UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

IMPLEMENTACIÓN DE MIDDLEWARE ROS PARA ROBOT DE SERVICIO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA

PABLO HEVIA-KOCH

SANTIAGO DE CHILE
JUNIO 2012

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

IMPLEMENTACIÓN DE MIDDLEWARE ROS PARA ROBOT DE SERVICIO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA

PABLO HEVIA-KOCH

PROFESOR GUÍA:
JAVIER RUIZ-DEL-SOLAR

MIEMBROS DE LA COMISIÓN:
HÉCTOR MILER AGUSTO ALEGRÍA
MAURICIO ALFREDO CORREA PÉREZ

Esta memoria fue parcialmente financiada por el Proyecto FONDECYT 1090250

SANTIAGO DE CHILE
JUNIO 2012

RESUMEN DE LA MEMORIA
PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRICISTA
POR: PABLO HEVIA-KOCH
FECHA: JUNIO 2012
PROF. GUÍA: JAVIER RUIZ-DEL-SOLAR

IMPLEMENTACIÓN DE MIDDLEWARE ROS PARA ROBOT DE SERVICIO

El objetivo del presente trabajo de memoria consiste en implementar el middleware ROS en el robot de servicio Bender, así como implementar un sistema de navegación con soporte para SLAM online, e integrar dicho sistema junto con los módulos preexistentes del robot, al tiempo de actualizarlos a una nueva versión de URBI.

Este proyecto es parte del trabajo de mejora continua realizado con el robot Bender, del Laboratorio de Robótica del Departamento de Ingeniería Eléctrica de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile y enmarcado dentro de la participación de este robot en el campeonato mundial de robótica Robocup, en su categoría Robocup@Home.

Tras una presentación del contexto y antecedentes respecto a las características del *middleware*, se procedió a implementar un sistema de navegación en base al *middleware* ROS utilizando el algoritmo de SLAM Online GMapping. Para esto, se implementan diversos módulos ROS: control de la base robótica, publicación de odometría, manejo de marcos de referencia, generación de mapas de costos, módulo SLAM, lecturas del sensor láser, planificador de rutas local y global, teleoperación, servidor de mapas de entorno y localización.

Implementados los módulos de navegación en ROS, se procedió a actualizar los módulos UObjects existentes a URBI 2.7.4 y finalmente a integrar el sistema ROS con el sistema URBI, ambos residentes en computadores diferentes y bajo distintos sistemas operativos.

Finalizado el desarrollo del sistema y su integración, se realizaron diversas pruebas para verificar el módulo de navegación desarrollado en base a ROS, así como la integración de éste a los módulos preexistentes basados en URBI que fueron actualizados y la comunicación entre ambos componentes. En base a estas pruebas, se verifica que el trabajo realizado cumple a cabalidad los objetivos planteados, tanto al cumplir las necesidades de navegación (evasión de obstáculos, planificación dinámica de rutas, SLAM online), como al mantener las funcionalidades preexistentes e integrar los sistemas antiguos dentro de la nueva configuración del sistema.

Para Pablo y Matilde....

Agradecimientos

"The real voyage of discovery consists not
in seeking new landscapes but in having new eyes."

-M. Proust

Agradecer a todos quienes me han acompañado en este proceso es una tarea para nada fácil, porque siendo tantas las personas que me gustaría nombrar, es casi seguro dejar alguna en el tintero. A pesar de lo anterior, haré el intento de hacerle justicia a cada uno de ustedes, quienes nombrados o no, significan mucho en mi vida.

En primer lugar quisiera agradecer a mi madre, la mujer más valiente que conozco, por atreverse a realizar el viaje más difícil de todos, el de conocerse a uno mismo. Quisiera agradecer también a mi padre, por mostrarme como la justicia y la igualdad pueden no solo ser un sueño, si no un camino de búsqueda.

A mi hermana, la persona más importante del mundo, y a quien dedico este trabajo, por enseñarme lo que es el amor incondicional. A mis abuelos, mi tía Cecilia y mi tío Pablo por su cariño, sus consejos y ser un constante modelo a seguir en casi todos los aspectos de mi vida.

Me es importante agradecer también a mis hermanos Fernando y Ramiro, que han sido una inspiración y un apoyo constante en mi vida, y que a pesar de la distancia, han sido partícipes de una u otra forma de este proceso.

