UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Interacción entre webcam y brazo robot para el posicionamiento del efector final

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: **Joaquín Macanás Valera**

Director: Jorge Feliu Batlle

Codirector: Carlos Alberto Díaz Hernández

Cartagena, Septiembre 2014

Al comienzo de este proyecto me gustaría dar las gracias a mi familia por su apoyo incondicional en mis estudios desde siempre y a todos los compañeros y profesores que han hecho crecer en mí la pasión por la ingeniería electrónica y la robótica. Pero en especial:

A Juan Luis y a Pablo por estos 4 estupendos años de convivencia que los han convertido en una segunda familia.

A Samper por ser compañero infatigable de risas y trabajo en los momentos más difíciles.

Una gran etapa finaliza, una mejor comienza.

ÍNDICE

Índice de figuras

Capítulo 1

OBJETO DEL PROYECTO

1. Objeto del proyecto

1.1. Marco de trabajo

El presente proyecto tiene como objetivo desarrollar un sistema robótico de posicionamiento guiado por visión artificial. El proyecto se enmarca dentro de las líneas de investigación del Departamento de Ingeniería de Sistemas y Automática así como dentro del grupo de investigación Neurocor, en cuyo laboratorio se ha realizado el desarrollo del mismo.

Aunque el proyecto se encuentra dentro de dichas líneas de investigación, se desmarca significativamente de ellas por ser un proyecto pionero en esta universidad. Esto es lo que le confiere su gran valor añadido. Introduce tecnologías nunca utilizadas en el departamento en proyectos o desarrollos anteriores, como la captación 3D de objetos y entornos en nube de puntos o la manipulación robótica con el middleware de última generación ROS, lo cual le confiere al proyecto un perfil investigador remarcable.

1.2. Descripción

La arquitectura del sistema está formada por un brazo robótico Schunk Powerball Lightweight Arm LWA 4P, una cámara RGBD Kinect de Microsoft® y el middleware robótico del momento, ROS.

En la siguiente infografía se explica cómo interaccionan todos los elementos del sistema. Como se puede ver también se ha establecido un trabajo cooperativo entre dos PC con diferentes versiones de Linux los cuales se comunican mediante sockets TCP/IP.

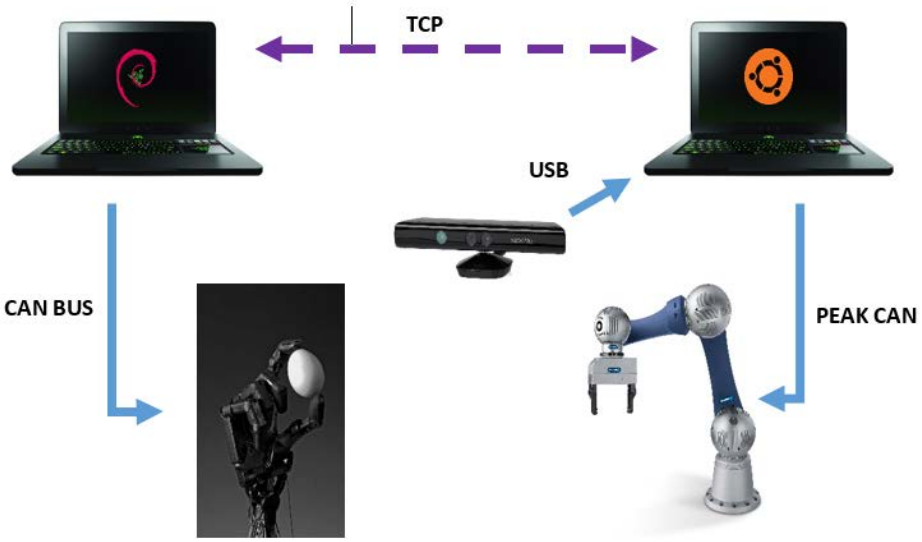


Ilustración 1.1 - Arquitectura del sistema

El objetivo último es que el robot sea capaz de posicionarse en el punto exacto que le permita a su efector final coger el objeto que el sistema de posicionamiento en 3D haya detectado.

Dentro de ROS nos apoyaremos sobre dos paquetes orientados a la robótica y la visión artificial, los cuales serán el paquete de control y gestión robótica *Moveit!* y la librería de visión artificial PCL (Point Cloud Library).

Ilustración 1.2 - Logo de Moveit!

Ilustración 1.3 - Logo de PCL

1.3. Relación de objetivos del proyecto

Este proyecto plantea por tanto una amplia lista de objetivos que se han de cumplir a su finalización. A continuación enumeran de forma detallada todos y cada uno de ellos.

- Familiarización, aprendizaje y uso avanzado de Linux.

- **Aprendizaje de programación orientada a objetos en C++ y familiarización con el lenguaje interpretado Python**.

- **Aprendizaje y desarrollo de sistemas basados en ROS**.

- Estudio de sistemas de visión para la discriminación de objetos.

- **Creación de un sistema de captación de 3D y tratamiento de imágenes con reconocimiento de objetos en base a la librería de visión PCL**.

- Configuración de un sistema robótico real para su funcionamiento en base a comandos de trabajo.

- **Gestión de dicho sistema real a través del paquete de manipulación *Moveit!***.

- **Creación de un sistema de comunicación entre ordenadores basado en sockets de comunicación TCP/IP para la sincronización entre la mano robótica y el brazo**.

- Integración de múltiples sistemas bajo una misma plataforma de trabajo conjunta.

- **Diseño de piezas mecánicas de ensamblaje para unir físicamente el brazo Schunk Powerball LWA-4P a la mano robótica ShadowHand**.

De los objetos arriba descritos se puede ver cómo algunos de ellos están remarcados en negrita. Esos objetivos no se contemplaban en la rúbrica inicial, por lo que son ampliaciones del proyecto en un principio planteado. Todas estas ampliaciones dotan al proyecto de un gran valor añadido, pues exceden con creces los requisitos iniciales de desarrollo propuestos.

Capítulo 2

OBJETO DEL PROYECYO

2. INTRODUCCIÓN

2.1. Orígenes de la automática y la robótica

Desde los inicios de la ingeniería el hombre siempre ha intentado crear máquinas y dispositivos que hiciesen más fácil y cómoda su vida, pues no es sino éste su fin último. Al afán por crear máquinas automáticas, término proveniente del griego *"automatos"*, se remonta a los mecanismos animados de Herón de Alejandría, datados en el 85 d.C. Dichos mecanismos se movían mediante dispositivos hidráulicos, palancas y poleas. Cabe destacar que por aquel entonces únicamente se atisbaba un fin lúdico en tales artefactos.

Desde entonces diferentes han sido las culturas que han seguido los inicios de los griegos y han desarrollado a lo largo de la historia nuevos artefactos automáticos. Un buen ejemplo de ello fue la cultura árabe que desde el siglo VIII al XV crearon dispositivos automatizados ya no sólo con carácter lúdico, sino de utilidad práctica, como dispensadores de agua para beber o lavarse, todos ellos claro está, destinados a la realeza. Otros ejemplos de aquellas primeras incursiones en los autómatas son el *Hombre de hierro* de Alberto Magno (1204 - 1282) o la *Cabeza parlante* de Roger Bacon (1214 – 1294).

Fue ya a finales del siglo XVIII y a principios del XIX cuando los ingenios mecánicos diseñados, principalmente para la industria textil, dieron comienzo a la verdadera era de la automatización, y que posteriormente darían a luz a la de la robótica. Importantes desarrollos de este período son la hiladora giratoria de Hargreaves (1770), la hiladora mecánica de Crompton (1779) o el telar mecánico de Cartwright (1785). Pero sin duda el que marcó la diferencia fue el telar de Joseph Jacquard, creado en 1801, el cual utilizaba una cinta de papel perforada con un código que servía de programa para el funcionamiento de la máquina. Éste fue el punto de inflexión para el inicio de la automatización industrial.

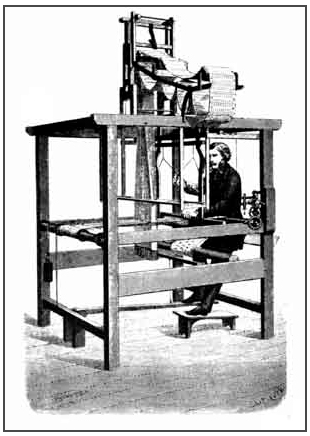


Ilustración 2.1 - Telar de Jacquard

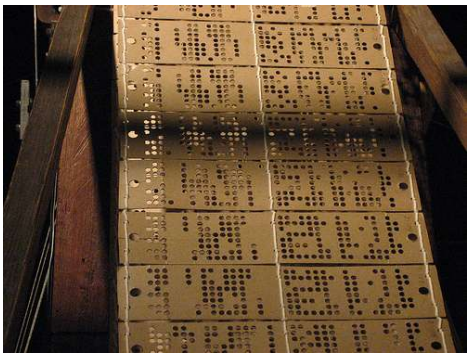


Ilustración 2.2 - Tarjetas perforadas con el programa para el telar

¿Pero cómo nace el término robot? Para encontrar la respuesta tenemos que remontarnos al año 1921, cuando el escritor checo Karel Capek estrenó en el teatro nacional de Praga la obra *Rossum's Universal Robot (R.U.R.)*. El origen es la palabra eslava *robota*, cuyo significado es el de trabajo realizado de manera forzada. El argumento de la obra se basa en que los robots de R.U.R. eran máquinas androides fabricadas a partir de una "fórmula" obtenida por un genio científico llamado Rossum. En la obra los robots servían a los humanos desarrollando todo tipo de trabajos físicos, hasta que al final se rebelan contra ellos y los acaban asesinando a todos, a excepción de uno de sus creadores, con la frustrada esperanza de que les enseñase a reproducirse.

Pero el término hubiera sido pronto olvidado si no hubiese sido mundialmente popularizado por el escritor americano de origen ruso Isaac Asimov, que es sin duda el máximo impulsor de esta idea. En octubre de 1945 publicó una historia en la que por primera vez enunció sus "tres leyes de la robótica", que han terminado por hacerse mundialmente conocidas:

1. Un robot no puede perjudicar a un ser humano, ni con su inacción permitir que un ser humano sufra daño.

2. Un robot ha de obedecer las órdenes recibidas de un ser humano, excepto si tales órdenes entran en conflicto con la primera ley.

3. Un robot debe proteger su propia existencia mientras tal protección no entre en conflicto con la primera o segunda ley.

También se le atribuye a Asimov la creación del término “robotics” (robótica).

2.2. Historia y desarrollo de la robótica de manipulación

La robótica de manipulación se aleja un poco de lo que es la idea original de robot, más centrada en máquinas de aspecto humanoide capaces de imitar el comportamiento humano. Este tipo de robótica (la más extendida a día de hoy) tiene sus comienzos en los tele-manipuladores. El objetivo primero que motivó la creación de estos dispositivos era poder manipular elementos radiactivos sin poner en riesgo la salud del operador. Fue R.C. Goertz, del Argonne National Laboratory quien en 1948 desarrolló el primer tele-manipulador con dicho propósito, el cual consistía en un dispositivo mecánico maestro-esclavo. El manipulador maestro, situado en una zona segura, era movido directamente por el operador, mientras que el esclavo, situado en la zona de contacto con los elementos radioactivos, y unido mecánicamente al maestro, reproducía fielmente los movimientos de éste. El operador además de poder observar a través de un grueso cristal el resultado de sus acciones, sentía a través del dispositivo maestro, las fuerzas que el esclavo ejercía sobre el entorno. Más fueron los dispositivos que se crearon y ya con cierta electrónica en su interior. Debido a eso, pronto otras industrias se interesaron por ellos, como la submarina a lo largo de los sesenta y la aeroespacial en los setenta.

El advenimiento de la robótica de manipulación se produjo cuando se sustituyó al operario por un programa informático que controlaba los movimientos y acciones del manipulador, pasándose a llamar ahora por ello, robot.

Fue George C. Devol, ingeniero e inventor norteamericano, quien siendo autor de varias patentes estableció las bases del robot industrial moderno. En 1954 concibió la idea de un dispositivo de transferencia de artículos programada, la cual patentó en Estados Unidos en 1961. Posteriormente crearía junto con Joseph F. Engelberger, director de ingeniería de la división aeroespacial de la empresa *Manning Maxwell & Moore*, la *Consolidated Controls Corporation*, para la utilización de máquinas industriales, que posteriormente se convertiría en *Unimation (Universal Automation)*. Fueron los primeros en instalar un robot en

la fábrica de General Motors en Trenton, New Jersey; un *Unimate* (1960) para fundición por inyección.

En Japón pronto se superó el desarrollo americano gracias al impulso de Nissan, que creo la primera asociación robótica del mundo en 1972, la *Asociación de Robótica Industrial de Japón (JIRA).* Dos años después los estadounidenses hicieron lo propio creando el *Instituto de Robóticas de América.*

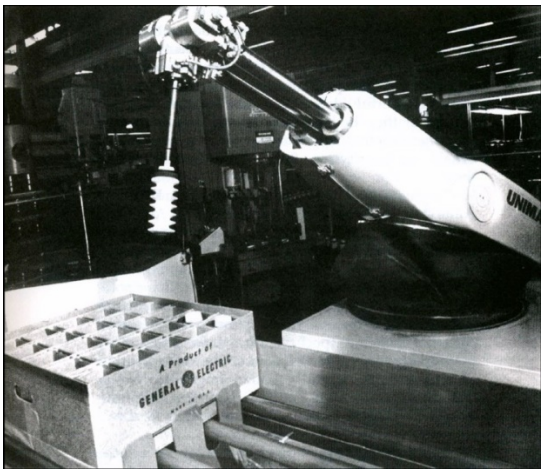


Ilustración 2.3 - Robot Unimate

En Europa la industria robótica llegó un poco después. No fue hasta 1973 cuando la firma sueca ABB construyó el primer robot de accionamiento totalmente eléctrico, el IRb6.

La evolución de los robots manipuladores en la industria durante los últimos cuarenta años ha sido espectacular. La mejora en la precisión, velocidad de actuación, repetibilidad y exactitud de movimientos ha sido drástica. Han permitido un aumento en el flujo de trabajo y en la calidad de los productos elaborados soberbia, siendo elementos fundamentales en cualquier cadena de montaje. Pero no se queda ahí su campo de actuación, pues han copado todo tipo de trabajos en la industria y fuera de ella, como pueden ser: robots espaciales (brazos para lanzamiento y recuperación de satélites, vehículos de exploración lunar, robots para construcción y mantenimiento de hardware en el espacio); robots para aplicaciones submarinas y subterráneas (exploración submarina, instalación y mantenimiento de cables telefónicos submarinos, limpieza e inspección de tuberías y drenajes subterráneos, inspección de sistemas de refrigeración de centrales nucleares); robots militares (desactivación de bombas,

robots centinelas experimentales dedicados a patrullar áreas críticas); robots móviles industriales (robots bomberos para patrullar fábricas, robots bibliotecarios, robots andantes con piernas); aplicaciones médicas (prótesis robotizadas, sistemas de ayuda a discapacitados); aplicaciones agrícolas (sembrado y recogida de cosechas, robot para esquilar ovejas); y un largo etcétera.

2.3. Brazos manipuladores en el mercado actual

En el mercado existen una gran cantidad de modelos, siendo la inmensa mayoría destinados a la industria. En esta breve reseña nos centraremos en los modelos de otras firmas que se encuadran más en el perfil del Powerball de Schunk, brazos de pequeña envergadura orientados a tareas de manipulación precisa o simulación de comportamientos antropomórficos.

Ilustración 2.4 - KUKA LWR

KUKA LWR

Robot de 7 ejes de la prestigiosa firma KUKA. Con un alcance de 0,8 m y peso máximo de carga 8 es el principal competidor del Powerball. Tiene un volumen de trabajo de 1,7 m^3. Se está usando en muchos laboratorios por su gran calidad de posicionamiento y la velocidad de respuesta de los motores.

Ilustración 2.5 - KINOVA JACO

KINOVA JACO

Este innovador robot de 6 DOF, diseñado tanto para asistencia personal como para tareas de investigación, es especialmente ligero con sólo 5,7 kg pudiéndose instalar en sillas de ruedas y similares para tareas de ayuda a discapacitados. Su carga máxima es de 1,5 kg, más que suficiente para las tareas diarias de una persona. De estructura de fibra de carbono e impermeable es de gran durabilidad. Puede funcionar con batería de 24 V o con la red eléctrica y su consumo se asemeja al de una bombilla. Tiene un alcance de 0,9 m y puede moverse hasta una velocidad de 20 cm/s.

BARRET WAM

Ilustración 2.6 - Barret WAM

El brazo WAM es un manipulador de alta precisión. Según la firma es el único brazo que se vende en el mundo con capacidad de transmisión directa con soporte de Transparent Dynamics™ entre los motores y las articulaciones, por lo que su control de las fuerzas de contacto es robusta e independiente de sensores mecánicos de fuerza o par. Está construido para superar a los robots convencionales con una inigualable destreza similar a la humana.
El brazo WAM™ está disponible en dos configuraciones principales, 4 grados de libertad y 7 grados de libertad, ambos con cinemática de apariencia humana.

UNIVERSAR ROBOTS UR5

Ilustración 2.7 - UR5

El UR5 es un manipulador de 6 ejes con una carga máxima de trabajo de 5 kg. Sus articulaciones tienen un rango de movimiento ±360° y pueden moverse a una admirable velocidad de 180°/s. Con un peso de 18,4 kg y fabricado en aluminio y plástico ABS no es especialmente pesado. El alcance que tiene es de 0,85 m.

Ilustración 2.8 - MOTOMAN SIA5D

YASKAWA MOTOMAN SIA5D

Según la empresa el SIA5D es un manipulador de 7 ejes de alto rendimiento con sorprendente libertad de movimiento y la capacidad de trabajar en áreas muy estrechas. Es capaz de reorientar los codos sin que el efector final se vea alterado. Tiene una carga máxima de 5 kg, un alcance vertical de 1,007 m y de 0,559 m horizontal. Por sus características este modelo tiene un enfoque más industrial.

Capítulo 3

LINUX, EL SISTEMA OPERATIVO DE TODOS Y PARA TODOS

3. Linux, el Sistema Operativo de todos y para todos

En este capítulo se pretende dar una visión general de lo que es el sistema operativo de libre distribución Linux, las ventajas y desventajas del mismo, y cómo se ha usado y por qué en este proyecto.

3.1. Orígenes y creación

La historia de Linux comenzó mucho antes de lo que la mayoría de gente piensa, ya que en 1969, Ken Thompson, de AT&T Bell Laboratories, desarrolló el sistema operativo Unix, adaptándolo a las necesidades de un entorno de investigación, sin saber la importancia que llegaría a tener su trabajo. Un año después Dennis Ritchie (creador del lenguaje de programación C), colaboró con Ken Thompson para pasar el código del sistema Unix a C. Lo que convierto a Unix en un sistema operativo transportable.

Unix creció gradualmente hasta convertirse en un producto de software estándar, distribuido por muchos vendedores tales como Novell e IBM. Sus primeras versiones fueron distribuidas de forma gratuita a los departamentos científicos de informática de muchas universidades de renombre.

En 1972, los laboratorios Bell empezaron a emitir versiones oficiales de Unix y a otorgar licencias del sistema a distintos usuarios. En 1975, Berkeley lanzó su propia versión de Unix (BSD). Esta versión de Unix se convirtió en la principal competidora de la versión de los laboratorios Bell de AT&T, pero no era la única ya que en 1980, Microsoft desarrolló una versión de Unix para PC llamada Xenix.

En 1991 esta organización desarrolló el SistemaV versión 4, que incorporaba casi todas las características que se encuentran en el SistemaV versión 3, BSD versión 4.3, SunOS y Xenix. Como respuesta a esta nueva versión, varias compañías, tales como IBM y Hewlett Packard, establecieron la Open Software Foundation (OSF) para crear su propia versión estándar del Unix.

Debido a la proliferación de versiones de Unix en las décadas anteriores, el Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) desarrollo un estándar del Unix independiente para el American National Standards Institute (ANSI). Este nuevo estándar ANSI del Unix se llama Portable Operating System Interface for Computer Environments (POSIX). Este estándar define una norma universal a la cual se deben adherir todas las versiones de Unix.

En esa época, los estudiantes utilizaban un programa llamado Minix, que incorporaba diferentes características de Unix. Minix fue creado por el profesor Andrew Tanenbaum. Director del Departamento de Sistemas de la Universidad de Vrije, Amsterdam. Profesor de Arquitectura de Ordenadores y Sistemas Operativos. Licenciado en el MIT, y doctorado en la Universidad de Berkeley, California. En 1992 participó en el debate con Linux sobre la idea de este utilizar un núcleo monolítico en vez de los basados en un micro núcleo que Tanenbaum creía que serían la base de los sistemas operativos futuros.

Era el año 1991 y Linus Torvalds, que en aquel entonces era un estudiante de informática de la Universidad de Helsinki, empezó a programar las primeras líneas de código de un sistema operativo(finalmente llamado LINUX) como una afición y sin poderse imaginar la gran repercusión que traería.

Hubo una primera versión no oficial de Linux 0.01, pero esta solo incluía el comienzo del núcleo, estaba escrita en lenguaje ensamblador y asumía que uno tenía acceso a un sistema Minix para su compilación.

El 5 de octubre de 1991, Linus anuncio la primera versión oficial de Linux (versión 0.02). Con esta versión Linus pudo ejecutar Bash (GNU Bourne Again Shell) y gcc (El compilador GNU de C). Desde aquel entonces se han hecho muchísimas versiones con ayuda de programadores de todo el mundo.

Linux es un sistema operativo compatible con Unix, sus dos características principales y que los diferencian del resto de los sistemas operativos que encontramos en el mercado son:

1. Es software libre, esto significa que no tenemos que pagar por el uso del mismo.

2. El sistema viene acompañado del código fuente (el sistema lo forman el núcleo del sistema (kernel) más un gran número de librerías que hacen posible su utilización).

Las plataformas en las que en un principio se empezó utilizar Linux son: Pentium, Pentium Pro, Pentium II/III/IV, Amiga y Atari, también existen versiones para su utilización en otras plataformas, como Alpha, ARM, MIPS, PowerPC y SPARC. Actualmente cualquier sistema soporta Linux, incluso existen smartphones que ya lo utilizan.

En los últimos tiempos, ciertas casas de software comercial han empezado a distribuir sus productos para Linux y la presencia del mismo en empresas aumenta rápidamente por la excelente relación calidad-precio que se consigue con Linux.

3.2. Distribuciones

La filosofía del software libre ha sido bien recibida por muchas compañías y comunidades de programadores, lo que ha dado como resultado las distribuciones de Linux, que son la forma habitual de hacer uso de dicho sistema operativo. Cada distribución personaliza el entorno visual y algunas de las funciones del sistema, dependiendo de para qué público esté más enfocado, ya sean usuarios comunes, empresas, desarrolladores…

A continuación se presentan algunas de las distribuciones más importantes de Linux:

El creador de CentOS fue Lance Davis. Es una distribución de Linux basada en código fuente libre disponible de Red Hat Enterprise Linux. Cada versión de CentOS es mantenida durante 7 años con actualizaciones de seguridad. Las versiones nuevas son liberadas cada 2 años y actualizadas regularmente para dar soporte al hardware nuevo. La primera versión de CentOS fue lanzada en mayo de 2004 y la última versión estable es la CentOS 6.2, lanzada en diciembre de 2011. CentOS se utiliza básicamente para la administración de sistemas.

	Es una distribución Linux creada por Red Hat, que fue una de las más populares en los entornos de usuarios domésticos. La versión 1.0 fue presentada el 3 de noviembre de 1994. Originalmente Red Hat Linux fue desarrollado exclusivamente dentro de Red Hat, con la sola realimentación de informes de usuarios que recuperaban de fallos y contribuciones a los paquetes de software incluidos; y no contribuciones a la distribución como tal. Esto cambió tardíamente en el 2003 cuando Red Hat Linux se fusionó con el Proyecto Fedora Linux orientado a la comunidad de usuarios. El nuevo plan es extraer el código base de Fedora para crear nuevas distribuciones de Red Hat Enterprise Linux. Actualmente la versión gratuita es Fedora.
	Fedora es un sistema operativo para uso doméstico, que se caracteriza por su velocidad. Es desarrollado por una comunidad de usuarios alrededor de todo el mundo. Es gratuito y libre, tanto para utilizarlo como para compartirlo o para conocer su funcionamiento. La versión actual para descargar es: Fedora 20
	Distribución francesa basada en RedHat y muy accesible para quienes se inician. Tras la fusión con Conectiva cambió su nombre por Mandriva. Esta distribución de Linux, dispone de varias versiones para usos distintos. Son las siguientes: - Mandriva One 2011 - Powpack 2011 - Enterprise Server 5.2
	Debian es otra de las distribuciones más utilizadas de Linux. Se está trabajando para ofrecer Debian con otros núcleos, en especial con el Hurd. El Hurd es una colección de servidores que se ejecutan sobre un micro-núcleo (como Mach) para implementar las distintas funcionalidades. El Hurd es software libre producido por el proyecto GNU. Debian siempre mantiene al menos tres versiones en mantenimiento activo: estable, en pruebas e inestable. La versión estable actual de Debian es la 7.6. La publicación en pruebas (testing) contiene paquetes que aún no han sido aceptados en la rama estable, pero están a la espera de ello. La principal ventaja de usar esta publicación es que tiene versiones más recientes del software. La publicación inestable llamada sid y es donde tiene lugar el desarrollo activo de Debian. Generalmente, esta publicación es la que usan los desarrolladores y otros que quieren estar a la última.

Ubuntu es una distribución Linux que ofrece un sistema operativo orientado a ordenadores de escritorio proporcionando también soporte para servidores.
Basada en Debian GNU/Linux, Ubuntu se centra en la facilidad de uso, los lanzamientos regulares (cada 6 meses) y la facilidad en la instalación. Ubuntu es patrocinado por Canonical Ltd., una empresa privada fundada y financiada por el empresario sudafricano Mark Shuttleworth. Cabe destacar que Canonical Ltd., ofrece multitud de aplicaciones para descargar.
El nombre de la distribución proviene del concepto zulú y xhosa de ubuntu, que significa "humanidad hacia otros" o "yo soy porque nosotros somos". Ubuntu es un movimiento sudafricano encabezado por el obispo Desmond Tutu, quien ganó el Premio Nobel de la Paz en 1984 por sus luchas en contra del Apartheid en Sudáfrica. El sudafricano Mark Shuttleworth, mecenas del proyecto, se encontraba muy familiarizado con la corriente. Tras ver similitudes entre los ideales de los proyectos GNU, Debian y en general con el movimiento del software libre, decidió aprovechar la ocasión para difundir los ideales de Ubuntu. El eslogan de Ubuntu – "Linux para seres humanos" (en inglés "Linux for Human Beings") – resume una de sus metas principales: hacer de Linux un sistema operativo más accesible y fácil de usar.

3.3. Comandos básicos para el manejo de Linux

Como este proyecto, a la vez que divulgar los avances, investigaciones y desarrollos que se han llevado a cabo, también pretende ser una especie de manual para futuros trabajos que se desarrollen en el departamento, es fundamental que se haga un breve repaso de los comandos básicos de trabajo por terminal.

Es de gran importancia tener claros algunos de estos comandos y su estructura, pues en el capítulo dedicado a la configuración del sistema, instalación de paquetes y puesta en marcha del robot muchos de ellos serán usados y no se explicarán en dicho momento de forma explícita.

Información del sistema

1. **arch**: mostrar la arquitectura de la máquina (1).
2. **uname -m**: mostrar la arquitectura de la máquina (2).
3. **uname -r**: mostrar la versión del kernel usado.
4. **dmidecode -q**: mostrar los componentes (hardware) del sistema.
5. **hdparm -i /dev/hda**: mostrar las características de un disco duro.
6. **hdparm -tT /dev/sda**: realizar prueba de lectura en un disco duro.

7. **cat /proc/cpuinfo**: mostrar información de la CPU.
8. **cat /proc/interrupts**: mostrar las interrupciones.
9. **cat /proc/meminfo**: verificar el uso de memoria.
10. **cat /proc/swaps**: mostrar ficheros swap.
11. **cat /proc/version**: mostrar la versión del kernel.
12. **cat /proc/net/dev**: mostrar adaptadores de red y estadísticas.
13. **cat /proc/mounts**: mostrar el sistema de ficheros montado.
14. **lspci -tv**: mostrar los dispositivos PCI.
15. **lsusb -tv**: mostrar los dispositivos USB.
16. **date**: mostrar la fecha del sistema.
17. **cal 2011**: mostrar el almanaque de 2011.
18. **cal 07 2011**: mostrar el almanaque para el mes julio de 2011.
19. **date 041217002011.00**: colocar (declarar, ajustar) fecha y hora.
20. **clock -w**: guardar los cambios de fecha en la BIOS.

Apagar (reiniciar o cerrar sesión)

1. **shutdown -h now**: apagar el sistema (1).
2. **init 0**: apagar el sistema (2).
3. **telinit 0**: apagar el sistema (3).
4. **halt**: apagar el sistema (4).
5. **shutdown -h hours:minutes &**: apagado planificado del sistema.
6. **shutdown -c**: cancelar un apagado planificado del sistema.
7. **shutdown -r now**: reiniciar (1).
8. **reboot**: reiniciar (2).
9. **logout**: cerrar sesión.

Archivos y directorios

1. **cd /home**: entrar en el directorio “home”.
2. **cd ..**: retroceder un nivel.
3. **cd ../..**: retroceder 2 niveles.
4. **cd:** ir al directorio raíz.
5. **cd ~user1**: ir al directorio user1.
6. **cd -**: ir (regresar) al directorio anterior.
7. **pwd**: mostrar el camino del directorio de trabajo.
8. **ls**: ver los ficheros de un directorio.
9. **ls -F**: ver los ficheros de un directorio.
10. **ls -l**: mostrar los detalles de ficheros y carpetas de un directorio.
11. **ls -a**: mostrar los ficheros ocultos.
12. **ls *[0-9]***: mostrar los ficheros y carpetas que contienen números.
13. **tree**: mostrar los ficheros y carpetas en forma de árbol comenzando por la raíz.(1)
14. **lstree**: mostrar los ficheros y carpetas en forma de árbol comenzando por la raíz.(2)

15. **mkdir dir1**: crear una carpeta o directorio con nombre 'dir1'.
16. **mkdir dir1 dir2**: crear dos carpetas o directorios simultáneamente (Crear dos directorios a la vez).
17. **mkdir -p /tmp/dir1/dir2**: crear un árbol de directorios.
18. **rm -f file1**: borrar el fichero llamado 'file1'.
19. **rmdir dir1**: borrar la carpeta llamada 'dir1'.
20. **rm -rf dir1**: eliminar una carpeta llamada 'dir1' con su contenido de forma recursiva. (Si lo borro recursivo estoy diciendo que es con su contenido).
21. **rm -rf dir1 dir2**: borrar dos carpetas (directorios) con su contenido de forma recursiva.
22. **mv dir1 new_dir**: renombrar o mover un fichero o carpeta (directorio).
23. **cp file1**: copiar un fichero.
24. **cp file1 file2**: copiar dos ficheros al unísono.
25. **cp dir /* .**: copiar todos los ficheros de un directorio dentro del directorio de trabajo actual.
26. **cp -a /tmp/dir1 .**: copiar un directorio dentro del directorio actual de trabajo.
27. **cp -a dir1**: copiar un directorio.
28. **cp -a dir1 dir2**: copiar dos directorios al unísono.
29. **ln -s file1 lnk1**: crear un enlace simbólico al fichero o directorio.
30. **ln file1 lnk1**: crear un enlace físico al fichero o directorio.
31. **touch -t 0712250000 file1**: modificar el tiempo real (tiempo de creación) de un fichero o directorio.
32. **file file1**: salida (volcado en pantalla) del tipo mime de un fichero texto.
33. **iconv -l**: listas de cifrados conocidos.
34. **iconv -f fromEncoding -t toEncoding inputFile > outputFile**: crea una nueva forma del fichero de entrada asumiendo que está codificado en fromEncoding y convirtiéndolo a ToEncoding.
35. **find . -maxdepth 1 -name *.jpg -print -exec convert "{}" -resize 80×60 "thumbs/{}" \;**: agrupar ficheros redimensionados en el directorio actual y enviarlos a directorios en vistas de miniaturas (requiere convertir desde ImagemagicK).

Encontrar archivos

1. **find / -name file1**: buscar fichero y directorio a partir de la raíz del sistema.
2. **find / -user user1**: buscar ficheros y directorios pertenecientes al usuario 'user1'.
3. **find /home/user1 -name *.bin**: buscar ficheros con extensión '. bin' dentro del directorio '/ home/user1'.
4. **find /usr/bin -type f -atime +100**: buscar ficheros binarios no usados en los últimos 100 días.
5. **find /usr/bin -type f -mtime -10**: buscar ficheros creados o cambiados dentro de los últimos 10 días.

6. **find / -name *.rpm -exec chmod 755 '{}' \;**: buscar ficheros con extensión '.rpm' y modificar permisos.
7. **find / -xdev -name *.rpm**: Buscar ficheros con extensión '.rpm' ignorando los dispositivos removibles como cdrom, pen-drive, etc.…
8. **locate *.ps**: encuentra ficheros con extensión '.ps' ejecutados primeramente con el command 'updatedb'.
9. **whereis halt**: mostrar la ubicación de un fichero binario, de ayuda o fuente. En este caso pregunta dónde está el comando 'halt'.
10. **which halt**: mostrar la senda completa (el camino completo) a un binario / ejecutable.

Montando un sistema de ficheros

1. **mount /dev/hda2 /mnt/hda2**: montar un disco llamado hda2. Verifique primero la existencia del directorio '/ mnt/hda2'; si no está, debe crearlo.
2. **umount /dev/hda2**: desmontar un disco llamado hda2. Salir primero desde el punto '/ mnt/hda2.
3. **fuser -km /mnt/hda2**: forzar el desmontaje cuando el dispositivo está ocupado.
4. **umount -n /mnt/hda2**: correr el desmontaje sin leer el fichero /etc/mtab. Útil cuando el fichero es de solo lectura o el disco duro está lleno.
5. **mount /dev/fd0 /mnt/floppy**: montar un disco flexible (floppy).
6. **mount /dev/cdrom /mnt/cdrom**: montar un cdrom / dvdrom.
7. **mount /dev/hdc /mnt/cdrecorder**: montar un cd regrabable o un dvdrom.
8. **mount /dev/hdb /mnt/cdrecorder**: montar un cd regrabable / dvdrom (un dvd).
9. **mount -o loop file.iso /mnt/cdrom**: montar un fichero o una imagen iso.
10. **mount -t vfat /dev/hda5 /mnt/hda5**: montar un sistema de ficheros FAT32.
11. **mount /dev/sda1 /mnt/usbdisk**: montar un usb pen-drive o una memoria (sin especificar el tipo de sistema de ficheros).

Espacio de disco

1. **df -h**: mostrar una lista de las particiones montadas.
2. **ls -lSr |more**: mostrar el tamaño de los ficheros y directorios ordenados por tamaño.
3. **du -sh dir1**: Estimar el espacio usado por el directorio 'dir1'.
4. **du -sk * | sort -rn**: mostrar el tamaño de los ficheros y directorios ordenados por tamaño.
5. **rpm -q -a –qf '%10{SIZE}t%{NAME}n' | sort -k1,1n**: mostrar el espacio usado por los paquetes rpm instalados organizados por tamaño (Fedora, Redhat y otros).

6. **dpkg-query -W -f='${Installed-Size;10}t${Package}n' | sort -k1,1n**: mostrar el espacio usado por los paquetes instalados, organizados por tamaño (Ubuntu, Debian y otros).

Usuarios y grupos

1. **groupadd nombre_del_grupo**: crear un nuevo grupo.
2. **groupdel nombre_del_grupo**: borrar un grupo.
3. **groupmod -n nuevo_nombre_del_grupo viejo_nombre_del_grupo**: renombrar un grupo.
4. **useradd -c "Name Surname " -g admin -d /home/user1 -s /bin/bash user1**: Crear un nuevo usuario perteneciente al grupo "admin".
5. **useradd user1**: crear un nuevo usuario.
6. **userdel -r user1**: borrar un usuario ('-r' elimina el directorio Home).
7. **usermod -c "User FTP" -g system -d /ftp/user1 -s /bin/nologin user1**: cambiar los atributos del usuario.
8. **passwd**: cambiar contraseña.
9. **passwd user1**: cambiar la contraseña de un usuario (solamente por root).
10. **chage -E 2011-12-31 user1**: colocar un plazo para la contraseña del usuario. En este caso dice que la clave expira el 31 de diciembre de 2011.
11. **pwck**: chequear la sintaxis correcta el formato de fichero de '/etc/passwd' y la existencia de usuarios.
12. **grpck**: chequear la sintaxis correcta y el formato del fichero '/etc/group' y la existencia de grupos.
13. **newgrp group_name**: registra a un nuevo grupo para cambiar el grupo predeterminado de los ficheros creados recientemente.

Permisos en ficheros (usar "+" para colocar permisos y "-" para eliminar)

1. **ls -lh**: Mostrar permisos.
2. **ls /tmp | pr -T5 -W$COLUMNS**: dividir la terminal en 5 columnas.
3. **chmod ugo+rwx directory1**: colocar permisos de lectura ®, escritura (w) y ejecución(x) al propietario (u), al grupo (g) y a otros (o) sobre el directorio 'directory1'.
4. **chmod go-rwx directory1**: quitar permiso de lectura ®, escritura (w) y (x) ejecución al grupo (g) y otros (o) sobre el directorio 'directory1'.
5. **chown user1 file1**: cambiar el dueño de un fichero.
6. **chown -R user1 directory1**: cambiar el propietario de un directorio y de todos los ficheros y directorios contenidos dentro.
7. **chgrp group1 file1**: cambiar grupo de ficheros.
8. **chown user1:group1 file1**: cambiar usuario y el grupo propietario de un fichero.
9. **find / -perm -u+s**: visualizar todos los ficheros del sistema con SUID configurado.

10. **chmod u+s /bin/file1**: colocar el bit SUID en un fichero binario. El usuario que corriendo ese fichero adquiere los mismos privilegios como dueño.
11. **chmod u-s /bin/file1**: deshabilitar el bit SUID en un fichero binario.
12. **chmod g+s /home/public**: colocar un bit SGID en un directorio –similar al SUID pero por directorio.
13. **chmod g-s /home/public**: desabilitar un bit SGID en un directorio.
14. **chmod o+t /home/public**: colocar un bit STIKY en un directorio. Permite el borrado de ficheros solamente a los dueños legítimos.
15. **chmod o-t /home/public**: desabilitar un bit STIKY en un directorio.

Atributos especiales en ficheros (usar "+" para colocar permisos y "-" para eliminar)

1. **chattr +a file1**: permite escribir abriendo un fichero solamente modo append.
2. **chattr +c file1**: permite que un fichero sea comprimido / descomprimido automaticamente.
3. **chattr +d file1**: asegura que el programa ignore borrar los ficheros durante la copia de seguridad.
4. **chattr +i file1**: convierte el fichero en invariable, por lo que no puede ser eliminado, alterado, renombrado, ni enlazado.
5. **chattr +s file1**: permite que un fichero sea borrado de forma segura.
6. **chattr +S file1**: asegura que un fichero sea modificado, los cambios son escritos en modo synchronous como con sync.
7. **chattr +u file1**: te permite recuperar el contenido de un fichero aún si este está cancelado.
8. **lsattr**: mostrar atributos especiales.

Archivos y ficheros comprimidos

1. **bunzip2 file1.bz2**: descomprime in fichero llamado 'file1.bz2'.
2. **bzip2 file1**: comprime un fichero llamado 'file1'.
3. **gunzip file1.gz**: descomprime un fichero llamado 'file1.gz'.
4. **gzip file1**: comprime un fichero llamado 'file1'.
5. **gzip -9 file1**: comprime con compresión máxima.
6. **rar a file1.rar test_file**: crear un fichero rar llamado 'file1.rar'.
7. **rar a file1.rar file1 file2 dir1**: comprimir 'file1', 'file2' y 'dir1' simultáneamente.
8. **rar x file1.rar**: descomprimir archivo rar.
9. **unrar x file1.rar**: descomprimir archivo rar.
10. **tar -cvf archive.tar file1**: crear un tarball descomprimido.
11. **tar -cvf archive.tar file1 file2 dir1**: crear un archivo conteniendo 'file1', 'file2' y'dir1'.
12. **tar -tf archive.tar**: mostrar los contenidos de un archivo.
13. **tar -xvf archive.tar**: extraer un tarball.

