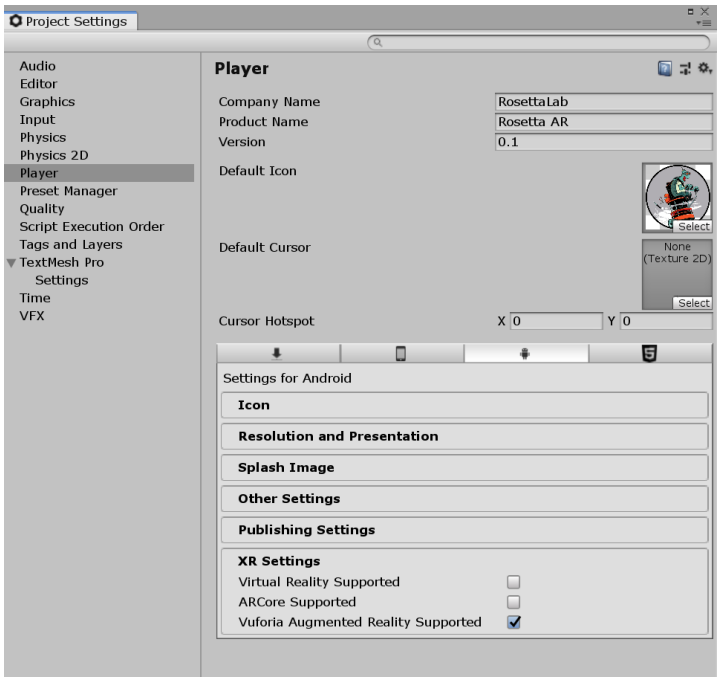

Figura 66. Player Settings.

En esta ventana, dentro de la pestaña de "*XR Settings"* se debe activar la opción *Vuforia Augemented Reality Supported en* las plataformas en las cuales será compatible la aplicación (en este caso será Android).

Esto provoca que se genere un *GameObject* del tipo *AR Camera* de Vuforia y que se importen al proyecto varios objetos y *scripts* propios de Vuforia, dentro de los *Assets* del proyecto. Para configurar Vuforia (esencialmente el comportamiento de la cámara) es necesario abrir en el Inspector de Unity el componente *Vuforia Configuration.* Aquí se debe introducir la licencia de desarrollo de Vuforia. Todas las aplicaciones de Vuforia utilizan una clave única que se obtiene a través de *Vuforia License Manager.* Para obtenerla, se debe crear una cuenta de desarrollador en el portal de desarrollo de Vuforia[23]. Una vez registrado en el sistema, se debe ir a la pestaña *"Develop"*, y, dentro de esta vista, seleccionar *"License Manager*" y luego *"Get Development Key*". Una vez creada la licencia se deben copiar los caracteres debajo de "*License Key*" (Figura 67) e introducirlos dentro del proyecto de Unity. (Figura 68).

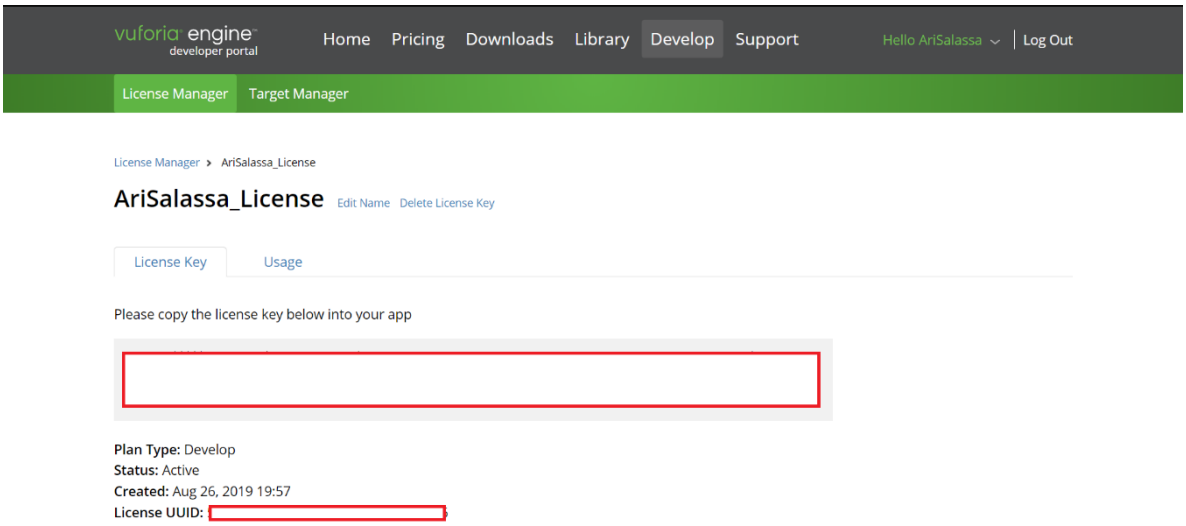

Figura 67. Vuforia Licence Manager, gestión de licencias para desarrollar en Unity.

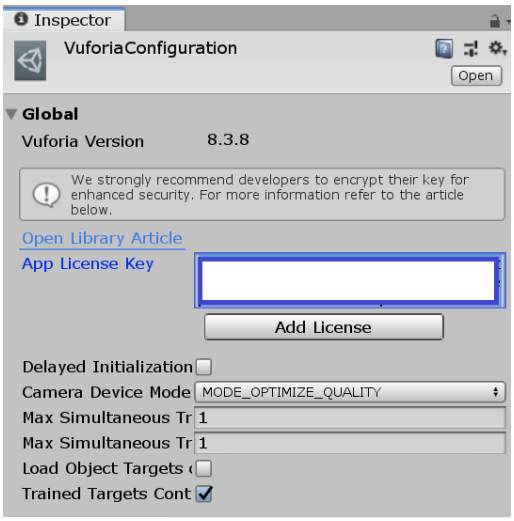

Figura 68. Lugar donde debe colocarse la clave obtenida desde el portal de Vuforia.