Fuera del ambiente familiar, quiero agradecer a mis compañeros de departamento Nico y Juanin, por su grata compañia, sus consejos y particulares puntos de vista. A Sebastián Rivas, por los momentos de risa compartidos, así como por las infinitas gestiones que realizó en mi nombre en la universidad. A mis vecinas Claudia y Nancy, por su cariño y su amistad siempre ligada a algún rico té compartido.

A mis compañeros de laboratorio y colegas durante varios años, Mauricio, Isao, Pablo, Paul y Fernando, cuya amistad, consejos y guía han servido como una fuente de inspiración y apoyo constante estos últimos 5 años. A Javier, por entregarme las oportunidades necesarias y el apoyo para este y otros trabajos.

A mis amigas Valentina Fiedler, Jasmin Pardakhty y Rosemarie Wiener, que a pesar de los vaivenes de la vida y las distancias geográficas en apariencia irremontables, han estado siempre ahí para mi.

Agradecer a todos, en definitiva. A todos los que han compartido algún momento conmigo en estos últimos años, y que tal vez sin saberlo, crearon los momentos que de a poco me han ido transformando en la persona que soy hoy.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Los sistemas robóticos, son sistemas cada vez más complejos, e incluye una gran cantidad de componentes, tanto de hardware como de software, que interactúan entre sí y con el entorno. Dada lo anterior, el control del sistema, así como la sincronización y comunicación entre sus módulos es un problema no trivial de resolver.

ROS[1] [2] es un proyecto de software creado por la empresa WillowGarage [8] que no solo permite resolver el problema de comunicación de procesos (es decir, entregar funcionalidad de *middleware*), si no que también otorga funcionalidades extra, como la existencia de librerías asociadas, y un entorno de desarrollo y administración de módulos y librerías.

Desde el lanzamiento de su primera versión en marzo de 2010 *Boxturtle* [3], ROS ha sido incorporado en variados proyectos robóticos, lo que ha generado una comunidad mundial de usuarios y desarrolladores que mantienen y actualizan tanto el sistema base ROS, como diversos paquetes y librerías asociados que le otorgan funcionalidades de distinto tipo.

En la actualidad, existen más de 2000 librerías que entregan funcionalidades varias, incluyendo procesamiento de imágenes, navegación, control de hardware, y más. Todas estas librerías son *open-source*[2] y pueden ser integradas de forma modular a un sistema basado en ROS. Dado el crecimiento de ROS, así como su actual ubicuidad en sistemas robóticos tanto comerciales como de investigación, es beneficioso adoptarlo en el robot Bender para resolver los problemas actuales existentes y de paso adquirir nuevas funcionalidades, orientado a la competición Robocup@Home.

[1] Robot Operating System

[2] El software de código abierto, en inglés *open-source*, es software desarrollado y distribuido de forma libre y gratuita, por lo que el acceso al código fuente es público, y existe libertad de utilizarlo y modificarlo a opción.

1.1.1. Robocup@Home

La liga Robocup@Home [1] es una categoría dentro del campeonato internacional Robocup. El objetivo de la liga es incentivar el desarrollo de tecnología robótica de servicio y asistiva con alta relevancia para aplicaciones futuras de orden doméstico. Es la competencia internacional anual más grande de robots de servicio autónomo.

Para evaluar los robots, existe una serie de pruebas estándar que determinan la habilidad y rendimiento de estos, tanto en ambientes hogareños no estandarizados, como en tiendas y supermercados. Las áreas principales evaluadas son: Interacción y cooperación entre humanos y robots, navegación y mapeo en ambientes dinámicos, visión computacional y reconocimiento de objetos y rostros en condiciones de iluminación natural, manipulación de objetos, conductas adaptativas, integración de conductas, inteligencia ligada al contexto, estandarización e integración de sistema, entre otros.

Fundada en 2006, la liga Robocup@Home ha crecido llegando a tener 25 equipos participando en su versión del año 2011. El equipo de la Universidad de Chile en esta liga es UChile Homebreakers [5], que con su robot Bender cuenta con 2 premios a la innovación, y una posición máxima del 5º lugar, obtenida en Robocup 2010 Singapore.

1.1.2. Robot de Servicio Bender

Bender es el robot del equipo UChile Homebreakers, creado a partir de un proyecto del curso Seminario de Diseño en Mecatrónica. Una imagen del robot en su versión actual puede verse en la Figura 1.1.