14. **tar -xvf archive.tar -C /tmp**: extraer un tarball en / tmp.
15. **tar -cvfj archive.tar.bz2 dir1**: crear un tarball comprimido dentro de bzip2.
16. **tar -xvfj archive.tar.bz2**: descomprimir un archivo tar comprimido en bzip2
17. **tar -cvfz archive.tar.gz dir1**: crear un tarball comprimido en gzip.
18. **tar -xvfz archive.tar.gz**: descomprimir un archive tar comprimido en gzip.
19. **zip file1.zip file1**: crear un archivo comprimido en zip.
20. **zip -r file1.zip file1 file2 dir1**: comprimir, en zip, varios archivos y directorios de forma simultánea.
21. **unzip file1.zip**: descomprimir un archivo zip.

Paquetes Deb (Debian, Ubuntu y derivados)

1. **dpkg -i package.deb**: instalar / actualizar un paquete deb.
2. **dpkg -r package_name**: eliminar un paquete deb del sistema.
3. **dpkg -l**: mostrar todos los paquetes deb instalados en el sistema.
4. **dpkg -l | grep httpd**: mostrar todos los paquetes deb con el nombre "httpd"
5. **dpkg -s package_name**: obtener información en un paquete específico instalado en el sistema.
6. **dpkg -L package_name**: mostar lista de ficheros dados por un paquete instalado en el sistema.
7. **dpkg –contents package.deb**: mostrar lista de ficheros dados por un paquete no instalado todavía.
8. **dpkg -S /bin/ping**: verificar cuál paquete pertenece a un fichero dado.

Actualizador de paquetes APT (Debian, Ubuntu y derivados)

1. **apt-get install package_name**: instalar / actualizar un paquete deb.
2. **apt-cdrom install package_name**: instalar / actualizar un paquete deb desde un cdrom.
3. **apt-get update**: actualizar la lista de paquetes.
4. **apt-get upgrade**: actualizar todos los paquetes instalados.
5. **apt-get remove package_name**: eliminar un paquete deb del sistema.
6. **apt-get check**: verificar la correcta resolución de las dependencias.
7. **apt-get clean**: limpiar cache desde los paquetes descargados.
8. **apt-cache search searched-package**: retorna lista de paquetes que corresponde a la serie «paquetes buscados».

Ver el contenido de un fichero

1. **cat file1**: ver los contenidos de un fichero comenzando desde la primera hilera.
2. **tac file1**: ver los contenidos de un fichero comenzando desde la última línea.
3. **more file1**: ver el contenido a lo largo de un fichero.
4. **less file1**: parecido al commando 'more' pero permite salvar el movimiento en el fichero así como el movimiento hacia atrás.

5. **head -2 file1**: ver las dos primeras líneas de un fichero.
6. **tail -2 file1**: ver las dos últimas líneas de un fichero.
7. **tail -f /var/log/messages**: ver en tiempo real qué ha sido añadido al fichero.

Manipulación de texto

1. **cat file1 file2 .. | command <> file1_in.txt_or_file1_out.txt**: sintaxis general para la manipulación de texto utilizando PIPE, STDIN y STDOUT.
2. **cat file1 | command(sed, grep, awk, grep, etc...) > result.txt**: sintaxis general para manipular un texto de un fichero y escribir el resultado en un fichero nuevo.
3. **cat file1 | command(sed, grep, awk, grep, etc...) » result.txt**: sintaxis general para manipular un texto de un fichero y añadir resultado en un fichero existente.
4. **grep Aug /var/log/messages**: buscar palabras "Aug" en el fichero '/var/log/messages'.
5. **grep ^Aug /var/log/messages**: buscar palabras que comienzan con "Aug" en fichero '/var/log/messages'
6. **grep [0-9] /var/log/messages**: seleccionar todas las líneas del fichero '/var/log/messages' que contienen números.
7. **grep Aug -R /var/log/***: buscar la cadena "Aug" en el directorio '/var/log' y debajo.
8. **sed 's/stringa1/stringa2/g' example.txt**: reubicar "string1" con "string2" en ejemplo.txt
9. **sed '/^$/d' example.txt**: eliminar todas las líneas en blanco desde el ejemplo.txt
10. **sed '/ *#/d; /^$/d' example.txt**: eliminar comentarios y líneas en blanco de ejemplo.txt
11. **echo 'esempio' | tr '[:lower:]' '[:upper:]'**: convertir minúsculas en mayúsculas.
12. **sed -e '1d' result.txt**: elimina la primera línea del fichero ejemplo.txt
13. **sed -n '/stringa1/p'**: visualizar solamente las líneas que contienen la palabra "string1".

Establecer carácter y conversión de ficheros

1. **dos2unix filedos.txt fileunix.txt**: convertir un formato de fichero texto desde MSDOS a UNIX.
2. **unix2dos fileunix.txt filedos.txt**: convertir un formato de fichero de texto desde UNIX a MSDOS.
3. **recode ..HTML < page.txt > page.html**: convertir un fichero de texto en html.
4. **recode -l | more**: mostrar todas las conversiones de formato disponibles.

Análisis del sistema de ficheros

1. **badblocks -v /dev/hda1**: Chequear los bloques defectuosos en el disco hda1.

2. **fsck /dev/hda1**: reparar / chequear la integridad del fichero del sistema Linux en el disco hda1.
3. **fsck.ext2 /dev/hda1**: reparar / chequear la integridad del fichero del sistema ext 2 en el disco hda1.
4. **e2fsck /dev/hda1**: reparar / chequear la integridad del fichero del sistema ext 2 en el disco hda1.
5. **e2fsck -j /dev/hda1**: reparar / chequear la integridad del fichero del sistema ext 3 en el disco hda1.
6. **fsck.ext3 /dev/hda1**: reparar / chequear la integridad del fichero del sistema ext 3 en el disco hda1.
7. **fsck.vfat /dev/hda1**: reparar / chequear la integridad del fichero sistema fat en el disco hda1.
8. **fsck.msdos /dev/hda1**: reparar / chequear la integridad de un fichero del sistema dos en el disco hda1.
9. **dosfsck /dev/hda1**: reparar / chequear la integridad de un fichero del sistema dos en el disco hda1.

Formatear un sistema de ficheros

1. **mkfs /dev/hda1**: crear un fichero de sistema tipo Linux en la partición hda1.
2. **mke2fs /dev/hda1**: crear un fichero de sistema tipo Linux ext 2 en hda1.
3. **mke2fs -j /dev/hda1**: crear un fichero de sistema tipo Linux ext3 (periódico) en la partición hda1.
4. **mkfs -t vfat 32 -F /dev/hda1**: crear un fichero de sistema FAT32 en hda1.
5. **fdformat -n /dev/fd0**: formatear un disco flooply.
6. **mkswap /dev/hda3**: crear un fichero de sistema swap.

Trabajo con SWAP

1. **mkswap /dev/hda3**: crear fichero de sistema swap.
2. **swapon /dev/hda3**: activando una nueva partición swap.
3. **swapon /dev/hda2 /dev/hdb3**: activar dos particiones swap.

Salvas (Backup)

1. **dump -0aj -f /tmp/home0.bak /home**: hacer una salva completa del directorio '/home'.
2. **dump -1aj -f /tmp/home0.bak /home**: hacer una salva incremental del directorio '/home'.
3. **restore -if /tmp/home0.bak**: restaurando una salva interactivamente.
4. **rsync -rogpav –delete /home /tmp**: sincronización entre directorios.
5. **rsync -rogpav -e ssh –delete /home ip_address:/tmp**: rsync a través del túnelSSH.

6. **rsync -az -e ssh –delete ip_addr:/home/public /home/local**: sincronizar un directorio local con un directorio remoto a través de ssh y de compresión.
7. **rsync -az -e ssh –delete /home/local ip_addr:/home/public**: sincronizar un directorio remoto con un directorio local a través de ssh y de compresión.
8. **dd bs=1M if=/dev/hda | gzip | ssh user@ip_addr 'dd of=hda.gz'**: hacer una salva de un disco duro en un host remoto a través de ssh.
9. **dd if=/dev/sda of=/tmp/file1**: salvar el contenido de un disco duro a un fichero. (En este caso el disco duro es "sda" y el fichero "file1").
10. **tar -Puf backup.tar /home/user**: hacer una salva incremental del directorio '/home/user'.
11. **(cd /tmp/local/ && tar c .) | ssh -C user@ip_addr 'cd /home/share/ && tar x -p'**: copiar el contenido de un directorio en un directorio remoto a través de ssh.
12. **(tar c /home) | ssh -C user@ip_addr 'cd /home/backup-home && tar x -p'**: copiar un directorio local en un directorio remoto a través de ssh.
13. **tar cf – . | (cd /tmp/backup ; tar xf –)**: copia local conservando las licencias y enlaces desde un directorio a otro.
14. **find /home/user1 -name '*.txt' | xargs cp -av –target-directory=/home/backup/ –parents**: encontrar y copiar todos los ficheros con extensión '.txt' de un directorio a otro.
15. **find /var/log -name '*.log' | tar cv –files-from=- | bzip2 > log.tar.bz2**: encontrar todos los ficheros con extensión '.log' y hacer un archivo bzip.
16. **dd if=/dev/hda of=/dev/fd0 bs=512 count=1**: hacer una copia del MRB (Master Boot Record) a un disco floppy.
17. **dd if=/dev/fd0 of=/dev/hda bs=512 count=1**: restaurar la copia del MBR (Master Boot Record) salvada en un floppy.

CD-ROM

1. **cdrecord -v gracetime=2 dev=/dev/cdrom -eject blank=fast -force**: limpiar o borrar un cd regrabable.
2. **mkisofs /dev/cdrom > cd.iso**: crear una imagen iso de cdrom en disco.
3. **mkisofs /dev/cdrom | gzip > cd_iso.gz**: crear una imagen comprimida iso de cdrom en disco.
4. **mkisofs -J -allow-leading-dots -R -V "Label CD" -iso-level 4 -o ./cd.iso data_cd**: crear una imagen iso de un directorio.
5. **cdrecord -v dev=/dev/cdrom cd.iso**: quemar una imagen iso.
6. **gzip -dc cd_iso.gz | cdrecord dev=/dev/cdrom -**: quemar una imagen iso comprimida.
7. **mount -o loop cd.iso /mnt/iso**: montar una imagen iso.
8. **cd-paranoia -B**: llevar canciones de un cd a ficheros wav.
9. **cd-paranoia – "-3"**: llevar las 3 primeras canciones de un cd a ficheros wav.

10. **cdrecord –scanbus**: escanear bus para identificar el canal scsi.
11. **dd if=/dev/hdc | md5sum**: hacer funcionar un md5sum en un dispositivo, como un CD.

Trabajo con la red (LAN y Wi-Fi)

1. **ifconfig eth0**: mostrar la configuración de una tarjeta de red Ethernet.
2. **ifup eth0**: activar una interface 'eth0'.
3. **ifdown eth0**: deshabilitar una interface 'eth0'.
4. **ifconfig eth0 192.168.1.1 netmask 255.255.255.0**: configurar una dirección IP.
5. **ifconfig eth0 promisc**: configurar 'eth0'en modo común para obtener los paquetes (sniffing).
6. **dhclient eth0**: activar la interface 'eth0' en modo dhcp.
7. **route -n**: mostrar mesa de recorrido.
8. **route add -net 0/0 gw IP_Gateway**: configurar entrada predeterminada.
9. **route add -net 192.168.0.0 netmask 255.255.0.0 gw 192.168.1.1**: configurar ruta estática para buscar la red '192.168.0.0/16'.
10. **route del 0/0 gw IP_gateway**: eliminar la ruta estática.
11. **echo "1" > /proc/sys/net/ipv4/ip_forward**: activar el recorrido ip.
12. **hostname**: mostrar el nombre del host del sistema.
13. **host www.example.com**: buscar el nombre del host para resolver el nombre a una dirección ip(1).
14. **nslookup www.example.com**: buscar el nombre del host para resolver el nombre a una direccióm ip y viceversa(2).
15. **ip link show**: mostar el estado de enlace de todas las interfaces.
16. **mii-tool eth0**: mostar el estado de enlace de 'eth0'.
17. **ethtool eth0**: mostrar las estadísticas de tarjeta de red 'eth0'.
18. **netstat -tup**: mostrar todas las conexiones de red activas y sus PID.
19. **netstat -tupl**: mostrar todos los servicios de escucha de red en el sistema y sus PID.
20. **tcpdump tcp port 80**: mostrar todo el tráfico HTTP.
21. **iwlist scan**: mostrar las redes inalámbricas.
22. **iwconfig eth1**: mostrar la configuración de una tarjeta de red inalámbrica.
23. **whois www.example.com**: buscar en base de datos Whois.

Tablas IP (cortafuegos)

1. **iptables -t filter -L**: mostrar todas las cadenas de la tabla de filtro.
2. **iptables -t nat -L**: mostrar todas las cadenas de la tabla nat.
3. **iptables -t filter -F**: limpiar todas las reglas de la tabla de filtro.
4. **iptables -t nat -F**: limpiar todas las reglas de la tabla nat.
5. **iptables -t filter -X**: borrar cualquier cadena creada por el usuario.
6. **iptables -t filter -A INPUT -p tcp –dport telnet -j ACCEPT**: permitir las conexiones telnet para entar.

7. **iptables -t filter -A OUTPUT -p tcp –dport http -j DROP**: bloquear las conexiones HTTP para salir.
8. **iptables -t filter -A FORWARD -p tcp –dport pop3 -j ACCEPT**: permitir las conexiones POP a una cadena delantera.
9. **iptables -t filter -A INPUT -j LOG –log-prefix "DROP INPUT"**: registrando una cadena de entrada.
10. **iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE**: configurar un PAT (Puerto de traducción de dirección) en eth0, ocultando los paquetes de salida forzada.
11. **iptables -t nat -A PREROUTING -d 192.168.0.1 -p tcp -m tcp –dport 22 -j DNAT –to-destination 10.0.0.2:22**: redireccionar los paquetes dirigidos de un host a otro.

Monitoreado y depurado

1. **top**: mostrar las tareas de linux usando la mayoría cpu.
2. **ps -eafw**: muestra las tareas Linux.
3. **ps -e -o pid,args –forest**: muestra las tareas Linux en un modo jerárquico.
4. **pstree**: mostrar un árbol sistema de procesos.
5. **kill -9 ID_Processo**: forzar el cierre de un proceso y terminarlo.
6. **kill -1 ID_Processo**: forzar un proceso para recargar la configuración.
7. **lsof -p $$**: mostrar una lista de ficheros abiertos por procesos.
8. **lsof /home/user1**: muestra una lista de ficheros abiertos en un camino dado del sistema.
9. **strace -c ls >/dev/null**: mostrar las llamadas del sistema hechas y recibidas por un proceso.
10. **strace -f -e open ls >/dev/null**: mostrar las llamadas a la biblioteca.
11. **watch -n1 'cat /proc/interrupts'**: mostrar interrupciones en tiempo real.
12. **last reboot**: mostrar historial de reinicio.
13. **lsmod**: mostrar el kernel cargado.
14. **free -m**: muestra el estado de la RAM en megabytes.
15. **smartctl -A /dev/hda**: monitorear la fiabilidad de un disco duro a través de SMART.
16. **smartctl -i /dev/hda**: chequear si SMART está activado en un disco duro.
17. **tail /var/log/dmesg**: mostrar eventos inherentes al proceso de carga del kernel.
18. **tail /var/log/messages**: mostrar los eventos del sistema.

Otros comandos útiles

1. **apropos ...keyword**: mostrar una lista de comandos que pertenecen a las palabras claves de un programa; son útiles cuando tú sabes qué hace tu programa, pero de sconoces el nombre del comando.

2. **man ping**: mostrar las páginas del manual on-line; por ejemplo, en un comando ping, usar la opción '-k' para encontrar cualquier comando relacionado.
3. **whatis ...keyword**: muestra la descripción de lo que hace el programa.
4. **mkbootdisk –device /dev/fd0 `uname -r`**: crear un floppy boteable.
5. **gpg -c file1**: codificar un fichero con guardia de seguridad GNU.
6. **gpg file1.gpg**: decodificar un fichero con Guardia de seguridad GNU.
7. **wget -r www.example.com**: descargar un sitio web completo.
8. **wget -c www.example.com/file.iso**: descargar un fichero con la posibilidad de parar la descargar y reanudar más tarde.
9. **echo 'wget -c www.example.com/files.iso' | at 09:00**: Comenzar una descarga a cualquier hora. En este caso empezaría a las 9 horas.
10. **ldd /usr/bin/ssh**: mostrar las bibliotecas compartidas requeridas por el programa ssh.
11. **alias hh='history'**: colocar un alias para un commando –hh= Historial.
12. **chsh**: cambiar el comando Shell.
13. **chsh –list-shells**: es un comando adecuado para saber si tienes que hacer remoto en otra terminal.
14. **who -a**: mostrar quien está registrado, e imprimir hora del último sistema de importación, procesos muertos, procesos de registro de sistema, procesos activos producidos por init, funcionamiento actual y últimos cambios del reloj del sistema.

3.4. El porqué de usar Linux en nuestro proyecto

Este proyecto se desarrolla íntegramente en entorno Linux, concretamente en la distribución Ubuntu 12.04 Precise. Esta versión, de las ofrecidas por Ubuntu, es la más extendida actualmente, pues pertenece a la modalidad LTS (Long Term Support), lo cual la hace la más estable y para la que más recursos disponibles existen.

El hecho de que se opte por Linux es que el middleware robótico ROS que usaremos para la gestión y comunicación del brazo y las cámaras corre sobre Ubunto (recomendado), aunque se encuentra disponible para otros sistemas operativos (Windows y Mac OS X), pero es en fase beta.

Existe una filosofía de software y código libre en torno a la robótica actualmente que permiten desde la comunidad de desarrolladores crear muchos proyectos de forma conjunta o apoyándose sobre los avances previos de otros.

3.4.1. Ventajas

Los sistemas operativos basados en Unix, como lo es Linux, son famosos por su robustez y estabilidad, gracias a su diseño modular de procesos que independiza de forma segura los diferentes programas haciendo que sea más difícil sufrir un bloqueo completo del sistema que obligue a reiniciar.

Otra ventaja es la rapidez y fluidez de este sistema operativo, así como los pocos recursos de hardware que necesita para funcionar. Esto hace posible que sea instalable en cualquier ordenador, incluso dispositivos empotrados (embebidos). Es de destacar que a diferencia de, por ejemplo Windows, un ordenador que corre Linux, con el paso del tiempo y del uso no reduce su rendimiento, por las características de arquitectura del propio sistema operativo.

Linux es multitarea y multiusuario, así como capaz de soportar de ejecución en tiempo real de forma nativa. Esto lo dota de una versatilidad y capacidades por encima de sus competidores. Además es capaz de correr casi cualquier aplicación compatible con Unix.

El diseño modular de Linux permite que el entorno gráfico de la interfaz de usuario sea un módulo más que se ejecuta en paralelo con los otros. Siendo así, soporta diferentes entornos gráficos, como pueden ser KDE, GNOME, XCFE…

También es un punto fuerte la orientación que este sistema tiene al desarrollo de proyectos e investigación, así como trabajos de la comunidad “DIY” (Do It Yourself).

Pero sin duda alguna, lo más importante y característico de este sistema operativo es que es totalmente gratuito y su código es libre, y existe una ingente cantidad de documentación gratuita, así como foros de ayuda y dudas. Cierto es que hay distribuciones de Linux que son de pago, como RedHat, pues ofrecen ciertos servicios de corte empresarial que requieren de un mantenimiento y unos desarrollos que sí hay que pagarlos. Pero otras, como Ubuntu, que es nuestro caso, son totalmente libres y abiertas.

3.4.2. Inconvenientes

El principal inconveniente que se les presenta a los usuarios de Linux es la complejidad de uso. Este sistema operativo está concebido para que todo se pueda hacer mediante comandos por consola. Aunque esto lo dota de gran potencia y versatilidad, la curva de aprendizaje de esta forma de trabajar, sobre todo para usuarios provenientes de Windows o MacOS, es bastante lenta. Éste se podría decir que es el principal hándicap que encuentran los usuarios al iniciarse en Linux, aunque las distribuciones de Ubuntu han avanzado mucho en ese aspecto y la gran mayoría de las tareas comunes pueden llevarse perfectamente a cabo sin tener que abrir un terminal. Se podría decir que el objetivo de convertirse en un sistema operativo masivo para el consumidor, como lo pueda ser Windows, está todavía un poco lejos de conseguirse.

Derivado de lo anteriormente comentado está la problemática que presenta la instalación de paquetes o programas. Sobre todo los utilizados para el desarrollo del presente proyecto. Requieren normalmente de numerosos comandos por consola, los cuales necesitan de dependencias y opciones que no son fácilmente utilizables. Esto quizá nos dé un poder y versatilidad muy alto, pero la documentación existente que la explica muchas veces es demasiado técnica y/o compleja, haciendo que no estemos seguros de si el comando que estamos usando realmente hará lo que queremos o nos hará meter la pata.

Aunque en defensa de Linux hay que decir que instrucciones por consola como `apt-get install` han mejorado notablemente la facilidad de instalación de los programas. A continuación se muestra a modo de comparativa la instalación de dos programas, uno en Linux y otro en Windows.

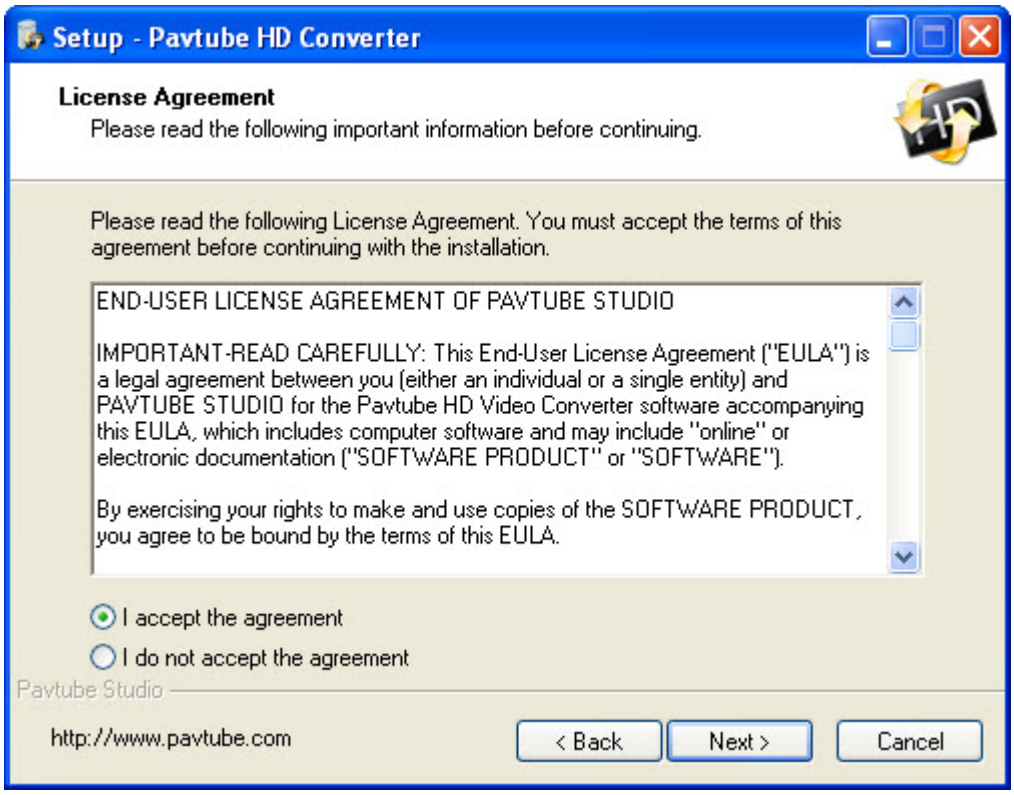


Ilustración 3.1 - Instalación de programa en Windows

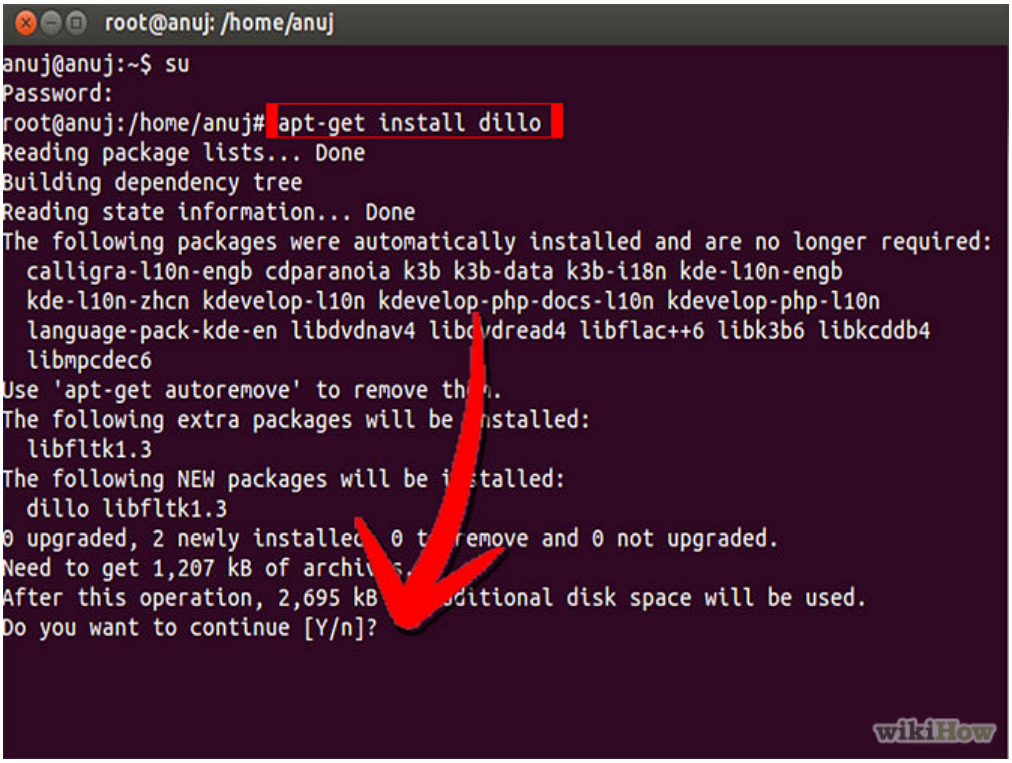


Ilustración 3.2 - Instalación de programa en Linux

Es posible también que cierto tipo de hardware no sea soportado si la comunidad o las fundaciones que respaldan Linux no han desarrollado los drivers pertinentes, o en su defecto los fabricantes de dicho software no han liberado licencias GPL para sus productos. Debido a esto la instalación de ciertos periféricos puede ser algo tediosa y frustrante, echando de menos el cómodo "instalar → siguiente → siguiente → finalizar" de Windows o Mac OS.

3.5. Instalación y configuración de Linux

En el *Anexo Nº1* se relata de forma detallada la manera de instalar Linux desde 0 en un PC y cómo ha de ser configurado en un principio para poder empezar a trabajar con él.

Capítulo 4

ROS – ROBOT OPERATING SYSTEM

4. ROS - Robot Operating System

Este capítulo se centrará en el núcleo de este proyecto, que es ROS. En él hablaremos de sus antecedentes y orígenes, así como de su arquitectura, funcionamiento, flujo de trabajo y demás conceptos relevantes para el entendimiento de este middleware robótico y su posterior uso.

4.1. Middlewares robóticos, definición y orígenes

En esta breve disertación acerca de los middleware robóticos nos apoyaremos en el trabajo hecho por Simone Ceriani y Martino Migliavacca, profesores del departamento de Electrónica, Información y Bioingeniería del Politécnico de Milán, en un informe interno que llevaron a cabo denominado "Advanced Methods of Information Technology for Autonomous Robotics".

Middleware es una adición relativamente nueva a los conceptos de la informática. Ganó popularidad en la década de 1980 como una solución para envolver sistema heredado e interconectarlo con nuevas aplicaciones.

Aunque su difusión es reciente, la introducción del término "middleware" se puede datar en 1968, cuando d'Agapeyeff lo utilizó por primera vez para identificar la parte del software que se puede compartir entre diferentes aplicaciones. En su idea original, d'Agapeyeff describe una pirámide inversa (Fig. 4.1) que describe una arquitectura de software primitivo en el que las aplicaciones de alto nivel se construyen utilizando un conjunto de rutinas de nivel medio (middleware). Esto, a su vez, se basa en un pequeño conjunto de rutinas de servicio.

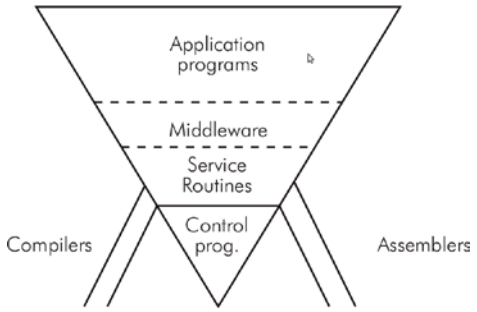


Ilustración 4.1 - Pirámide invertida de d'Agapeyeff

4.1.1. Definición de “middleware”

Un middleware es un software que conecta otros componentes de software o aplicaciones. En general, un middleware consiste en un conjunto de servicios que permite que varios procesos se ejecutan en una o más máquinas para interactuar. Esta tecnología se desarrolló para proporcionar la interoperabilidad necesaria para fomentar la difusión de las arquitecturas distribuidas coherentes, que se utilizan con mayor frecuencia para apoyar y simplificar el desarrollo de complejos aplicaciones distribuidas. Incluye servidores web, servidores de aplicaciones y herramientas similares que apoyan el desarrollo de aplicaciones. El middleware es la base de las tecnologías modernas de información basados en XML, SOAP, servicios Web y arquitectura orientada a servicios.

4.1.2. Dónde trabaja dicho middleware

El middleware se establece "en el medio", entre aplicaciones de software que pueden ejecutarse en diferentes sistemas operativos. Es similar a la capa media de una única arquitectura de sistema de tres niveles, excepto por que se estira a través de múltiples sistemas o aplicaciones. Los ejemplos incluyen software de telecomunicaciones, supervisores de transacciones y software de mensajería y gestión de colas.

La distinción entre el sistema operativo y la funcionalidad del middleware es, hasta cierto punto, arbitraria. Mientras las funciones del núcleo central sólo pueden ser proporcionados por el propio sistema operativo, algunas funcionalidades proporcionadas previamente por separado por middleware vendido ahora está integrado en los sistemas operativos. Un ejemplo típico es la

pila TCP/IP para las telecomunicaciones, hoy en día se incluye en casi todos los sistemas operativos.

El middleware se puede imaginar como una capa que se encuentra entre el código de la aplicación y de la infraestructura de tiempo de ejecución. El middleware consiste generalmente en una biblioteca de funciones, y permite una serie de aplicaciones para usar estas funciones de la biblioteca común en lugar de volver a crearlos para cada aplicación.

4.1.3. El porqué de usar un middleware

El middleware es muy útil por una serie de razones, a la hora de diseñar sistemas grandes y/o distribuidos. Algunos de los aspectos más relevantes se representan a continuación:

Portabilidad: el middleware ofrece un modelo de programación común a través de los límites del lenguaje y/o la plataforma, así como a través de sistemas de distribuidos. Gracias a esto, es posible que las aplicaciones desarrolladas en diferentes lenguajes (por ejemplo, Java y C++) y ejecutados en diferentes sistemas operativos (por ejemplo, Windows y Linux) sin ningún esfuerzo específico por el programador, sean capaces de cooperar.

Fiabilidad: los middlewares son desarrollados y probados por separado de la aplicación final. Esto permite que el programador de la aplicación abstraerse de los aspectos de bajo nivel y usar (y reutilizar) una librería bien probada.

Gestión de la complejidad: los aspectos de bajo nivel podrían ser gestionados por las librerías adecuadas que abstraen aspectos específicos del sistema operativo o del hardware. Esto simplifica el desarrollo y reduce la probabilidad de errores.

4.1.4. Tipos y funcionalidades de los middlewares

El software de middleware se puede dividir por funcionalidades y objetivo de las aplicaciones en tres categorías principales: de aplicación específica, de intercambio de información y de gestión y middleware de apoyo.

El middleware de aplicación específica proporciona servicios para diversas clases de aplicaciones, tales como los servicios de base de datos distribuida,

procesamiento datos-distribuidos/transacción-de-objetos y servicios especializados para la computación móvil y multimedia.

El middleware para intercambio de información la gestiona el a través de una red. Se utiliza para tareas como la transferencia de datos, emisión de órdenes, recepción de respuestas, comprobación de estado y resolución de puntos muertos.

El middleware de gestión y apoyo es responsable de localizar los recursos, la comunicación con los servidores, el manejo de la seguridad y los fallos y la monitorización del rendimiento.

4.1.5. El concepto de middleware en la robótica

Los sistemas robóticos son generalmente sistemas complejos construidos sobre diversos componentes de hardware y software, como sensores y actuadores, así como planificadores y algoritmos de control.

En general, en cada robot se ejecuta un software que se encarga de leer los datos de los sensores, extraer la información que necesitan de ellos, calcular la secuencia de acciones para llevar a cabo una tarea determinada y controlar los actuadores para ejecutar las acciones. Utilizando un enfoque personalizado, habrá una sola aplicación monolítica que se encargará de todas estas tareas, lo que hace que el mantenimiento del código sea duro y elimina cualquier posibilidad de reutilizar código y su intercambio entre diferentes proyectos.

En tal escenario, con tantos componentes de hardware y software que necesitan comunicarse y colaborar para alcanzar una meta, es exactamente donde un middleware puede ayudar mejorando la organización, la capacidad de mantenimiento y la eficiencia del código. Toda la aplicación se puede estructurar en muchas pequeñas tareas concretas, como "obtener una lectura del sensor", "extraer características de algunos datos", "impulsar los motores a cierta velocidad". Los diferentes componentes pueden intercambiar datos a través de un canal de comunicación común provisto por el middleware, haciendo uso de interfaces que sean compatibles entre las diferentes aplicaciones. De esta manera, se vuelve muy fácil compartir y reutilizar el código entre diferentes proyectos, o cambiar un algoritmo para conseguir un poco de funcionalidad, ya que sólo es necesario mantener la misma interfaz. Como ejemplo, si se necesita pasar de un sensor de proximidad a otro, es posible escribir un nuevo componente que

comparta la misma interfaz y actualizarlo sin modificar el resto de la aplicación. Este concepto puede extenderse a aplicaciones grandes y complejas, en las que el uso de un middleware puede mejorar claramente la organización del código general y reducir el esfuerzo de programación.

4.2. Middlewares robóticos anteriores a ROS

Hay muchos proyectos en torno a intentar lanzar un middleware para la robótica, que comparten casi los mismos conceptos y objetivos básicos, pero muchos de ellos existen sólo como propuesta y otros no han seguido siendo desarrollados. En esta breve revisión nos centraremos en los proyectos que ganaron más popularidad en la comunidad robótica y que todavía son apoyados y mantenidos de manera activa.

4.2.1. OROCOS

La idea de iniciar un proyecto de software libre para el control de robot nació en diciembre de 2000, sobre la lista de distribución de EURON, motivada por más de dos décadas de fracasos y experiencias decepcionantes al tratar de utilizar un software de control robótico comercial para la investigación avanzada en este campo. En ese momento no había software de propósito general de control robótico disponible, con código libre claro está, haciendo a OROCOS un proyecto innovador que interesó a mucha gente. El proyecto ha sido financiado por la Comisión Europea con la participación de tres socios: K. U. Leuven en Bélgica, LAAS Toulouse en Francia y KTH de Estocolmo en Suecia.

Ilustración 4.2 - Logo de OROCOS

El proyecto OROCOS aspira a convertirse en un paquete de propósito general y software robótico libre:

- Bajo licencia Open Source como LGPL, con extrema modularidad y flexibilidad, de manera que un desarrollador puede construir su sistema desde cero, mientras que otros pueden contribuir a los módulos en los que están interesados, sin la necesidad de tratar con el código de todo el sistema.

- De la más alta calidad, desde el punto de vista de la documentación técnica y la ingeniería de software.

- Independiente (pero si es posible compatible) de los fabricantes de robots comerciales, alegando que el código OROCOS debería convertirse en un inevitable estándar abierto.

- Para todo tipo de dispositivos robóticos y plataformas de ordenador.

- Con componentes de software para la cinemática, dinámica, planificación, control, sensorización, detección de hardware, etc. Los componentes están implementados en el sentido de objetos de software, pudiendo ser agregados o eliminados de forma dinámica de una red, y eso ofrece sus servicios a través de una interfaz de lenguaje de programación neutral e independiente.

OROCOS se compone de 4 bibliotecas de C++, las cuales introduciremos brevemente en los próximos párrafos.

El Kit de Herramientas de Tiempo Real (RTT, "Real-Time Toolkit"), no es una aplicación en sí misma, sino que proporciona la infraestructura y las funcionalidades para crear aplicaciones de robótica en C++. El énfasis está en tiempo real, aplicaciones interactivas en línea y aplicaciones basadas en componentes.

La biblioteca de tiempo real (RTT) permite a los diseñadores de aplicaciones construir aplicaciones de control en tiempo real basadas en componentes altamente configurables e interactivas.

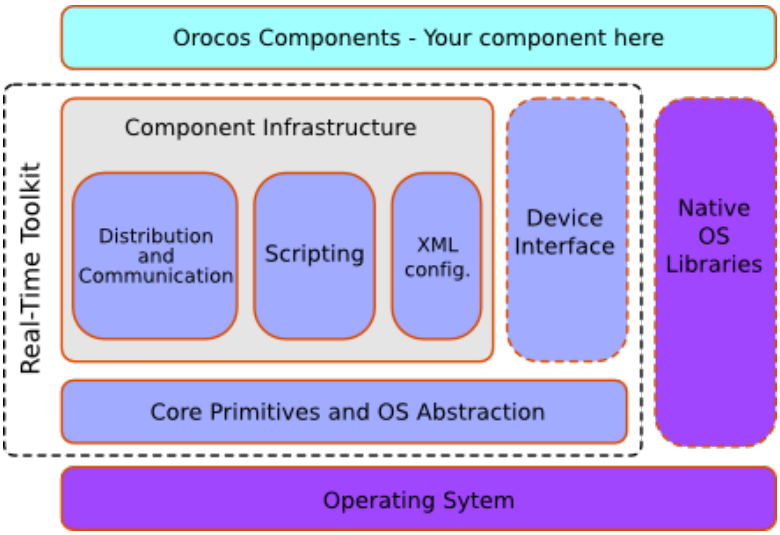


Ilustración 4.3 - Estructura del Kit de Herramientas de Tiempo Real de OROCOS

El RTT permite a los componentes ejecutarse en sistemas operativos de tiempo real y ofrece la posibilidad de crear scripts para tiempo real. También la comunicación y la distribución de componentes de la API y la configuración XML (Fig. 4.3).

Un componente de RTT se puede escribir por ejemplo para controlar dispositivos que van desde sensores hasta robots completos, para capturar y trazar un flujo de datos, para sintonizar un algoritmo en tiempo de ejecución o para conectarse a una interfaz de usuario.

Cada componente es una extensión de la primitiva "TaskContext", un objeto activo que ofrece puertos por hilos seguros y eficientes para (lock-free) intercambio de datos, que puede reaccionar a los eventos, comandos de proceso, o ejecutar máquinas de estados finitos en tiempo real estricto.

Los componentes se pueden configurar en línea a través de una interfaz de propiedad (valores set/get) y archivos XML.

La Librería de Componentes de Orocos (OCL, "Orocos Components Library") proporciona algunos componentes listos para usar. Todos los componentes están construidos sobre RTT, y algunos de ellos pueden utilizar las librerías KDL o BFL.

Es posible encontrar componentes de trabajo para interactuar con los dispositivos de hardware, para manejar rutas de acceso y planificar tareas, para

controlar diversos tipos de robots, así como para grabar y visualizar los flujos de datos y para poder depurar toda la aplicación.

La Librería de Cinemática y Dinámica de Orocos (KDL, "Kinematics and Dynamics Library) es una librería de C++ que permite calcular cadenas cinemáticas en tiempo real. La KDL desarrolla un marco independiente de aplicaciones para el modelado y el cálculo de las cadenas cinemáticas, tales como robots, modelos humanos biomecánicos, figuras animadas por ordenador, máquinas herramientas, etc.

Proporciona también librerías de clases de objetos geométricos (punto, marco, línea...), de cadenas cinemáticas de varias familias (serie, humanoide, paralelo, móvil...), y su especificación de movimiento e interpolación.

La Librería de Filtro Bayesiano de Orocos (BFL, "Bayesian Filtering Library") proporciona un marco de trabajo de aplicaciones independiente para la inferencia en redes bayesianas dinámicas, es decir, algoritmos de procesamiento recursivo de la información y su estimación basados en la regla de Bayes, como los Filtros de Kalman (extendidos), Filtros de Partículas (métodos Monte secuenciales), etc. Estos algoritmos pueden, por ejemplo, ser ejecutados en la parte superior de los servicios en tiempo real, o ser utilizados para la estimación en aplicaciones de cinemática y dinámica.