[23] Link del portal: https://developer.vuforia.com/

El paso siguiente es crear los *targets* que identificará la cámara. Dentro de la plataforma de desarrollo de Vuforia, también se tiene un "*Target Manager*". Aquí se genera una base de datos con los *targets* necesarios para el desarrollo de una aplicación. Para generarla, dar click a "*Add Database*" y seleccionar el lugar donde correrá la base de datos. En este caso, se ha seleccionado "*Device*", dando lugar a que los targets se almacenen en el propio dispositivo. (Figura 69).

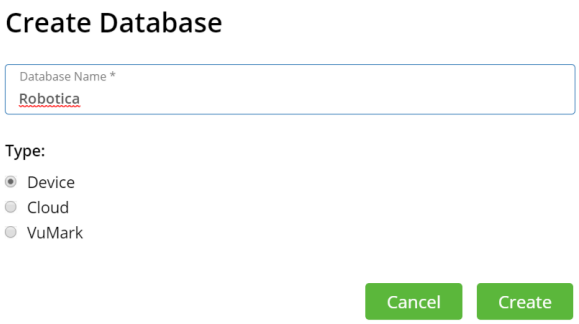

Figura 69. Selección del lugar donde se guardarán los targets de la base de datos de Vuforia.

Una vez creada, si entramos dentro de ella, se puede comenzar a añadir los *targets* de la aplicación. Estos *targets* pueden ser en forma de imagen, cubo, cilindro o un objeto 3D. Cuando se han añadido, el algoritmo de Vuforia detecta las características de la imagen y le otorga una puntuación en estrellas, dependiendo de la generación de características necesarias para realizar el reconocimiento. (Figura 70).

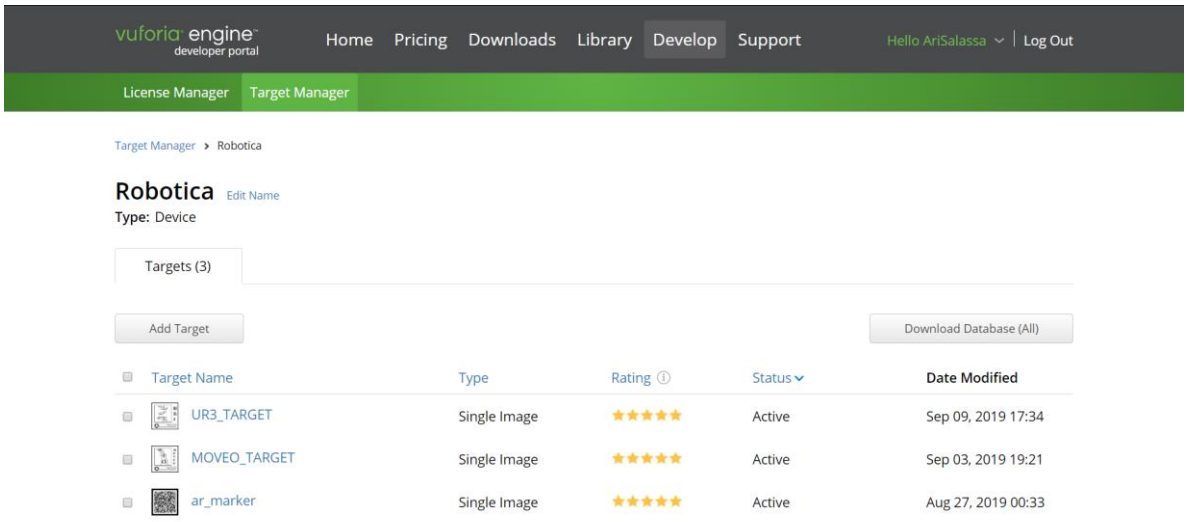

Figura 70. Lista de targets, con sus correspondientes atributos (nombre, tipo, puntuación, estado, fecha de modificación).

Cuando todos los *targets* han sido generados y procesados, se pueden seleccionar los targets de interés, y descargarlos desde el botón *"Download Database"*, seleccionando como plataforma de desarrollo *"Unity Editor"*. Esto generará un archivo tipo "*.unitypackage*" que contiene todos los elementos necesarios para incluir los *targets* en

la aplicación. Para ello, se debe ejecutar el paquete cuando se tenga el entorno de desarrollo abierto con el proyecto. Se abrirá una ventana para importar los elementos necesarios listos para utilizar en el desarrollo.

Una vez realizado esto, se puede incluir el *target* en la escena a través de *click derecho –> Vuforia Engine –> Image*. Así se generará un *target* en forma de imagen y, si se ha importado correctamente el *target*, debería ser el *target* por defecto en el módulo de configuración del *GameObject ImageTarget* que se acaba de crear. En caso contrario, o en caso de que se esté trabajando con diferentes targets o bases de datos de *targets*, se puede seleccionar el *target* deseado desde el Inspector. (Figura 71).

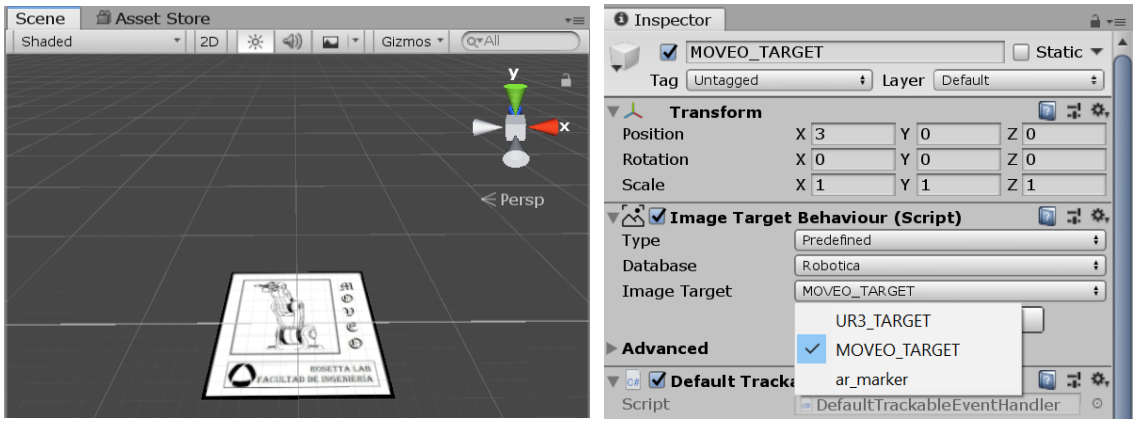

Figura 71. Inserción del target de vuforia como un GameObject del tipo Image en la escena de Unity.