Figura 1.1: Robot Bender

Bender es un robot que presenta un grado de complejidad bastante alto, donde interactúan diversos sistemas, tanto de software como de hardware, en tiempo real y en conjunto para realizar las tareas a cumplir. La idea principal detrás de su diseño corresponde a tener una plataforma abierta

y flexible donde probar nuevos desarrollos, y que al mismo tiempo tenga un aspecto abordable y natural, que facilite la interacción entre humanos y el robot.

Hardware

El robot bender está compuesto por una base móvil, modelo Pioneer 3-AT, sobre la cual se encuentra montado el torso del robot, a través de un perfil de aluminio atornillado a la parte superior de la base. Sobre el torso se encuentran montados ambos brazos y la cabeza. El robot tiene una altura de $155[cm]$ y un ancho máximo de $69[cm]$.

A continuación, se presenta una descripción de cada elemento del robot:

Torso En su torso, el robot cuenta con un tablet PC como plataforma principal de procesamiento, un HP2710p, que cuenta con un procesador Intel Core2Duo de 1.2GHz y 2GB DDRII 667MHZ RAM. El dispositivo, con sistema operativo Windows XP Tablet PC edition, permite conectividad vía Ethernet y 802.11b. La pantalla del tablet permite: (i) Visualizar información relevante al usuario, como imágenes, videos, navegador de internet; y (ii) el ingreso de datos e información a través de su pantalla táctil, a modo de interacción.

Cabeza La cabeza del robot incorpora dos cámaras: una cámara CCD estándar, modelo Philips ToUCam III-SPC900NC, en la posición del ojo izquierdo; y una cámara térmica, modelo FLIR TAU 320, en la posición del ojo derecho. Ambas cámaras se conectan al tablet PC a través de USB.

La cabeza puede ejecutar movimientos de paneo tanto de forma vertical como horizontal y además tiene la capacidad de mostrar emociones. Esto se logra a través de varios servomotores que mueven la boca, las cejas y las orejas del robot, además de varios LED de color ubicados en los ojos. Las expresiones y movimientos son controladas utilizando hardware dedicado basado en un PIC18F4550, que comunican el hardware de la cabeza con el tablet PC vía USB.

Visión 3D Ubicado bajo el torso, existe un Microsoft Kinect, que permite en base a tecnología *"Light Coding"*, obtener una visión tridimensional de su entorno. Esto es utilizado para detectar personas, así como para detectar objetos durante el proceso de agarre de estos.

Brazos El robot cuenta con dos brazos diseñados para permitirle al robot manipular objetos, y tienen suficiente fuerza como para levantar un vaso grande con agua o una taza de café. Cada brazo cuenta con 6 grados de libertad: dos en el hombro, dos en el codo, uno en la muñeca, y uno en la mano de apertura monodireccional. Los actuadores son 8 servomotores Dynamixel, 2 modelo RX-28 y 6 modelo RX-64, que son controlados desde el tablet PC, a través de USB.

Brazo Grip El brazo grip es un tercer brazo, ubicado en la base móvil, que permite manipular objetos a nivel del suelo. El brazo grip cuenta con 5 grados de libertad: dos en el hombro, uno en el codo, uno en la muñeca, y uno en la mano de apertura bidireccional. Los actuadores

son 5 servomotores Dynamixel, 3 modelo RX-64 y 2 modelo RX-28, controlados desde el tablet PC por USB.

Plataforma Móvil Todas las estructuras anteriores están montadas sobre una plataforma móvil modelo Pioneer 3-AT. Esta plataforma cuenta con cuatro ruedas traccionadas a pares, que permiten al robot moverse y girar a través de derrapamiento. Sobre la base va montado también un sensor láser Hokuyo URG04-LX utilizado para la navegación del robot. Sobre esta misma base se encuentra un computador Alienware M11x R3, con procesador Intel i7, que controla el sensor láser, la base móvil, y ejecuta algoritmos de navegación y mapeo. Este computador se conecta al tablet PC del pecho a través de un cable Ethernet.

Software

Los componentes principales de la arquitectura de software pueden verse en la Figura 1.2.