4.2.2. Orca

Orca es un marco de trabajo de código abierto para el desarrollo de sistemas robóticos basados en componentes. Proporciona los medios para definir y desarrollar los bloques de construcción que pueden ser montados juntos para formar sistemas robóticos arbitrariamente complejos, desde dispositivos individuales hasta redes de sensores distribuidos.

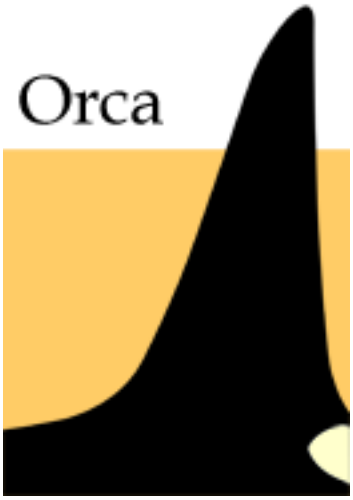


Ilustración 4.4 - Logo de Orca

Proyecto Orca surgió de OROCOS en KTH de Estocolmo en 2003. El objetivo principal señalado por sus desarrolladores es la reutilización del software, que definen un factor clave para seguir avanzando en la investigación robótica y la industria.

Orca intenta conseguir este objetivo centrándose en tres conceptos:

- Permitir la reutilización del software mediante la definición de un conjunto de interfaces de uso común.

- Simplificar la reutilización del software, proporcionando librerías con una API conveniente de alto nivel.

- Animar a la reutilización del software mediante el mantenimiento de un depósito de componentes.

Orca se define a sí misma como un sistema basado en componentes sin restricciones, lo que significa que quiere proporcionar una infraestructura para construir software robótico, pero no obligar al usuario a reescribir todo su código desde cero. De esta manera, Orca adopta un enfoque de Ingeniería de Software Basado en Componentes sin aplicar ninguna restricción arquitectónica adicional y utiliza una librería de código abierto comercial para la comunicación y la definición de la interfaz.

Orca también proporciona algunas librerías para aplicaciones comunes y herramientas para simplificar el desarrollo de componentes pero los hace

estrictamente opcionales para mantener pleno acceso al subyacente motor de comunicación y servicios.

La principal diferencia entre Orca y OROCOS radica en la herramienta sobre la que el motor de la comunicación se construye. Los desarrolladores de Orca han sustituido a CORBA, desarrollado desde 1991 y utilizado en OROCOS, por un moderno marco de trabajo de ZeroC: Internet Communication Engine (ICE).

Los desarrolladores ICE critican duramente el uso de CORBA en la proyectos actuales, alegando que se basa en viejas suposiciones, desarrolladas con tecnologías obsoletas y difíciles de usar. Orca se basa profundamente en ICE, como se puede notar observando su organización.

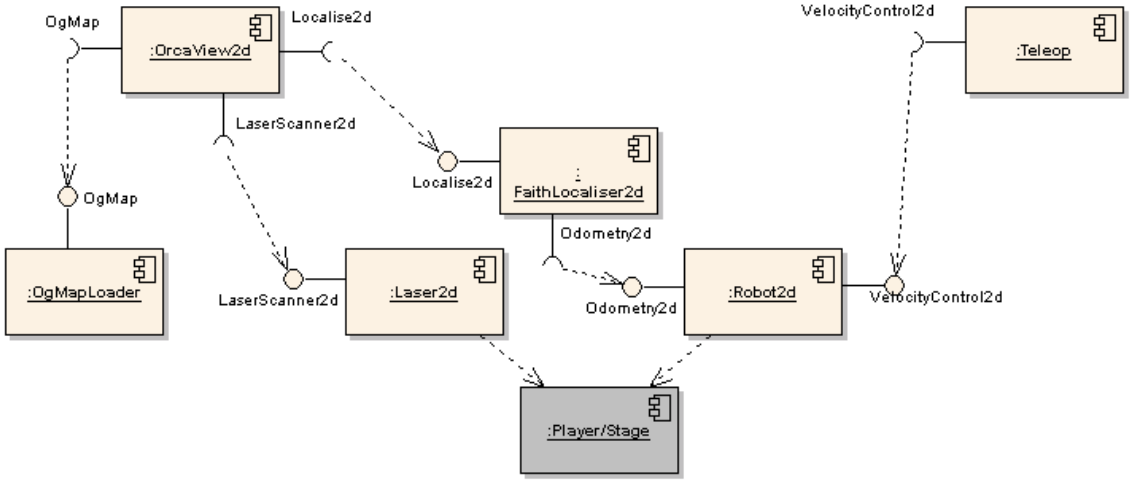


Ilustración 4.5 - Ejemplo de estructura de un sistema gestionado por Orca

Orca está compuesto por algunos servicios básicos, heredados de ICE, que forman la infraestructura y un conjunto de componentes comunes, que pueden ser utilizados y ampliados para construir aplicaciones personalizadas. Los servicios principales son:

Registro IceGrid: ofrece un servicio de nombres, un mapeo de nombres de interfaces lógicas a direcciones físicas. En la actualidad es la única manera para que los componentes se encuentren el uno al otro.

El Servicio IceStorm es un servicio de eventos, que se utiliza para desacoplar los publicadores de mensajes de los suscriptores. Típicamente, hay un servicio IceStorm por host.

La aplicación es, por tanto, construida por componentes, que se conocen entre sí a través del registro IceGrid y se comunican utilizando el servicio Icestorm.

Los componentes pueden interactuar con el hardware, utilizando controladores e interfaces de controladores, que deben ser coherentes en la misma familia de dispositivos (por ejemplo, los escáneres láser pueden tener diferentes controladores, pero la misma interfaz).

Los desarrolladores de Orca proporcionan algunos controladores para hardware popular, como cámaras, escáneres láser, plataformas de robots y controladores manuales, así como algunos de los componentes relacionados con tareas comunes, como los planificadores de meta y ruta, los algoritmos de visión por computador, controlador de movimiento, etc.

4.2.3. YARP

El último middleware robótico que abordaremos en esta comparativa será YARP, el cual en los últimos años ha sido bastante popular, pero que, como todos los demás, con la llegada de ROS han quedado desbancados de sus nichos de mercado por el que está llamado a ser el estándar internacional de gestión y comunicación robótica.

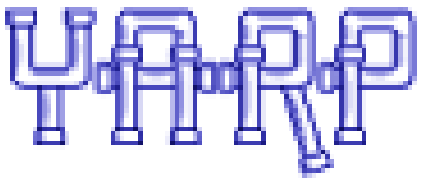


Ilustración 4.6 - Logo de YARP

YARP son las siglas de "Yet Another Robotic Plattform", que significa *sin embargo otra plataforma robótica*, y es, como no podía ser de otra manera, software gratuito y de código libre. Es un conjunto de bibliotecas, protocolos y herramientas para mantener los módulos y dispositivos adecuadamente

desacoplados. Es un middleware sin ningún deseo o expectativa de estar en control del sistema. YARP definitivamente no es un sistema operativo.

Proyectos de robots son a menudo callejones evolutivos sin salida, desapareciendo el software y hardware que producen sin dejar rastro. Las causas más habituales incluyen dependencias de dispositivos no muy comunes u obsoletos o librerías, y la dispersión de un ya pequeño grupo de usuarios. En robótica humanoide, un pequeño campo con un apetito ávido de nuevos dispositivos, se experimenta una gran cantidad de rotación de esta naturaleza. YARP es un intento de hacer que el software robótico sea más estable y de larga duración, sin comprometer la capacidad de cambiar constantemente los sensores, actuadores, procesadores y redes. Ayuda a organizar la comunicación entre sensores, procesadores y actuadores para que los acoplamientos débiles sean reforzados, permitiendo que la evolución gradual del sistema sea mucho más fácil.

El modelo YARP de comunicación es neutro para transporte, de modo que el flujo de datos se desacopla de los detalles de las redes y protocolos subyacentes (permitiendo utilizar varios simultáneamente, clave para una evolución progresiva). YARP utiliza una metodología para la interconexión con dispositivos (sensores, actuadores, etc.) que alienta nuevamente la articulación flexible y puede hacer cambios menos perjudiciales en los dispositivos. Al mismo tiempo, YARP no espera estar al cargo; se intenta minimizar el problema de "arquitecturas" incompatibles, "marcos de trabajo", y "middleware" (también conocido en este contexto como "muddleware").

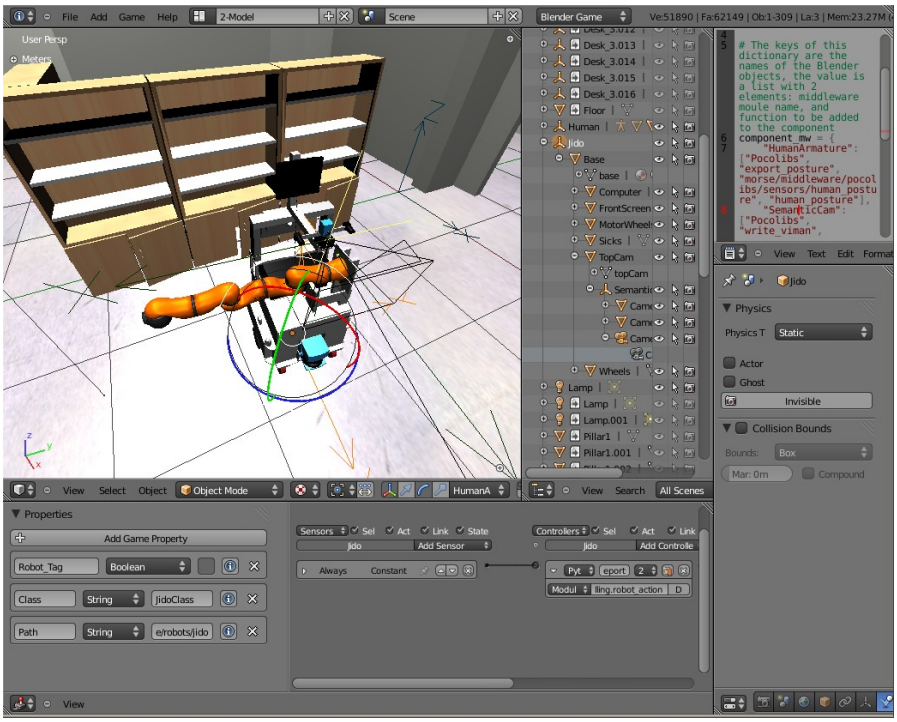


Ilustración 4.7 - Simulador MORSE para el uso es YARP

YARP está escrito por y para investigadores en robótica, particularmente la robótica humanoide, que se encuentran con una complicada pila de hardware para controlar con una igualmente complicada pila de software. En el momento de escribir, correr una decente percepción visual, auditiva y táctil mientras se realiza el elaborado control de un motor en tiempo real requiere una gran cantidad de cómputo. La forma más sencilla y escalable para hacer esto ahora mismo es tener un conjunto de ordenadores. Cada año lo que una máquina puede hacer crece, pero también lo hacen la demanda de las aplicaciones. En definitiva, YARP es un conjunto de herramientas que han encontrado útiles para la satisfacción de las necesidades computacionales para el control de varios robots humanoides.

Los componentes de YARP se pueden dividir en:

- libYARP_OS – hace de interfaz con el sistema operativo (uno o varios) y permite un sencillo flujo de datos a largo de diferentes hilos en diferentes máquinas. YARP está diseñado para ser neutral en cuanto al SO, y ha sido usado con Linux, Microsoft Windows, Mac OS X y Solaris. Este middleware utiliza la librería "open-source ACE" (ADAPTATIVE Communication Enviroment), que es portable entre un gran rango de

entornos, y YARP hereda esa portabilidad. Además YARP está casi íntegramente escrito en C++.

- libYARP_sig – lleva a cabo tareas comunes de proceso de señal (audio y vídeo) de manera abierta fácilmente interconectada entre otras librerías comúnmente usadas, por ejemplo OpenCV.

- libYARP_dev – hace de interfaz con dispositivos comunes usados en robótica, como framegrabbers, cámaras digitales, tarjetas de control de motores, etc.

Estos componentes se mantienen por separado. El componente principal es libYARP_OS, que deberá estar disponible antes de poder utilizar el resto de componentes.

Para el funcionamiento en tiempo real, la sobrecarga de la red tiene que ser minimizada, por lo que YARP está diseñado para funcionar en una red aislada o detrás de un firewall. Si se exponen máquinas que funcionan YARP a internet, es de esperar que el robot algún día pueda ser controlado remotamente sin que podamos evitarlo.

Para la conexión con el hardware, estamos a merced de los sistemas operativos que las empresas particulares optan por dar soporte. La librería libYARP_dev está estructurada para interactuar fácilmente con código suministrado por el fabricante, pero protegiendo el resto del sistema de dicho código de manera que futuros reemplazos de hardware no impliquen problema alguno. YARP no reducirá los requisitos impuestos por su hardware actual, sólo hará que los cambios futuros sean más fáciles.

YARP tiene en consecuencia tres niveles de configuración: sistema operativo, hardware, y nivel robot. El primer nivel de configuración únicamente concierne a aquellos usuarios que estén planeando compilar YARP en un nuevo sistema operativo.

El segundo nivel es el hardware. Una nueva adición en una plataforma existente o una nueva plataforma por completo puede requerir la preparación de algunos controladores YARP de dispositivos. Éstos son, a todos los efectos, clases de C++ que admiten métodos para acceder al hardware, que se implementa

normalmente a través de llamadas a funciones cualesquiera que facilite el proveedor de hardware. Esto viene típicamente en la forma de una DLL o una biblioteca estática.

Por último, se pueden preparar archivos de configuración para una nueva plataforma robótica.

4.3. ROS

Y finalmente llegamos al middleware robótico que está llamado a ser el estándar de la industria durante bastantes años. Con la elección de esta nueva tecnología se ha sido pionero en el Departamento de Sistemas y Automática. Hemos sido los primeros en desarrollar un sistema basado en esta plataforma, con las dificultadas y complejidad que eso entraña.

Esta parte del proyecto, al igual que la de Visión Artificial, se pueden catalogar como proyectos de investigación, pues todos los conocimientos que han sido necesarios para llevar a cabo el proyecto se han conseguido a través del autoaprendizaje y la experiencia propia. Esto ha supuesto un importante hándicap en el desarrollo de este TFG, otorgándole así un gran valor añadido.

El trabajo desarrollado en ROS no se ha ceñido al marco del presente trabajo, sino que durante el curso también se diseñó un proyecto final de un robot móvil con algoritmos de evitación de obstáculos gestionado remotamente por ROS y conexión inalámbrica WiFi. Dicho proyecto está recogido como código libre en el portal GitHub para servir a futuros estudiantes de ejemplo para el desarrollo de una plataforma completa ROS.

Antes de comenzar a explicar qué es realmente ROS, cómo está estructurado y cómo funciona, es importante comentar un poco la complejidad de este software y su curva de aprendizaje.

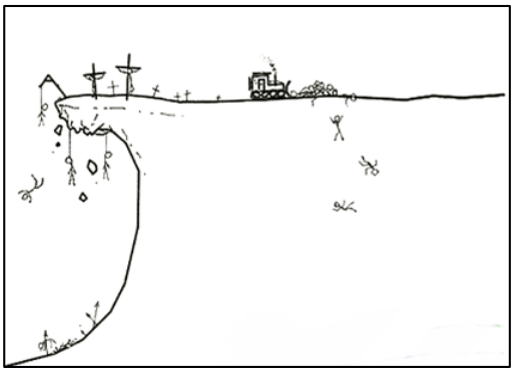


Ilustración 4.8 - Curva de aprendizaje ROS

En la Fig. 4.8 se muestra la curva de aprendizaje de ROS a modo de pequeña sátira, aunque no lo es tanto. Entender cómo está organizado ROS, el sistema de comunicación interna, la gestión de paquetes, etc., puede ser bastante caótico cuando no hay otra persona que lo domine y pueda explicarlo y resolver dudas.

El dominio completo de un sistema como ROS no se alcanza en pocos meses o con la lectura de unos cuantos tutoriales. Requiere de varios años para ser un experto en él y muchas constancia, debido a la gran velocidad de actualización que sufre. Es por tanto, tarea obligada, advertir al lector de que zambullirse en el mundo de la robótica y ROS es una ardua tarea que tarda bastante en dar sus frutos y no apta para pusilánimes.

4.3.1. Definición

ROS es una iniciativa reciente de Willow Garage, un laboratorio de investigación, fundada a finales de 2006 para acelerar el desarrollo de la robótica no destinada al uso militar y crear avances en software de robótica.

Como Willow Garage produce tanto de software como de hardware, ROS se ha probado en sus robots a través de una serie de hitos que muestran el progreso del desarrollo.

ROS se define como un sistema de meta-operativo de código abierto, destacando que quiere ser algo más que un middleware. Proporciona los servicios que se esperan de un sistema operativo, incluyendo abstracción de hardware, control de dispositivos de bajo nivel, implementación de la funcionalidad de uso común, paso de mensajes entre procesos y de gestión de paquetes. Su objetivo

principal, como para Orca, es permitir la reutilización de código en la investigación y desarrollo de la robótica y liberar paquetes de software listos para utilizarse en un gran conjunto de tareas comunes.

ROS ofrece también una comunidad web para compartir y colaborar, y repositorios para almacenar y distribuir paquetes de software.

4.3.2. Historia de ROS

ROS es un proyecto grande con muchos ancestros y contribuyentes. La necesidad de un marco de colaboración abierta era algo que tenían muy claro muchas personas en la comunidad de investigación de la robótica, y muchos proyectos se han creado para alcanzar este objetivo.

Varios esfuerzos de la Universidad de Stanford a mediados de la primera década del s. XXI que incluían IA integrada, como la de Stanford AI Robot (STAIR) y el programa de Robots Personales (PR), crearon prototipos internos de sistemas de software flexibles y dinámicos destinados a la robótica. En 2007, Willow Garage, una cercana y visionaria incubadora de robótica, ofreció importantes recursos para ampliar estos conceptos mucho más allá y crear implementaciones bien probadas. El esfuerzo se ha visto impulsado por innumerables investigadores que contribuyeron con su tiempo y experiencia para desarrollar las ideas centrales de ROS y sus paquetes de software fundamentales. En todo momento, el software fue desarrollado en el uso de la licencia de código abierto BSD, y poco a poco se ha convertido en una plataforma ampliamente utilizada por la comunidad de investigación robótica.

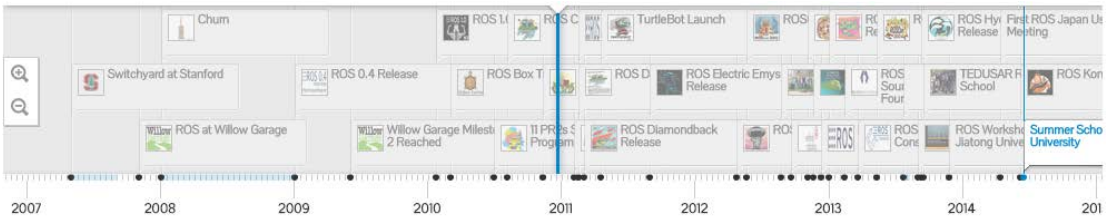


Ilustración 4.9 - Cronología de ROS

Desde el principio, ROS se desarrolló en múltiples instituciones y para múltiples robots, incluyendo muchas instituciones que recibieron robots PR2 de Willow Garage. A pesar de que habría sido mucho más sencillo para todos los contribuyentes poner su código en los mismos servidores, en los últimos años, el

modelo "federado" ha resultado ser una de las grandes fortalezas del ecosistema ROS. Cualquier grupo puede iniciar su propio repositorio de código ROS en sus propios servidores, y mantener la propiedad y el control completamente. No se necesita permiso de nadie. Si deciden poner su repositorio a disposición del público, pueden recibir el reconocimiento y el crédito que se merecen por sus logros, y beneficiarse de información técnica específica y mejoras, como en todos los proyectos de software de código abierto.

El ecosistema de ROS se compone actualmente de decenas de miles de usuarios en todo el mundo, trabajando en dominios que van desde proyectos de aficionados en sus casas a grandes sistemas de automatización industrial.

4.3.3. Arquitectura de ROS

ROS se organiza en tres capas:

- Nivel del sistema de archivos: recursos ROS almacenados en el disco duro.
- Nivel gráfico computacional: una red de procesos punto a punto de ROS que se están gestionando los datos de forma conjunta.
- Nivel comunitario: recursos ROS que permitan a las comunidades separadas al software de intercambio y conocimiento.

El núcleo del sistema es el Gráfico Computacional, que se describe como una red distribuida de procesos, llamados nodos, diseñado individualmente y después débilmente acoplados en tiempo de ejecución. Esta red está compuesta por algunos elementos básicos:

- Los nodos son procesos que llevan a cabo el cómputo de realizar alguna tarea (por ejemplo, leer telémetro láser, control de motores, localización, planificación de la trayectoria...).
- El servicio Maestro proporciona el registro de nombres y de consulta para el resto de la Computación Gráfica.
- El servidor de parámetros permite que los datos sean almacenados mediante clave en una ubicación central.

- Los mensajes son utilizados por los nodos para comunicarse entre sí. Un mensaje es una estructura de datos que comprende campos tipados.

- Los topics se utilizan para identificar el contenido de un mensaje. Los mensajes se enrutan a través de un sistema de publicación/suscripción de transporte. Un nodo envía un mensaje mediante su publicación en un topic, y otro nodo lee mediante la suscripción a ese topic.

- Los servicios se utilizan cuando el modelo publicador/suscriptor no es apropiado con respecto a la solicitud/respuesta de interacciones. Un nodo ofrece un servicio, que se define como un par de estructuras de petición y respuesta. Un nodo cliente envía la solicitud y espera la respuesta.

- Las bolsas son formatos para guardar y reproducir datos de mensajes ROS.

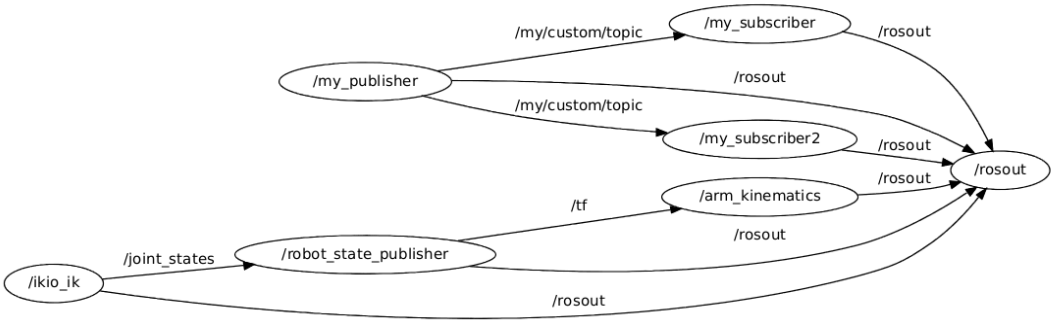


Ilustración 4.10 - Gráfico Computacional

El ROS maestro actúa como un servicio de nombres en el ROS Computación Gráfica. Almacena información de topics y servicios de registro de nodos ROS.

Los nodos se comunican con el Maestro para reportar su información de registro. A medida que estos nodos se comunican con el Maestro, pueden recibir información sobre otros nodos inscritos y hacer las conexiones según proceda. El protocolo más común que se utiliza en un ROS se llama TCPROS, que utiliza sockets TCP/IP estándar. El Maestro también hará callbacks a estos nodos cuando esta información de registro cambie, que permite que los nodos puedan crear dinámicamente conexiones con los nuevos nodos que comienzan a ejecutarse.

Los nodos se conectan a otros nodos directamente, mientras que el Maestro sólo proporciona información de búsqueda. Los nodos que se suscriben a un topic solicitarán conexiones desde los nodos que publican en ese topic, y establecerán esa conexión a través de una conexión de protocolo acordado.

Esta arquitectura permite una operación disociada, donde los nombres son el medio principal por el que los sistemas más grandes y complejos se pueden construir. Por ejemplo, cada nodo sensor publica exploraciones, sin el conocimiento de si alguien está suscrito. Todos los nodos filtro se suscriben a las exploraciones, sin el conocimiento de si alguien los está publicando. Los nodos pueden ser iniciados, cerrados y reiniciados en cualquier orden, sin inducir ninguna condicione de error.

Los nombres tener un papel muy importante en ROS: nodos, topics, servicios y parámetros, todos tienen nombres. Cada biblioteca cliente ROS soporta comandos de reasignación de nombres, lo que significa que un programa compilado puede ser reconfigurado en tiempo de ejecución para operar en una topología de Computación Gráfica diferente.

Los desarrolladores de ROS distribuyen una gran cantidad de componentes listos para usar, como controladores de hardware para muchos dispositivos y algoritmos para un amplio conjunto de tareas. ROS también se distribuye como una imagen de arranque, construida en Ubuntu GNU/Linux, que puede ser instalada en cualquier ordenador para conseguir un ecosistema ROS completo en minutos.

4.3.4. Creación de archivos

ROS es un sistema multiplataforma y por tanto admite distintos lenguajes de programación. Los programas pueden ser redactados en C++, Python, Octave o LISP, mientras que los mensajes se redactan en un lenguaje neutral (IDL). Esto otorga una gran flexibilidad a los programadores, pues pueden adaptar los desarrollos a sus conocimientos en lenguajes de programación.

También es importante dominar la estructura de un archivo XML y de configuración YAML, ya que los paquetes vienen definidos en el primer lenguaje y la mayoría de los archivos de configuración en el segundo. Otro tipo de archivo para la ejecución de paquetes es el LAUNCH. Su estructura no es muy compleja,

pero si el usuario necesita información detallada puede encontrar más información en los siguientes enlaces.

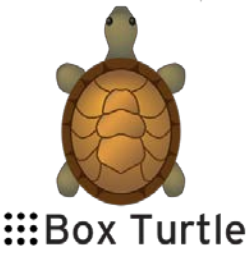
http://yaml.org/spec/current.html#id2502311

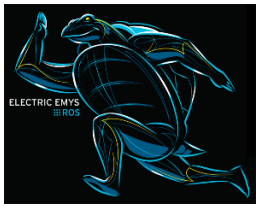
http://wiki.ros.org/roslaunch/XML

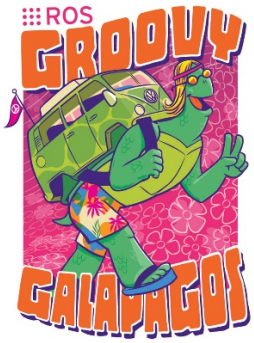
4.3.5. Versiones de ROS

ROS, como cualquier sistema de cierta envergadura, es actualizado con gran frecuencia. Normalmente estas actualizaciones tienen repercusiones menores, afectando a alguna librería en particular o algún paquete. Sin embargo, los desarrolladores de ROS, a lo largo de su historia lo han estructurado en versiones, cuyo nombre viene en orden alfabético según su lanzamiento cronológico.

Estas versiones de ROS tienen una peculiaridad que es bastante molesta, y es que existe incompatibilidad de trabajo entre distintas versiones. Esto puede suponer un problema cuando se tienen diferentes plataformas de trabajo hardware, las cuales tienen sus controladores desarrollados en versiones diferentes de ROS. La comunicación entre ellos puede ser complicada, teniendo que recurrir a otros métodos de comunicación más engorrosos.

Versión	Fecha de lanzamiento	Logo	Descripción
ROS 1.0	22 Enero 2010		Tras una extensa fase de documentación y pruebas (también conocido como "Milestone 3"), ROS 1.0 es presentado, estableciendo muchos de los componentes y APIs que todavía hoy se utilizan.
ROS Box Turtle	1 Marzo 2010		La primera distribución de ROS propiamente dicha, ya con su nombre clave. La distribución es ahora un concepto clave para ROS, con la mayoría de los usuarios confiando en una distribución específica.

ROS C Turtle	3 Agosto 2010		ROS C Turtle es la segunda distribución. El primer lote de PR2s fue distribuido con una versión pre-lanzamiento de C Turtle. Consistía básicamente en una actualización de las librerías disponibles en la versión anterior.
ROS Diamondblack	2 Marzo 2011		La tercera distribución de ROS. Contiene alrededor de 40 nuevos paquetes, incluyendo soprote para la Kinect, paquetes creados por la comunidad ROS y una versión estable de PCL. Ha sido diseñado para ser más pequeño, ligero y configurable que su anterior versión.
ROS Electric Emys	30 Agosto 2011		La cuarta distribución de ROS. Incluye librerías estables para arm_navigation y PCL, así como expande el soporte para muchas plataformas nuevas como Android y Arduino.
ROS Fuerte Tuetle	30 Abril 2012		ROS Fuerte Turtle es la quinta versión de distribución de ROS ROS. ROS Fuerte tiene importantes mejoras que hacen más fácil la integración con otros marcos de trabajo y herramientas de software. Esto incluye una nueva versión del sistema de compilación, la migración hacia el marco de Qt, y transición continua a librerías standalone.

ROS Groovy Galapagos	31 Diciembre 2012		ROS Groovy Galápagos es la sexta versión de distribución de ROS. En esta versión se han centrado en la infraestructura central de ROS para que sea más fácil de usar, más modular, más escalable, el trabajo a través de un mayor número de sistemas operativos/ arquitecturas de hardware / robots y lo más importante, para involucrar más a la comunidad ROS.
ROS Hydro Medusa	9 Septiembre 2013		ROS Hydro Medusa es la séptima versión de distribución de ROS. En esta versión se han centrado en la conversión de muchos de los paquetes de ROS al nuevo sistema catkin de construcción al tiempo mejorar y arreglar componentes centrales de ROS. Además existen muchas mejoras en herramientas como rviz y rqt. Esta versión de ROS se ha esforzado para depender de las versiones de dependencias de canonical, en lugar de empaquetar sus propias versiones. Esto ocurre con PCL, Stage y Gazebo y otras librerías que solían personalizarse y lanzarse por separado de manos de la comunidad ROS. Hydro dispone también de una integración mejorada con Gazebo.
ROS Indigo Igloo	22 Julio 2014		La octava versión de ROS diseñada especialmente para la nueva distribución de Ubuntu, 14.04 LTS (Trusty.)

4.3.6. Gazebo

ROS ofrece una potente herramienta de simulación, Gazebo, que permite recrear infinidad de entornos y situaciones en las que probar robots, sensores, actuadores, etc.

Es un simulador 3D con físicas de sólido rígido que permite la detección de colisiones o la interacción entre diferentes objetos o robots dentro del mismo. De esta manera, aunque no se disponga en un momento dado del robot real es posible probar los desarrollos en un entorno simulado seguro y comprobar si el sistema responde tal y como se ha diseñado.

La gran ventaja de Gazebo es que para que los robots simulados que representa funcionen no hay que publicar en topics especiales, se mantiene a la escucha de los topics reales del robot. Esto soluciona en gran medida las cosas. Hay que decir que esto es así porque los desarrolladores al hacer el paquete de simulación de su robot para Gazebo lo definen para que lea y publique en los mismos topics que el modelo real, si no, Gazebo no respondería debidamente. Esto es importante tenerlo en cuenta de cara a hacer un desarrollo propio.

Gazebo se puede lanzar directamente o a través de un archivo ".launch" donde venga especificado qué se desea mostrar. En caso de querer abrir el simulador con espacio de trabajo vacío escribiremos:

```
$ roslaunch gazebo_worlds empty_world.launch
```

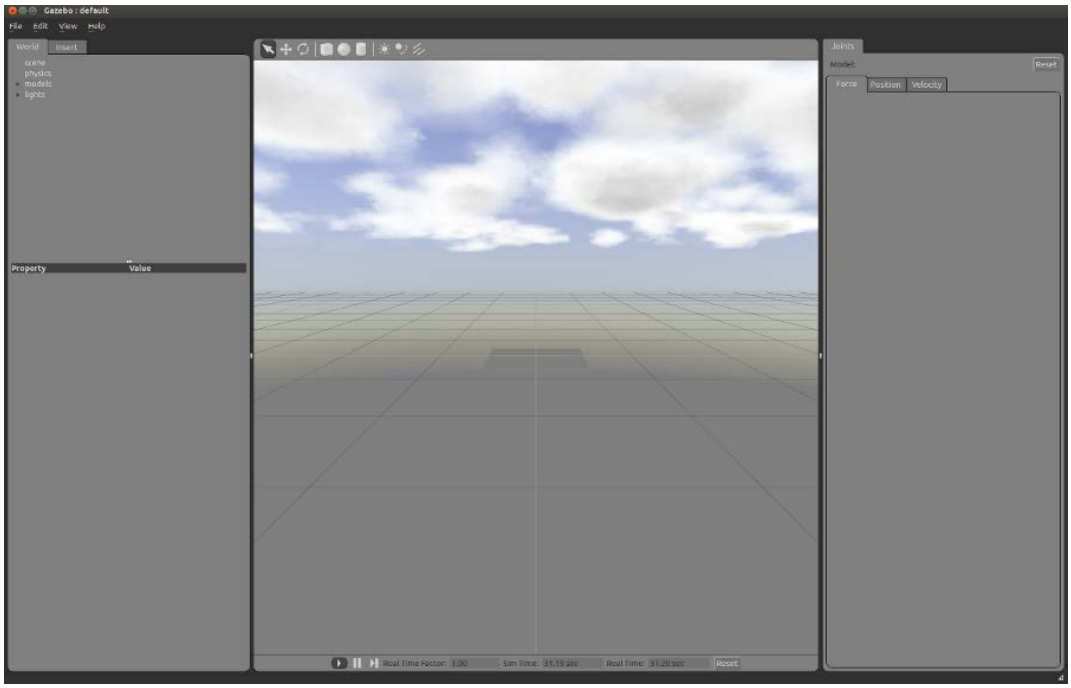


Ilustración 4.11 - Simulador Gazebo

Para más información acerca de Gazebo se recomienda visitar los siguientes enlaces:

http://wiki.ros.org/simulator_gazebo http://gazebosim.org/

4.3.7. RVIZ

RVIZ es una potentísima herramienta de visualización 3D que ofrece ROS. En ella se pueden integrar todo tipo de elementos (visión, control, navegación…) y controlarlos de forma interactiva. Su uso es bastante intuitivo y permita la simulación de trabajo entre sistemas cooperativos. Ofrece multitud de opciones para variar la situación en la que se está visualizando, así como la adición y extracción de elementos de la escena, como marcadores, nubes de puntos PCL, sistemas de referencia, trayectorias, etc.

En el capítulo en el que se profundiza en *Moveit!* se verá la importancia que tiene esta herramienta a la hora del trazado de trayectorias, pues permite ver antes de que se ejecute en el robot real cuál será el trazado que recorrerá el brazo para llegar de una posición a otra especificada.

El aspecto que muestra RVIZ es el siguiente:

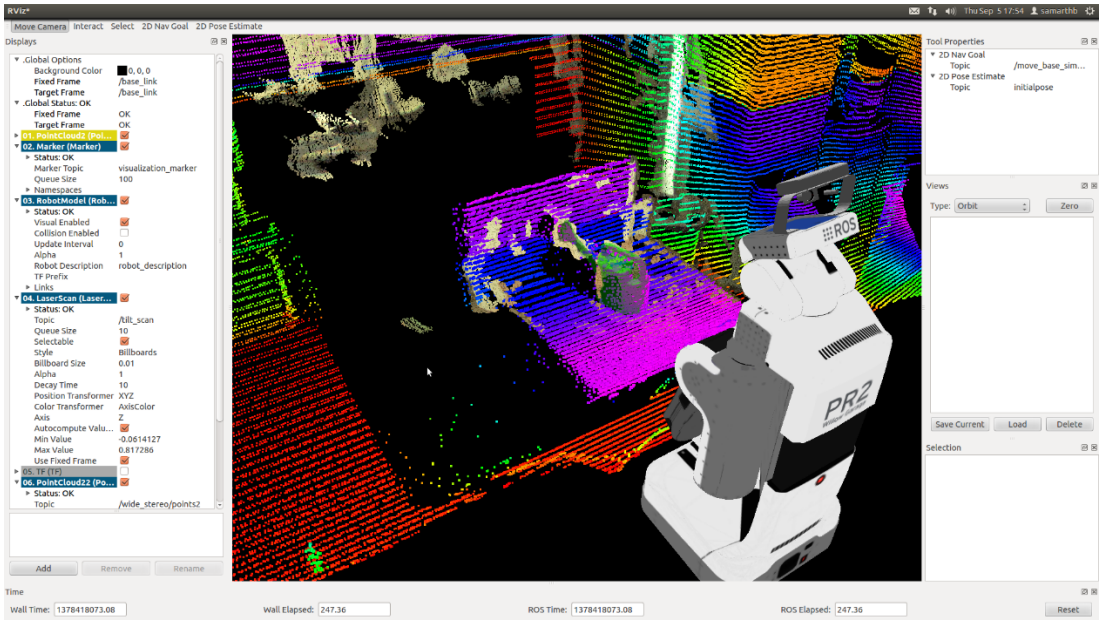


Ilustración 4.12 - RVIZ

En el ejemplo mostrado en la Fig. 4.12 vemos una representación de un sistema formado por un robot PR2 el cual dispone de un escáner láser. Vemos en la barra lateral izquierda cómo se muestran los elementos representados (robot, láser, nube de puntos captada por el láser…). A la derecha hay una barra de propiedades y en la parte inferior se puede observar que se está monitorizando el tiempo de simulación.

Para más información acerca de RVIZ visitar el siguiente enlace:

http://wiki.ros.org/rviz/UserGuide

4.3.8. Instalación, configuración y guía de iniciación

En el *Anexo Nº2* se adjunta una guía detallada de cómo se instala ROS en un PC con Ubuntu 12.04, en nuestro caso la versión Groovy, que es la que se ha utilizado para el proyecto. También se adjunta una pequeña guía y consejos para iniciarse en el uso de este software.

Capítulo 5

HARDWARE UTILIZADO

5. Hardware utilizado

Este capítulo está dedicado a la descripción del hardware utilizado para el desarrollo del presente proyecto. En él abordaremos las características del brazo robótico utilizado, la cámara Kinect del sistema de visión y las piezas que hubo que diseñar específicamente para poder acoplar el brazo a la mano ShadowHand.

Ilustración 5.1 - Logo Schunk

Ilustración 5.2 - Logo Microsoft Kinect

5.1. Powerball Lightweight Arm LWA 4P

El brazo utilizado para el proyecto es de la firma Schunk, y el modelo es el especificado en el título del presente apartado. Este brazo dispone de 6 grados de libertad, es decir, posicionamiento en los tres ejes (x, y, z) y orientación con respecto a esos tres ejes (yaw, pitch, roll). Esto nos ofrece total operatividad a la hora de posicionar el efector final.

Ilustración 5.3 - Schunk Powerball LWA 4P (pinza no incluida)

5.1.1. Características técnicas principales

El Schunk Powerball se presenta como un brazo manipulador revolucionario en lo que respecta al diseño de sus articulaciones, las cuales integran en un mismo dispositivo dos motores con ejes de giro perpendiculares. Esto le otorga una gran maniobrabilidad en un cuerpo muy reducido, con sólo dos eslabones en su cadena cinemática.

Las articulaciones ofrecen, por tanto, ±170° de giro en cada eje, como se muestra en la siguiente imagen.

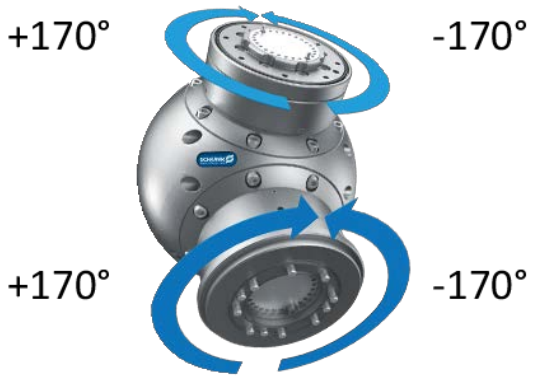


Ilustración 5.4 - Giros de la articulación Powerball

Estas innovadoras articulaciones tienen la siguiente estructura interna:

Ilustración 5.5 - Estructura interna de la articlación Powerball

En ella podemos ver cómo se compone de la electrónica de control [1], el encoder [2], el motor [3], el engranaje Harmonic Drive® [4], el freno de mantenimiento [5] y un eje hueco para el cableado interno [6].