ii. Configuración de ROS#

Descargar el repositorio de *rossharp*[24] y copiar la carpeta *RosSharp* dentro de la carpeta *Assets* del proyecto. Es necesario que Unity esté utilizando *.NET Framework 4.x,* ya que es requerido por *RosBridgeClient.* Para configurarlo:

- En el menú de Unity, ir a *Edit -> Project Settings -> Player.*
- En el Inspector seleccionar *Others Settings -> Configuration.*
- Setear *Scripting Runtime Version** a *.Net4.x Equivalent.*

Una vez realizado esto, *RosBridgeClient* y *UrdfImporter* están incluidos en el proyecto. Una vez que los *plugins* han sido cargados, aparecen nuevos ítems en el menú superior (Figura 72) y en la generación de nuevos *GameObjects* (Figura 73).

[24] Link del repositorio: https://github.com/siemens/ros-sharp.

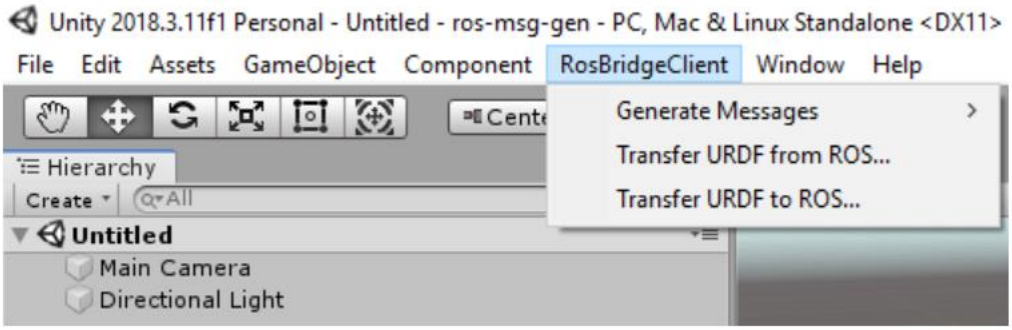

Figura 72. Impacto de incluir ROS# en el menú superior de Unity.

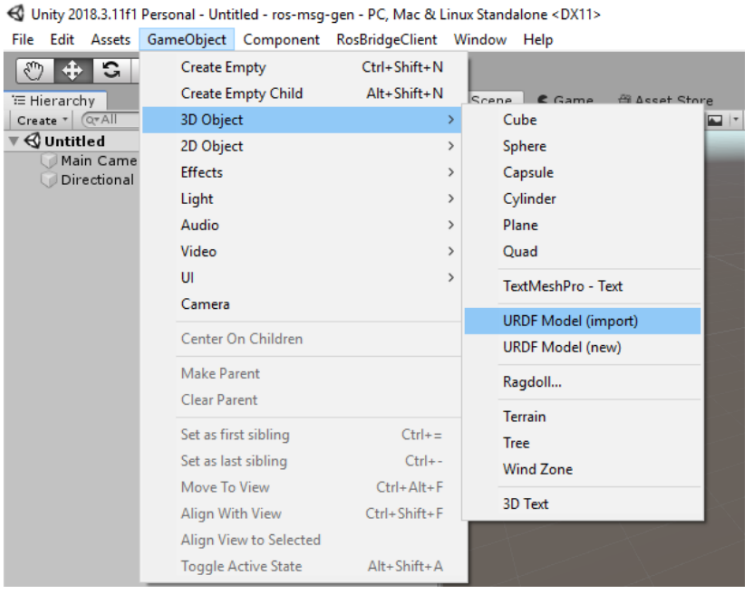

Figura 73. Posibilidad de trabajar con robots en URDF al haber incorporado ROS#.

Además, varios *scripts* y *GameObjects* ahora están contenidos en la carpeta *Assets*. Estos logran diferentes funcionalidades, como consumir y publicar tópicos, llamar y anunciar nuevos servicios, etc.[25]

iii. Configuración de compilación.

En este apartado se exponen los ajustes necesarios de para generar el archivo ".apk" para poder instalar la aplicación en dispositivos Android.

El primer ajuste que se debe hacer es configurar la cámara *ARCamera* para que sea utilizable en los dispositivos móviles. Dentro de las opciones de *Rendering Path,* seleccionar *Legacy Deferred (light prepas).* (Figura 74).

[25] Link de ejemplos de aplicación con ROS#: https://github.com/siemens/ros-sharp/wiki/User_App_ROS_ApplicationExamplesWithROSConnection.

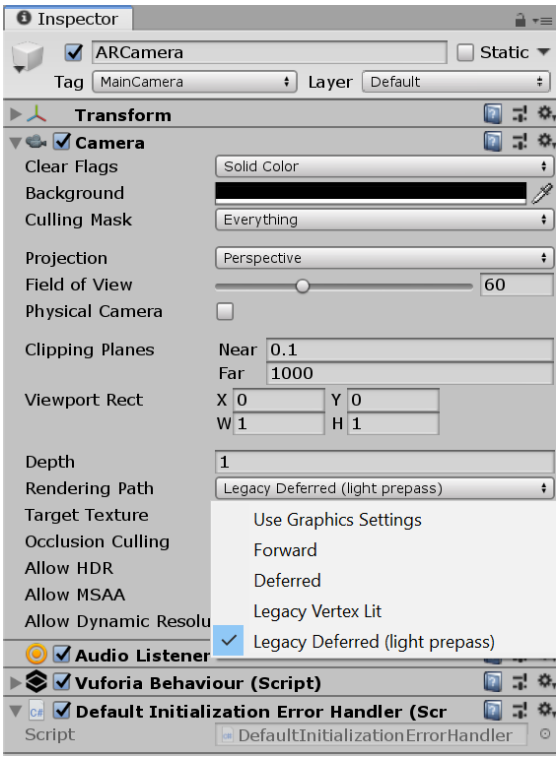

Figura 74. Configuración de la cámara principal de Vuforia.