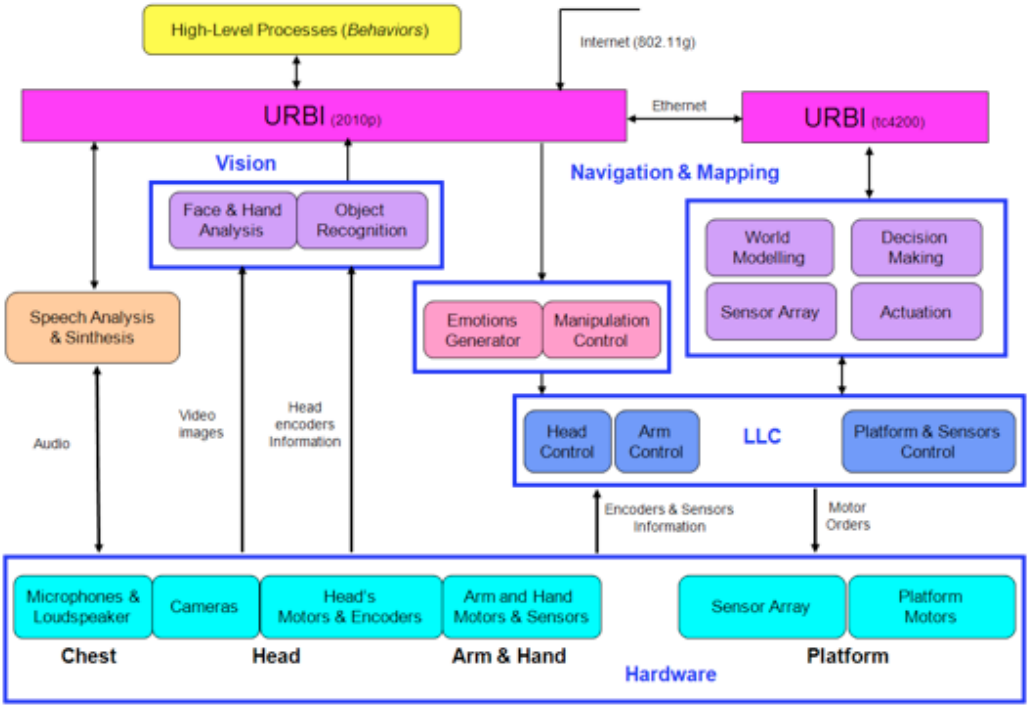


Figura 1.2: Arquitectura de Software del sistema

Las tareas de síntesis y analisis de voz, visión computacional (reconocimiento de objetos, caras, manos y gestos) y el control de manipulación de objetos se ejecutan en el tablet PC HP2170p, bajo Windows XP Tablet PC edition; mientras que las tareas de navegación y generación de mapas a desarrollar lo harán en el computador Alienware M11x R3, bajo Ubuntu Linux. Los módulos de control de bajo nivel (*Low-Level-Control*, o *LLC*) se ejecutan en hardware dedicado basado en PIC.

Los módulos que residen en el tablet PC son implementados a través de UObjects[3] y controlados a través de URBI [6], y será necesario comunicarlos al sistema a desarrollar.

1.1.3. Aporte del Trabajo a Bender

La implementación de un *middleware* basado en ROS es un paso necesario para estandarizar el desarrollo en el robot Bender, así como para permitir la integración de personas nuevas al equipo de trabajo. Esto debido a que el sistema integrado a ROS será de alta modularidad e independencia, lo que permitirá usar las habilidades específicas de desarrollo de cada persona en módulos independientes y acotados sin que cada integrante del equipo deba conocer el funcionamiento completo del sistema. Además, la existencia de un estándar, permitirá la reutilización de código abierto desarrollado por terceros, lo cual es esencial para poder focalizar el desarrollo en elementos nuevos, minimizando el tiempo utilizado en rehacer soluciones ya existentes.

Ligado a esto, la implementación del nuevo sistema de navegación entregará mayor flexibilidad al sistema, además de funcionalidades que hoy en día se encuentran ausentes: simulación completa del sistema, registro de operación del sistema, reemplazo transparente de módulos y abstracción de hardware.

1.2. Objetivos

El objetivo general del trabajo consistirá en instalar ROS en los computadores del robot Bender, determinar la estructura de comunicación y módulos del sistema, e integrar parte de los módulos ya existentes al nuevo sistema. Es esencial que dado que el proyecto se encuentra enmarcado en la competencia Robocup@Home, el sistema implementado sea capaz de realizar las tareas necesarias asociadas, como SLAM online, edición rápida de mapas e inclusión de información semántica a estos (como identificar zonas por nombre).