Este robot ofrece una serie de ventajas que lo hacen destacar ante sus competidores, como pueden ser:

- Es móvil, pudiendo ser usado independientemente de forma opcional.
- Capacidad de operación con batería (baja capacidad 12 Ah).
- Permite una jornada de trabajo (8 horas) sin cargar la batería.
- Bajo consumo de energía a 24 V (DC)
- Gran par, velocidad y repetibilidad para una rápida aceleración, tiempos de ciclo corto y gran estabilidad del proceso.
- La completa integración del control, regulador y electrónica de potencia hace que no se requiera un armario de control.
- Sistema de cambios rápidos para mejor mantenimiento.
- Posibilidad de conexión directa de pinzas y herramientas.
- Cableado totalmente interno
- Ampliable sin molestos cables
- Construcción ligera y novedoso diseño que permite un ratio de peso/carga de 2:1.

5.1.2. Comunicación

La comunicación en este robot se lleva a cabo mediante CAN BUS. Para poder interactuar con el mismo desde un ordenador común hemos de hacer uso de un adaptador entre el bus de comunicación y un puerto estándar USB. Ésta es la función que cumple el PCAN-USB de la compañía PEAK System.

Ilustración 5.6 - PCAN-USB de PEAK System

Para hacer uso de este dispositivo es necesaria la instalación de los drivers del mismo, que están disponibles en la página web de la compañía. Este proceso de instalación se ilustrará paso a paso en el capítulo de configuración del sistema.

Una vez instalados, el protocolo de comunicación se hace invisible al usuario, lo que facilita en gran medida su utilización.

5.1.3. Elementos de seguridad

El brazo dispone de dos sistemas de protección. Por un lado la caja de alimentación del mismo donde se encuentran el transformador y conversor DC-DC que lo alimentan, así como todas las protecciones pertinentes contra sobretensiones y/o sobrecorrientes.

Por otro lado tenemos la seta de emergencia, con botón de rearme tras la actuación. Cada vez que conectemos la alimentación del robot será necesario presionar el botón de rearme, que está iluminado en color amarillo cuando no permite movimiento en el brazo. De igual manera, si hemos pulsado la seta para detener el movimiento del robot por cualquier causa, habrá que liberar la seta

girándola y posteriormente rearmar el botón para continuar trabajando con el robot.

5.1.4. Especificaciones técnicas completas

En el *Anexo Nº3* se presentan las características técnicas completas del robot.

En el *Anexo Nº4* se presentan las características técnicas completas del adaptador PCAN-USB.

5.2. Microsoft Kinect

Para la captación de imágenes del sistema de visión artificial se ha usado un sensor Kinect. Esta cámara RGBD perteneciente a la videoconsola XBOX 360 de Microsoft ha sido ampliamente usada por la comunidad científica e ingenieril, especialmente en el campo de la robótica. El hecho de que haya sido utilizada por tantos investigadores se debe a su reducido precio (unos 150€) en comparación con otros sistemas de visión estéreo específicos, además de que los resultados ofrecidos son de una calidad bastante aceptable para la mayoría de aplicaciones.

Todo lo anterior, unido a la liberación del SDK, así como el desarrollo de paquetes de drivers libres como OpenNi han hecho de él un dispositivo muy útil y versátil.

5.2.1. Características técnicas principales

La cámara Kinect, como bien se detalla en la Fig. 5.7, está formada por una hilera (array) de 4 micrófonos para poder detectar la voz en distintas ubicaciones con precisión. En el centro tiene una cámara estándar con resolución 4xVGA (1280 x 960) y luego un sensor y un emisor de rayos infrarrojos con una separación equivalente a la de los ojos humanos. El trabajo conjunto entre la cámara RGB y los sensores de profundidad permiten asociar a cada píxel de la imagen unas coordenadas en unos ejes cartesianos referenciados al plano que forman entre sí sensor y cámara. La cámara es capaz de capturar imágenes a una velocidad de 30 fps.

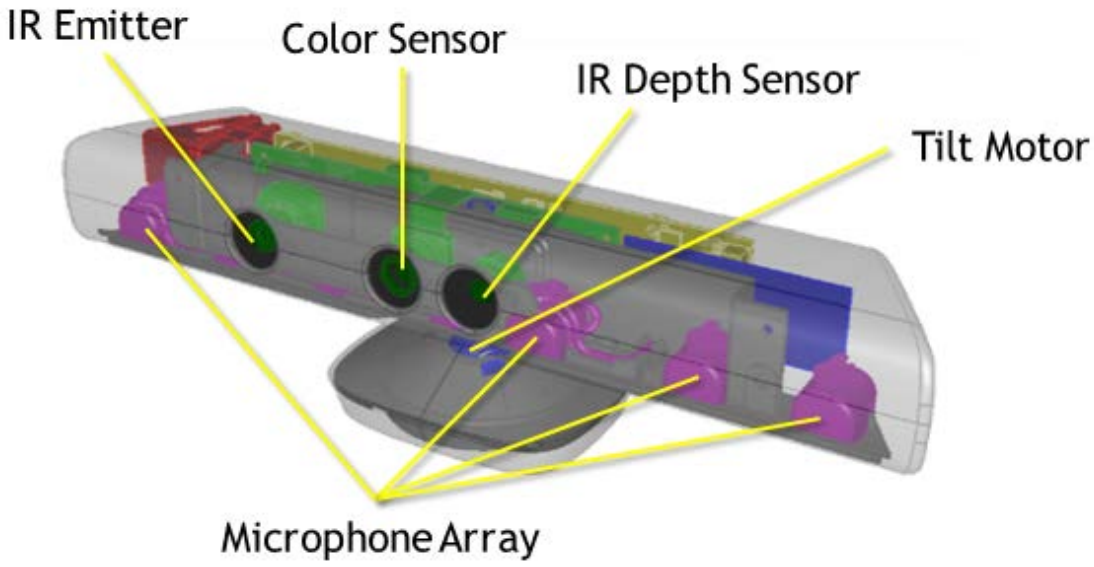


Ilustración 5.7 - Sensor Kinect (componentes)

El array de micrófonos dispone de un convertidor analógico-digital de 24 bit con cancelación de eco y supresión de ruido. La codificación del audio es en modulación de ancho de pulso (PCM) en formato mono.

La óptica de la cámara permite un ángulo de visión de 43° en el plano vertical y de 57° en el plano horizontal.

La base de la cámara Kinect integra un motor con una serie de engranajes que le permiten variar la inclinación del sensor en función de las necesidades de enfoque. Con esto tenemos la posibilidad de variar el plano de captura sin necesidad de utilizar una superficie inclinada variable, lo cual, a la hora de hacer calibraciones para el sistema de visión artificial es de gran ayuda pues podemos saber exactamente los grados con respecto al suelo. Este motor permite variar la inclinación en ±27°.

El sistema, para poder determinar en qué orientación se encuentra dispone de un acelerómetro de 2G/4G/8G, configurado para el rango de 2G, con un límite de exactitud de un grado.

5.2.2. Comunicación y alimentación

La comunicación del sistema de visión estereoscópica Kinect es a través de USB. La alimentación igualmente se hace a través de cable USB, proveniente de

la fuente de alimentación correspondiente. Ambos cables se unen en un solo conductor entes de llegar al dispositivo. En la Fig. 5.8 se pueden apreciar los componentes de la alimentación y la comunicación del sistema Kinect.

Ilustración 5.8 - Componentes de alimentación y comunicación del sistema Kinect

Capítulo 6

DISEÑO DE ACOPLES Y SOPORTES

6. Diseño de acoples y soportes

Para poder llevar a cabo el proyecto, en el cual es necesaria la sincronización entre el brazo Schunk LWA 4P y la mano ShadowHand, se hace necesario un montaje conjunto entre ambas. El extremo del brazo ofrece una interfaz de conexión especialmente indicada para productos Schunk. Para otro tipo de herramientas, la firma ofrece una abrazadera ajustable (Fig. 6.1), aunque para nuestro caso no era útil por las dimensiones de la base de la ShadowHand.

Ilustración 6.1 - FWS 115

Siendo así, se optó por el diseño de un nuevo acople para ensamblar la mano con el brazo. Para su diseño fue preciso tomar medidas de la abrazadera anteriormente citada con pie de rey ya que no se disponía de los planos de la misma. De la base de la ShadowHand sí disponíamos de los planos, lo que facilitó en cierta medida el diseño del acople.

6.1. Acople Powerball-ShadowHand

Para el diseño de la pieza se ha utilizado el software de diseño industrial SolidWorks® 2013 x64 Edition. La pieza se ha construido en aluminio para que sea ligera manteniendo una rigidez suficiente. Era importante el hecho de que el peso fuera reducido (600 g aprox.) ya que la mano tiene de por sí un peso de 3,2 kg, y la carga máxima de trabajo del brazo es de 6 kg.

El acople se diseñó en dos piezas complementarias que se cierran por medio de tornillos para poder ajustarla a los dos elementos a unir evitando juegos. En la Fig. 6.2 se aprecia la forma del acople, donde en un lado se copia el diseño de

la abrazadera FWS 115 y en el otro extremo tenemos la forma de la base de la ShadowHand.

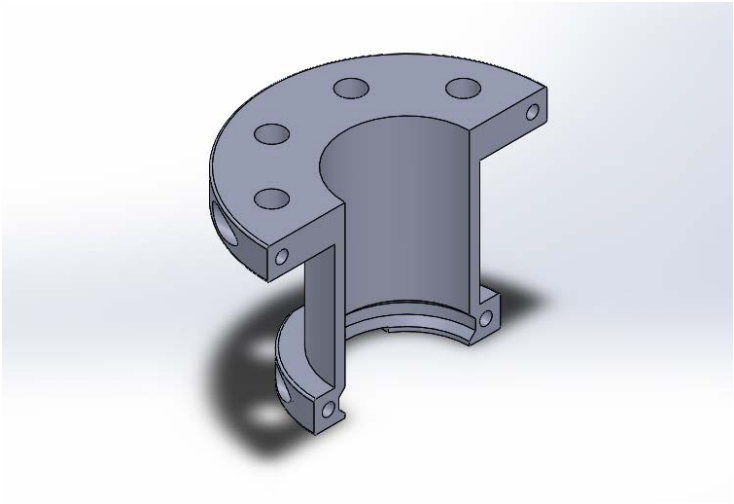


Ilustración 6.2 - Acople (parte simétrica)

Los planos con todas las medidas y especificaciones se adjuntan en el *Anexo Nº5*.

Debido a que las medidas en la parte del Powerball se tuvieron que tomar manualmente se cometieron ciertos errores que luego hubo que subsanar haciendo un refrentado en esa cara eliminando medio milímetro en el taller del S.A.I.T., y luego haciendo un pulido de alta calidad en el Departamento de Ingeniería de Materiales y Fabricación para reducir la fricción con la última articulación del robot.

6.2. Soporte para el sistema de visión

De cara a tener un sistema de visión regulable en altura también se diseñó un soporte consistente en 2 piezas, una de las cuales se mantiene fija al soporte del robot y la otra se desplaza longitudinalmente por una ranura ubicada en el extremo de la otra.

Para fijar una altura determinada las piezas disponen de multitud de orificios para fijarlos con tornillería adecuada. El ensamblaje de ambas piezas tiene el aspecto que se muestra en la Fig. 6.3.

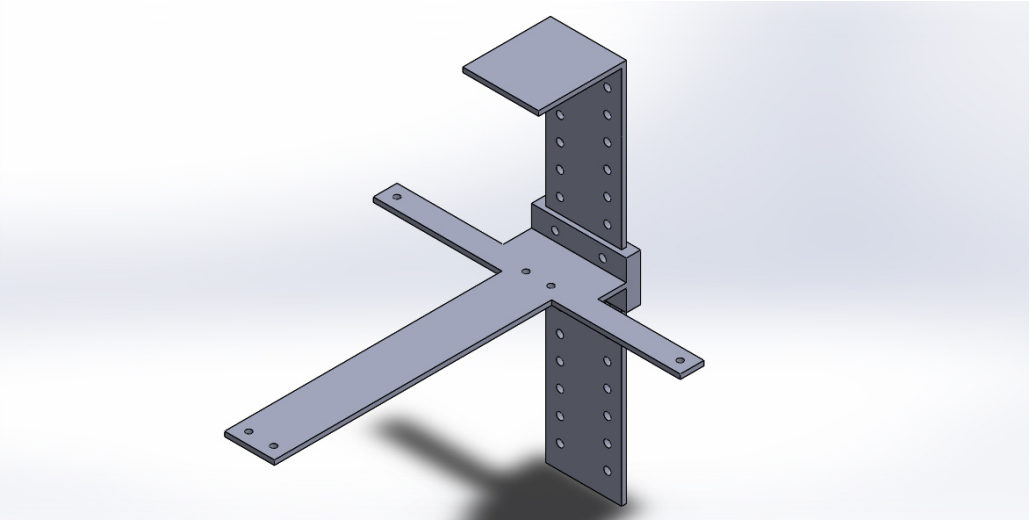


Ilustración 6.3 - Soporte del Sistema de Visión

Los planos de las dos piezas ilustradas arriba se adjuntan en el *Anexo Nº6*. La pieza se ha llevado a cabo en acero inoxidable, pues aporta la rigidez y estabilidad necesarias, no siendo el peso de la misma una característica crítica.

Capítulo 7

VISIÓN ARTIFICIAL

7. Visión artificial

En este capítulo se abordará el software utilizado para el sistema de reconocimiento de objetos. Se verán las posibles alternativas que se estudiaron en un principio y cuál finalmente fue el seleccionado para la aplicación que nos compete.

7.1. Introducción

La visión artificial es un campo que incluye métodos para adquirir, procesar, analizar y comprender imágenes y, en general, metadatos del mundo real con el fin de producir información numérica o simbólica, por ejemplo, en las formas de decisión. Un punto importante en el desarrollo de este campo ha sido el de duplicar la capacidad de la visión humana por vía electrónica percibiendo y comprendiendo una imagen. Esta interpretación de imágenes se puede ver como la extracción de información simbólica a partir de los datos contenidos en la imagen utilizando modelos basados en conocimientos geométricos, físicos, estadísticos, así como en la teoría del aprendizaje. La visión artificial también ha sido descrita como la empresa de automatización e integración de una amplia gama de procesos y representaciones para la percepción visual.

Como disciplina científica que es, la visión artificial se refiere a la teoría de los sistemas artificiales que extraen información de las imágenes. Los datos a registrar pueden tomar muchas formas, tales como secuencias de vídeo, puntos de vista de varias cámaras o datos multidimensionales desde un escáner médico. Como disciplina tecnológica, la visión artificial trata de aplicar sus teorías y modelos para la construcción de sistemas de visión por computador.

Sub-dominios de la visión artificial incluyen la reconstrucción de escenas, detección de eventos, seguimiento por video, reconocimiento de objetos, aprendizaje, indexación, estimación de movimiento, y restauración de la imagen.

La visión artificial es un campo que se ha expandido de manera espectacular por en nuestra vida cotidiana sin siquiera darnos cuenta. No es una disciplina científica e ingenieril que quede confinada en complejos industriales o

laboratorios de investigación. Sin ir más lejos, en la palma de nuestra mano tenemos multitud de algoritmos de visión que nos facilitan la vida. Es el caso de los archiconocidos smartphones. En ellos tenemos aplicaciones capaces de reconocer rostros a la hora de tomar fotografías, hacer un matching de patrones para poder llevar a cabo una imagen panorámica, detección de objetos en aplicaciones de realidad aumentada, detección de personas, etc.

Ilustración 7.1 - Detección de objetos en una aplicación de realidad aumentada

En la industria es ampliamente utilizada para tests de calidad, conteo de unidades en una cinta transportadora de alta velocidad, clasificación de productos por forma y color, etc.

7.2. Sistemas de visión alternativos disponibles

En el momento en el que se definió el proyecto se optó por el uso de visión artificial como sistema de adquisición de datos del entorno para poder determinar la ubicación del objeto que se desease coger y desplazar el extremo del brazo robot hasta la citada posición.

En dicho momento se estudiaron las posibles alternativas para poder llevar a cabo la tarea. Como primeras aproximaciones estudió el uso de Matlab y OpenCV.

7.2.1. Matlab

Matlab es un potentísimo paquete de software científico e ingenieril ampliamente usado en todo el mundo. Matlab se divide toolboxes (paquetes de herramientas) de los que se sirve el núcleo de la aplicación para llevar a cabo todos los cálculos que se le requieran.

Matlab también implementa su propio lenguaje de programación, en caso de necesitar crear aplicaciones ejecutadas dentro de él y valiéndose de su potencia y utilidades de cálculo. Este lenguaje de programación deriva de C y las diferencias no escapan más allá de sutiles diferencias en la indexación de vectores, la puntuación o la definición de bloques de control.

En nuestro caso el toolbox que analizamos fue el de Visión Artifial (Computer Vision System Toolbox™), enfocado al tratamiento de imágenes.

El Computer Vision System Toolbox™ proporciona algoritmos, funciones y aplicaciones para el diseño y simulación de la visión por computador y sistemas de procesamiento de vídeo. Se puede llevar a cabo la detección de objetos y su rastreo, detección de características y extracción de las mismas, comparación de características para determinar correspondencia, visión estéreo, calibración de la cámara, y las tareas de detección de movimiento. El toolbox también proporciona herramientas para el procesamiento de vídeo, incluyendo E/S, visualización de vídeo, anotación en objetos, diseño de gráficos y composición. Los algoritmos están disponibles como funciones de MATLAB, Objetos de Sistema™, y bloques Simulink®.

Esta opción, aun siendo muy potente y tener la ventaja de que ya habíamos trabajado con Matlab en otros trabajos, no era viable por la casi imposible comunicación con ROS. Disponíamos de Matlab en Windows y ROS ha de ser ejecutado Linux para que funcione correctamente. Además habría que desarrollar una interfaz de comunicación que excede sobremanera los objetivos del proyecto.

7.2.2. OpenCV

La segundo opción que se barajó para acometer la tarea de la detección de objetos en un entorno de trabajo y la obtención de sus coordenadas fue OpenCV, utilizado de forma masiva en el ámbito científico técnico, tanto por sus grandes capacidades como por ser de libre distribución.

OpenCV (Open Source Computer Vision Library) es una librería de código fuente abierto de visión por computador y software de aprendizaje automático. OpenCV fue construido para proporcionar una infraestructura común para aplicaciones de visión por computador y para acelerar el uso de la visión artificial

en los productos comerciales. Al ser un producto de licencia BSD, OpenCV hace que sea fácil para las empresas utilizar y modificar el código a su conveniencia.

La biblioteca cuenta con más de 2.500 algoritmos optimizados, que incluye un amplio conjunto de algoritmos de visión por computador y aprendizaje automático con tecnología de última generación. Estos algoritmos se pueden utilizar para detectar y reconocer rostros, identificar objetos, clasificar las acciones humanas en videos, seguir movimientos de cámara, seguir objetos en movimiento, extraer modelos 3D de objetos, producir nubes de puntos 3D de cámaras estéreo, unir imágenes para producir una imagen de alta resolución de una escena completa, encontrar imágenes similares de una base de datos, eliminar los ojos rojos de las imágenes tomadas con flash, seguir los movimientos de los ojos, reconocer paisajes y establecer marcadores para cubrir la imagen en aplicaciones de realidad aumentada, etc. OpenCV tiene más de 47 mil personas en su comunidad de usuarios y el número estimado de descargas es superior a 7 millones. La librería se utiliza ampliamente en las empresas, grupos de investigación y por los organismos gubernamentales.

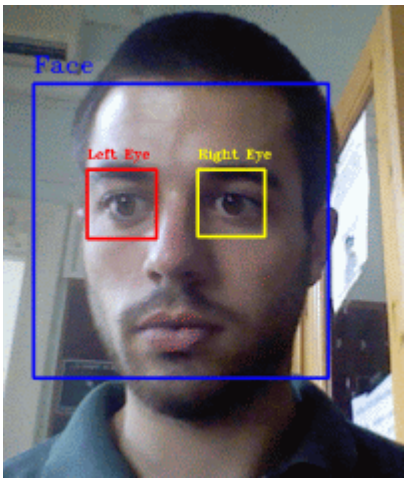


Ilustración 7.2 - Detección ocular y facial con OpenCV

Junto con empresas bien establecidas como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota, que emplean la librería, hay muchas nuevas empresas como Applied Minds, VideoSurf y Zeitera, que hacen un amplio uso de OpenCV. Usos derivados de OpenCV abarcan todo el rango de aplicaciones, como juntar imágenes de streetview, detectar intrusiones en vídeos de vigilancia en Israel, equipos de vigilancia de minas en China, ayudar a los robots en la navegación y la recogida objetos en Willow Garage, detección de piscina

accidentes por ahogamiento en Europa, genarar arte interactivo en España y Nueva York, la comprobación pistas para escombros en Turquía, la inspección de las etiquetas de los productos en las fábricas de todo el mundo o la detección rápida de rostros en Japón.

Cuenta con interfaces de C++, C, Python, Java y MATLAB y es compatible con Windows, Linux, Android y Mac OS. OpenCV se inclina principalmente hacia las aplicaciones de visión en tiempo real y se aprovecha de instrucciones MMX y SSE cuando está disponible. Interfaces con todas las funciones CUDA y OpenCL se están desarrollando de forma activa actualmente. Hay más de 500 algoritmos y cerca de 10 veces la cantidad de funciones que componen o apoyan esos algoritmos. OpenCV está escrito de forma nativa en C ++ y tiene una interfaz de plantillas que funciona a la perfección con los contenedores STL.

En un principio se comenzó a trabajar con este sistema, tomando como libro de referencia y aprendizaje *Learning OpenCV Computer Vision with the OpcenCV Library* de Gary Bradski y Adrian Kaebler. Este libro nos confirió gran cantidad de los conceptos que tuvimos que aprender acerca del tratamiento de imágenes para un sistema robótico. La importancia del uso de filtros para la detección de bordes o la calibración de cámaras para su uso en sistemas estereoscópicos fue adquirida a través de la lectura de este libro.

Finalmente este potente sistema se descartó por la problemática que presentaba configurar el sistema de cámaras estereoscópico. Se disponía en el laboratorio de par de cámaras industriales de la firma IDS, en concreto unas uEye, que no eran exactamente el mismo modelo, pues ofrecían resoluciones distintas. El hecho de configurar el sistema y calibrarlo era bastante complejo e inducía bastante error en la alineación de los objetivos ya que el soporte de las cámaras no estaba diseñado con gran precisión.

Por otro lado, aunque existe un puente de comunicación entre OpenCV y ROS, las cámaras industriales anteriormente citadas tienen un hardware que no está soportado por los controladores de OpenCV. Requerían del uso de la API de las mismas para poder obtener las imágenes, y posteriormente, portarlas a OpenCV, y finalmente mandar los mensajes necesarios a ROS. Esto suponía un

desarrollo excesivo y sin garantías de funcionamiento, por lo que también se descartó.

7.3. PCL – Point Cloud Library

Y por último llegamos al sistema que finalmente se ha escogido para el desarrollo de este proyecto. Mencionar que hemos sido pioneros en este departamento en el desarrollo de visión artificial, pues este complejo y novedoso método de captación y procesamiento de imágenes no se había llevado a cabo nunca en el Departamento, siendo nosotros los que hemos llevado a cabo un trabajo de investigación para su aprendizaje y puesta en marcha en solitario.

7.3.1. Descripción

El Point Cloud Library (o PCL) es un proyecto abierto a gran escala para el procesamiento de imágenes 2D/3D y de nube de puntos. El marco de trabajo PCL contiene numerosos algoritmos de última generación, incluyendo el filtrado, estimación de características, reconstrucción de superficies, registro, ajuste de modelos y segmentación. Estos algoritmos se pueden utilizar, por ejemplo, para filtrar los valores extremos de los datos con ruido, unir nubes de puntos 3D, los segmentar partes relevantes de una escena, extraer puntos clave y calcular descriptores para reconocer objetos en el mundo basándose en su apariencia geométrica, o crear superficies a partir de nubes de puntos y visualizarlos, por nombrar algunos.

PCL es liberado bajo los términos de la tercera cláusula de la licencia BSD y es un software de código abierto. Es gratuito para el uso comercial y la investigación.

PCL es multiplataforma, y ha sido compilado e implementado satisfactoriamente en Linux, Mac OS, Windows y Android / iOS. Para simplificar el desarrollo, PCL se divide en una serie de librerías de código más pequeños, que pueden ser compiladas por separado. Esta modularidad es importante para la distribución de PCL en plataformas con limitaciones computacionales o de tamaño limitado. El la Fig. 7.3 tenemos un gráfico de cómo dichas librerías se interconectan y dependen unas de otras.

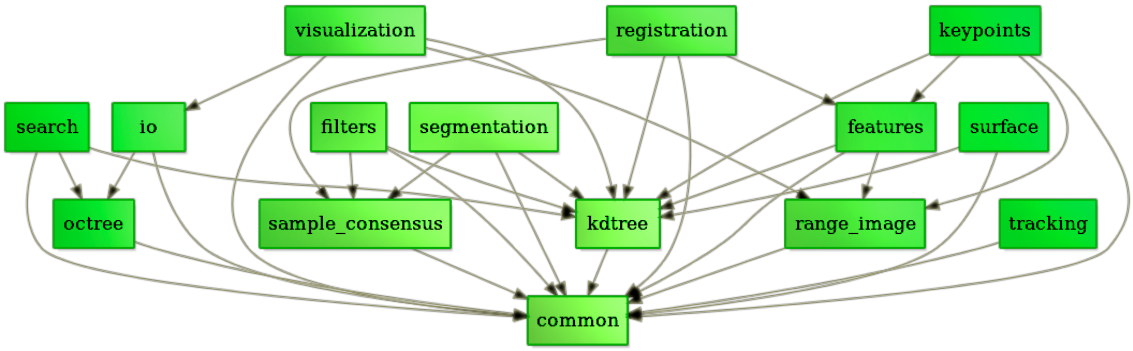


Ilustración 7.3 - Dependencias en las librerías de PCL

PCL es un proyecto ambicioso y con muy buenas expectativas de futuro. Es por esto que está siendo desarrollado por una ingente cantidad de empresas, universidades e institutos tecnológicos. En la Fig. 7.4 se puede ver quiénes son los implicados en el desarrollo de este gran sistema de visión.

Como se observa el número de implicados es enorme, contando con multinacionales de primer nivel como Google o la empresa de sistemas de visualización con la tecnología más puntera del mercado, Nvidia.

Ilustración 7.4 - Desarrolladores de PCL

7.3.2. Estructura de PCL

Lo novedoso del sistema que se presenta es la forma en la que se estructura la información. Una nube de puntos es una estructura de datos utilizada para representar una colección de puntos multidimensionales y se utiliza comúnmente para representar datos en tres dimensiones. En una nube de puntos 3D, los puntos representan generalmente las coordenadas geométricas X, Y, y Z de una superficie

muestreada subyacente. Cuando la información de color está presente, la nube de puntos se convierte en 4D.

Las nubes de puntos se pueden adquirir a partir de sensores, tales como cámaras estéreo, escáneres 3D, o cámaras de tiempo de vuelo, o generados a partir de un programa de ordenador de forma sintética. PCL soporta de forma nativa las interfaces OpenNI 3D, y por lo tanto se puede adquirir y procesar datos de dispositivos tales como las cámaras PrimeSensor 3D, la Microsoft Kinect, o la Asus XTionPRO.

Éste fue uno de los puntos que nos hizo decantarnos por el uso de esta tecnología. También el hecho de haber acudido al ROS-RM 2014 Workshop en la Universidad de Alicante el 23 de julio de 2014, donde Federico Tombari, una eminencia a nivel internacional en lo que a visión 3D se refiere, especialmente PCL, y algunos otros asistentes, nos recomendaron encarecidamente el uso de un sensor Kinect por la fluidez en el trabajo y los buenos resultados arrojados a un coste muy bajo.

7.3.3. Instalación y primeros pasos

En el *Anexo Nº7* se hace una descripción detallada de cómo se instalan las librerías en PCL y cuáles son los conceptos básicos que hay que tener en cuenta a la hora de enfrentarse a la creación de un nuevo programa basado en la captación y análisis de una nube de puntos.

Capítulo 8

MOVEIT!

8. Moveit!

Este capítulo se centra en la herramienta que ha permitido interactuar con el brazo Schunk Powerball. Este paquete para ROS es uno de sus puntos fuertes en la robótica de manipulación, pues incluye una serie de librerías e interfaces para el cálculo de trayectorias, posicionamiento, evitación de obstáculos, etc., con un gran potencial y que permiten un rápido desarrollo de proyectos robóticos así como un flujo de trabajo acelerado.

MoveIt! es la vanguardia del software para la manipulación móvil, incorporando los últimos avances en planificación de movimiento, manipulación, percepción 3D, cinemática, control y navegación. Proporciona una plataforma fácil de usar para el desarrollo de aplicaciones de robótica avanzada, evaluando nuevos diseños de robots y productos robóticos de construcción integrada para uso industrial, comercial, I+D y otros dominios.

De cara a la consecución del presente proyecto tuvimos la oportunidad de asistir al ROS-RM Workshop 2014 en la UA. De los ponentes que allí asistieron, todos de primerísimo nivel, uno destacaba por ser uno de los creadores de ROS y del paquete Moveit!, Sachin Chitta. Sachin es el referente internacional en cuanto a la robótica de manipulación y nos dio una clase magistral acerca del sistema desarrollado por él.

Moveit! se está utilizando a gran escala actualmente, y ya no sólo en el campo experimental y de la investigación, sino también en el ámbito industrial por firmas de primer nivel como ABB o KUKA. Como prueba de ello en la siguiente tabla se recogen 65 modelos que ya incorporan Moveit! y trabajan de forma totalmente operativa con él.

ABB IRB2400	ABB IRB6640	Aldebaran Nao
Boston Dynamics Atlas	BioRob Arm	Cerberus
Comau NM45	Cyton Veta	Demining robot
Dr. Robot	Fanuc m10ia	Fraunhofer Care-O-bot
Hoap3	HRP-4 (simulation)	HRP2
IRB2400	Kawada Hiro	Kinova Jaco

KUKA LWR4	KUKA OmniRob	KUKA youBot
Motoman SIA10d (2)	Motoman SIA20	Motoman SIA5
Pi Robot	Pioneer P3AT	Pisa Velvet Gripper
Robonaut2	Schunk 7DOF	Schunk Dextreous Hand
Summit XL-Terabot	TUM Rosie	Universal Robots UR10
Aldebaran Romeo	Arbotix PhantomX Pincher	Barrett WAM
CKBot	ClamArm	CloPeMa Robot
Denso robot (vs060)	DIY Mobile Manipulator	DLR-HIT Hand
Fraunhofer Rob@Work	HDT arm with Base	Hiro (Nextage)
Hubo	iarm ABB	iCub
Korus Homemate robot	KUKA LBR	Kuka Leightweight Arm
Lego NXT	Lyncmotion servo erector set	Meka M3 Robot
Neuronics Katana	PAL Robotics REEM	PAL Robotics REEM-C
Willow Garage PR2	Rethink Robotics Baxter	Robonaut
Schunk LWA	Schunk Powerball	Shadow Robot Arm and Hand
Universal robot UR5	X-WAM	

8.1. Arquitectura del sistema

La arquitectura del paquete es organiza como aparece en el siguiente gráfico:

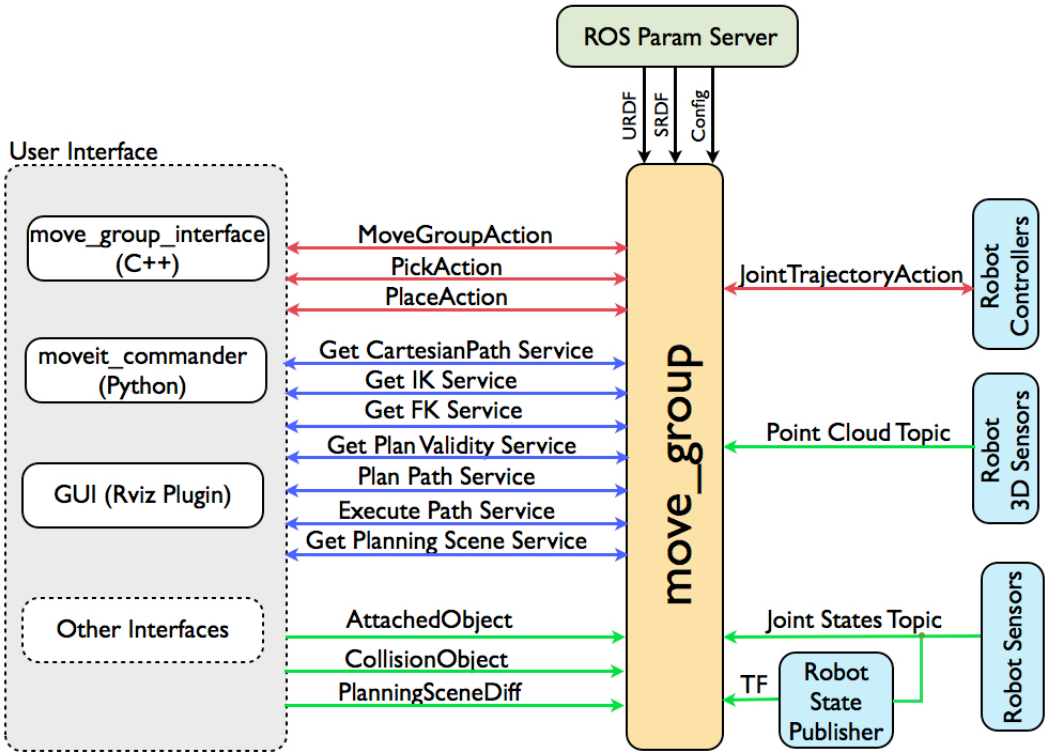


Ilustración 8.1 - Arquitectura de Moveit!

En el gráfico superior se detalla la estructura del nodo principal que ofrece Moveit!, llamado "move_group". Este nodo sirve de integrador, enlazando todos los componentes individuales en un cuerpo que puede ofrecer un conjunto de acciones ROS y servicios para el usuario.

8.1.1. Interfaz de usuario

A la izquierda viene representada la interfaz de usuario. Los usuarios tienen acceso a las acciones y servicios ofrecidos por el nodo de tres formas:

- **Mediante C++**: usando el paquete "move_group_interface" que ofrece una interfaz fácilmente configurable para el control de "move_group" en C++.
- **Mediante Python**: usando el paquete "moveit_commander".
- **Mediante Interfaz Gráfica**: usando el plugin "Motion Planning" para RVIZ

El nodo “move_group” puede ser también configurado usando el servidor de parámetros de ROS para que obtenga de ahí el archivo URDF y SRDF del robot.

8.1.2. Configuración

El nodo “move_group” usa el servidor de parámetros de ROS para obtener estos tres tipos de información:

1. URDF – “move_group” busca el parámetro “robot_description” en el servidor para obtener el archivo URDF del robot.

2. SRDF - “move_group” busca el parámetro “robot_description_semantic” en el servidor para obtener el archivo SRDF asociado al robot. Normalmente el SRDF se crea una vez usando el Asistente de Configuración Moveit!.

3. Configuración Moveit! - “move_group” busca en el servidor otras especificaciones de configuración de Moveit!, como pueden ser límites de articulaciones, cinemáticas, planificaciones de movimiento, percepción, etc. Los archivos de configuración de estos componentes se generan automáticamente a través del Asistente de Configuración Moveit! y son guardados en la carpeta “config” del correspondiente paquete Moveit! para el robot.

8.1.3. Interfaz con el robot

El nodo “move_group” se comunica a través de topics y acciones con el robot real. A través de esta comunicación puede obtener información del mismo (posición de las articulaciones, velocidad de los motores, etc.), registrar nubes de puntos y otra información que puedan enviar los sensores y por supuesto enviarle órdenes a los controladores del robot.

8.2. Planificación de movimientos

Moveit! es capaz de calcular las trayectorias que deben seguir los miembros del robot en función de las especificaciones que se le marquen.

8.2.1. El plugin de planificación de movimientos

Moveit! trabaja con planificadores de movimiento a través de plugin interfaz. Este le permite comunicarse y usar diferentes planificadores procedentes de diferentes librerías, haciéndolo fácilmente ampliable. La interfaz con el planificador se lleva a cabo a través de un servicio o acción de ROS (ofrecida por el nodo "move_group"). Los planificadores por defecto para "move_group" están configurados usando OMPL y la interfaz de Moveit! para OMPL creada con el Asistente de Configuración.

8.2.2. La solicitud de planificación de movimiento

La solicitud planificación de movimiento especifica claramente lo que se desea que el planificador de movimiento haga. Normalmente, se le pedirá al planificador de movimiento mover un brazo a una ubicación diferente (en el espacio articular) o el efector final a una nueva pose. Las colisiones se comprueban por defecto (incluyendo auto-colisiones). Puede adjuntarse un objeto al efector final (o cualquier parte del robot), por ejemplo, si el robot recoge un objeto. Esto permite que el planificador de movimiento tenga en cuenta el objeto durante el movimiento. También se pueden especificar restricciones para comprobar las limitaciones consustanciales proporcionadas por MoveIt!, como por ejemplo:

- Limitaciones de posición - restringen la posición de un enlace a estar dentro de una región del espacio.

- Limitaciones Orientación - restringen la orientación de un enlace dentro de los límites de roll, pitch o yaw.

- Limitaciones Visibilidad - restringen un punto en un eslabón dentro de los límites del cono de visibilidad para un sensor concreto.

- Restricciones de articulación - restringen una articulación a estar entre dos valores

- Limitaciones especificadas por el usuario – se pueden especificar limitaciones propias a través de un callback definido por el usuario.

8.2.3. Resultado de la planificación de movimiento

El nodo “move_group” generará una trayectoria deseada en respuesta a la solicitud. Esta trayectoria moverá el brazo (o cualquier grupo de articulaciones) a la ubicación deseada. Hay que tener en cuenta que el resultado de “move_group” es una trayectoria y no sólo un camino. “move_group” utilizará las velocidades y aceleraciones máximas deseadas (si se especifica) para generar una trayectoria que obedece a las limitaciones de velocidad y aceleración a nivel de articulaciones.

8.2.4. OMPL

OMPL (Open Library Planning Library) es una librearía de planificación del movimiento de código abierto que implementa principalmente planificadores de movimiento aleatorios. MoveIt! integra directamente OMPL y utiliza los planificadores de movimiento de dicha librería por defecto. Los planificadores en OMPL son abstractos; es decir, OMPL no tiene ningún concepto del robot. En lugar de ello, MoveIt! configura OMPL y proporciona el back-end a OMPL para trabajar con problemas en robótica.

8.3. Cinemática

8.3.1. El plugin de cinemática

MoveIt! utiliza una infraestructura de plugins, especialmente dirigida a permitir a los usuarios escribir sus propios algoritmos de cinemática inversa. La cinemática directa y la determinación de los jacobianos se integran dentro de la propia clase “RobotState”. El plugin por defecto de cinemática inversa para MoveIt! se configura mediante un programa de cálculo numérico basado en el jacobiano. Este plugin se configura automáticamente en el Asistente de Configuración de Moveit!. El plugin que elegiremos por defecto será el conocido como KDL.

8.3.2. El plugin IKFast

A menudo, los usuarios pueden optar por aplicar sus propios algoritmos cinemáticos, por ejemplo, el PR2 tiene sus propios algoritmos. Un enfoque popular

para la implementación de un algoritmo cinemático de este tipo es utilizar el paquete IKFast para generar el código C++ necesario para trabajar con nuestro robot particular.

Para profundizar más en la arquitectura y los componentes de Moveit!, así como su funcionamiento, se recomienda visitar el siguiente enlace:

http://moveit.ros.org/documentation/concepts/

8.4. El Asistente de Configuración Moveit!

Para poder trabajar con cualquier robot, Moveit! incorpora un asistente que nos permite configurar todos los detalles necesarios para generar los archivos que le servirán al paquete para poder reconocer y comunicarse con dicho robot. Este asistente trabaja a través de una interfaz gráfica que hace más llevadero este proceso.