Para configurar las opciones de compilación se debe ir nuevamente a *Player Settings*, siguiendo los mismos pasos que se mencionaron anteriormente y seleccionar "*Other Settings*". Para generar un ".apk" optimizado es recomendable utilizar "*IL2CPP*". Al construir un proyecto usando *IL2CPP*, Unity convierte el código en lenguaje *intermedio* de los *scripts* y ensambla a C ++, antes de crear un archivo binario nativo (.exe, apk, .xap, por ejemplo) para la plataforma elegida. Para realizar la compilación es necesario especificar el hardware del microprocesador del dispositivo (*Target Architectures*). Si no se está seguro, o si se quiere abarcar la mayor cantidad de dispositivos, seleccionar todas las opciones. (Figura 75).

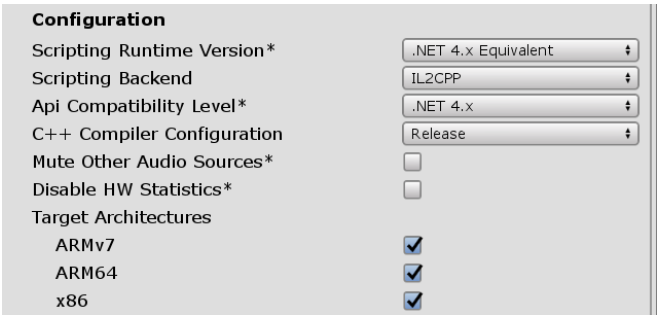

Figura 75. Configuración de las opciones de compilación. Selección de arquitecturas.

El último paso de configuración es configurar la API necesaria para el renderizado de gráficos. Se recomienda deseleccionar la opción "*Auto Graphics API*" y dejar *OpenGLES3* en la cima de la lista. (Figura 76).

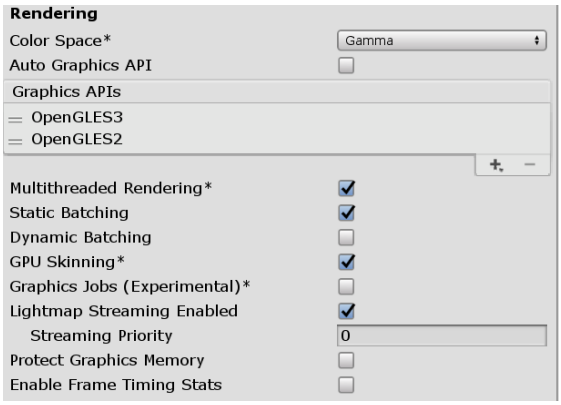

Figura 76. Opciones de compilación. Configuración del renderizado.

Por otro lado, la primera vez que se realice un proyecto para Android, Unity necesitará conocer la ubicación de la carpeta donde se ha instalado *Android SDK.* Como se está utilizando *IL2CPP* es además necesario el *Android Native Development Kit* (NDK), que contiene las herramientas precisas para construir las bibliotecas necesarios y finalmente producir el paquete de salida (*".apk"*)[26].

[26] Link para una información más detallada de los pasos de configuración a seguir cuando se trabaja con proyectos de Android en Unity: https://docs.unity3d.com/560/Documentation/Manual/android-sdksetup.html.

6. Resultado final

Finalmente, se obtuvo un sistema global funcional integrando los cinco componentes desarrollados en el informe: nodo *Orchestrator* (gestor de conexiones), nodo RR (interfaz con robot real), nodo *Pro.Sim.* (simulador industrial virtual), nodo *A.R. Engine* (simulación con realidad aumentada) y nodo *Control GUI* (interfaz de Matlab *teach-pendant*).

En la Figura 77 se puede apreciar un esquema del sistema resultante. Dentro del mismo, se han reutilizado lo que engloba la sección "Robot Real" (debajo del nodo *Real Robot*) y la sección "Matlab" (debajo del nodo *Control GUI*). En el primer caso se ha utilizado como banco de pruebas físico el robot MOVEO. En el segundo caso se ha reutilizado y adaptado la interfaz de control desarrollada por la cátedra de Robótica I.