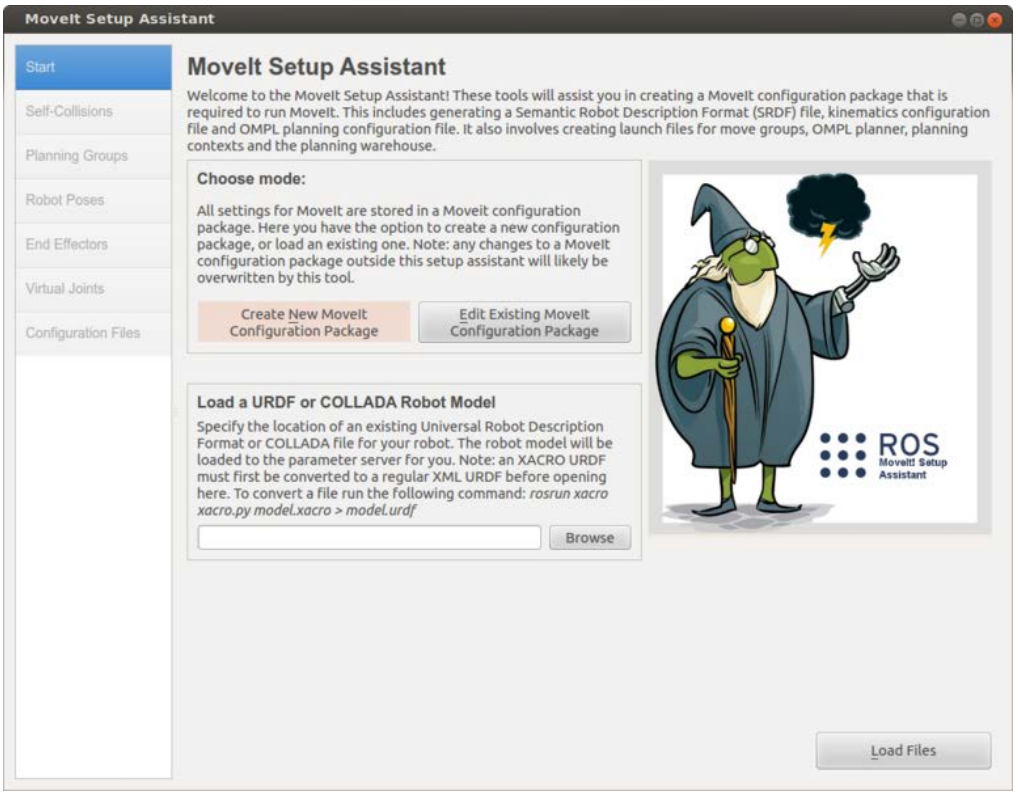


Ilustración 8.2 - Asistente de Configuración Moveit!

Como se puede ver, nos permite crear un nuevo paquete o modificar uno existente. En el caso de querer crear uno sólo debemos cargar el archivo URDF

que describe el robot. En caso de querer modificar un paquete existente sólo habrá que cargar la dirección en la que se creó anteriormente dicho paquete.

Existen una serie de botones en la parte izquierda de la ventana que sirven para configurar el robot. En ellos podemos:

- Generar la matriz de auto-colisiones: permite saber si entre todos los eslabones y articulaciones se puede producir alguna colisión en un movimiento dado.

- Definir un grupo: permite definir un conjunto de articulaciones sobre las que se podrá definir una trayectoria.

- Definir poses del robot: permite guardar posiciones definidas del robot que sean útiles, como reposo, plegado, completamente estirado…

- Definir efectores finales: es posible añadir al extremo del robot la herramienta que vaya a portar para poder calcular con precisión el posicionamiento, en caso de tener que llevar a cabo tareas de recogida, atornillado, etc.

- Definir articulaciones virtuales: es posible añadirle al robot articulaciones virtuales, articulaciones que no existen físicamente. Esto permite crear una articulación ficticia que en los parámetros del robot servirá como punto de anclaje con el mundo físico a la hora de definir las trayectorias.

- Generar los archivos: el último botón de la columna nos permite generar todos los archivos necesarios para crear el paquete Moveit! asociado a nuestro robot en la carpeta que nosotros deseemos. Es muy importante añadir dicho paquete al ~/.bashrc tras generarlo o no podremos usarlo. No se dispone de archivo “setup.bash”, por lo que habrá que usar la directiva *export*.

8.5. Instalación, configuración y primeros pasos

En el *Anexo Nº8* se adjunta una guía de instalación de Moveit! y su configuración.

Capítulo 9

SOFTWARE DESARROLLADO

9. Software desarrollado

En este capítulo se desglosará todo el software que se ha desarrollado en el proyecto. Se explicará en detalle el funcionamiento del mismo y algunas de las características más reseñables del código. El software se puede dividir en 3 partes: visión artificial, robot y sincronizaciones.

9.1. Visión Artificial

El código del sistema de visión se divide en 4 archivos. Uno principal que se encarga de crear los ficheros que guardarán la información obtenida por el reconocimiento de la cámara, llamar al programa de reconocimiento 3D y al de control del brazo y hacer de interfaz de usuario para que éste elija el objeto que se quiere coger de entre los que se han detectado.

El segundo programa es el encargado de obtener las imágenes de la cámara. Con él se consigue la información necesaria para crear una pequeña base de datos de objetos y escenas con las que posteriormente se podrá trabajar. Este programa se ha creado en base al código ofrecido en el siguiente enlace, el cual ha sido ligeramente modificado para adecuarse a los fines del presente proyecto.

http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI_tutorial_1:_Installing_and_testing

El tercer programa es el que se encarga de extraer de una escena cualquiera los objetos que se encuentran depositados sobre la mesa. Es capaz de discretizar la información y obtener los clústers individuales que conformarán los modelos que servirán para poder identificar los objetos que se encuentren en la escena experimento.

El último programa es el que realmente hace el reconocimiento de objetos. Este programa trabaja con modelos y escenas almacenados en unas carpetas específicas y acaba escribiendo en un fichero las coordenadas del centroide de los objetos detectados en la escena.

A continuación se presenta un flujograma simplificado del modo de operación del sistema de visión que se ha diseñado:

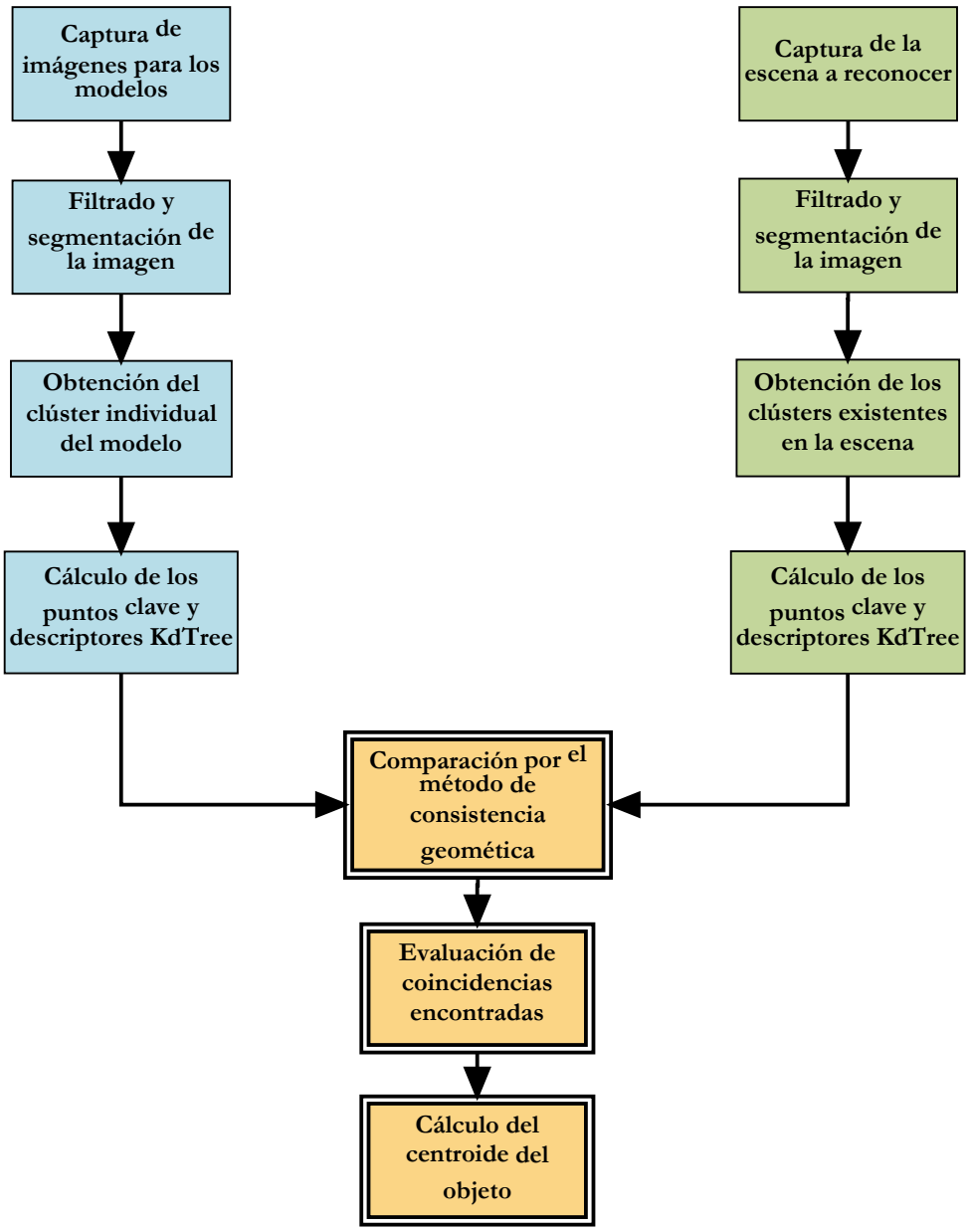


Ilustración 9.1 - Flujograma del programa de reconocimiento de objetos 3D

9.1.1. Programa principal

El programa principal se vale de una estructura (`objetos_conocidos`) que se ha diseñado con el fin de identificar los objetos para poder organizar las llamadas a los otros programas que conforman el sistema de visión. Dicha estructura almacena el nombre del objeto, un booleano que confirma si existe en

la escena o no y la ruta para ejecutar el programa de reconocer objetos con los parámetros necesarios.

Esta estructura es usada en una función (`rellenar_objetos`) que se ha creado para rellenar toda esa información dentro de un bucle de control *for*.

El programa al iniciarse lo primero que lleva a cabo es la creación de un archivo de salido donde se guardarán las coordenadas de los objetos detectados (`centroides.txt`). Se crea la estructura de los objetos y se rellena con los valores por defecto.

A continuación se llama al programa de reconocimiento de objetos varias veces dentro de un bucle para que analice la escena comparando con cada uno de los modelos que se tienen y devuelva si dicho objeto existe en la escena. El siguiente paso a seguir es mostrarle al usuario los objetos que se han encontrado en la escena y pedirle cuál quiere recoger. Dependiendo de la respuesta obtenida se hace una llamada al sistema para ejecutar el programa de control del robot con unos parámetros u otros. Si de esta llamada se obtiene una respuesta satisfactoria se entrega un mensaje de que todo se ha llevado a cabo con éxito y si no, se muestra un mensaje de error.

9.1.2. Programa de obtención de imágenes

Este programa genera una interfaz de visualización de la información que está captando el Kinect. Para ello crea una serie de objetos nube de puntos con información RGB en los cuales se guardará la escena que se desea capturar.

El visualizador incluye una serie de opciones para reiniciarlo, cerrarlo u obtener una imagen. Este último proceso se realiza cuando se pulsa la barra espaciadora. Cada vez que dicha tecla es pulsada el programa transforma la nube de puntos asociada en archivo con extensión ".pcd" que guardará la información de la escena en la carpeta que se especifique en la ruta.

Este programa también puede utilizarse como visualizador de escenas que se hayan capturado con anterioridad, simplemente pasándole como parámetro "-v" y la ruta del archivo *pcd*.

Para más información de cómo funciona el código se recomienda ver el archivo *cpp* en detenimiento.

9.1.3. Programa de obtención de modelos individuales

Para el diseño de este programa se han creado 3 funciones que son llamadas durante la ejecución del *main*.

La función `eliminar_elementos_fuera_mesa` consigue quitar información irrelevante de la escena. Lo primero que hace es eliminar los puntos que quedan fuera de la nube de puntos en base a una distancia umbral que se le defina. A continuación le aplica un filtro a la imagen, el cual permite eliminar la información que no se encuentre en el rango de 0,9 a 1,3 m del sensor. Con esto se consigue acotar la escena a la ubicación espacial de la mesa. Por último esta función guarda en fichero la nube de puntos filtrada.

La función `suavizar_nube` coge la nube de puntos y le aplica un suavizado en los puntos que la componen, eliminando el ruido que pueda llevar por errores de captación por parte del sensor. Así en posteriores tratamientos se podrán extraer mejor los modelos individuales.

La función `count_Input_Files` simplemente se encarga de saber cuántos archivos hay en un directorio que se le pase al programa como parámetro para saber cuántas veces debe ejecutarse el programa. Esto es muy útil pues permite trabajar con todos los archivos que queramos de una vez.

El programa comienza evaluando los archivos que tiene que tratar mediante la función anteriormente explicada y establece un bucle *for* con las repeticiones pertinentes. Se carga la nube de puntos y se le aplican los dos filtros que se han explicado al comienzo de este apartado.

Para poder separar objetos dentro de una escena es necesario hacer una segmentación. En este caso se ha llevado a cabo una segmentación RANSAC de tipo planar. Con ella se consigue eliminar el plano dominante de la escena, en este caso la mesa. En un principio, antes de diseñar la función `eliminar_elementos_fuera_mesa` se tuvo que llevar a cabo más de una vez dicha segmentación para poder eliminar la pared y la mesa. Gracias a dicha función el código ha sido optimizado con respecto a su tiempo de ejecución.

Al final del programa se ha diseñado un algoritmo bastante complejo para el guardado de los clústers segmentados que los organiza en múltiples carpetas y les asigna nombre y numeraciones en concordancia a su ubicación, así como en base al tratamiento que han recibido.

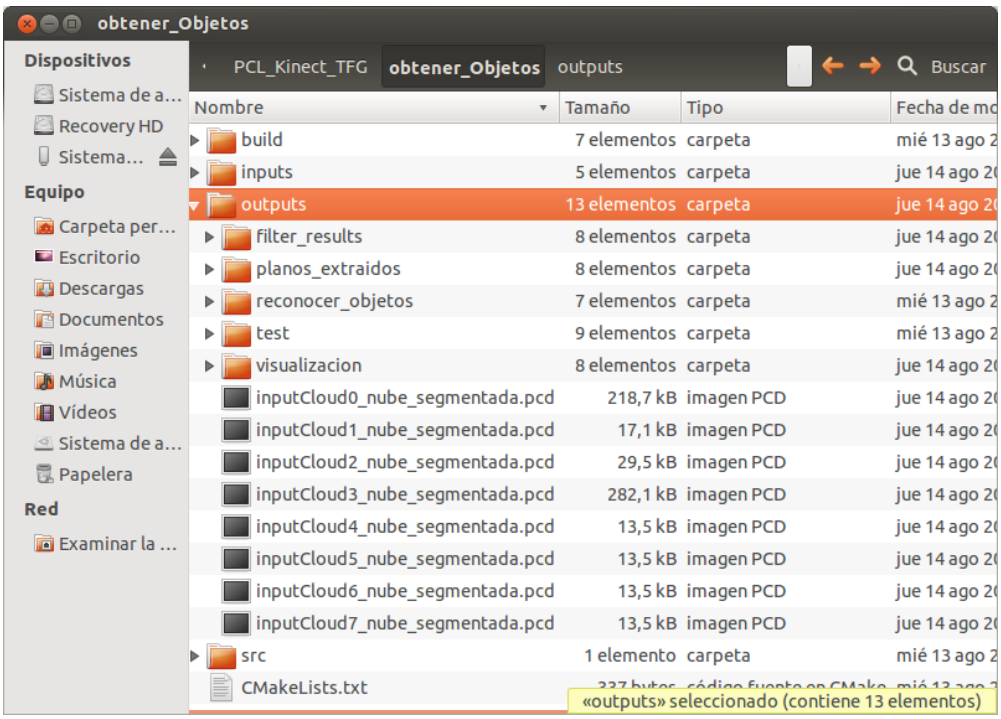


Ilustración 9.2 - Estructura de guardado de archivos

9.1.4. Programa de detección de objetos 3D

Este programa se encarga de identificar los objetos presentes en una escena con los modelos obtenidos anteriormente y guardados en una base de datos.

El programa recibe como parámetros la escena que se vaya a analizar y el modelo que se quiere buscar. Para facilitar la identificación de la escena se le pasa un filtro en el cual se elimina el suelo y la pared del fondo, quedándose sólo la mesa con los objetos.

A continuación se calculan las normales de la superficie que generaría un punto con sus vecinos más próximos. También se le disminuye la resolución tanto a la escena como a los modelos para agilizar los cálculos de la extracción de los puntos clave que es llevada a cabo justo después. Los puntos clave serán puntos de referencia en la nube que permitirán compararla con otra para evaluar si existen coincidencias.

Los procesos de identificación de objetos necesitan de un descriptor que contenga las características del modelo que representan. Existen varios tipos de descriptores en función del algoritmo de identificación que utilizan. En este caso se utiliza el KdTree asociado a los puntos clave. Se buscan todos los vecinos cercanos que no exceden una distancia especificada y se guarda dicha información en un vector de correspondencias. Estas operaciones se llevan a cabo tanto en la escena como en el modelo facilitado.

Con este último proceso ya se puede proceder a la creación de los clústers. Los clústers pueden ser comparados por varios métodos. Durante la realización del proyecto se utilizó el algoritmo Hough 3D y el de consistencia geométrica, dando mejores resultados el segundo.

Por último el programa muestra por pantalla los resultados obtenidos de la comparación de los clústers. Se indica si se han encontrado coincidencias con alguno de los objetos de la escena y en tal caso se calcula la matriz de rotación y traslación del objeto con respecto al modelo, es decir, muestra los giros y desplazamientos que ha habido que aplicarle al modelo para que se encaje en el objeto encontrado en la escena. Los modelos se han generado alineados con el eje focal de la cámara. En una ventana se visualiza la escena con el objeto detectado en color rojo para que el usuario pueda comprobar que la detección se ha realizado correctamente.

La última operación que se lleva a cabo es la de calcular el centroide de la nube de puntos asociada al objeto detectado y guardar esa información en un fichero, que servirá de base de datos para el programa controlador de la mano.

A modo de conclusión del apartado, señalar que el índice de aciertos, una vez ajustados los parámetros es del 99%. La única corrección o solución que tuvo que plantearse fue la incorporación de un segmento transversal al bote de ambientador ya que éste era confundido en numerosas ocasiones con la botella debido a la resolución y ruido del sensor.

El resultado que obtendríamos por pantalla tras la detección sería el siguiente:

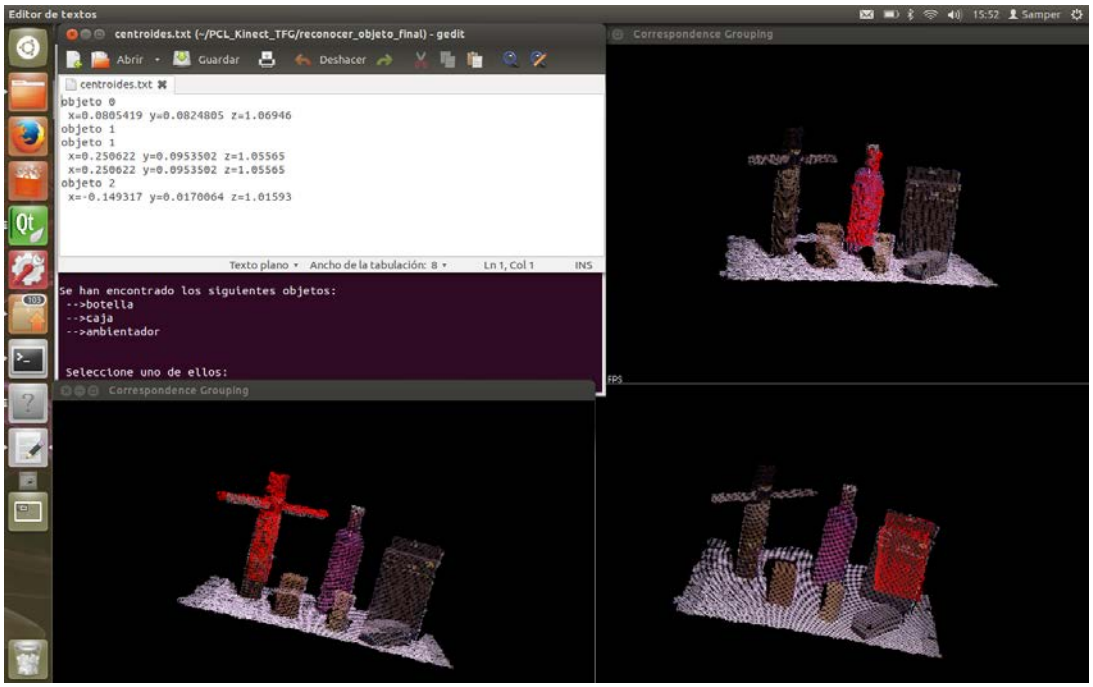


Ilustración 9.3 - Resultados de la detección en 3D

9.2. Robot Schunk Powerball

El programa que controla el robot se ha diseñado utilizando la interfaz *move_group*. Esta interfaz está formada por un conjunto de clases que permiten interactuar con todos los elementos del robot, definir posiciones, restricciones, obstáculos, etc. Una librería muy completa que llegar a conocerla a la perfección requiere de mucha experiencia y trabajo.

9.2.1. Funciones propias creadas

Al margen de la ejecución del *main* se han creado dos funciones auxiliares a las cuales se les llama en diversas ocasiones durante la ejecución del programa.

La primera de ellas es una función que permite visualizar la trayectoria que Moveit! calcula para una posición final dada. Esta función recibe como parámetros varios objetos asociados a la creación de la trayectoria del robot, como el plan o los mensajes de publicación de estado. Gracias a un *flag* que se le pasa es capaz de decir al usuario si la trayectoria se ha podido calcular correctamente o no. En caso afirmativo la muestra en RVIZ y en caso contrario comunica el error.

La segunda función es la encargada de transformar las coordenadas obtenidas del centroide del objeto a agarrar por la Kinect al sistema de referencia asociado al robot. La transformación matricial a la que se somete el punto que llega como parámetro tiene en cuenta el cambio de sistemas de referencia entre la Kinect y el brazo, y además entre el brazo y la palma de la ShadowHand. Matemáticamente la transformación tendría esta definición:

$$\begin{pmatrix} X_{robot} \\ Y_{robot} \\ Z_{robot} \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c_\alpha & -s_\alpha & 0 \\ 0 & s_\alpha & c_\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_{\pi/2} & 0 & s_{\pi/2} & 0 \\ 0 & 1 & 0 & 0 \\ -s_{\pi/2} & 0 & c_{\pi/2} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c_{\pi/2} & -s_{\pi/2} & 0 & 0 \\ s_{\pi/2} & c_{\pi/2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot$$

$$\cdot \begin{pmatrix} 1 & 0 & 0 & X_{kr} \\ 0 & 1 & 0 & Y_{kr} \\ 0 & 0 & 1 & Z_{kr} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & X_{sh} \\ 0 & 1 & 0 & Y_{sh} \\ 0 & 0 & 1 & Z_{sh} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \\ 1 \end{pmatrix}$$

Se obtienen las coordenadas que utilizará Moveit! para posicionar el robot a través de tres rotaciones que orientan los ejes de coordenadas para que coincidan con los del brazo. El ángulo alfa es la inclinación de la Kinect con respecto a la horizonal.

Posteriormente se le aplican dos transformaciones de traslación. Una con coordenadas “kr” que corresponden al desplazamiento de la base de la cámara a la base del robot. Y otra con coordenadas “sh” pertenecientes a la separación del extremo del robot y la palma de la mano. El resultado del cálculo es guardado en una estructura de dato propia que permite guardas poses (posición y orientación).

9.2.2. Programa principal

El programa principal o *main* se inicia recibiendo como argumento el objeto que haya decido coger el usuario de entre los que ha encontrado el programa de visión. Las primeras instrucciones están destinadas a iniciar el nodo de ROS que soportará la ejecución.

A continuación se define una restricción para el robot en su quinta articulación. Se le especifica que no debe moverse más allá del rango de ±90° a la hora de calcular las trayectorias, ya que se produciría una colisión entre el brazo y la propia mano:

```
//ESTABLECEMOS UN RESTRICCIÓN DE MOVIMIENTO EN LA 5ª ARTICULACIÓN PARA QUE NO CHOQUE LA MANO

moveit_msgs::JointLimits mano;
mano.joint_name = "arm_5_joint";
mano.has_position_limits = true;
mano.min_position = -1.5708;
mano.max_position = 1.5708;
mano.has_velocity_limits = true;
mano.max_velocity = 2.0;
mano.has_acceleration_limits = true;
mano.max_acceleration = 0.8;
```

Ilustración 9.4 - Definición de restricción Moveit!

Lo siguiente que se ejecuta es la lectura desde el fichero generado por el programa de visión de las coordenadas del objeto seleccionado. Este fichero guarda en diferentes líneas consecutivas el nombre del objeto detectado y las coordenadas XYZ de su centroide. Si no se encuentra el objeto en el fichero el programa aborta el experimento. Si por el contrario sí se encuentra, se envían estas coordenadas a la función de transformación explicada en el punto anterior.

Una vez hecho esto se establece la sincronización con la mano, que se explica en detalle en el apartado *9.3 Sincronización*. Tras esto se define la orientación que debe tener el efector final (mano) para coger el objeto.

Las líneas de código posteriores tienen la función de crear todos los objetos necesarios para construir una trayectoria en un conjunto de articulaciones y obtener la información necesaria de ella en tiempo de ejecución. En estas líneas también se inicia el visualizador RVIZ para mostrar en él las trayectorias que se calculen a lo largo del programa.

En cuanto se recibe la confirmación de que la mano está lista (todas las sincronizaciones que no devuelven un valor exitoso abortan la ejecución del programa) se calcula la trayectoria posible, se visualiza y se ejecuta. Una vez completada se le comunica a la mano que ya está en la posición que adecuada para llevar a cabo el agarre.

Si llegados a este punto la mano es capaz de coger bien el objeto se procede a retirarlo de la mesa, levantándolo 2 cm y desplazándolo hacia atrás 20 cm. Una vista en detalle de este código:

```
//Código de espera hasta que la mano ha cogido el objeto
flag = conexion.recibir_datos();
if(flag != 2){
    ROS_INFO("La mano no ha podido llevar a cabo el agarre de forma optima.");
    ROS_INFO("Abortando la operacion...");
    return 0;
}else
 target_pose1.position.x = target_pose1.position.x + 0.02;
 target_pose1.position.y = target_pose1.position.y - 0.2;

 group.setPoseTarget(target_pose1);
 success = group.plan(my_plan);
 visualizar(success, my_plan, display_publisher, display_trajectory);
 group.move();
```

Ilustración 9.5 - Código de retirar el objeto de la mesa

Se le comunica a la mano que el objeto se ha retirado de la mesa y se espera la confirmación de que tras retirar el objeto éste no se ha deslizado de la mano.

Acto seguido se establecen unas nuevas coordenadas objetivo, que serán la posición donde se deberá depositar el objeto agarrado. Se visualiza la trayectoria y se ordena al robot ejecutarla. El programa finaliza cuando, tras haber llegado a la posición indicada, la mano confirma que ha podido soltar el objeto de forma satisfactoria.

9.3. Sincronización

En este apartado se detallan las clases y funciones creadas en para poder establecer la comunicación y sincronización entre el brazo y la mano. En realidad la comunicación es entre los dos ordenadores que controlan la mano y el brazo respectivamente.

9.3.1. Presentación de librerías y funciones principales

Una vez desarrollados y expuestos todos los módulos en los que se desglosa el presente Trabajo de Fin de Grado, queda por definir el nexo de unión que comunica todos estos paquetes: las librerías de comunicación y sincronización.

Para sincronizar ambos robots y comunicar los dos ordenadores responsables del control, se decidió desarrollar un conjunto de librerías que aglutinasen todas las funciones relativas al uso de sockets TCP/IP. Las pautas y premisas consideradas para el desarrollo fueron las siguientes:

- Existen dos programas que involucran comunicaciones: uno de ellos en Debian, encargado de controlar la mano y otro, en Ubuntu, responsable del movimiento del brazo.

- El programa de Ubuntu realizará las operaciones que se consideren oportunas para adquirir los datos del sistema de visión artificial e informar al sistema de Debian del objeto que se debe agarrar.

- Deben definirse pautas y procedimientos que aseguren la sincronización y contemplen medidas de parada segura en caso de fallo.

Partiendo de estas hipótesis y principios de diseño, se generó una librería denominada "socket.h", con funciones para abrir sockets, establecer la comunicación con otro dispositivo y enviar/recibir datos. Aunque es posible encontrar todo el código en el cd adjunto, a continuación se comentarán las funciones y comportamientos más interesantes.

El código de sincronización que se ejecuta dentro del programa de control del robot hace las veces de servidor en la comunicación y el de la mano funciona como cliente. Cuando se crea el objeto de la clase *socket_shadow_powerball* el constructor inicia la comunicación y espera que se conecte algún cliente, en nuestro caso la mano.

De observar el código se extrae que se emplean dos funciones principalmente:

- enviar_datos(). Encargada de enviar la cadena de datos recibida como argumento y esperar una confirmación. El valor de retorno es de tipo booleano e indica si la confirmación ha sido recibida o no.

- recibir_datos(). Encargada de gestionar la recepción de datos. Esta función espera recibir una cadena de datos coincidente con el string recibido como argumento. El valor de retorno es de tipo booleano, devolviéndose false si la cadena recibida no coincide con la esperada o si concluye el tiempo máximo de espera para la recepción.

9.3.2. Flujo de funcionamiento

Se tiene que el flujo de trabajo y funcionamiento es el siguiente:

1. Establecimiento de la comunicación e inicio de las sincronizaciones (todas las funciones necesarias las realiza el constructor de la clase, siendo necesario únicamente verificar el valor de la variable booleana problemas_comunicación a través de la función getProblemas()).

2. El Software de control del brazo recibe la información, del sistema de visión, del objeto a coger (dimensiones, posición en el espacio…) y envía el tipo de objeto a través del socket abierto (función enviar_datos()).

3. El Software de control de la mano recibe los datos sobre el objeto a agarrar y verifica si dicho objeto se encuentra en la base de datos de objetos "agarrables o conocidos" (fichero dimensiones_objeto.txt). Este proceso de recepción extraordinaria y verificación se realiza por medio de las funciones recibirObjeto() y decodificar_objeto().

4. El robot manipulador Shadow Hand inicia los procesos de lectura de sensores y verificación de actuadores.

5. Una vez preparada, la mano comunica al programa de control del brazo su estado: "mano preparada".

6. Tras recibir la orden anterior, el brazo posiciona al manipulador para que realice el agarre, enviando "posición recogida alcanzada" cuando ha finalizado el posicionamiento del efector final.

7. Conocida su correcta posición, la Shadow Hand inicia el proceso de agarre, siguiendo los valores del algoritmo cinemático y estabilizando la presión ejercida acorde a unos valores de referencia.

8. El software de control del manipulador envía al controlador del brazo la sentencia: "agarre efectuado" (función enviar_datos()).

9. El brazo LWA4P eleva la mano dos centímetros y la retrae 20 cm para que el objeto quede en suspensión.

10. Concluido el movimiento 9, el software de control del brazo comunica su estado: "retirada de mesa".

11. El programa encargado de controlar la mano inicia los controles antideslizamiento.

12. Una vez estabilizado y asegurado el agarre, el brazo es informado de la situación: “no deslizamiento”

13. A continuación, el brazo se desplaza a la posición de entrega del objeto y señala al software de control de la mano que debe soltar el objeto: “posición final alcanzada”.

14. La mano procede a soltar el objeto y adoptar la posición inicial. Tras finalizar estas tareas, este módulo del programa global envía “mano terminada” antes detenerse.

15. Una vez recibido el mensaje anterior, el robot adopta la posición inicial, quedando preparado para repetir el proceso.

16. Detención de los motores y fin del programa.

En el siguiente esquema se resumen todos estos pasos, contemplándose únicamente los fallos debidos a comunicaciones.

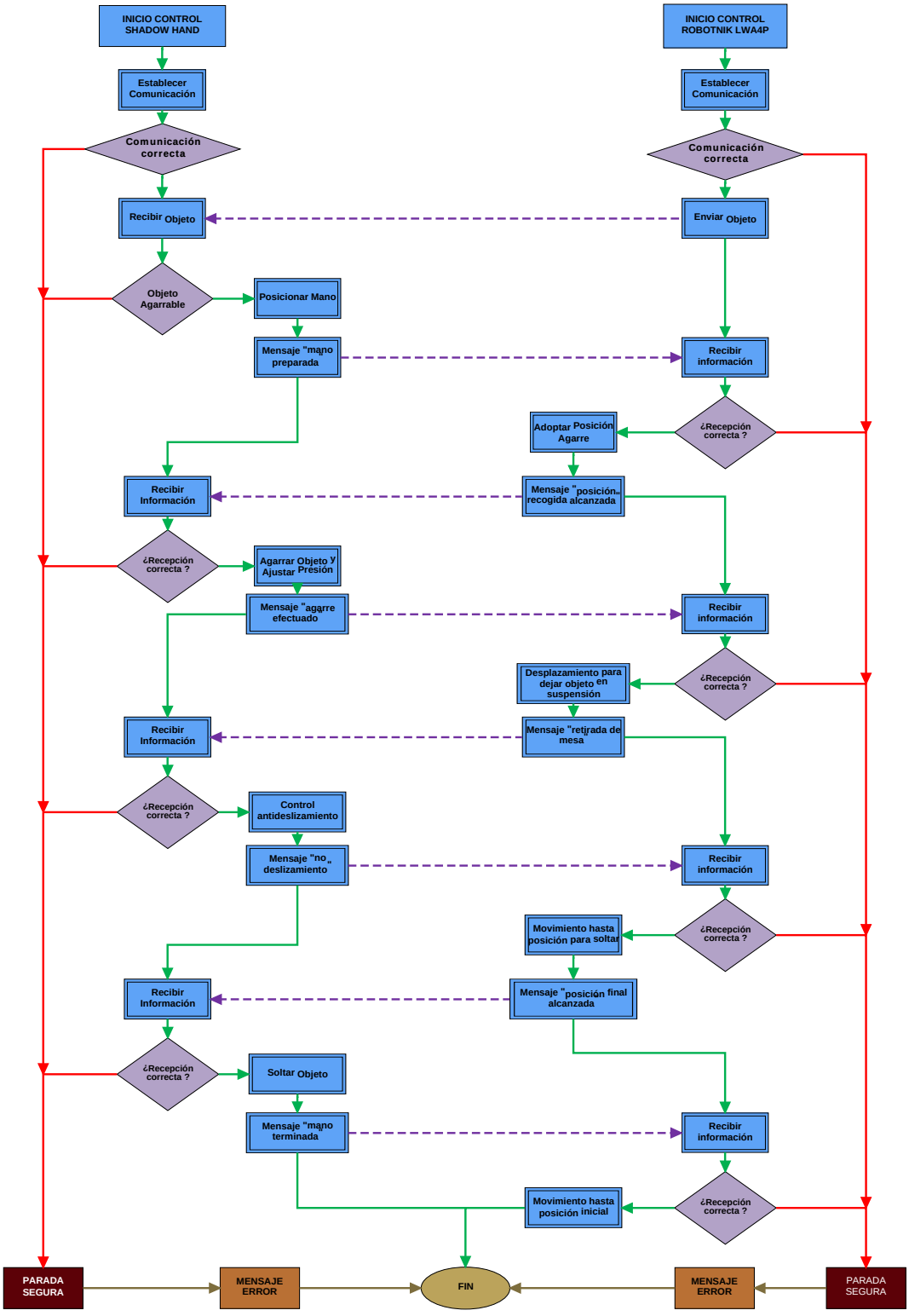


Ilustración 9.6 - Flujograma del sistema de sincronización

9.3.3. Medidas de seguridad y comprobaciones

Como se puede observar en la imagen del código, todas las funciones devuelven un valor booleano. En caso de que algún procedimiento o paso de los

citados en el apartado anterior falle, se dará inicio a un protocolo de finalización consistente en mostrar un mensaje de error y detener el módulo problemático (controlador de la mano o controlador del brazo).

Las comunicaciones se han establecido de modo tal que todos los envíos y recepciones verifiquen el estado del socket. De este modo, si alguno de los extremos se detuviese, el extremo contrario sería consciente de que ha acontecido algún problema e iniciaría una parada segura. Para ello:

- En las funciones de envío de datos se espera que el receptor envíe una confirmación (valor true) en menos de 1 segundo para informar de que no ha habido problemas. En caso de no recibirse, la función devolvería false, provocando que el programa usuario inicie tareas de parada.

- En las funciones de recepción de datos, puesto que se conocen los mensajes a enviar, se comprueba la validez de la información recibida por medio de comparaciones. En caso de recibirse información incompleta o datos inesperados, la función finaliza indicando la existencia de problemas.

Capítulo 10

CONFIGURACIÓN DEL SISTEMA

10. Configuración del sistema

En este capítulo se enumerarán de forma detallada los pasos que se han seguido para la puesta en marcha del proyecto, de manera que cualquier otra persona pueda recrear el experimento a posteriori.

10.1. Instalación de Ubuntu 12.04

Este paso queda definido en el *Anexo Nº1*.

10.2. Instalación de ROS Groovy Galapagos

Este paso queda definido en el *Anexo Nº2*.

10.3. Instalación de PCL

Este paso queda definido en el *Anexo Nº7*.

10.4. Instalación de Moveit!

Este paso queda definido en el *Anexo Nº8*.

10.5. Instalación y configuración del paquete del Schunk Powerball

En este apartado se explicarán todos los pasos a seguir a la hora de instalar y configurar todo el software necesario para controlar el brazo.

10.5.1. Instalación del paquete “schunk_robots”

Lo primero que se debe hacer es crear una carpeta en nuestro directorio personal (llamada robotnik por ejemplo), donde su ubicarán todos los subpaquetes.

```
$ mkdir robotnik
$ cd robotnik
```

A continuación instalamos el paquete de Schunk en dicho directorio.

```
$ rosinstall .
https://raw.github.com/ipa320/schunk_robots/groovy_dev/gr
oovy.rosinstall
```

Es posible que la instrucción anterior devuelva un error diciendo que no se encuentra la distribución de ROS instalada, y que es necesario que se especifique. En tal caso hay que introducir el anterior comando con un modificador al final que establece la distribución que hay instalada.

```
$ rosinstall .
https://raw.github.com/ipa320/schunk_robots/groovy_dev/gr
oovy.rosinstall /opt/ros/groovy
```

A continuación exportamos la ruta del paquete al ~/.bashrc y lo actualizamos.

```
$ echo “export
ROS_PACKAGE_PATH=/home/nombre_usuario/robotnik:$ROS_PACKA
GE_PATH” >>~/.bashrc
$ source ~/.bashrc
```

Para que funcione correctamente el robot se deben instalar algunas dependencias adicionales relacionadas con paquetes de manipulación con brazos robóticos.

```
$ sudo apt-get install ros-groovy-pr2-controllers ros-
groovy-arm-navigation ros-groovy-arm-navigation-
experimental ros-groovy-audio-common ros-groovy-pr2-
power-drivers ros-groovy-pr2-gui
```

Por ultimo hay que construir los paquetes para que ROS pueda hacer uso de ellos.

```
$ rosdep install  schunk_robots
$ rosmake schunk_robots
```

Es posible instalar componentes individuales de los controladores en lugar del paquete completo como se ha hecho. En caso de no querer utilizar ROS se pueden instalar los drivers independientes de bajo nivel "ipa_canopen_core" o instalar únicamente los drivers de bajo nivel para ROS "ipa_canopen_ros".

10.5.2. Instalación y configuración del adaptador PCAN-USB

Lo primero que debemos hacer es descargar la última versión de los drivers de la página oficial de PEAK System. Para ello nos dirigimos al siguiente enlace y la descargamos:

```
http://www.peak-system.com/fileadmin/media/linux/index.htm
```

Actualmente la última versión disponible es la 7.12, descargaremos esa. A continuación es necesario descomprimir el archivo y guardarlo en nuestro directorio personal. Si dejamos el nombre de la carpeta igual al del archivo descargado será "peak-linux-driver-7.12". Siendo así entramos al directorio y ejecutamos la siguiente serie de comandos. Es importante que el dispositivo no esté conectado al ordenador durante esta parte.

```
$ cd peak-linux-driver-7.12
$ make clean
$ make NET=NO_NETDEV_SUPPORT
$ sudo make install
```

Una vez hecho esto es cuando se puede conectar el adaptador al PC. Los siguientes comandos son para verificar el correcto funcionamiento del mismo.

```
$ lsmod | grep pcan
$ ls /dev/pcanusb0
$ cat /proc/pcan
```

Con el útlimo comando introducido deberá aparecer por pantalla un resumen de las especificaciones del driver instalado y su configuración. Lo más importante es que justo debajo del término "-ndev-" esté "-NA-", lo que implicará que la configuración ha sido correcta.

```
Kernel 2.6.24.7-92.fc8
==========================

*------------ PEAK-Systems CAN interfaces (www.peak-system.com) ------------
*-------------------------  Release_20080220_n  -------------------------
*--------------- [mod] [isa] [pci] [dng] [par] [usb] [pcc] ---------------
*--------------------- 1 interfaces @ major 248 found ----------------------
*n -type- ndev --base-- irq --btr- --read-- --write- --irqs-- -errors- status
32   usb  -NA- ffffffff 255 0x001c 00000000 00000000 00000000 00000000 0x0000
```

Este proceso se convertirá en bastante familiar para el usuario pues la interfaz de comunicación CAN tiene una molesta predilección a desconfigurarse cuando se produce alguna actualización en el sistema o cambio en los paquetes de ROS. Se explicará con detalle en el apartado de problemas comunes.

10.5.3. Encendido del robot

Para encender el robot es fundamental que esté conectado a la red eléctrica y que el interruptor general esté encendido. Otro punto fundamental es que la seta de emergencia no se encuentre pulsada o que el botón de rearme no esté encendido. Una vez verificado lo anterior hay que iniciar los controladores del motor y la interfaz de comunicación. Para ello:

```
$ roslaunch schunk_lwa4p lwa4p.launch
```

En un terminal, para arrancar o parar los motores podemos hacerlo a través de llamadas a servicios.

```
$ rosservice call /arm_controler/init
$ rosservice call /arm_controler/halt
```

Si queremos probar las demos que ofrece Schunk, así como ejecutar un monitor de mensajes y procesos del robot que es bastante útil habremos de introducir:

```
$ roslaunch schunk_lwa4p dashboard.launch
```

10.6. Creación del paquete Moveit!

Para poder utilizar el robot es necesario crear un paquete con el Asistente de Configuración Moveit!. Para llevar a cabo este apartado de la configuración se

recomienda haber leído con anterioridad el capítulo *8.4 El Asistente de Configuración Moveit!* donde se explica el asistente y el *Anexo Nº8*.

Para crearlo cargaremos el archivo URDF que se encuentra en:

```
~/robotnik/schunk_robots/schunk_lwa4p/urdf/lwa4p.urdf.xacro
```

Una vez cargado hay que calcular la matriz de auto-colisiones y generar el grupo de movimiento. El grupo lo llamaremos "arm" para que coincida con el que el propio robot tiene definido en sus topics y nodos y que no existan malentendidos.

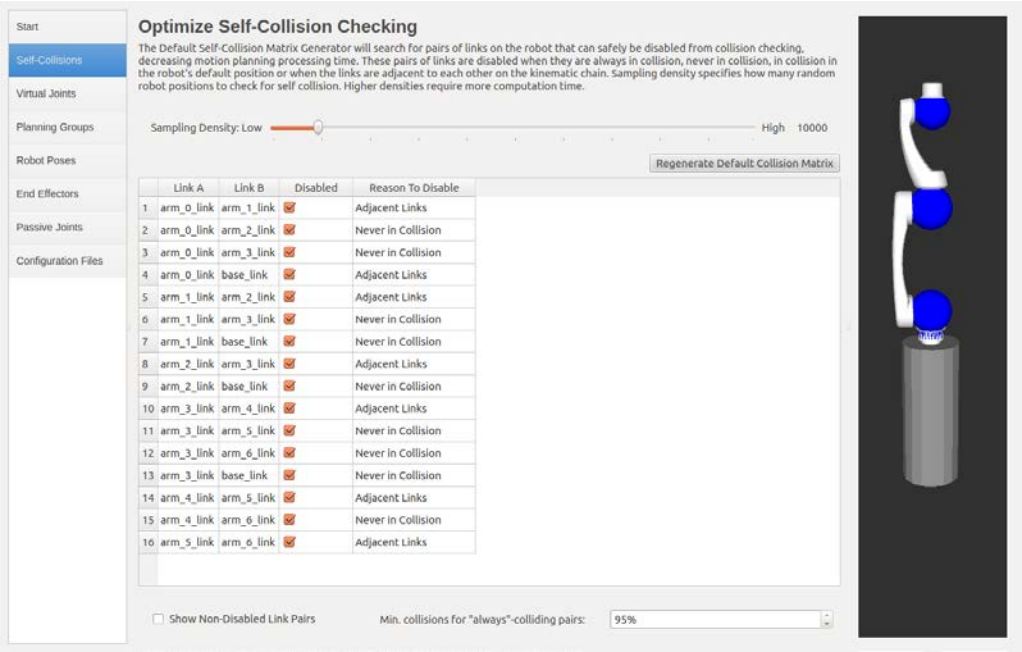


Ilustración 10.1 - Generación de la matriz de colisiones

A la hora de crear el grupo añadiremos todas las articulaciones y eslabones. También se añadirá una cadena cinemática que vaya desde la base al extremo del robot.

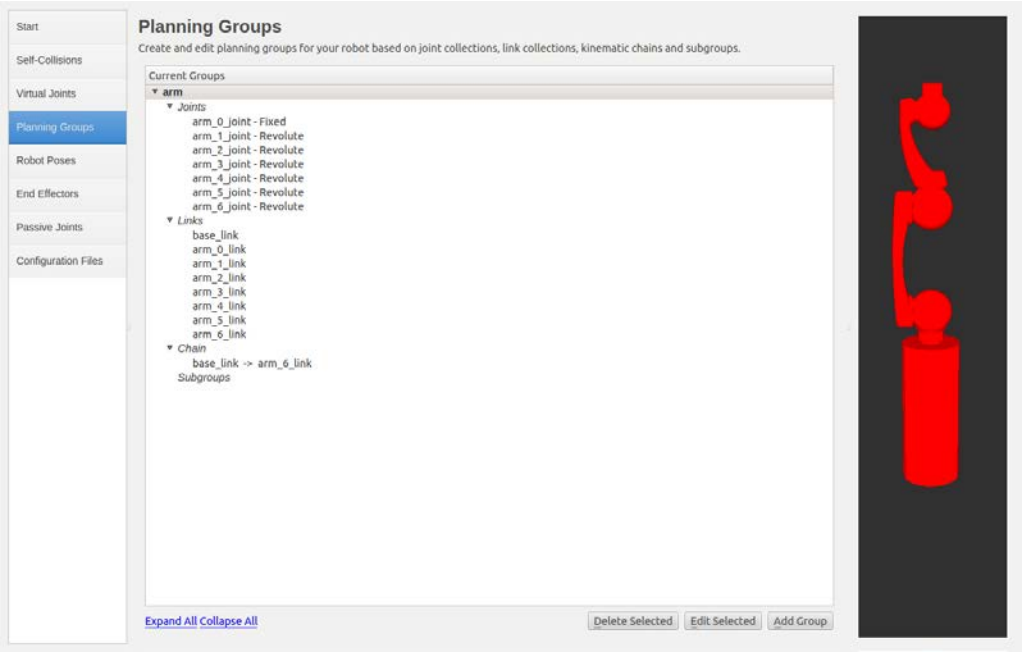


Ilustración 10.2 - Creación del grupo "arm"

Es posible que sea necesario añadir una articulación virtual para anclar el robot al escenario de simulación que presenta RVIZ. En ese caso se crearía una articulación con los parámetros abajo descritos:

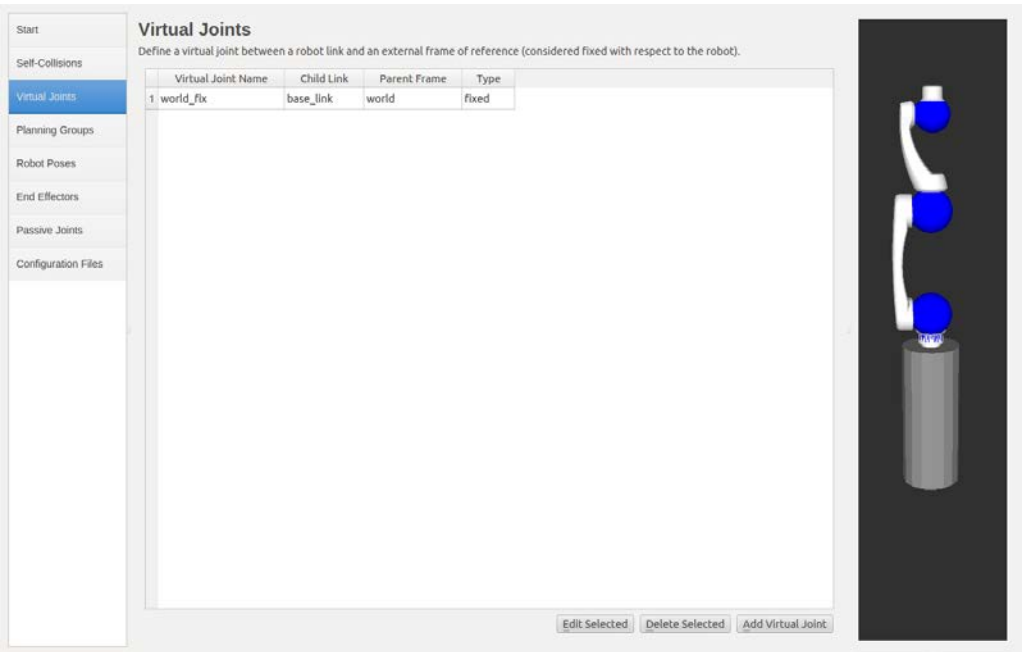


Ilustración 10.3 - Adición de una articulación virtual

La opción de añadir efectores finales no la definiremos pues para el sistema que se ha usado, la versión de la ShadowHand que hay disponible en el laboratorio es antigua y no tiene soporte ROS. Debido a eso, la comunicación entre el brazo

y la mano ha tenido que ser un desarrollo propio mediante sincronizaciones y sockets TCP/IP. Las correcciones en la posición final se han definido en el capítulo *9 Software desarrollado* mediante cálculos geométricos y programación manual.

El último paso de la configuración en el Asistente es la generación de los archivos. Para este ejemplo se ha creado una carpeta llamada "powerball_moveit" donde se alojarán todos los archivos y directorios que se generen.

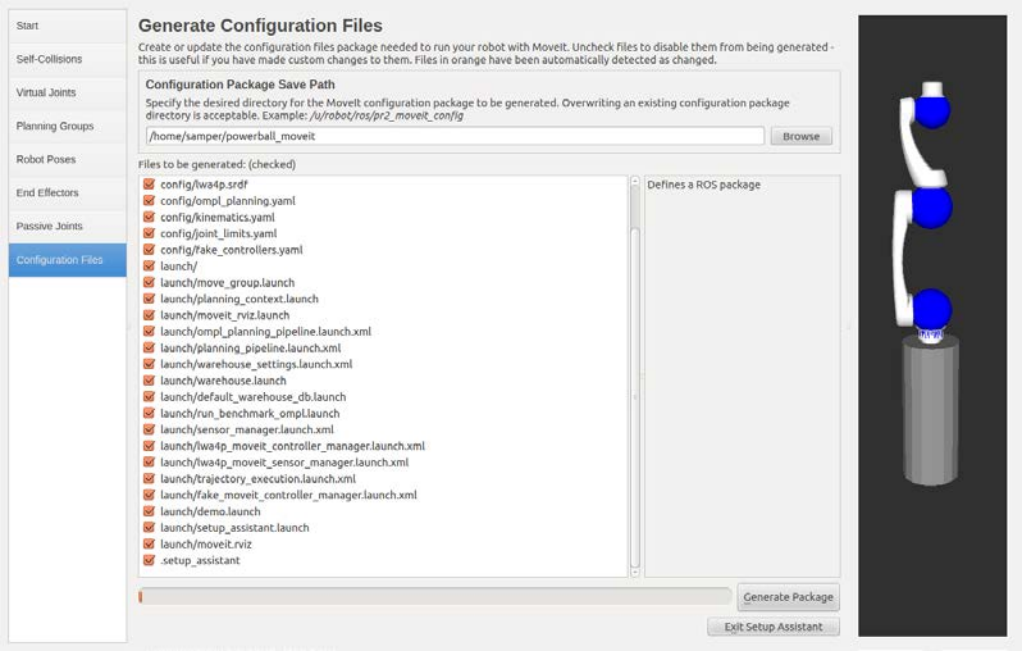


Ilustración 10.4 - Generación de archivos Moveit!

Una vez hecho esto ya tendremos creado nuestro paquete Moveit!. Ahora sólo falta añadirlo al ~/.bash para poder utilizarlo. En la línea que ya existe con el comando *export* se añaden dos puntos ":" y la ruta del paquete "/home/nombre_usuario/powerball_moveit".

También es necesario construir el paquete mediante la herramienta *cmake*. Se guardarán todos los archivos que se generen en una carpeta que llamaremos "build". Para ello introducimos los siguientes comandos (suponemos que estamos en el directorio del paquete):

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

Y lo añadimos también al ~/.bashrc:

```
$ echo “source
/home/nombre_usuario/powerball_moveit/build/devel/setup.b
ash
$ source ~/.bashrc
```

10.6.1. Modificación de archivos de configuración

Moveit! genera por defecto unos archivos de configuración que posiblemente den problemas a la hora de trabajar con el robot real. Dentro te la carpeta “config” existe un archivo llamado “joint_limits.yaml”. En ese archivo se especifican las velocidades y aceleraciones máximas de cada articulación con la siguiente estructura (mostrada en este ejemplo una sola articulación):

```
joint limits
    arm_1_joint:
        has_velocity_limits: true
        max_velocity: 2
        has_acceleration_limits: true
        max_acceleration: 0.4
```

Este valor debe reducirse a un valor igual o ligeramente inferior al que venga especificado en el archivo de configuración del robot real. Este archivo se encuentra en:

```
/home/nombre_ususario/robotnik/schunk_robots/
schunk_lwa4p/config/controller.yaml
```

En él exiten dos parámetros: “ptp_vel” y “ptp_acc”. Estos parámetros definen la velocidad a la que el robot ha de moverse entre dos puntos consecutivos interpolados en una trayectoria. Si Moveit! tiene definida una velocidad mayor creerá que la trayectoria se puede llevar a cabo en menos tiempo del que en realidad lo hará. Esto arrojará un error de “TIMEOUT reached” y abortará el movimiento. Si se modifican los archivos como se ha explicado anteriormente este problema quedará solventado.

10.6.2. Creación del archivo del controlador

Para que Moveit! pueda ser capaz de controlar el robot real es necesario crear un nuevo archivo, o mejor dicho, dos.

En primer lugar hay que crear en la carpeta “config” un archivo de configuración llamado “controllers.yaml”. Este archivo deberá contener el siguiente código:

```
controller_list:
 - name: arm_controller
   action_ns: follow_joint_trajectory
   type: FollowJointTrajectory
   default: true
   joints:
     - arm_0_joint
     - arm_1_joint
     - arm_2_joint
     - arm_4_joint
     - arm_3_joint
     - arm_6_joint
     - arm_5_joint
```

El siguiente archivo que se debe crear ha de hacerse dentro de la carpeta “launch”. Este archivo tendrá el nombre “lwa4p_moveit_controller_manager.launch” y su contenido ha de ser el siguiente:

```
<launch>
 <!-- Set the param that trajectory_execution_manager
needs to find the controller plugin -->
 <arg name="moveit_controller_manager"
default="moveit_simple_controller_manager/MoveItSimpleCon
trollerManager" />
 <param name="moveit_controller_manager" value="$(arg
moveit_controller_manager)"/>
 <!-- load controller_list -->
 <rosparam file="$(find
powerball_moveit)/config/controllers.yaml"/>
</launch>
```

10.7. Errores comunes y solución

A lo largo del desarrollo del proyecto los errores y problemas que se han encontrado han sido numerosos y diversos. A continuación se relatan algunos de los más comunes y que librarán a los siguientes estudiantes de los innumerables quebraderos de cabeza que nosotros sí tuvimos que sufrir y solventar.

10.7.1. "waiting for node 3"

Es posible que alguna vez que intentemos inicializar el robot, cuando empiece a numerar los nodos a los que va llamando se quede bloqueado esperando en el "node 3". Este problema tremendamente habitual se debe a la comunicación por el PCAN. Revise que todo está bien conectado y si es así, reinstale el controlador tal y como se especifica en el apartado *10.5.2 Instalación y configuración del adaptador PCAN-USB*.

10.7.2. "TIMEOUT reached"

Este problema ocurre cuando la trayectoria real tarda más tiempo en ejecutarse que la definida por Moveit!. La solución de este problema se explica de forma detallada en el apartado *10.6.1 Modificación de archivos de configuración*.

10.7.3. "paquete no encontrado"

Es posible que cuando intentemos lanzar una aplicación desde un archivo ".launch" obtengamos una salida como la que sigue (ejemplo del archivo "demo.launch" que nos genera el Asistente de Moveit!):

```
[demo.launch] is neither a launch file in package
[powerball_moveit] nor is [powerball_moveit] a launch
file name
```

Que ROS devuelva dicho mensaje implica que no sabe que ese paquete existe. Puede ser debido a que no ha sido instalado o que, de haberlo sido, no se ha añadido correctamente en ~/.bash. A lo largo de este proyecto se han dado muchos ejemplos de añadir paquetes al ~/.bash, por lo que no se volverá a repetir el proceso.

10.7.4. "current control error exceeds x.x"

Actualmente el robot presenta un problema a la hora de intentar seguir trayectorias que Moveit! le envía. Al parecer el robot acumula error en cada punto en los que se subdivide la trayectoria y al poco de empezar a ejecutarla aborta el proceso alegando que se ha superado el error máximo establecido. Dicho error se especifica en el archivo:

```
/home/nombre_ususario/robotnik/schunk_robots/
schunk_lwa4p/config/controller.yaml
```

Independientemente del error que se establezca este fallo se repite continuamente. Actualmente la empresa Robotnik, comercializadora de los productos robóticos de Schunk en España, no ha sabido darnos una solución para solventar dicho problema. Es vital, de cara a futuros trabajos por parte del grupo NEUROCOR, que este problema sea corregido cuanto antes.

Capítulo 11

FUTUROS DESARROLLOS

11. Futuros desarrollos

Para dar por terminada la presente memoria del Trabajo de Fin de Grado se proporcionarán pautas e ideas con las que continuar las investigaciones y desarrollos realizados con el proyecto que nos atañe. Antes de enumerar nuevos recorridos y plantear la realización de desarrollos más complejos que asienten su base en los conceptos descritos a lo largo de estas páginas, se considera importante mejorar algunos aspectos y características:

- Mejora del sistema de visión artificial para reconocer y dimensionar totalmente los objetos detectados.

- Crear una interfaz gráfica con la que comunicarse con los robots e integrar en ésta todos los mensajes de error y avisos. Se recomienda seguir los principios del desarrollo de SCADAS.

- Desarrollo de algoritmos de evitación de obstáculos en las trayectorias.

- Desarrollar un protocolo de comunicación más sólido, embebido en ROS.

- Intentar proporcionar al software de control de la mano comportamientos que permitan el autoaprendizaje para lograr agarres estables.

Por otro lado, surgen numerosas ideas que parten de las bases establecidas por este proyecto, de entre las que destacan:

- Uso de un sensor LeapMotion que permita que el robot copie los movimientos realizados por un usuario. Este desarrollo supone un trabajo gemelo al presentado ya que sustituye los sistemas de visión artificial y algoritmos de movimiento por acciones de captación de movimientos humanos y emulación de los mismos.

- Trabajo cooperativo entres dos brazos manipuladores (brazo más herramienta de manipulación).

- Adaptación del trabajo para agarrar objetos dinámicos o en movimiento.

- Adhesión de un escáner a un brazo robótico para capturar el entorno y generar mapas tridimensionales. En colaboración podría trabajar un segundo brazo con un manipulador que agarrase objetos escondidos o de difícil acceso.

- Crear IronMan.

12. Bibliografía

[1] A. Barrientos, L. F. Peñín, C. Balaguer y R. Aracil, Fundamentos de Robótica, Madrid: McGraw-Hill, 1997.

[2] Escuela Técnica Superior de Ingeniería Informática - UPV, «Blog Historia de la Informática,» 22 Diciembre 2011. [En línea]. Available: http://histinf.blogs.upv.es/2011/12/23/historia-de-linux/.

[3] ROS, «Moveit! Robots,» [En línea]. Available: http://moveit.ros.org/robots/.

[4] KUKA, «KUKA LBR iiwa - Lightweight Robot | KUKA Laboratories,» [En línea]. Available: http://www.kuka-labs.com/en/service_robotics/lightweight_robotics/.

[5] Kinova Robotics, «Products Research. Robotic arm for research, rehabilitation | Kinova Robotics,» [En línea]. Available: http://kinovarobotics.com/products/jaco-research-edition/.

[6] Barrett Technology, Inc., «Barrett Technology, Inc. - Products - WAM™ Arm,» [En línea]. Available: http://www.barrett.com/robot/products-arm.htm.

[7] UNIVERSAL ROBOTS, «Brazo robótico flexible y ecológico de Universal Robots,» [En línea]. Available: http://www.universal-robots.es/ES/Productos.aspx.

[8] YASKAWA, «Motoman: Robots MOTOMAN de YASKAWA para todas las aplicaciones industriales,» [En línea]. Available: http://www.motoman.es/es/productos/robots/?no_cache=1.

[9] elav, «Tips: Más de 400 comandos para GNU/Linux que deberías conocer :D - Desde Linux,» DesdeLinux - Usemos Linux para ser libres, 2012. [En línea]. Available: http://blog.desdelinux.net/mas-de-400-comandos-para-gnulinux-que-deberias-conocer/.

[10] S. Ceriani y M. Migliavacca, «Middleware in robotics,» de *Internal report for "Advanced Methods of Information Technology for Autonomous Robotics"*, 2012.

[11] YARP, «YARP: What exactly is YARP?,» [En línea]. Available: http://wiki.icub.org/yarpdoc/what_is_yarp.html.

[12] ROS, «ROS.org | Powering the world's robots,» [En línea]. Available: http://www.ros.org/.

[13] ROS, «Documentation - ROS Wiki,» [En línea]. Available: http://wiki.ros.org/.

[14] Schunk, «Powerball Lightweight Arm LWA 4P: SCHUNK Mobile Greifsysteme,» [En línea]. Available: http://mobile.schunk-microsite.com/en/produkte/produkte/powerball-lightweight-arm-lwa-4p.html.

[15] Microsoft, «Kinect for Windows Sensor Components and Specifications,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/jj131033.aspx.

[16] PEAK-System, «PCAN-USB: PEAK-System,» [En línea]. Available: http://www.peak-system.com/PCAN-USB.199.0.html?&L=1.

[17] Wikipedia Org., «Computer vision - Wikipedia, the free encyclopedia,» [En línea]. Available: http://en.wikipedia.org/wiki/Computer_vision.

[18] R. Rao y J.-H. Chen, «CSE 455 Computer Vision,» *UX CSE Vision Faculty.*

[19] Mathworks, «Computer Vision System Toolbox - MATLAB & Simulink - MathWorks España,» [En línea]. Available: http://www.mathworks.es/products/computer-vision/.

[20] OpenCV, «ABOUT | OpenCV,» [En línea]. Available: http://opencv.org/about.html.

[21] OPEN PERCEPTION FOUNDATION, «PCL - Point Cloud Library (PCL),» [En línea]. Available: http://pointclouds.org/.

[22] Universidad de León, «PCL/OpenNI tutorial 1: Installing and testing - Robótica - ULE,» [En línea]. Available: http://robotica.unileon.es/mediawiki/index.php/PCL/OpenNI_tutorial_1:_Installing_and_testing.

[23] ROS, «MoveIt!,» [En línea]. Available: http://moveit.ros.org/.

[24] ROS, «Moveit Tutorials for the PR2 — pr2_moveit_tutorials documentation,» [En línea]. Available: http://docs.ros.org/hydro/api/pr2_moveit_tutorials/html/.

[25] Beej, «Client-Server Background,» [En línea]. Available: http://beej.us/guide/bgnet/output/html/multipage/clientserver.html.

[26] Stack Overflow, «C++ - socket descriptor closes after assignment - Stack Overflow,» [En línea]. Available: http://stackoverflow.com/questions/22189184/socket-descriptor-closes-after-assignment.

[27] A. Parguelas, «Programación Básica de Sockets en Unix para Novatos,» [En línea]. Available: http://es.tldp.org/Tutoriales/PROG-SOCKETS/prog-sockets.html.

[28] S. N. U. UG, «Linux Howtos: C/C++ -> Sockets Tutorial,» [En línea]. Available: http://www.linuxhowtos.org/C_C++/socket.htm.

[29] Chuidiang, «Sockets en C de Unix/Linux,» 4 Febrero 2007. [En línea]. Available: http://www.chuidiang.com/clinux/sockets/sockets_simp.php.

[30] G. Bradsky , Learning OpenCV [computer vision with the OpenCV Library], O'Reilly, 2008.

[31] D. L. Baggio, Mastering OpenCV with practical computer vision projects, Packt Pub, 2012.

[32] M. Lutz, Learning Python 2nd ed. Covers Python 2.3, O'Reilly, 2003.

ANEXO Nº1

INSTALACIÓN Y CONFIGURACIÓN DE LINUX

INSTALACIÓN Y CONFIGURACIÓN DE LINUX

Este anexo recoge todos los pasos a seguir para instalar de forma satisfactoria Linux en un PC y configurarlo con las aplicaciones básicas e indispensables para el posterior trabajo en ROS y el desarrollo de aplicaciones. Partiremos de que el sistema operativo que tenemos de inicio es Windows.

A. Obtención de la ISO

Lo primero es dirigirnos a la página web oficial de Ubuntu y descargar la ISO que contiene el sistema operativo. Aunque ya se encuentra disponible la versión 14.04 LTS debemos descargar la 12.04.5 LTS (Precise Pangolin), pues es la que tiene soporte completo actualmente para las versiones de Groovy e Hydro de ROS. Por tanto, nos dirigimos al siguiente enlace:

http://releases.ubuntu.com/precise/

Y en él descargamos, preferiblemente, la versión de 64 bits, si nuestro sistema lo soporta, claro está.

B. Creación de la unidad de arranque

Una vez tengamos la ISO descargada es necesario guardarla en algún soporte físico desde el cual el ordenador pueda arrancar y cargar el sistema operativo. Este soporte físico puede ser un DVD o una unidad de memoria en estado sólido (pendrive). Lo más cómodo y económico es utilizar un pendrive. Cierto es que algunos equipos presentan problemas a la hora de intentar arrancar desde una unidad USB, porque algunas placas base no alimentan dichos puertos hasta haber cargado la partición de arranque del disco duro principal.

Para crear un DVD desde una imagen ISO existen multitud de herramientas a nuestra disposición. Windows dispone por defecto de un grabador

preinstalado que nos permite hacerlo. En caso de querer hacer un trabajo más personalizado y/o profesional, también existen a disposición del usuario suites de grabación como las de Nero o Roxio, siendo éstas de pago. De igual manera encontramos multitud de aplicaciones de libre distribución.

La otra opción es crear un dispositivo de arranque en un pendrive. Para ello se recomienda utilizar el programa *Universal USB Installer*. Dicho programa puede ser descargado de forma gratuita en el siguiente enlace.

http://www.pendrivelinux.com/universal-usb-installer-easy-as-1-2-3/

Para crear el dispositivo de arranque simplemente tenemos que seguir los sencillos pasos que se especifican que el siguiente enlace, perteneciente a la propia página oficial de Ubuntu.

http://www.ubuntu.com/download/desktop/create-a-usb-stick-on-windows

Como bien se indica en el tutorial anteriormente citado, simplemente hay que seleccionar la ISO de la ubicación en nuestro PC donde la tengamos y seleccionar el dispositivo USB que va a ser formateado para convertirse en el dispositivo de arranque.

C. Configuración del arranque del PC

El siguiente paso a seguir es cambiar la prioridad de arranque del ordenador. Para ello es necesario reiniciar el PC y entrar en la BIOS. Dependiendo del fabricante de la placa base, el método para entrar a la configuración de la BIOS requerirá una combinación de teclas distinta. Por regla general, durante la imagen de carga de la placa, es necesario presionar F2, F8 o similar.

Una vez accedamos al menú de configuración hemos de entrar en el submenú de arranque y establecemos la prioridad de arranque. Como se puede ver existe una lista de unidades en las que el sistema puede buscar la información para el arranque. Lo que debemos hacer es establecer como primer dispositivo la unidad USB que hayamos creado o el DVD.

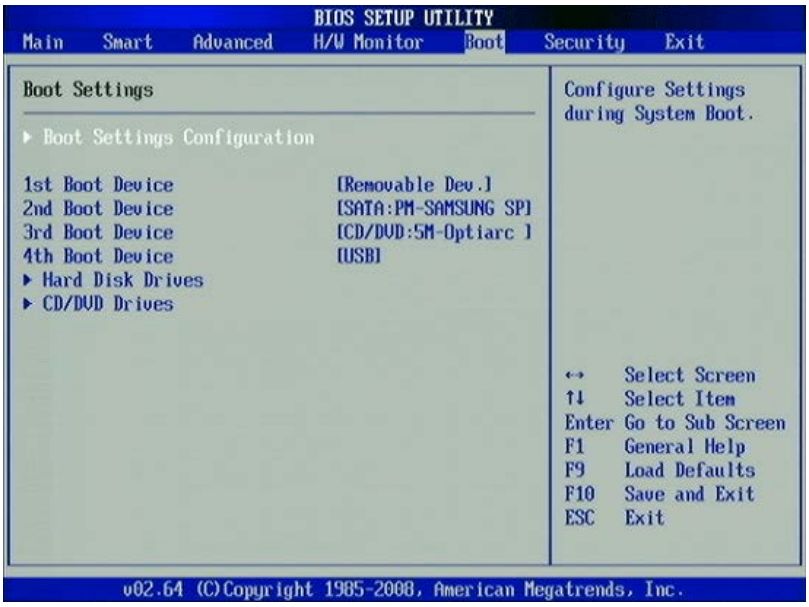


Ilustración 12.1 - Menú de arranque de la BIOS

Una vez establecido el orden, guardamos la configuración y reiniciamos.

D. Instalación de Ubuntu

Una vez que el ordenador arranque Ubuntu iniciará su ventana y nos dirá si queremos simplemente probarlo a instalarlo. Obviamente seleccionaremos instalación.

Ilustración 12.2 - Instalación Ubuntu

Ilustración 12.3 - Instalación en disco compartido

De igual manera seleccionaremos el idioma que nos interese tener en nuestro ordenador.

A la hora de decidir dónde se alojará el sistema operativo, como vemos en la Fig. 12.3, podemos elegir entre instalarlo conjuntamente con nuestro sistema

operativo actual, borrar todo el disco duro y únicamente tener Linux o entrar en la opción de configuración avanzada donde podemos definir particiones en el disco a nuestro gusto y necesidad instalando Ubuntu en cualquiera de ellas.

Lo más sencillo para un novato será tomar la primera opción, cediéndole un espacio de al menos unos 20 o 30 GB para dicha partición. A partir de ahí simplemente hay que seguir las instrucciones de las ventanas que van apareciendo, donde se nos pedirá la configuración de usuario y su contraseña, y si deseamos que a la vez que se instala y configura el sistema operativo se descarguen las actualizaciones pertinentes. Es recomendable hacer esto, habiendo conectado el PC a una red WiFi o cableada con acceso a Internet.

E. Inicio e instalación de aplicaciones importantes

Una vez se haya instalado y configurado el sistema operativo tendremos ante nosotros el escritorio de Ubuntu. Dicho escritorio es bastante intuitivo, por lo que su uso no nos resultará complejo, ya estemos acostumbrados a Windows o a Mac OS X.

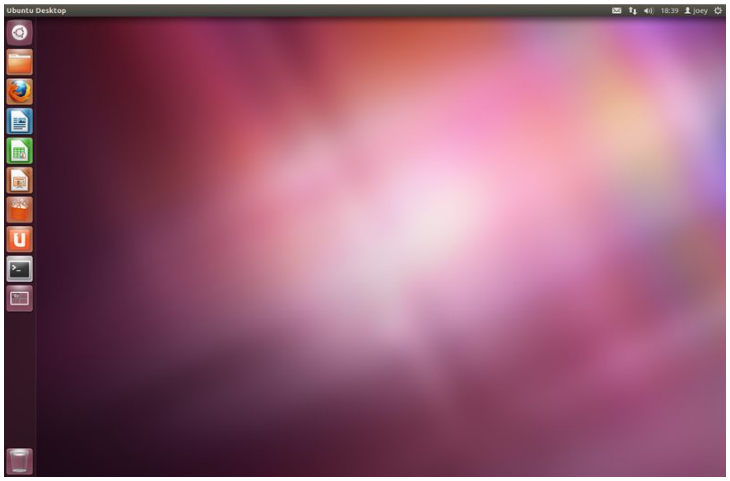


Ilustración 12.4 - Escritorio de Ubuntu

En el lateral izquierdo tenemos el dock, que es una barra con accesos directos a las aplicaciones que utilizaremos más a menudo. Dicha barra puede ser personalizada a nuestro antojo. A la hora del desarrollo de software las aplicaciones que más usaremos serán, por un lado el terminal de comandos, o *Shell*, que será nuestro controlador principal del PC y por otro lado, un buen editor de textos. Por defecto tenemos instalado Gedit, pero se recomienda el uso

de alguno más profesional, ya sean Qt Creator o Geany. Ambos pueden ser descargados de forma gratuita desde el Centro de Software de Ubuntu, aplicación preinstalada en el sistema operativo.

La última aplicación que será fundamental para el trabajo con Linux y ROS será el Gestor de Paquetes Synaptics. En él podremos ver qué paquetes y librerías tenemos instaladas en el equipo, así como instalar las que nos falten o eliminar las que no necesitemos. Tiene la ventaja de ser muy completo y de prescindir del terminal para tener que ejecutar las acciones de instalación, actualización o desinstalación.

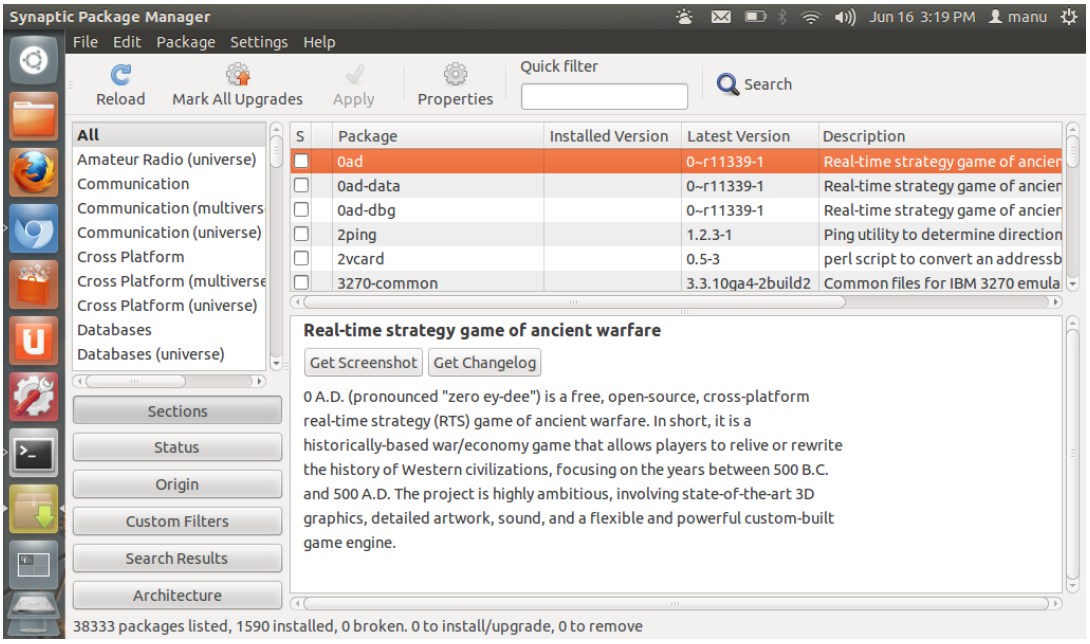


Ilustración 12.5 - Synaptics Package Manager

Tras haber realizado estos pasos, ya tenemos nuestro sistema listo para empezar a trabajar. Es importante no olvidar ejecutar en el terminal el siguiente comando para mantener actualizado el sistema, o hacerlo desde el Gestor de Actualizaciones del sistema.

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

ANEXO Nº2

INSTALACIÓN, CONFIGURACIÓN Y GUÍA DE INICIACIÓN A ROS

INSTALACIÓN, CONFIGURACIÓN Y GUÍA DE INSTALACIÓN DE ROS

En este anexo se verá cómo hay que instalar ROS en un PC con Ubuntu. En este proyecto instalaremos la versión Groovy Galapagos, pues el brazo Powerball de Schunk está completamente desarrollado para esta distribución.

A. Instalación

Lo primero que debemos haces es configurar las lista de repositorios disponible para descargar actualizaciones y software de ROS. Para ello abrimos una ventana nueva del terminal (CTRL+ALT+T) y ejecutamos el siguiente código.

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

Notar que todos los comandos presentados en el presente anexo y en los demás que conforman el proyecto vienen precedidos por el símbolo del $ (dollar). Esto es simplemente para identificar qué son comandos a introducir y qué no. Por tanto no se debe escribir dicho símbolo antes de los comandos que se introduzcan por consola.

El siguiente paso es configurar las llaves de descargas.

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

A continuación es importante saber si tenemos nuestro sistema actualizado. De cara a tener el PC con las últimas versiones de código de los repositorios ejecutamos el siguiente comando:

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

Una vez actualizado el sistema, se puede proceder a instalar el paquete completo de ROS. Llegados a este punto se presentan diferentes opciones de cara a la instalación. Podemos instalar el sistema completo con todos los paquetes, una versión de escritorio más "light", únicamente la base de ROS (core) o si ya lo tenemos, añadir paquetes individuales. Para más información acerca de estas opciones se recomienda visitar el siguiente enlace:

```
http://wiki.ros.org/groovy/Installation/Ubuntu
```

En el caso más general, y el nuestro se encuentra en esa tesitura, lo recomendable es instalar la versión completa, que cuenta con el núcleo ROS, rqt, rviz, librerías genéricas para robots, simuladores 2D/3D, navegación y percepción 2D/3D. Para ello ejecutamos el siguiente comando:

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

Linux conectará con los servidores de ROS y nos pedirá confirmación sobre si deseamos descargar los archivos necesarios para la instalación. Simplemente hemos de aceptar la descarga y esperar a que se vaya realizando.

Llegados a cierto punto nos aparecerá una pantalla en la que nos pedirá si queremos ejecutar el demonio de ROS para que éste siempre esté corriendo en un segundo plano sin necesidad de ejecutar el comando "roscore". Se recomienda no instalarlo a menos que se tenga la intención de crear aplicaciones muy específicas que lo requieran.

Una vez hayan terminado de instalarse todos los paquete de ROS es momento de inicializar las dependencias del sistema, que nos permitirán compilar nuevo código y utilizar de forma conjunta todos los paquetes que tengamos instalados en el sistema. Ejecutamos, por tanto, los siguientes comandos.

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

Es importante que las variables de entorno sean añadidas de forma automática al *bash* del terminal cada vez que sea habrá una nueva ventana. Más adelante se explicará lo que es el *bash* y para qué se utiliza. Explicaciones aparte, los comandos para añadir las variables al *bash* son:

```
$ echo “source /opt/ros/groovy/setup.bash” >> ~/.bashrc
$ source ~/.bashrc
```

Por ultimo nos queda instalar la herramienta “rosinstall”, que permite con un comando descargar gran cantidad de paquetes de software e instalarlos. Veremos que este comando será utilizado en el capítulo de configuración del robot. Para instalarlo únicamente debemos ejecutar por consola:

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

B. Configuración

En este apartado se hablará del *bash* y de la importancia que éste tiene en el flujo de trabajo con ROS. El *bash* es un archivo de configuración que carga todas las rutas y comandos que Linux puede reconocer en base a los programas que tiene instalados. También quedan reflejadas las variables del entorno y algunas asignaciones que no son motivo de estudio en el presente trabajo. Si lo abrimos con el siguiente comando:

```
$ gedit ~/.bashrc
```

Veremos que tiene el siguiente aspecto:

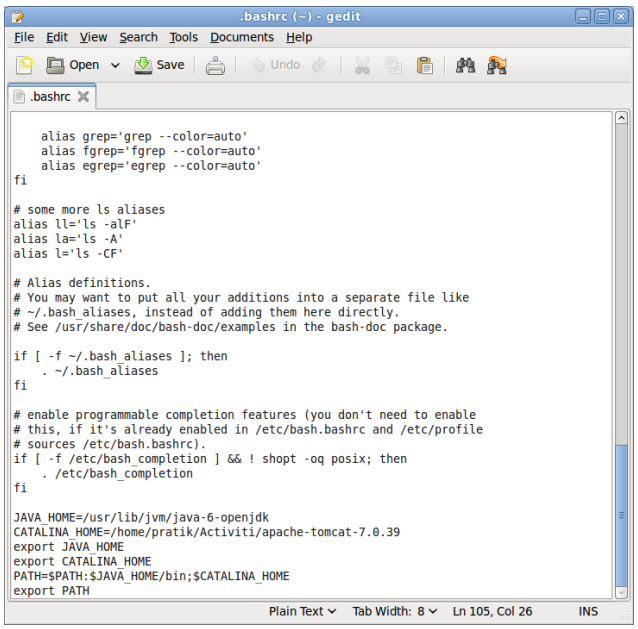


Ilustración 12.6 - bashrc

Todo lo que venga especificado en dicho archivo se ejecutará cada vez que se abra una nueva ventana del terminal. Si al instalar un nuevo paquete no tenemos la precaución de definir en el bash que ejecute el comando "source" al correspondiente archivo "setup.bash" de dicho paquete de nada habrá servido que lo hayamos instalado, pues el terminal no podrá ejecutarlo, ya que no lo encuentra en sus listas de comandos disponibles asociados a las aplicaciones instaladas.