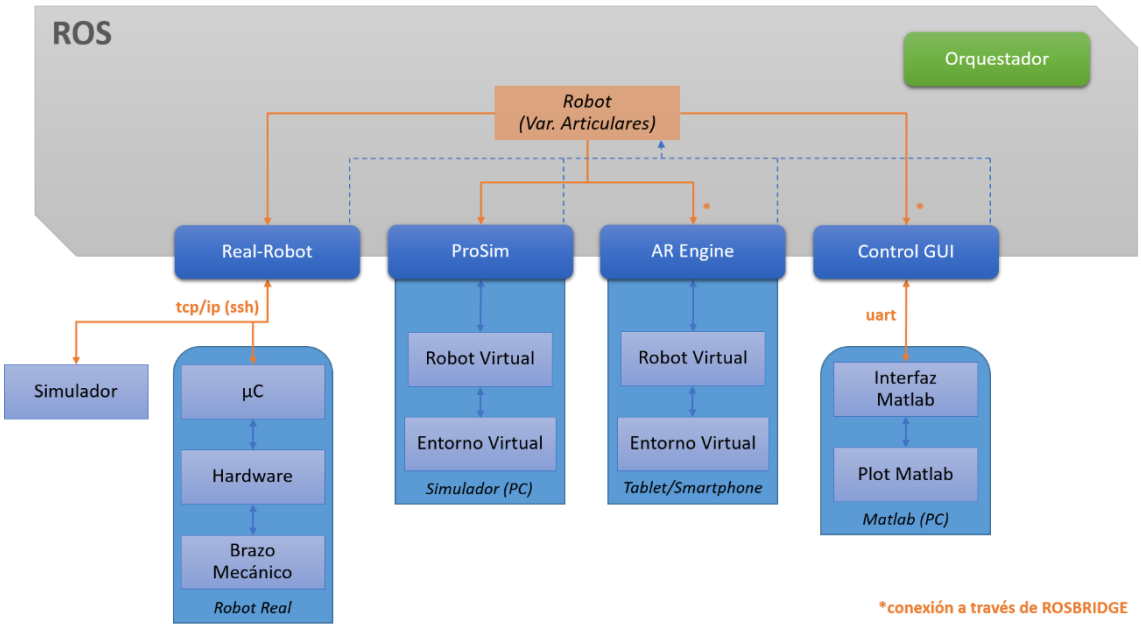

Figura 77. Esquematización del sistema global.

En la Figura 78 se muestra una representación resumida del sistema desarrollado que pone en evidencia la importancia de *Rosbridge.*

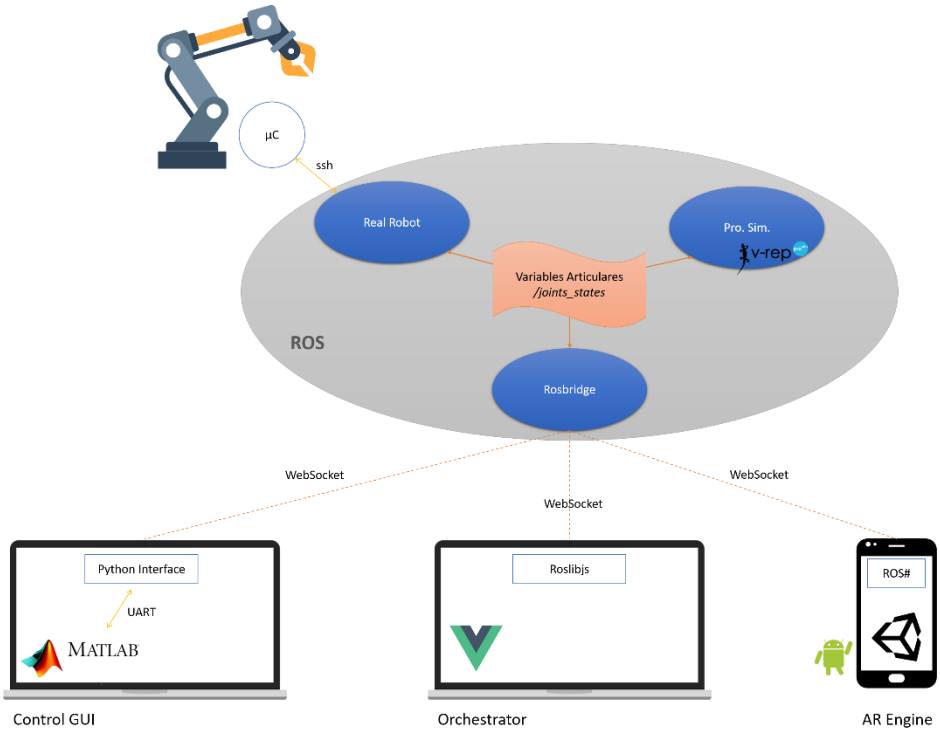

Figura 78. Funcionamiento global del sistema, considerando Rosbridge.

La Figura 79 pone de manifiesto el hardware requerido para correr el sistema completo. Nótese que los subsistemas pueden correr en una misma máquina (virtual o real) siempre y cuando haya compatibilidad de lenguajes y sistemas operativos. De esta manera los nodos *Real Robot, Rosbridge, Orchestrator* y *Pro.Sim.* se ejecutan en una máquina con sistema operativo Linux; el nodo *Control GUI* corre en una máquina con Windows; la aplicación de Unity corre en un emulador de Android; y, finalmente, Machinekit corre sobre una virtualización de LinuxCNC.

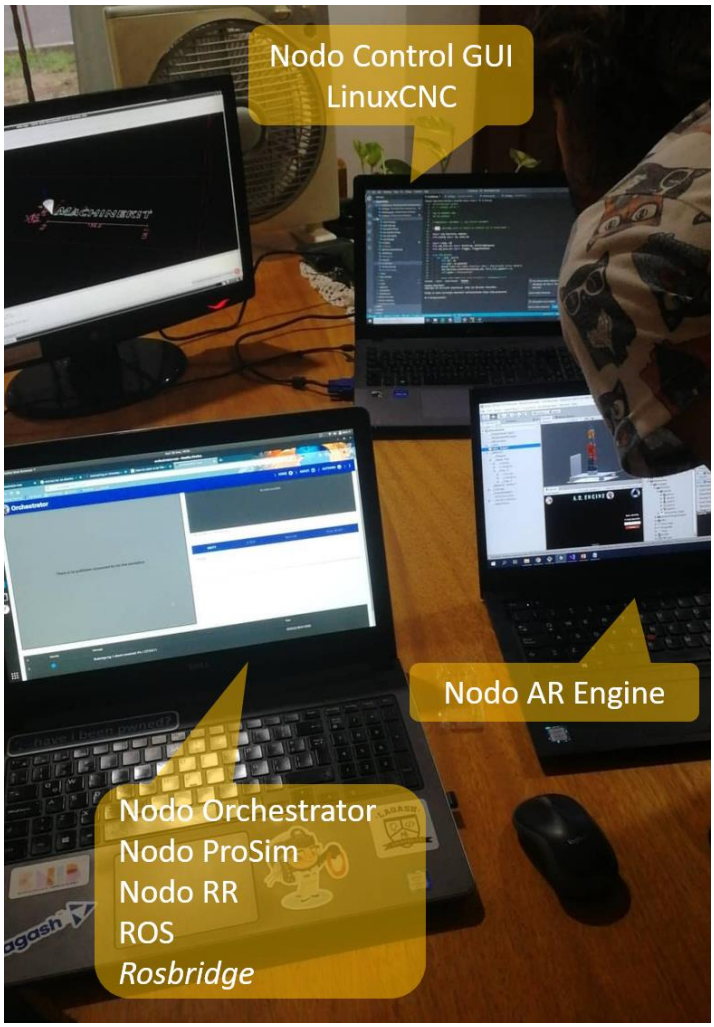

Figura 79. Pruebas integrales sobre el sistema.