Otro comando muy importante para añadir rutas al bash es el asociado a "ROS_PACKAGE_PATH". Existen paquetes, como los que genera Moveit! en su asistente de configuración, que no tienen un archivo "setup.bash" que se puede gestionar como en los otros. En este caso debemos operar de la siguiente manera para que el sistema sea capaz de ejecutar los programas incluidos en dicho.

```
$ echo "export ROS_PACKAGE_PATH=<<ubicación del paquete>>:$ ROS_PACKAGE_PATH">>~/.bashrc
```

Cambiamos la parte <<ubicación del archivo>> por la ruta hasta la carpeta que contiene el paquete y ya estaremos en disposición de utilizarlo. Si dicho paquete contiene otros sub-paquetes en su interior estos quedarán definidos uno tras otro separados por el dos puntos ":". Es importante que si necesitamos agregar más rutas lo hagamos en la misma línea del archivo *bashrc* utilizando el signo anteriormente definido. En caso de encontrarse dos órdenes "export" dentro del archivo *bashrc* obtendremos un error al intentar llamar a dichos paquetes, pues no reconocerá ni unos ni otros.

C. Guía básica de iniciación a ROS

Esta breve guía acerca de los conceptos básicos de funcionamiento en ROS está basa en los tutoriales que se ofrecen en la página oficial (enlace abajo). Para ampliar la información que aquí se detalla se recomienda encarecidamente llevar a cabo todos los tutoriales que en dicha página se ofrecen. Aunque el llevarlos a cabo no asegura ningún dominio de ROS sí que permite ir sentando unas bases para poder iniciarse en el sistema y crear nuestros primeros programas o entender otros ya creados para poder modificarlos si así lo necesitamos.

http://wiki.ros.org/ROS/Tutorials

I. Creación de un Workspace

Para poder crear programas que corran sobre la plataforma ROS es necesario que creemos un espacio de trabajo o workspace, lo cual nos facilitará en gran medida las cosas a la hora de compilar el programa y que el ROS Master sea capaz de reconocer nuestros nuevos programas como nodos ejecutables.

A partir de la versión Groovy Galapagos el sistema de creación de paquetes cambió. Anteriormente existía una herramienta conocida como "rosbuild". La citada herramienta ha sido sustituida por el sistema de creación y compilación de paquetes "Catkin", mucho más sencillo, rápido y eficiente que el anterior. Sigue estando disponible dicha herramienta para los "nostálgicos", pero no se recomienda su uso, pues el flujo de trabajo con "Catkin" es notablemente superior. Para más información acerca de "Catkin" se recomienda visitar el siguiente enlace:

```
http://wiki.ros.org/catkin
```

Para crear un workspace, que se llamará en este ejemplo "catkin_ws", deberemos ejecutar lo siguiente:

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

Aunque el workspace se encuentre vacío, ya que no hay ningún paquete dentro de la carpeta "src", sólo un simple enlace de *CMakeLists.txt*, la herramienta nos permite construirlo. Ahora sólo queda subir un nivel en el directorio y hacer uso del comando "catkin_make" para compilar todo aquello que se encuentre dentro de dicho workspace.

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

Este comando nos creará en nuestro directorio las carpetas "devel" y "build". Dentro de la carpeta "devel" encontramos diferentes archivos de configuración. El que nos debe interesar es el "setup.bash", que habrá que enlazar con el terminal para que los programas que se creen puedan ser lanzados por consola.

```
$ echo "source ~/catkin_ws/devel/setup.bash"
```

```
$ source ~/.bashrc
```

II. Navegación por el sistema de archivos de ROS

El código está diseminado a lo largo de muchos paquetes en ROS. Tener que hacer una búsqueda de alguno de ellos valiéndose de únicamente herramientas como “ls” o “cd” sería una labor tediosa y muy poco eficiente. Es por esto que ROS ofrece una serie de herramientas para hacer esta labor mucho más rápida y efectiva. Estas herramientas son “rospack”, “roscd”, “roscd log” y “rosls”.

rospack: ofrece información acerca de paquetes. Presenta varias opciones, y un ejemplo de ellas es “find”. Tiene la siguiente estructura:

```
# rospack find [package_name]
```

Y un ejemplo de uso sería el siguiente:

```
$ rospack find roscpp
```

Y devolvería el siguiente resultado por pantalla, que no es sino la ubicación del paquete en el disco duro. En caso contrario diría que el paquete no ha sido encontrado.

```
/opt/ros/groovy/share/roscpp
```

roscd: permite cambiar directamente a un paquete o biblioteca. Su estructura es la siguiente:

```
# roscd [locationname[/subdir]]
```

Y un ejemplo de uso sería:

```
$ roscd roscpp
```

roscd log: permite ir a la carpeta donde ROS guarda los archivos de registro. Si nunca se ha ejecutado ningún programa la instrucción devolverá un error.

rosls: permite listar los archivos existentes dentro de un paquete únicamente escribiendo su nombre, sin necesidad de escribir la ruta completa. Su estructura es la siguiente:

```
# rosls [locationname[/subdir]]
```

Y un ejemplo de uso nos daría este resultado:

```
$ rosls roscpp_tutorials

    cmake package.xml srv
```

ROS también permite la función de auto-completar rutas mediante la pulsación del tabulador. Con simplemente escribir parte del paquete y pulsar dicha tecla el sistema nos autocompleta la instrucción si lo escrito hasta entonces es único, y si no lo es, nos ofrece una lista de las posibles opciones.

III. Crear un paquete

En este punto se verá como crear un paquete mediante la herramienta de construcción "catkin". Para empezar es importante definir qué elementos indispensables ha de tener un paquete catkin para que sea considerado como tal. Esos componentes indispensables son:

- **package.xml**: contiene meta información sobre el paquete.
- **CMakeLists.txt**: especifica qué se debe crear al compilar y que dependencias existen.
- No puede existir más de un paquete en cada carpeta. Esto implica que no deben haber paquetes múltiples o anidados compartiendo el mismo directorio.

Dentro de un workspace la estructura que deben presentar los diferentes paquetes debe ser algo parecido a lo presentado a continuación:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file,
provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for
package_1
      package.xml        -- Package manifest for
package_1
    ...
```

```
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for
package_n
      package.xml        -- Package manifest for
package_n
```

Para crear el paquete suponemos que ya se ha creado un workspace con anterioridad. En este ejemplo crearemos un paquete llamado "beginner_tutorials" que dependerá de otros 3 paquetes para funcionar: "rospy", "roscpp" y "std_msgs".

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg beginner_tutorials std_msgs rospy
roscpp
```

Como se aprecia, el primer atributo es el nombre del paquete que se desea crear, y los siguientes son las dependencias que éste necesitará. Es posible añadir o eliminar dependencias posteriores modificando de forma manual los archivos *CMakeLists.txt* y *package.xml*. Para profundizar más en dichas modificaciones se recomienda leer los siguientes enlaces:

http://wiki.ros.org/ROS/Tutorials/CreatingPackage
http://wiki.ros.org/catkin/CMakeLists.txt
http://wiki.ros.org/catkin/package.xml

IV. Construir el paquete

Una vez creado el paquete, para hacerlo completamente funcional es necesario construirlo. Gracias a la herramienta *catkin* es realmente fácil este paso, pues mediante un comando podemos ejecutarlo todo. Estando ubicados en el workspace que se ha creado únicamente hay que ejecutar:

```
$ catkin_make
```

Antes de ejecutar el comando anterior es importante abrir el archivo *CMakeLists.txt* y modificarlo adecuadamente. Al crear el paquete se genera dicho archivo de forma automática, pero no está listo para usar. Se genera con muchas opciones, pero todas ellas vienen comentadas, con la intención de que el usuario descomente las que le interesan. Entre esas opciones se encuentra el enlace a librerías, la creación del archivo ejecutable correspondiente al programa o el establecimiento de las dependencias del paquete. Es muy importante que esas

líneas de código sean descomentadas si se quiere que el programa funcione correctamente.

V. Trabajo con los nodos

En el capítulo *4.3 ROS* ya se ha explicado lo que son los nodos. En este punto del anexo se profundizará en el trabajo con ellos.

El primer nodo que debe ejecutarse para poder trabajar con ROS es, obviamente, el nodo Maestro. Se inicia con el siguiente comando:

```
$ roscore
```

Obteniendo la siguiente información a la salida:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-
00251148e8cf/roslaunch-machine_name-13039.log
Checking log directory for disk usage. This may take
awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://machine_name:33919/
ros_comm version 1.4.7

SUMMARY
========

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

auto-starting new master
process[master]: started with pid [13054]
ROS_MASTER_URI=http://machine_name:11311/

setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

En el terminal que se ejecute este comando no se podrán seguir lanzando instrucciones, pues se queda manteniendo el proceso de "roscore". Abrimos una nueva ventana, o pestaña dentro de la ya creada ventana (CTRL+SHIFT+T) para poder ver otros comandos relacionados con los nodos.

El comando “rosnode” ofrece información acerca de los nodos que se están ejecutando en el momento de hacer la petición y presenta diferentes opciones a la hora de llamarlo. En caso de querer ver todos los nodos operativos basta con introducir:

```
$ rosnode list
```

Otro comando fundamental para iniciar un nodo es “rosrun”, el cual tiene la siguiente estructura:

```
$ rosrun [package_name] [node_name]
```

Un ejemplo que viene preinstalado para empezar a familiarizarse con el funcionamiento de ROS es el simulador de la tortuga. Consiste en una animación 2D que puede ser controlada y modificada a través de otros nodos. Para llamarla introducimos:

```
$ rosrun turtlesim turtlesim_node
```

Y se generará una ventana donde aparecerá la pequeña tortuga (mascota corporativa de ROS).

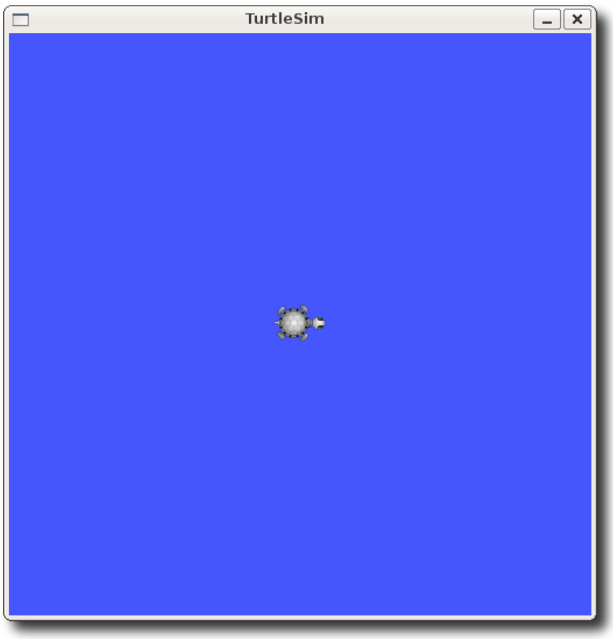


Ilustración 12.7 - Simulador TurtleSim

VI. Trabajo con topics

De la misma manera que los nodos fueron explicados en el capítulo *4.3 ROS* los topics también se abordaron en esa parte del trabajo, por lo que no repetiremos su definición.

Para poder ver cómo dos nodos se comunican a través de un topic, sin haber cortado el proceso del simulador de la tortuga en un nuevo terminal ejecutamos:

```
$ rosrun turtlesim turtle_teleop_key
```

Este nodo permite controlar la tortuga a través de las flechas del teclado. Estos dos nodos compartirán información a través de topics a los que están suscritos o en los que publican.

Existe un comando, "rostopic", que es muy útil a la hora de extraer información de los topics activos. Dispone de varias opciones, que se detallan a continuación:

```
rostopic bw     display bandwidth used by topic
rostopic echo   print messages to screen
rostopic hz     display publishing rate of topic
rostopic list   print information about active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

ROS también dispone de una herramienta muy útil, "rqt_graph", la cual genera un gráfico en el que se muestra la relación entre los nodos activos existentes y los topics que las comunican. En el ejemplo anteriormente iniciado, si ejecutamos esta herramienta mediante el comando abajo especificado, aparece en pantalla la siguiente aplicación.

```
$ rosrun rqt_graph rqt_graph
```

Ilustración 12.8 - rqt_graph

Como se observa, en azul aparece el nodo que publica información en el topic "/turtle1/command_valocity", mostrado en rojo. Y en verde se representa el nodo que está suscrito al citado topic. Esta herramienta de análisis puede ser muy útil de cara a entender cómo está funcionando un programa o a depurar su funcionamiento.

Otra herramienta que ofrece ROS para monitorizar procesos activos es "rqt_plot". Esta utilidad permite dibujar a modo de gráfico los datos publicados en un topic. Para lanzar esta aplicación introducimos por terminal:

```
$ rosrun rqt_plot rqt_plot
```

Si lo ejecutamos estando el anterior ejemplo ejecutándose podremos ver la posición de la tortuga tanto en el eje X como en el Y, si previamente le decimos que es eso lo que queremos visualizar abriendo un par de nuevos terminales e introduciendo en cada uno un comando de los abajo presentados:

```
$ /turtle1/pose/x
$ /turtle1/pose/y
```

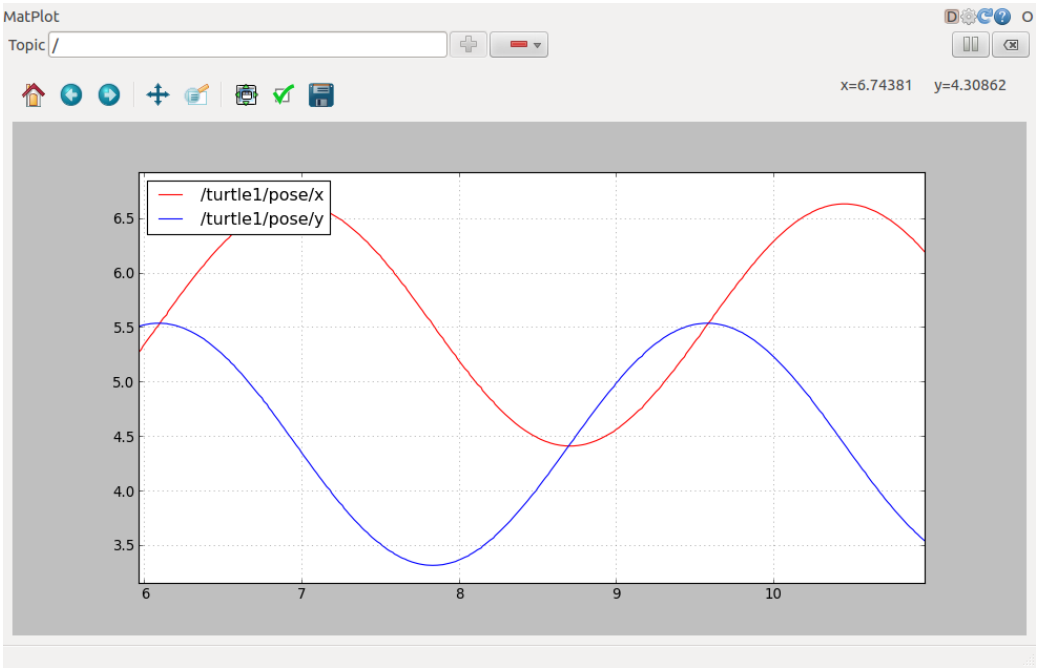


Ilustración 12.9 - rqt_plot

Para profundizar más en el uso de esta herramienta se recomienda visitar la página siguiente:

http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

VII. Servicios y parámetros

Como ya se vio en el capítulo 4.3, los servicios son otra forma de comunicación alternativa a los topics, donde los nodos pueden enviar peticiones y recibir respuestas de otros.

Una herramienta fundamental para obtener información de los servicios activos es "rosservice". Tiene varias opciones que ofrecen gran versatilidad:

```
rosservice list    print information about active
services
rosservice call    call the service with the provided
args
rosservice type    print service type
rosservice find    find services by service type
rosservice uri     print service ROSRPC uri
```

Otra herramienta muy útil es "rosparam". Ofrece la posibilidad de almacenar y manipular datos en el Servidor de Parámetros de ROS. Dicho

servidor puede almacenar enteros, floats, booleanos, diccionarios y listas. "rosparam" utiliza el lenguaje marcado YAML. El uso de esta herramienta es:

```
rosparam set            set parameter
rosparam get            get parameter
rosparam load           load parameters from file
rosparam dump           dump parameters to file
rosparam delete         delete parameter
rosparam list           list parameter names
```

Para profundizar más en el uso de esta herramienta se recomienda visitar la página siguiente:

http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

VIII. Uso de rqt_console y rqt_logger_level

A la hora de depurar el funcionamiento de un programa ROS ofrece unas herramientas llamadas "rqt_console" y "rqt_logger_level". La primera se ancla al marco de trabajo de registro de ROS y permite sacar por pantalla la salida de los nodos. La segunda permite cambiar el nivel de verbosidad de los nodos al tiempo que son ejecutados (DEBUG, WARN, INFO y ERROR).

Si ejecutamos estas dos herramientas en terminales distintos mediante los siguientes comandos, y posteriormente lanzamos el ejemplo anterior de la tortuga aparecerán en pantalla las siguientes ventanas.

```
$ rosrun rqt_console rqt_console
$ rosrun rqt_logger_level rqt_logger_level
```

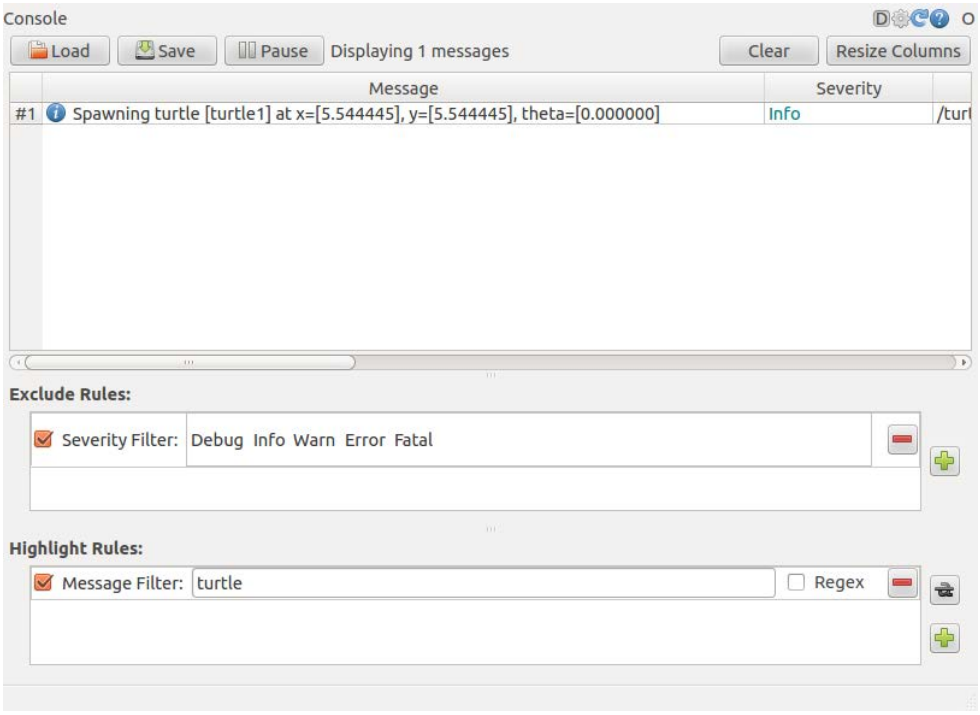


Ilustración 12.10 - rqt_console con turtlesim iniciado

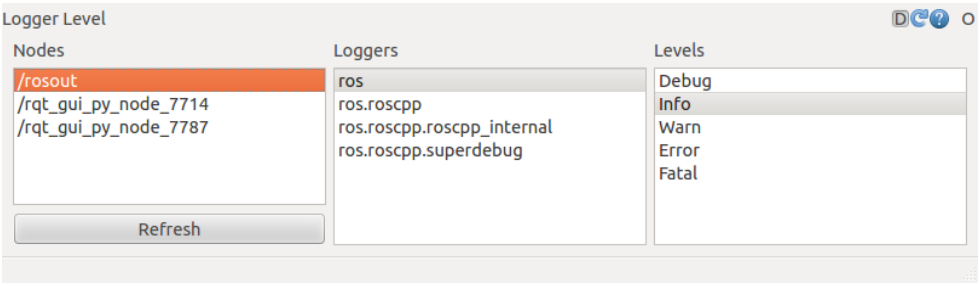


Ilustración 12.11 - rqt_logger_level

Dependiendo del nivel de verbosidad que se defina en el "rqt_logger_level" en la ventana de la "rqt_console" aparecerán unos mensajes u otros. Se especifica cuál es el nivel máximo de información deseado y en función de eso se mostrarán unos mensajes u otros. Los mensajes en ROS están priorizados, siendo su orden de mayor a menor prioridad el que sigue: fatal > error > warn > info > debug. Si se selecciona un nivel de verbosidad únicamente se obtendrán mensajes de ese nivel o superiores, obviándose los de inferior prioridad.

IX. Comando roslaunch y archivos .launch

El comando "roslaunch" es muy útil a la hora de lanzar a la vez varios nodos los cuales están especificados en un archivo de extensión ".launch". Gran cantidad de programas que se componen de diferentes nodos para poder operar suelen venir organizadas en archivos de este estilo para que su ejecución no

implique tener que estar llamando uno a uno todos los nodos que componen el programa. Esto aporta una gran fluidez y sencillez de trabajo.

La estructura del comando es la que sigue:

```
$ roslaunch [package] [filename.launch]
```

Ni que decir tiene que el paquete debe haber sido añadido anteriormente al ~/.bashrc para poder ser lanzado desde consola. Para entender la estructura de un archivo con extensión ".launch" se recomienda visitar los siguientes enlaces:

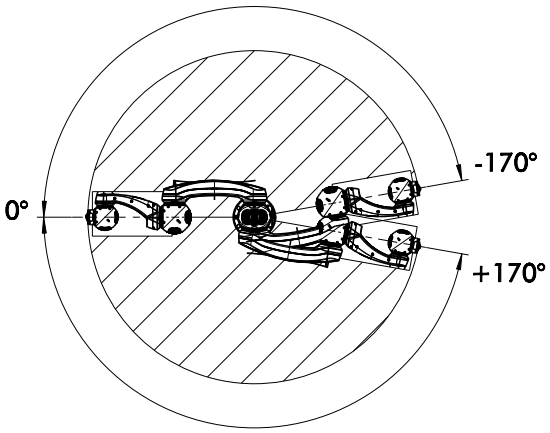
http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch
http://wiki.ros.org/roslaunch/XML/launch

ANEXO Nº3

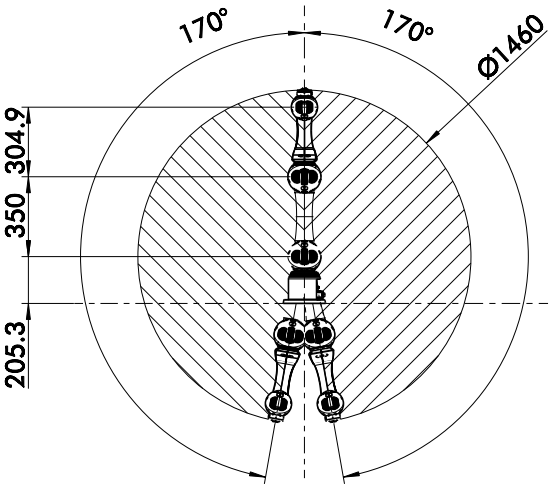
ESPECIFICACIONES TÉCNICAS SCHUNK POWERBALL

Technical data

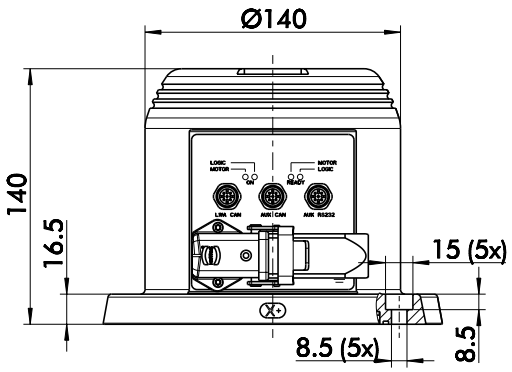
Working space
(Top view)

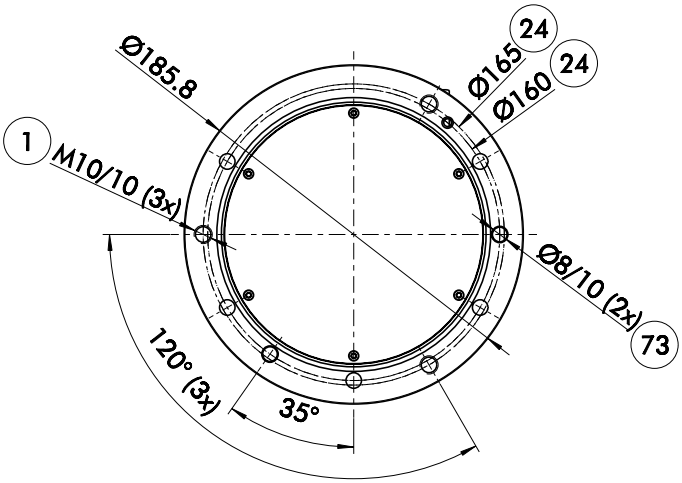


Working space
(Right side)



Robot foot





(1) Robot-side connection
(24) Bolt circle
(73) Fitting for centering pin

Designation	LWA 4P
ID	0306960
Type	6 DOF lightweight robot
Number of axes	6
Max. payload load [kg]	6
Repeat accuracy [mm]	±0.15
Position feedback	Pseudo-absolute position measuring
Drives	Brushless servomotors with permanent-magnet brake
Pan-tilt unit flange	Flat tool changer with free lines and power supply
Installation direction	Any
Dead weight [kg]	15

IP class [IP]	40	
Power supply	24 V DC / avg. 3 A / max. 12 A	
Interface	CANopen (CiA DS402:IEC61800-7-201)	
Axes	Speed with nominal load	Range
Axis 1	72°/s	±170°
Axis 2	72°/s	±170°
Axis 3	72°/s	±155.5°
Axis 4	72°/s	±170°
Axis 5	72°/s	±170°
Axis 6	72°/s	±170°
Grippers	WSG 50, PG-plus 70, MEG, SDH 2, SVH	
Changer	FWS 115	
Robot control system	ROS node (ROS.org) or KEBA CP 242/A (KEBA.com)	

ANEXO Nº4

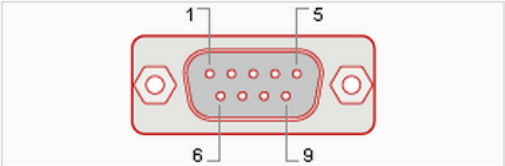
ESPECIFICACIONES TÉCNICAS PCAN-USB

ESPECIFICACIONES TÉCNICAS PCAN-USB

Las especificaciones técnicas del adaptador PCAN-USB de la firma PEAK System son las que se muestran a continuación.

- Adapter for the USB connection (USB 1.1, compatible with USB 2.0 and USB 3.0)
- Voltage supply via USB
- Bit rates from 5 kbit/s up to 1 Mbit/s
- Time stamp resolution approx. 42 µs
- Compliant with CAN specifications 2.0A (11-bit ID) and 2.0B (29-bit ID)
- CAN bus connection via D-Sub, 9-pin (in accordance with CiA® 102)
- NXP SJA1000 CAN controller, 16 MHz clock frequency
- NXP PCA82C251 CAN transceiver
- 5-Volt supply to the CAN connection can be connected through a solder jumper, e.g. for external bus converter
- Extended operating temperature range from -40 to 85 °C (-40 to 185 °F)

Pin assignment D-Sub



Pin	Pin assignment
1	Not connected / optional +5V
2	CAN-L
3	GND
4	Not connected
5	Not connected
6	GND
7	CAN-H
8	Not connected
9	Not connected / optional +5V

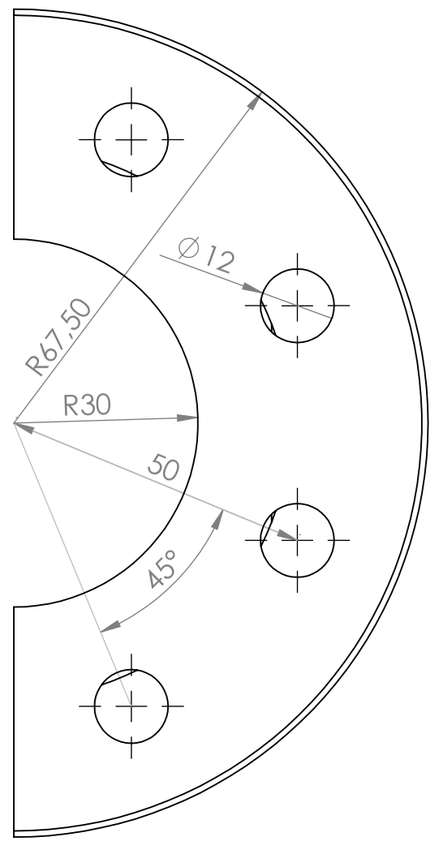
Optionally available:

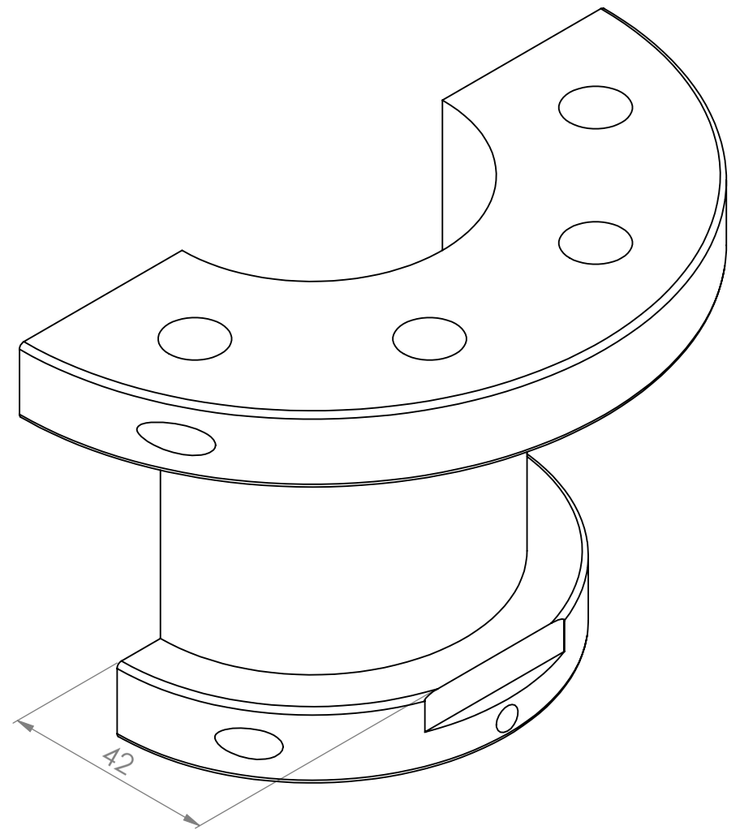
- Galvanic isolation on the CAN connection up to 500 V

ANEXO Nº5

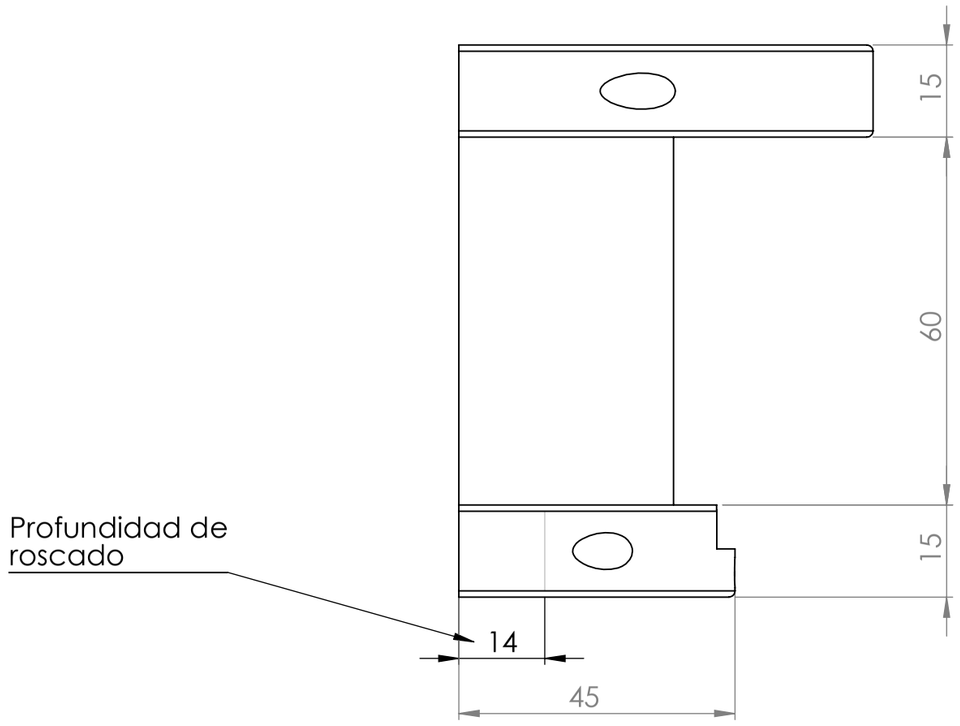
PLANOS DEL ACOPLE ENTRE LA SHADOWHAND Y EL POWERBALL

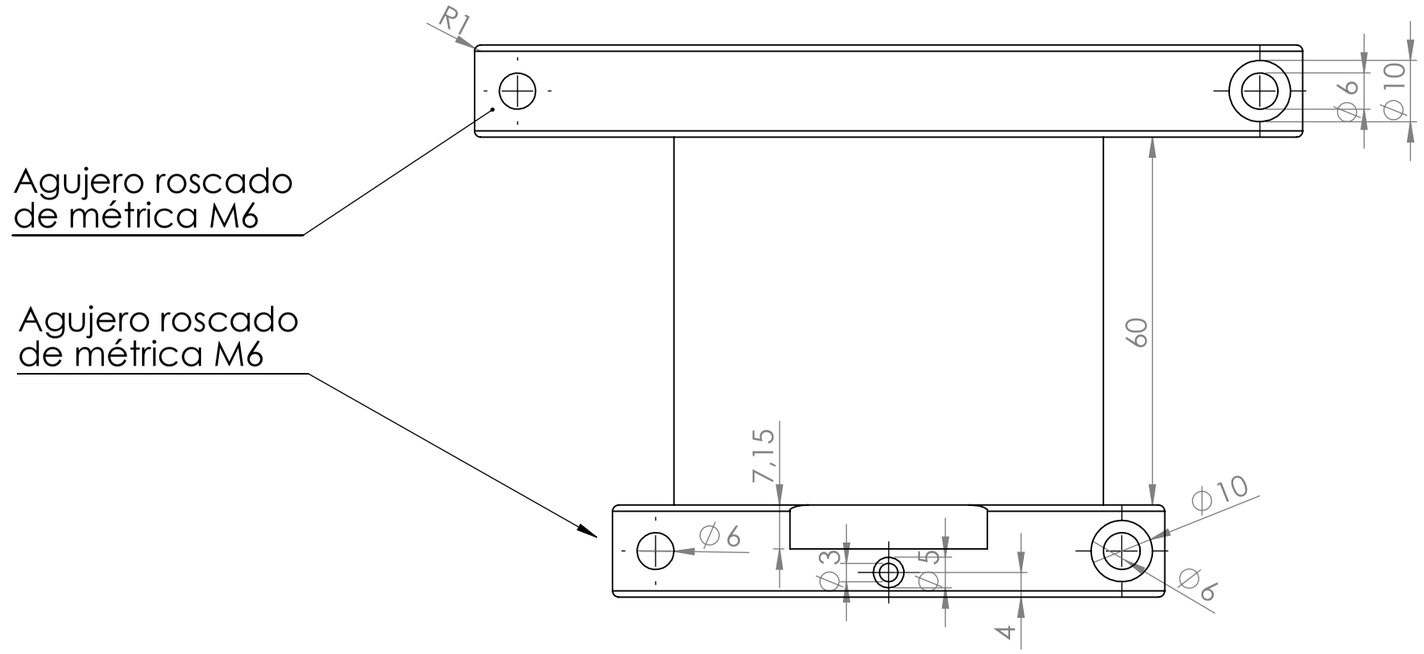
Universidad Politécnica de Cartagena





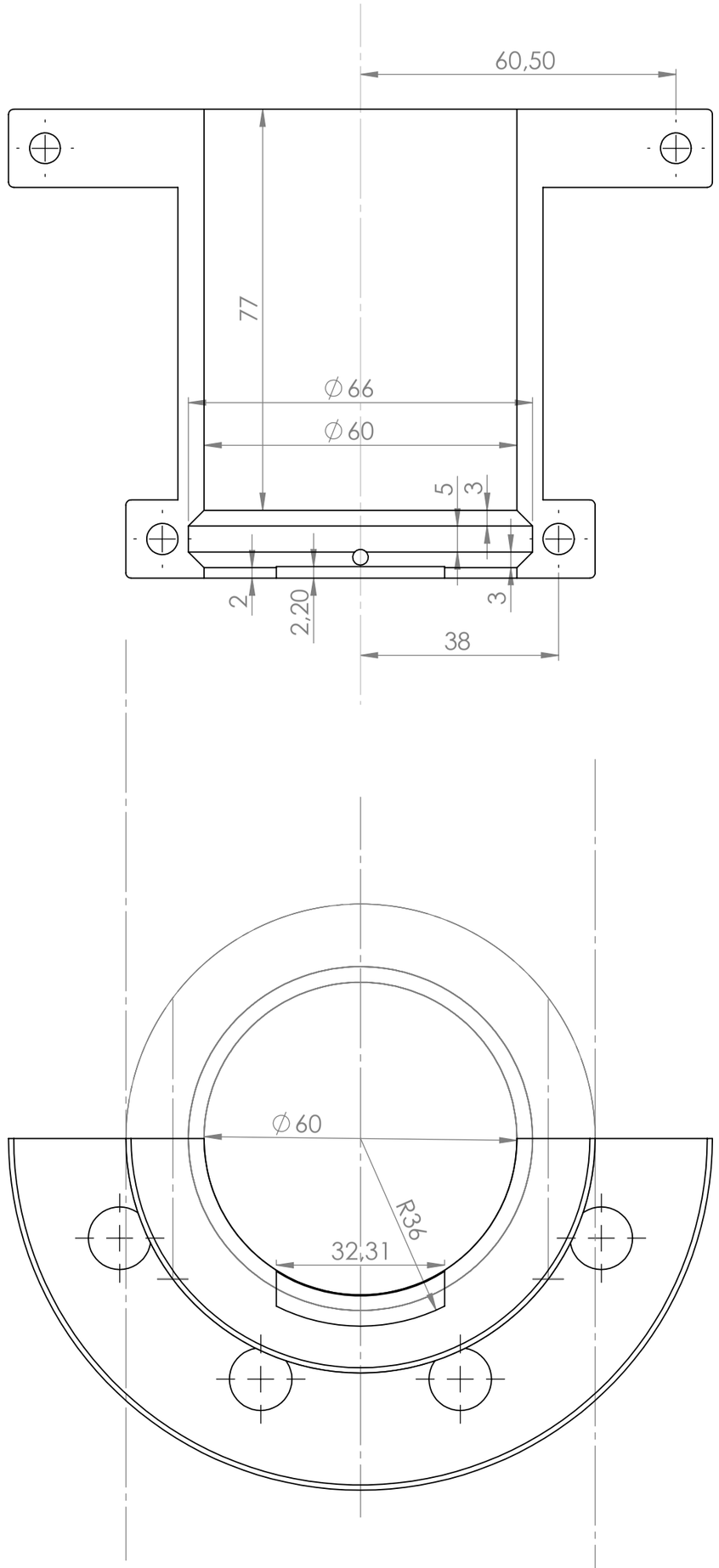
Todos los suavizados de bordes en la pieza son de radio = 1 mm

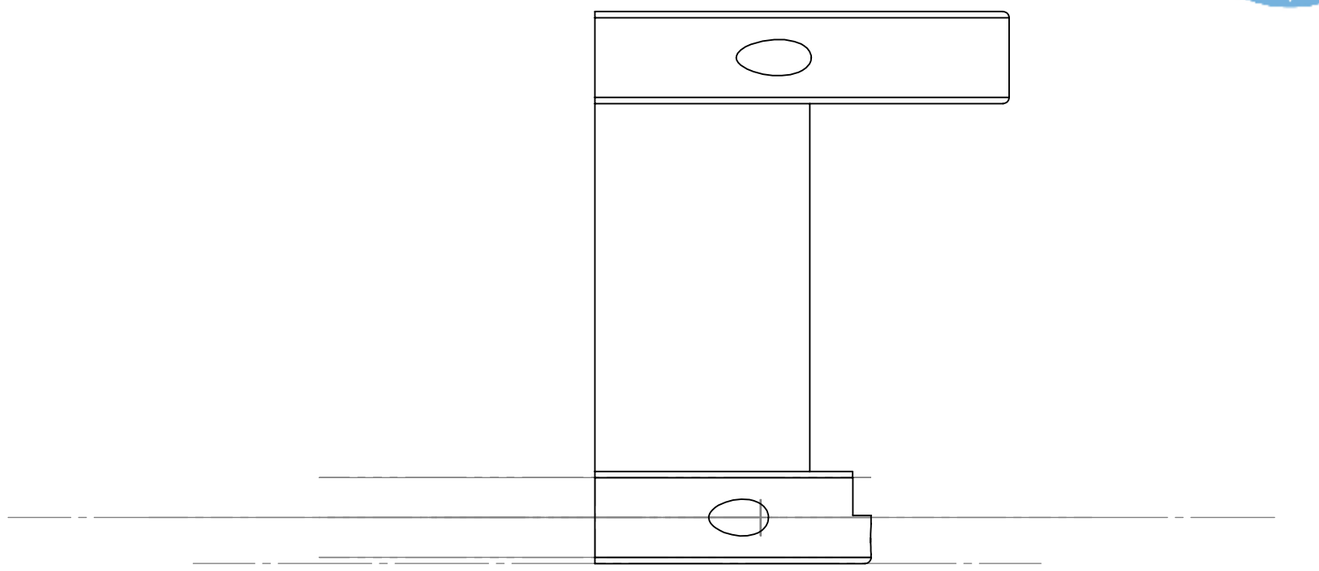


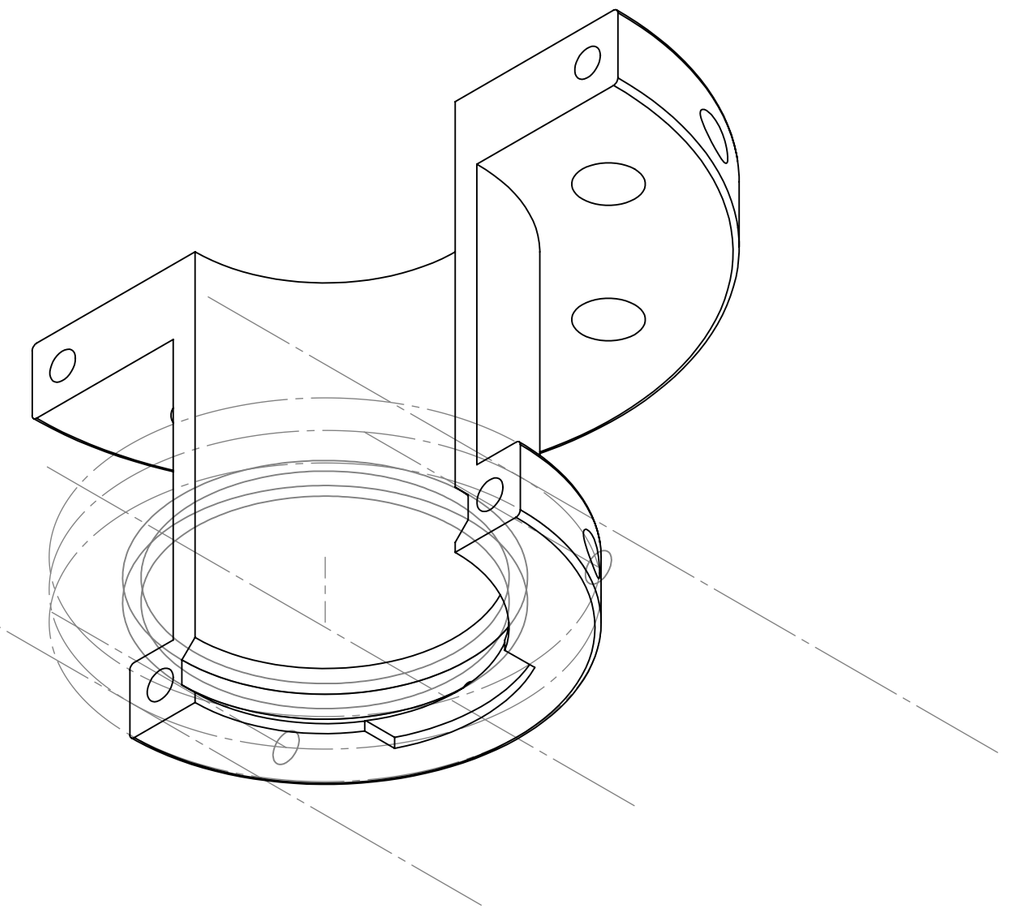


SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:			REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA	REVISIÓN
						Joaquín Macanás Valera	
	NOMBRE	FIRMA	FECHA			TÍTULO:	
DIBUJ.							
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL:		N.º DE DIBUJO: Acople brazo seccionado	A2
				PESO:		ESCALA:1:1	HOJA 1 DE 2

Universidad Politécnica de Cartagena





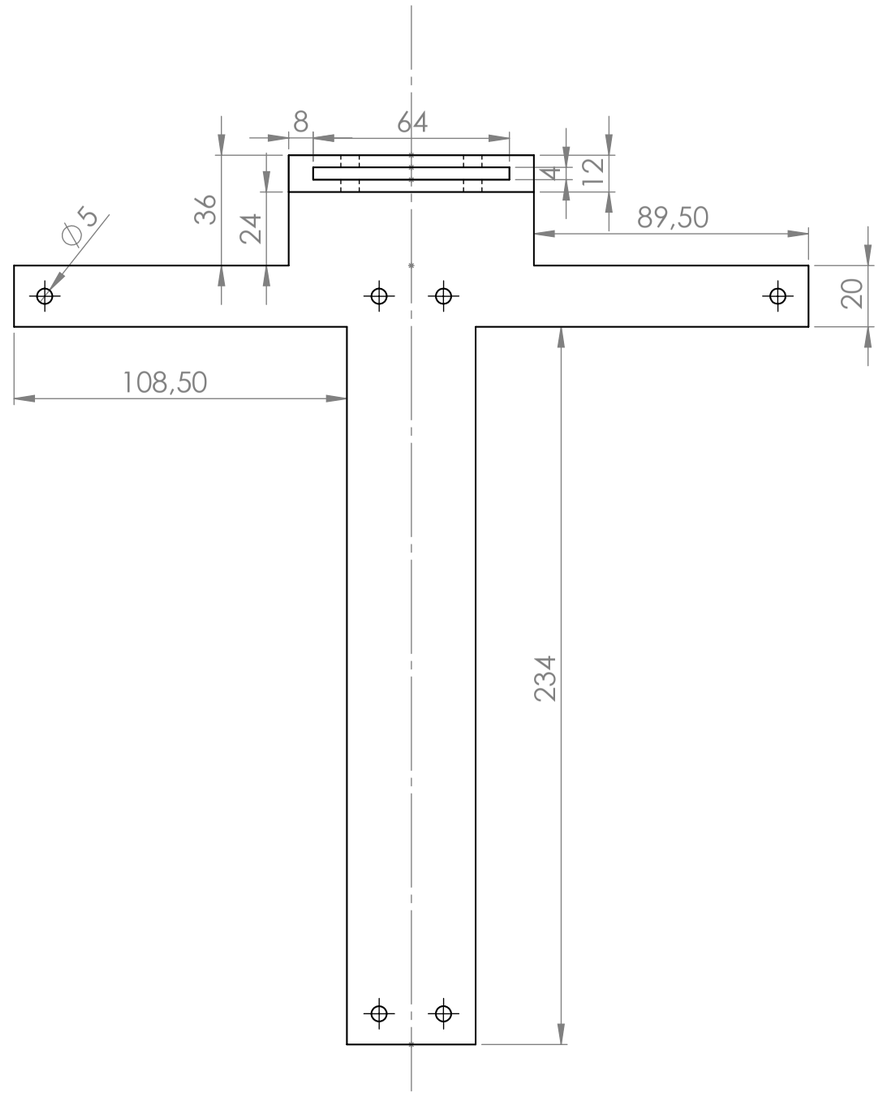


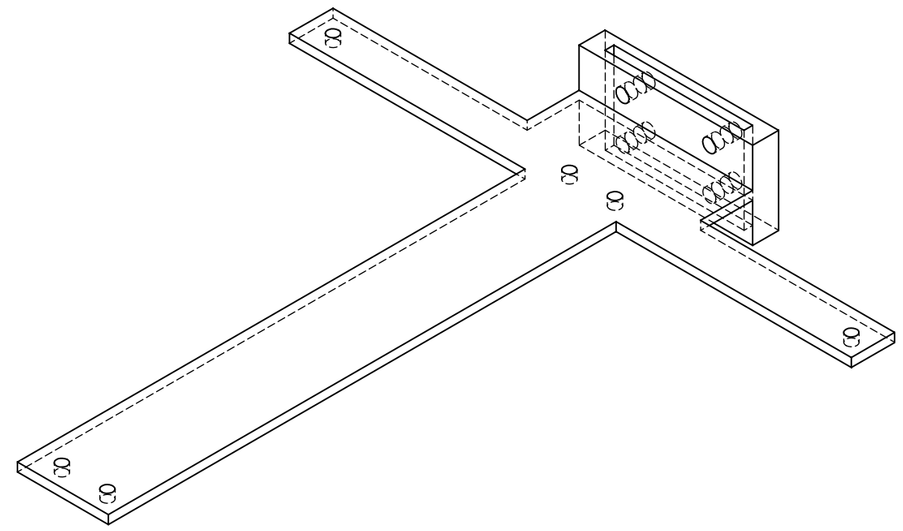
SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:				REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA / REVISIÓN
							Joaquín Macanás Valera
	NOMBRE	FIRMA	FECHA				TÍTULO:
DIBUJ.							
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL:			N.º DE DIBUJO: Acople brazo seccionado — A2
				PESO:			ESCALA:1:1 — HOJA 2 DE 2

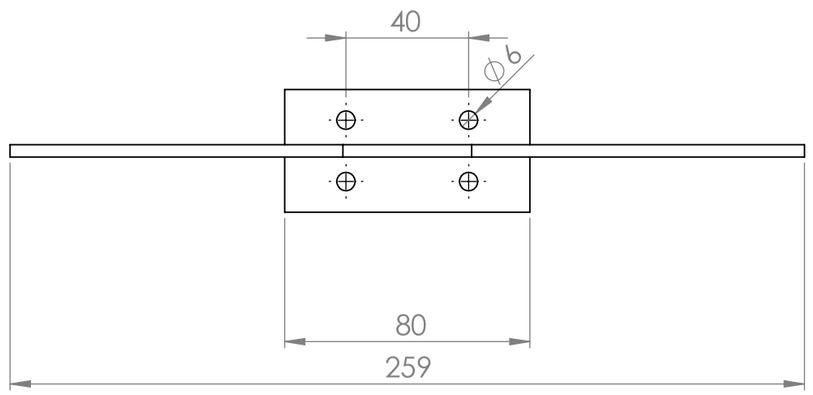
ANEXO Nº6

PLANOS DEL SOPORTE DEL SISTEMA DE VISIÓN

Universidad Politécnica de Cartagena

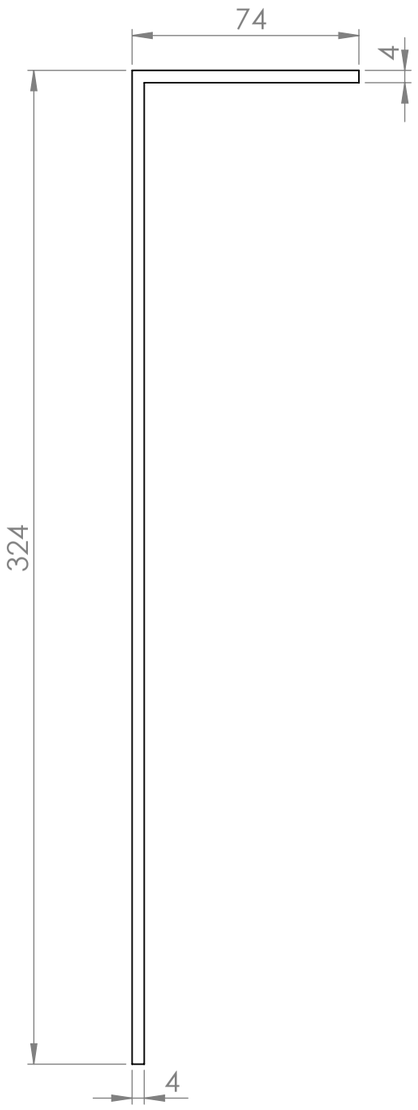


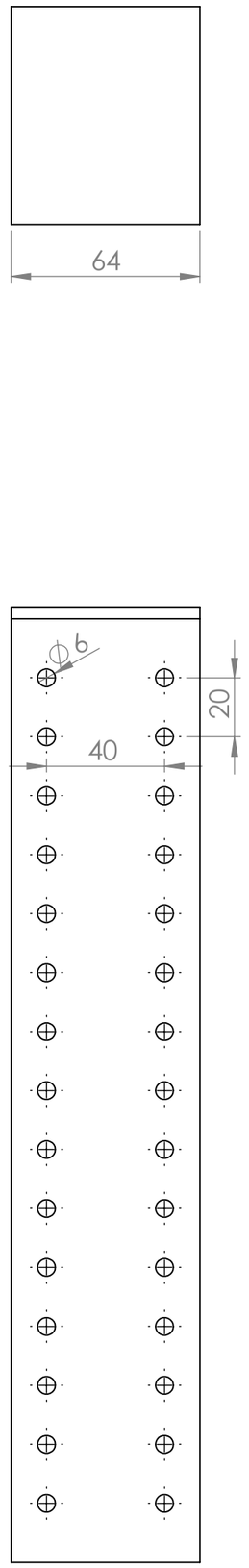


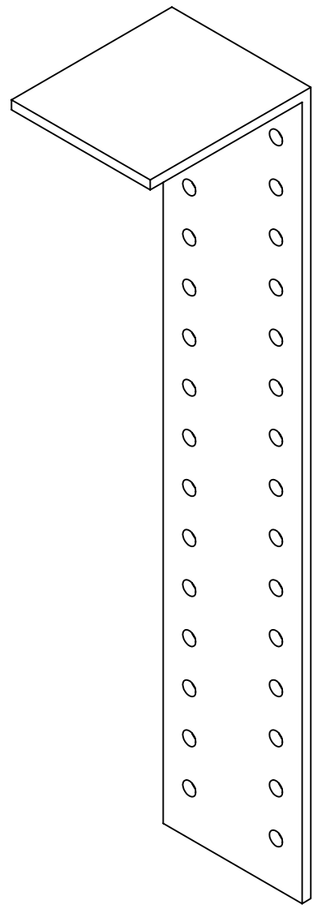


SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:		ACABADO:			REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA / REVISIÓN
						Joaquín Macanás Valera
	NOMBRE	FIRMA	FECHA			TÍTULO:
DIBUJ.						Soporte cámara horizontal
VERIF.						
APROB.						
FABR.						
CALID.				MATERIAL:		N.º DE DIBUJO / A2
				Acero inoxidable		
				PESO:		ESCALA:1:1 / HOJA 1 DE 1

Universidad Politécnica de Cartagena

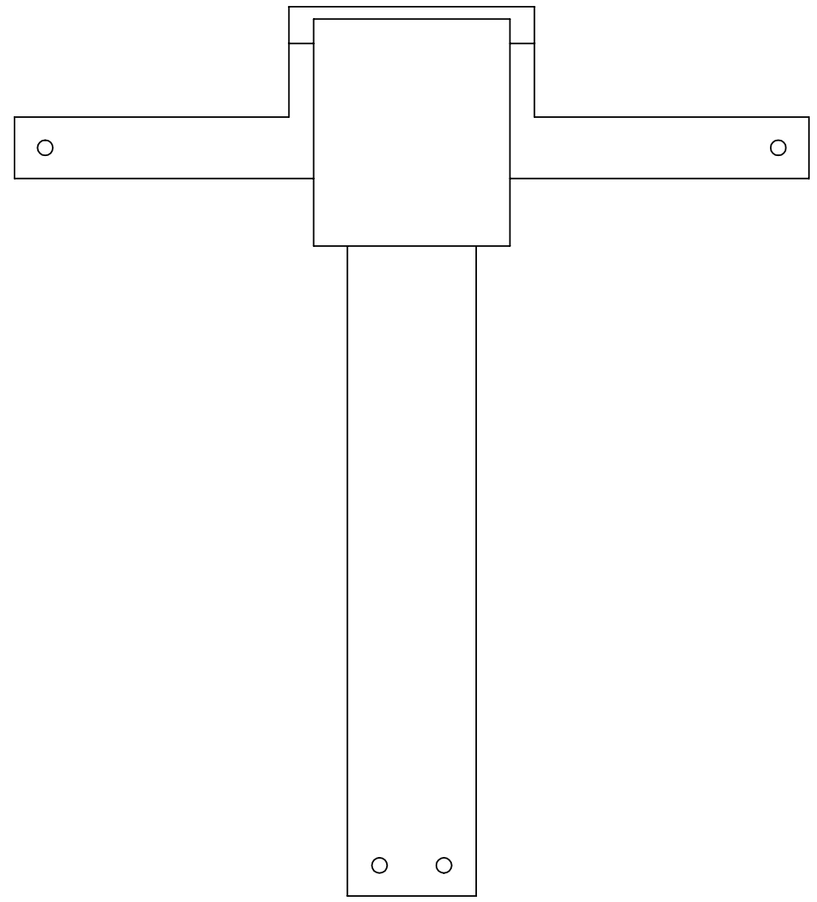


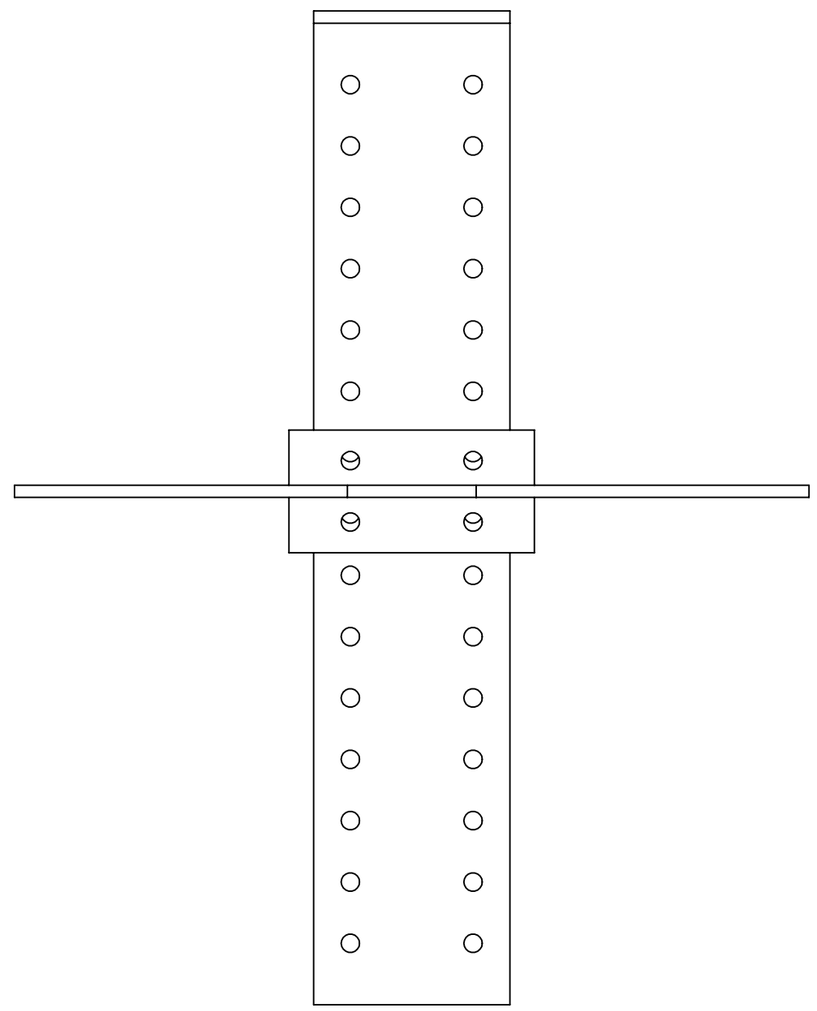


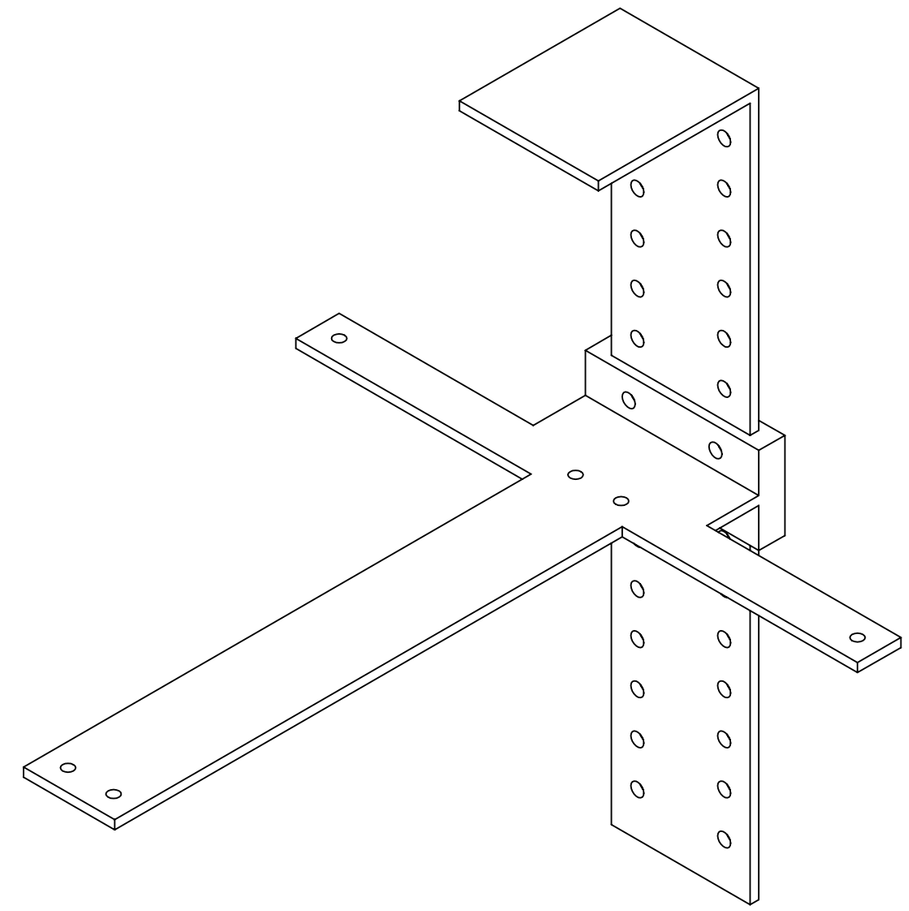


SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:			ACABADO:			REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA / REVISIÓN
	NOMBRE	FIRMA	FECHA				Joaquín Macanás Valera
DIBUJ.							TÍTULO: Soporte cámara vertical
VERIF.							
APROB.							
FABR.							
CALID.				MATERIAL: Acero inoxidable			N.º DE DIBUJO: A2
PESO:							ESCALA:1:1 / HOJA 1 DE 1

Universidad
Politécnica
de Cartagena







SI NO SE INDICA LO CONTRARIO: LAS COTAS SE EXPRESAN EN MM ACABADO SUPERFICIAL: TOLERANCIAS: LINEAL: ANGULAR:			ACABADO:			REBARBAR Y ROMPER ARISTAS VIVAS	NO CAMBIE LA ESCALA / REVISIÓN
	NOMBRE	FIRMA	FECHA				Joaquín Macanás Valera
DIBUJ.							TÍTULO:
VERIF.							Ensamblaje soporte cámara
APROB.							
FABR.							
CALID.				MATERIAL:			N.º DE DIBUJO: A2
				PESO:			ESCALA:1:1 / HOJA 1 DE 1

ANEXO Nº7

PCL – INSTALACIÓN Y PRIMEROS PASOS

PCL – INSTALACIÓN Y PRIMEROS PASOS

En este anexo se darán las directrices para poder instalar PCL en un ordenador con Ubuntu y posteriormente se explicarán algunos detalles acerca de los métodos más importantes para la adquisición de imágenes y su posterior tratamiento.

A. Instalación de PCL y OpenNI

PCL, al venir bajo el brazo de ROS viene por defecto instalado en las librerías de éste cuando instalamos su versión completa. Ahora bien, depende de la distribución que instalemos de ROS tendremos una y otra de PCL. La última versión que ha sido lanzada al mercado es la 1.7.1 (experimental). Con la distribución Groovy Galapagos, que es la que se ha usado en este proyecto, viene el paquete en su versión 1.6. Para gran cantidad de acciones es más que suficiente, pero para el análisis de formas 3D que se ha llevado a cabo en este proyecto mediante descriptores VFH sí era necesaria la última versión disponible estable (1.7).

La siguiente versión, Hydro Medusa, y por supuesto Indigo Igloo vienen con PCL instalado en las versiones 1.7 y 1.7.1 respectivamente.

En el caso de tener una versión de ROS más antigua, como en aquí sucede, es posible instalar PCL de forma independiente. Decir que PCL no depende de ROS para funcionar. No se requiere de ningún nodo operativo ni de que el ROS Master está trabajando en segundo plano.

Para instalar las librerías introducimos la siguiente instrucción:

```
$ sudo apt-get install libpcl-1.7-all libpcl-1.7-all-dev
libopenni-dev libopenni-sensor-primesense-dev -y
```

Con la instrucción anterior instalamos todas las librerías en la versión 1.7 y además instalamos los controlados de la Kinect, recogidos en las librerías de OpenNI.

En caso de no querer ROS o nuestra versión no coincida con la de PCL, o que simplemente queramos trabajar de forma independiente con estas librerías existe un repositorio privado en el cual está todo disponible. Para hacer uso de él únicamente se necesita añadirlo a la lista de repositorios disponibles de Ubuntu e instalar todo lo que en él se encuentra. Se procedería de la siguiente manera:

```
$ sudo add-apt-repository ppa:v-launchpad-jochen-
sprickerhof-de/pcl
$ sudo apt-get update
$ sudo apt-get install build-essential cmake libpcl-all
libpcl-all-dev -y
```

I. Librería adicional HDF5

Existe una librería llamada HDF5 que suele quedar omitida en la instalación y que es muy importante añadir para poder ejecutar ciertas funciones que generan descriptores de las nubes de puntos para identificarlas.

Para instalar esta librería lo más cómo y recomendable es hacerlo a través del Gestor del Paquetes Synaptics. Lo abrimos y en la barra de búsqueda tecleamos “HDF5”. De los diferentes paquetes que aparecen debemos seleccionar “h5utils”, “hdf5-tools” y “libhdf5-serial-dev”. Deberemos tener ya instalada con anterioridad (marcado con el cuadradito verde) el paquete “libhdf5-serial-1.8.4”. En caso contrario instalarlo también.

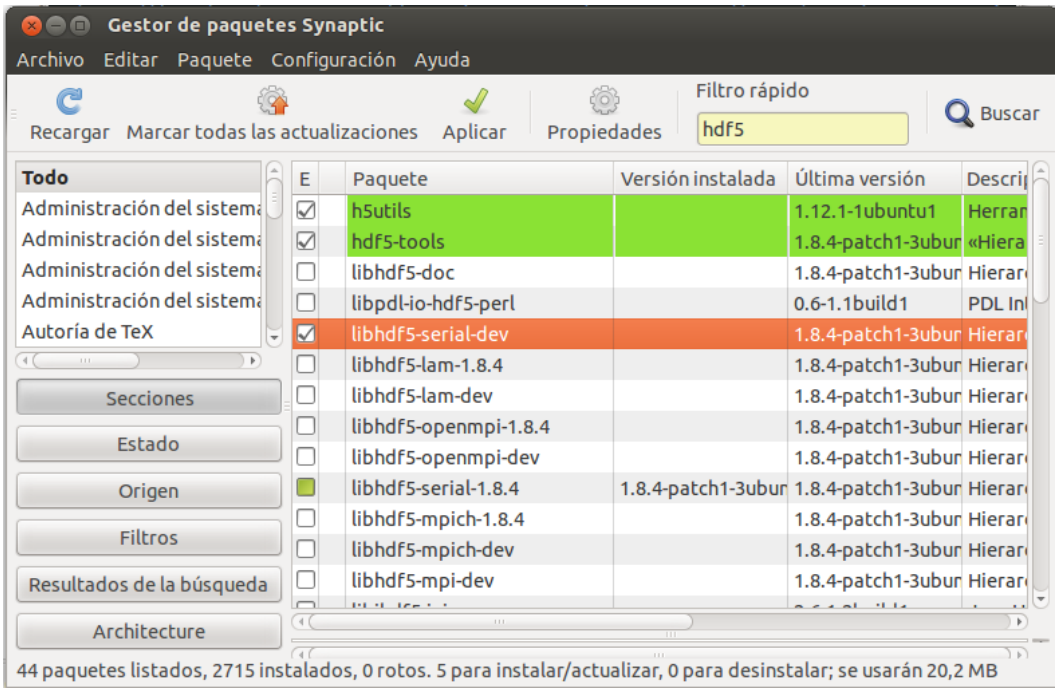


Ilustración 12.12 - Librerías HDF5

II. Archivo findFLANN.cmake

A la hora de compilar algún programa que precise de descriptores VFH para nubes de puntos será necesario hacer uso de la librería FLANN. Cuando se intenta compilar alguno de estos programas se suele obtener un error por consola que alerta de que no se ha encontrado el archivo "findFLANN.cmake" necesario para la construcción del programa. En realidad si buscamos este archivo está dentro del paquete de PCL, pero por algún motivo no sabe direccionar su ruta.

Sin embargo, la solución es bien sencilla. Basta con descargar dicho archivo desde el repositorio de PCL en GitHub y guardarlo en la carpeta del paquete al que pertenece el programa que queremos compilar. Hay que dejarlo fuera de cualquier subcarpeta. El enlace para encontrar el archivo en GitHub es:

https://github.com/PointCloudLibrary/pcl/tree/master/cmake/Modules

B. Primeros pasos en PCL

A la hora de iniciarse en el uso de las librerías de PCL es fundamental consultar los tutoriales que ofrece la propia página oficial de Point Cloud. Para

entender por tanto el código que se ha desarrollado en este proyecto se recomienda visitar el siguiente enlace:

http://pointclouds.org/documentation/tutorials/

Como introducción a PCL se repasarán las estructuras de datos que utiliza y algunos fundamentos básicos. Para ello nos apoyaremos en la documentación facilitada por Federico Tombari, PhD. en la Universidad de Bolonia y uno de los mayores expertos a nivel internacional en PCL, en su ponencia del *ROS-RM Workshop Alicante 2014*. No se harán revisiones de aplicaciones específicas pues para ello ya están los tutoriales anteriormente citados y tampoco es objeto del presente proyecto hacer un manual detallado de esta tecnología.

I. Representación de datos

La información 3D puede ser representada de diferentes formas. Principalmente puede ser dividida en 2 grandes grupos: organizada y desorganizada. Si los datos están desorganizados pueden presentarse como:

- **Nube de puntos**: lista desordenada de vértices.
- **Malla de polígonos**: lista desordenada de vértices y sus conexiones (topología).

Si los datos se presentan de forma organizada podrán ser como:

- **Mapa voxel binario**: una rejilla regular 3D de valores de densidad binarios.
- **Imagen de rangos**: una rejilla regular 2D de coordenadas 3D.

Los datos en 3D pueden almacenar información de una escena de 2 formas distintas:

- **3D puro**: representa únicamente la geometría de la escena.
- **RGB-D**: almacena la geometría de la escena así como la intensidad de color en los tres canales estándar (rojo, verde, azul).

Por consiguiente, los datos 3D pueden representar la realidad de 2 formas bien diferenciadas:

- Vista completa de 360° de la escena o el objeto (3D completo).
- Vista en 2.5D del objeto o la escena.

Como se puede apreciar hay multitud de formas de codificar la información proveniente de un sensor de profundidad. Al haber tantas codificaciones, parece lógico que exista una forma de migrar los datos entre unas y otras. En el siguiente gráfico se especifica en qué supuestos se deben hacer transformaciones entre distintos tipos de dato.

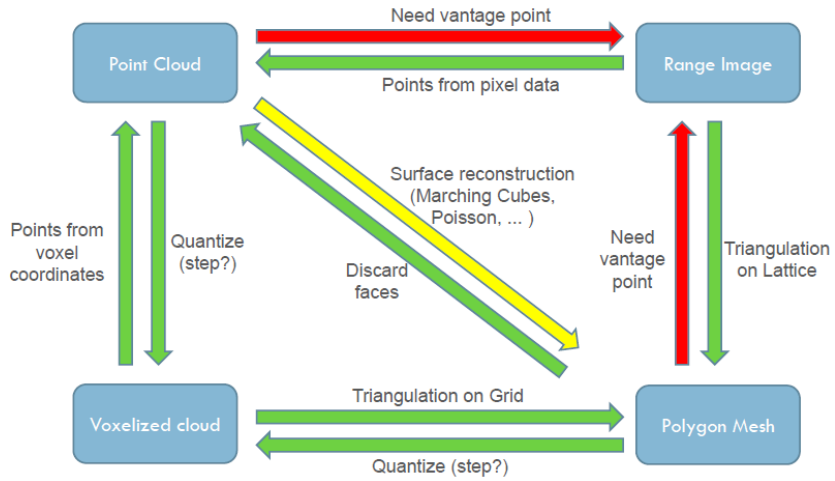


Ilustración 12.13 - Transformaciones entre tipos de dato

PCL es capaz de lidiar con datos tanto organizados como desorganizados. En caso de existir una estructura 2D subyacente disponible pueden ser usados esquemas de tratamiento eficientes (por ejemplo, imágenes integrales en vez de kd-trees para la búsqueda de vecinos cercanos).

Ambos tipos de dato son manejados por la misma estructura de datos "pcl::PointCLoud", una plantilla altamente personalizable. Los puntos pueden ser XYZ, XYZ+normals, XYZRGB, etc.

Las representaciones voxelizadas son implementadas sobre pcl::PointCloud + funciones de voxelización. La voxelización no es más que un proceso de discretización de la nube de puntos en pequeños cubos.

II. Metodología de trabajo

Normalmente el reconocimiento de objetos en escenas se hace a partir de datos en 2.5D, que no son sino vistas parciales de un modelos completo en 3D. También se puede hacer identificación entre 2 patrones 3D, pero no es lo usual pues las escenas no se pueden tener normalmente en 3D, sino que sólo tenemos una vista de ella, aunque el objeto lo sí lo tengamos modelado en 3D. Es por esto que lo que se suele hacer es obtener vistas en 2.5D del objeto a reconocer y luego hacer identificaciones con la escena.

Transformar un modelo 3D completo en uno 2.5D es muy sencillo, pues basta con elegir un punto de vista y obtener la vista. El proceso contrario implica una reconstrucción que no es precisamente sencilla. Es por esto que casi siempre se suele trabajar en el primer método descrito.

En la imagen inferior se puede comprobar cómo a partir de un modelo 3D de un tetrabrick se pueden obtener numerosas vistas del mismo en 2.5D.

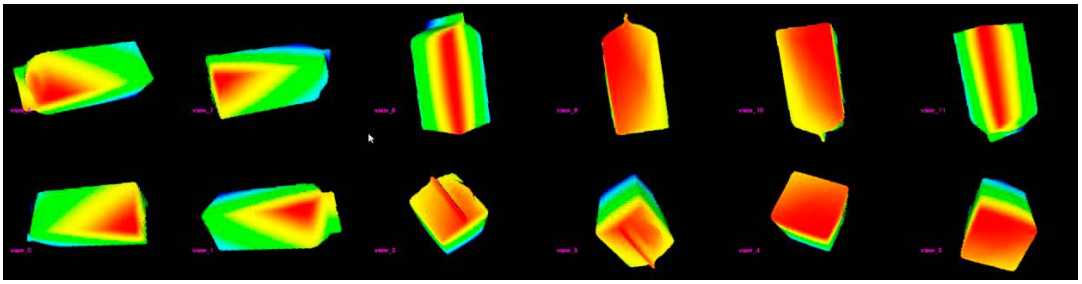


Ilustración 12.14 - Vistas 2.5D

ANEXO Nº8

INSTALACIÓN, CONFIGURACIÓN Y PRIMEROS PASOS DE MOVEIT!

INSTALACIÓN, CONFIGURACIÓN Y PRIMEROS PASOS DE MOVEIT!

A continuación se detallará cómo se puede instalar el paquete Moveit! en un PC que tenga como sistema operativo Ubuntu 12.04 y ROS Groovy Galapagos.

A. Instalación y configuración general

Para instalar el paquete completo introduciremos por terminal:

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

Si además queremos usar Moveit! con todos los paquetes del PR2 (el robot de Willow Garage) sobre el que se basan todos los tutoriales ofrecidos en la página oficial habrá que introducir el siguiente comando. Es totalmente recomendable para el que se está iniciando en el uso de este sistema.

```
$ sudo apt-get install ros-groovy-moveit-full-pr2
```

Tras la instalación es fundamental de cara a ejecutar cualquier sub-paquete de Moveit! actualizar las variables de entorno. Por tanto:

```
$ source ~/.bashrc
```

B. Instalación y configuración para desarrolladores

Esta instalación está indicada para aquellas personas que se vayan a dedicar a crear paquetes de Moveit! o a extender/mejorar los existentes. Lo que se necesita es tener instalada la versión base de ROS. En caso contrario dirigirse al siguiente enlace para ver cómo obtenerla.

```
http://www.ros.org/wiki/groovy/Installation/Ubuntu
```

Si hemos instalado la versión completa de ROS como se especificaba en el Anexo Nº2 el paso anterior no será necesario. Lo siguiente que hay que hacer es instalar el paquete "wstool". Posiblemente ya lo tengamos si tenemos la versión completa de ROS. En cualquier caso, introducimos el siguiente comando para conseguirlo:

```
$ sudo apt-get install Python-wstool
```

Ahora estamos en disposición de descargar todo el código necesario y guardarlo en una carpeta que crearemos y se llamará "moveit", en nuestro directorio personal.

```
$ source /opt/ros/groovy/setup.bash
$ mkdir moveit
$ cd moveit
$ mkdir src
$ cd src/
$ wstool init .
$ wstool merge https://raw.github.com/ros-
planning/moveit_docs/groovy-devel/moveit.rosinstall
$ wstool update
$ cd ..
```

Lo que ahora procede es verificar que todas las dependencias se instalan correctamente, algo fundamental para que los paquetes funcionen sin problemas.

```
$ rosdep install --from-paths src --ignore-src --
rosdistro groovy -y
```

Por último solo queda construir el paquete haciendo uso de la potente herramienta "catkin_make". Asumiendo que no hemos variado el directorio en el que nos encontramos desde la ejecución de la instrucción anterior, escribimos:

```
$ catkin_make
```

Por último, como tras la instalación de cualquier paquete nuevo, es importante añadir la ruta del paquete al ~/.bashrc. Para ello:

```
$ echo "source
/home/nombre_usuario/moveit/devel/setup.bash" >>
~/.bashrc
```

C. Primeros pasos en Moveit!

Para iniciarse en el mundo de Moveit! lo más aconsejable es seguir detenidamente los tutoriales ofrecidos por los propios desarrolladores en la página oficial. Dichos tutoriales se pueden encontrar a través del siguiente enlace:

http://moveit.ros.org/documentation/tutorials/