El sistema resultante constituye una base didáctica muy robusta y potente para el estudio de diferentes áreas de la robótica industrial en la facultad. Siendo potencialmente utilizable en diferentes cátedras de la carrera de mecatrónica: Robótica I y 2, Realidad Aumentada, Inteligencia Artificial, Microcontroladores y Electrónica de Potencia, Concepción y Fabricación Asistida por Computadora, Control y Sistemas, entre otras.

El sistema es utilizable tanto de manera local (cada alumno puede correr una instancia del sistema global) como de manera distribuida (los alumnos conectan diferentes nodos a un mismo sistema global). Esta última metodología podría ser de gran utilidad académica ya que cada estudiante puede estar como espectador y el control del sistema es otorgado al profesor a cargo.

Si bien se tomó como base el robot MOVEO para la realización del proyecto, se hicieron muchos desarrollos también para el robot UR3, quedando prácticamente integrado al sistema.

El sistema fue desarrollado para que la incorporación de un nuevo robot sea sencilla. De esta manera, el proyecto constituye una herramienta que puede ampliar las posibilidades de los robots presentes en la cátedra y de los que eventualmente se fabricarán en un futuro.

7. Mejoras propuestas – Futuros desarrollos

El sistema que se ha desarrollado es completamente funcional. Aun así, también se considera que este proyecto tiene mucho potencial de crecimiento y que, conforme se vayan volcando horas de trabajo sobre él, irán surgiendo múltiples oportunidades de mejora. A continuación, se mencionan alguna de ellas:

- Migración de ROS1 hacia ROS2*:* En el tiempo que se desarrollaba este proyecto, ROS lanzó una nueva versión mejorada de su framework, denominada ROS2. Adoptar esta plataforma actualizada implicaba un cambio sumamente drástico en el sistema, por lo que se optó por preservar ROS1. Si bien ROS2 es beneficioso pues ofrece muchas ventajas sobre ROS1 (corre en W10 además de correr en Ubuntu, tiene estándares de C++ más actualizados, utiliza Python 3.5 en vez de Python 2, tiene mejor comportamiento en tiempo real y soporte de WebSockets), los paquetes más populares y de mayor uso fueron desarrollados para ROS1 y no son compatibles (muchas veces hay incompatibilidad incluso entre distribuciones de ROS). Una mejora sería hacer una migración completa a ROS2, prescindir de *Rorbridge* de forma tal de tener un esquema de nodos más limpio, y actualizar en cada uno de los nodos la forma de conexión al nuevo entorno de ROS [49].
- Agregar validaciones adicionales del robot simulado: Actualmente cada uno de los nodos vuelca y consume información hacia y desde el tópico */joints_states* sin realizar ninguna validación respecto de si el nodo *Publisher* y los nodos *Subscribers* hacen referencia al mismo robot. Para sortear este problema se han pensado dos posibles soluciones. Una es setear el nombre del robot que se va a simular dentro de los parámetros globales de ROS y actuar en consecuencia en cada uno de los nodos. La otra solución es asignarle al nombre del tópico un nombre más descriptivo como: "*/[robot_name]/joints_states*". En cualquiera de los dos casos el orquestador debería ser capaz de discriminar las conexiones.
- Mejorar sincronía de movimientos de los diferentes nodos: En la lógica del sistema desarrollado, el nodo *Publisher* vuelca en el tópico */joints_states* valores articulares que el resto de los nodos *Subscribers* toman como consigna. A partir de allí, cada nodo evoluciona según su propia dinámica, que, si bien son bastante similares, no son exactamente iguales. La manera más rápida de solucionar este problema sería crear un tópico de *set-points*, donde sólo uno de los nodos puede ser el *Publisher*, y otro tópico de estado de la articulación donde sólo el nodo del robot real vuelca, en caso de estar presente, la información pertinente. De esta forma, las simulaciones enviarían simplemente consignas de posición, pero evolucionarían conforme el robot real vaya evolucionando. Otra manera implicaría mejorar drásticamente la dinámica y la física de los robots en las

simulaciones y agregar controladores de movimiento virtuales que reduzca la brecha de comportamiento entre lo real y lo simulado. Para implementar esta mejora sería necesaria una vasta experiencia en el modelado de cuerpos sólidos, por un lado, y en Unity y CoppeliaSim, por el otro. De esta forma se sabría con exactitud cómo ajustar las distintas variables de simulación en cada programa para obtener una representación dinámica más exacta.[27]

- Expandir la compatibilidad del nodo RR*:* El nodo *Real Robot* está preparado para conectarse con LinuxCNC porque ese ha sido el sistema operativo de control cargado en el robot disponible. Pero esto no tiene por qué ser así. El robot puede ser controlado por otros tipos de controladores que tengan un sistema operativo diferente, o bien, que no soporten sistemas operativos (como, por ejemplo, un microprocesador Atmel). La ventaja del desarrollo realizado es que dicho nodo puede evolucionar de forma tal de detectar el microprocesador utilizado en el robot y ejecutar los comandos que sean necesarios para que el mismo pueda ser visible dentro del entorno de ROS.
- Interfaz gráfica para el nodo RR*:* Desarrollar una pequeña GUI para gestionar los cambios de modo, conexión y volcado de *logs* así como se hizo con los demás nodos. Actualmente el cambio de modo se acciona mediante servicios ROS y los reportes se emiten por consola.
- Utilizar *Rosbridge* para el nodo RR*:* El nodo *Real Robot* fue uno de los primeros en ser desarrollados y se optó en ese momento por una comunicación SSH entre el ordenador que ejecuta el driver y el sistema operativo del robot remoto con LinuxCNC. A medida que el proyecto fue avanzando, se incorporó el paquete de *Rosbridge* para la comunicación de otros nodos. Una posible mejora sería comparar el desempeño de ambos y eventualmente incorporar *Rosbridge* al nodo RR, utilizando WebSockets y descartando la conexión SSH.
- Graficar información didáctica del robot*:* Como bien se mencionó anteriormente, este proyecto buscaba tener un alto contenido didáctico que sirviese como soporte a la cátedra de Robótica I. Un futuro incremento a este trabajo sería agregar la visualización de cómo y dónde se colocan los sistemas de referencia de cada una de las articulaciones según la convención de D-H, mediante el desarrollo de una pequeña animación en Unity que haga que el sistema de la base del robot vaya evolucionando hasta llegar al sistema del

[27] Esta óptica intentó abordarse desde un principio, pero los resultados no fueron satisfactorios. La falta de información de la dinámica los robots (amortiguamiento, rigidez, torque, etc.) y el trabajo que implicaría obtenerla, hicieron que fuera descartada y se optara por otras soluciones. En *CoppeliaSim* se colocó un controlador de movimientos idéntico para cada una de las articulaciones con ganancia proporciona e integral. En Unity se prescindió de utilizar los componentes "*HingleJoint*" *de* las articulaciones y el motor de física. En su lugar se decidió implementar transformaciones locales de cada articulación respecto a la anterior con cálculos puramente cinemáticos.

efector final, realizando todas las transformaciones algebraicas pertinentes en las articulaciones intermedias, y justificando así cada uno de los parámetros de la matriz representativa del robot.

- Utilizar la última versión de Unity para el nodo *A.R. Engine*: En el tiempo en que se desarrolló este proyecto, Unity pasó por múltiples actualizaciones. Esto no causó mayores inconvenientes dado que este programa presenta una buena compatibilidad entre versiones. Sin embargo, la frecuencia de actualizaciones (dos por mes en promedio) llevó a tomar la decisión de hacer el desarrollo en la versión 2018.4.12f1. Si bien los cambios a la hora de desarrollar la aplicación son, en el caso de este proyecto, imperceptibles, se presentaron algunos problemas de compatibilidad con el despliegue de la aplicación en algunos dispositivos Android. Versiones más recientes de Unity utilizan a su vez versiones más recientes de Android NDK (*Native Development Kit*), que pueden llegar a palear dichos problemas
- Simulación virtual interactiva del robot: Agregar objetos a la simulación en *CoppeliaSim* con los cuales el robot pueda interactuar. Ejemplo: incluir al robot en medio de una cadena de producción para que realice una tarea específica.
- Compatibilidad con robots paralelos y móviles*:* Sería interesante analizar, como trabajo a futuro, la compatibilidad del sistema presentado, con robots paralelos o robots móviles y hacer las adaptaciones que sean necesarias para agrandar el abanico de posibilidades.

8. Conclusión

Finalmente, logramos la concepción de un entorno de simulación para robótica el cuál puede ser utilizado en numerosas aplicaciones didácticas: banco de pruebas para brazos robóticos, herramienta de supervisión remota, ensayos en procesos productivos simulados, entre otras.

El mismo posibilita la simulación simultánea de un brazo robótico en un entorno virtual (Nodo *ProSim*), eventualmente inmerso en un proceso industrial simulado, en un entorno de realidad aumentada (Nodo *AR Engine*), controlable desde una interfaz *teach-pendant* (Nodo *Control GUI*) y con la posibilidad de conectar al sistema un robot real (Nodo *RealRobot*).

Gracias a sus diferentes módulos, este sistema es utilizable en desarrollos y en cátedras que involucren robótica industrial, realidad virtual, robótica cooperativa, sistemas de supervisión, automatización de procesos, entre otras áreas. Particularmente, dentro de la cátedra de robótica de la facultad, representa una herramienta que añade un gran valor agregado a cualquier brazo robótico que sea desarrollado siendo su adaptación al mismo relativamente fácil y sencilla.

Luego de haber realizado el trabajo y habiendo cumplido con los objetivos propuestos, hemos podido concluir con la formación que la facultad, a lo largo de muchos años, ha volcado sobre nosotros, dándole un cierre global a todos los conocimientos adquiridos. Esto ha quedado en evidencia al presentar el funcionamiento de un sistema integral de simulación de robots que utiliza tecnologías de punta en todos los frentes.

Pudimos combinar satisfactoriamente tanto los conocimientos adquiridos a lo largo de la carrera, como también los conocimientos adquiridos en el campo laboral, lo cual ha permitido que en este proyecto se incursione en cosas que a priori no presentaban compatibilidad, pero que luego han sido resultas satisfactoriamente. La motivación ha sido permanente alimentada en el desafío de encontrar soluciones y alternativas a los problemas que se presentaban a medida que desarrollaba el trabajo, y ha sido un motor en el desarrollo de este proyecto y en el aprendizaje de nuevas tecnologías.

Es grato destacar que durante la realización del proyecto se trabajó mucho con un robot desarrollado por colegas de la carrera como su proyecto final de estudios. El mismo, también fue realizado dentro del laboratorio Rosetta Lab bajo la tutoría de los profesores a cargo de la cátedra de Robótica I. De igual manera, este nuevo proyecto que hemos desarrollado busca dejar la puerta abierta a innumerables oportunidades de mejoras y de aportes para aquellos que se sientan desafiados a hacerlo, deseando mejorar así la calidad de la enseñanza de nuestros compañeros y futuros colegas.

9. Referencias

[1]. Gattas, Samir; Torres, Rodrigo (2019). *Desarrollo Sistema Robótico Educativo*. Facultad de Ingeniería – Universidad nacional de Cuyo.

[2]. *Modelo CAD 3D robot MOVEO original* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://github.com/BCN3D/BCN3D-Moveo

[3]. *Calameo – La realidad aumentada y la robótica* [fecha de consulta: septiembre 2019]. Disponible en: https://es.calameo.com/read/0032992276a4d576f6f6c

[4]. Elias Matsas, George; Christopher Vosniakos (2015). *Design of a virtual reality training system for human-robotic collaboration in manufacturing tasks.* National Technical University of Athens.

[5]. Yun Suen Pai, Hwa Jen Yap, Ramesh Singh (2014). *Augmented reality-based programming, planning and simulation of a robotic work cell*. University of Malasya.

[6]. Martinez Nogueroles, Miguel (2019). *Supervisión mediante realidad aumentada de un robot industrial.* Universidad de Alicante.

[7]. *Wikipedia - Desarrollo ágil de software* [fecha de consulta: septiembre 2019]. Disponible en: https://es.wikipedia.org/wiki/Desarrollo_%C3%A1gil_de_software

[8]. *Proyectosagiles.com – Qué es SCRUM* [fecha de consulta: septiembre 2019]. Disponible en: https://proyectosagiles.org/que-es-scrum/

[9]. *GitLab Docs* [fecha de consulta: septiembre 2019]. Disponible en: https://docs.gitlab.com/

[10]. *ROS.org* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://www.ros.org/

[11]. *ROS Documentation* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: http://wiki.ros.org/

[12]. *Coppelia Robotics* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: http://www.coppeliarobotics.com/

[13]. *CoppeliaSim User Manual* [en línea] [fecha de consulta: enero 2020]. Disponible en: http://www.coppeliarobotics.com/helpFiles/

[14]. *Lua 5.3 Reference Manual* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://www.lua.org/manual/5.3/

[15]. *VueJS – Guía de Uso* [fecha de consulta: noviembre 2019]. Disponible en: https://es.vuejs.org/v2/guide/index.html

[16]. *Manual de Unity* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://docs.unity3d.com/es/530/Manual/UnityManual.html

[17]. *Microsoft, Documentación de C#* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: https://docs.microsoft.com/es-es/dotnet/csharp/

[18]. *Unity Scripting API* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://docs.unity3d.com/es/530/ScriptReference/index.html
[19]. *Unity Forum* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://forum.unity.com/
[20]. *Unity Asset Store* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: https://assetstore.unity.com/
[21]. *Getting Started with Vuforia Engine in Unity* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://library.vuforia.com/articles/Training/getting-started-with-vuforia-in-unity.html
[22]. *Vuforia Support Center* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://developer.vuforia.com/support
[23]. *Documentación Matlab* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: https://la.mathworks.com/help/matlab/
[24]. *Machinekit Documentation* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://www.machinekit.io/docs/
[25]. *LinuxCNC Documents* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: http://linuxcnc.org/documents/
[26]. *ROS - Client Libraries* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: http://wiki.ros.org/Client%20Libraries
[27]. *Rosbridge_server* [en línea] [fecha de consulta: octubre 2019]. Disponible en: http://wiki.ros.org/rosbridge_server
[28]. *Wikipedia - WebSocket* [fecha de consulta: septiembre 2019]. Disponible en: https://es.wikipedia.org/wiki/WebSocket
[29]. *VMWare – Worktation Player* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://www.vmware.com/ar/products/workstation-player.html
[30]. *Python 2.7 Documentation* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://docs.python.org/2/
[31]. *Welcome to Paramiko!* [en línea] [fecha de consulta: octubre 2019]. Disponible en: http://www.paramiko.org/
[32]. *Python Interface* [en línea] [fecha de consulta: noviembre 2019]. Disponible en: http://linuxcnc.org/docs/2.6/html/common/python-interface.html
[33]. *Roslibpy* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://roslibpy.readthedocs.io/
[34]. *Graphical User Interface with Tk* [en línea] [fecha de consulta: noviembre 2019]. Disponible en: https://docs.python.org/2.7/library/tk.html
[35]. *Welcome to pySerial's documentation* [en línea] [fecha de consulta: noviembre 2019]. Disponible en: https://pythonhosted.org/pyserial/

[36]. *Etherlogic.com – Virtual Serial Port Emulator* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: http://www.eterlogic.com/Products.VSPE.html

[37]. *ROS / ROS 2 Interface* [en línea] [fecha de consulta: enero 2020]. Disponible en: http://www.coppeliarobotics.com/helpFiles/en/rosInterf.html

[38]. *ROS - Urdf* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: http://wiki.ros.org/urdf

[39]. *Cob_srvs* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: http://wiki.ros.org/cob_srvs

[40]. *ROS# User Documentation* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://github.com/siemens/ros-sharp/wiki

[41]. *Lean Touch Documentation* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: http://carloswilkes.com/Documentation/LeanTouch

[42]. *Wikipedia - Singleton* [fecha de consulta: septiembre 2019]. Disponible en: https://es.wikipedia.org/wiki/Singleton

[43]. *Roslibjs* [en línea] [fecha de consulta: octubre 2019]. Disponible en: http://wiki.ros.org/roslibjs

[44]. *Vuetify* [fecha de consulta: noviembre 2019]. Disponible en: https://vuetifyjs.com/en/

[45]. *Vue Router - Introduction* [fecha de consulta: noviembre 2019]. Disponible en: https://router.vuejs.org/

[46]. *What is Vuex?* [fecha de consulta: noviembre 2019]. Disponible en: https://vuex.vuejs.org/

[47]. *LuaRocks* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://luarocks.org/

[48]. *LuaRocks, f-strings* [en línea] [fecha de consulta: octubre 2019]. Disponible en: https://luarocks.org/modules/hisham/f-strings

[49]. *ROS2 Documentation* [en línea] [fecha de consulta: diciembre 2019]. Disponible en: https://index.ros.org/doc/ros2/

[50]. *Universal Robots* [en línea] [fecha de consulta: septiembre 2019]. Disponible en: https://www.universal-robots.com/