En base a lo anterior, se propone una serie de objetivos específicos asociados a la realización del objetivo general:

- Determinar la estructura de módulos del sistema en los diversos computadores.
- Determinar los métodos de comunicación entre los módulos del sistema.
- Instalación de sistemas operativos en los computadores a utilizar.
- Instalación de sistema de navegación basado en ROS con capacidad de integración de información semántica al mapa.
- Integración y validación de módulo SLAM Online al robot.

[3] Objetos de código implementados en interfaz con el sistema URBI. Son revisados en mayor detalle en el Capítulo 2 del presente texto.

- Reimplementación de módulos UExec, UVision2, USpeech, UMainGui, UHead, ULaserTracker y UPersonTracker de URBI 1.0 a URBI 2.7.4
- Integración de ROS con sistema URBI para *scripting*[4].

1.3. Estructura de la memoria

La presente memoria se encuentra dividida en 6 capítulos, incluyendo el actual. El primer capítulo corresponde a la introducción de la memoria, y presenta las motivaciones y objetivos del trabajo a realizar. En el segundo capítulo se presenta el contexto teórico del trabajo. El capítulo tercero muestra el desarrollo del sistema de navegación en sí, incluyendo las decisiones de diseño tomadas y la implementación específica realizada, bajo el contexto de utilizar el *middleware* ROS. El capítulo cuarto, detalla la integración del sistema desarrollado en el capítulo tercero, con el resto de los módulos existentes en el robot Bender, y la adaptación de estos para funcionar con la nueva versión del software URBI. En el capítulo quinto se presentan pruebas de funcionamiento del sistema completo, incluyendo la integración del sistema de navegación con los módulos antiguos existentes en el robot Bender. Finalmente, en el capítulo sexto se presenta un análisis del trabajo hecho, y conclusiones respecto a lo realizado, así como sugerencias para posible trabajo futuro.

[4]Un lenguaje de *scripting* es un lenguaje de programación orientado a la generación de "libretos", pequeños programas no compilados que automatizan procesos, y que pueden ser ejecutados de forma interpretada

Capítulo 2

Antecedentes

2.1. ROS

ROS es un meta-sistema operativo[1] para robots que provee una serie de servicios orientados a facilitar el desarrollo de software para robots, así como su operación. Entre los servicios ofrecidos presenta abstracción de hardware, control de bajo nivel, comunicación entre procesos, y manejo de paquetes de software. Además, incluye herramientas y librerías para facilitar el desarrollo, compilación y ejecución de código de forma distribuida en diversas plataformas de procesamiento.

ROS está diseñado teniendo en mente la reutilización de código, por lo cual se espera que sea lo más transparente posible. Esto significa que debe ser posible integrar ROS con librerías ya existentes, así como entornos de desarrollo para robótica, elemento esencial en el trabajo que se plantea en el presente documento. Por otra parte, ROS soporta actualmente diversos lenguajes, C++, Python y Lisp, mientras se desarrolla soporte para Java y Lua. Esto permite una mayor flexibilidad a la hora de programar y de implementar código ya existente.

2.1.1. Organización de Grafo Computacional de ROS

El grafo computacional de ROS representa todos los procesos y los distintos tipos de conexiones que pueden existir entre ellos, además de los mensajes enviados entre sí. Existe una serie de conceptos que representan cada uno de los elementos involucrados, que serán explicados a continuación:

[1] Se denomina meta-sistema operativo puesto que si bien entrega características propias de un sistema operativo, como comunicación entre procesos, se ejecuta encima de otro sistema operativo completo, como Linux.

Nodos

Los nodos son los procesos ejecutables que realizan el cómputo. En un sistema de control robótico se espera que existan varios nodos, donde cada uno se encargue de ejecutar una misión particular. Por ejemplo, un nodo puede encargarse de leer el dispositivo laser, otro de realizar el planeamiento de trayectorias, otro otorga una interfaz gráfica de control, etc.

Master

El programa *master* es el proceso central que mantiene un registro de todo el grafo computacional. Es quien permite la comunicación entre los nodos. Además, cuenta con un servidor de parámetros que permite centralizar cierta información.

Mensaje

Los nodos se comunican entre sí enviándose mensajes. Un mensaje es una estructura de datos simple basada en campos de datos con tipo, que pueden ser tipos primitivos (integer, floating point, boolean, etc.) o arreglos de estos. En ROS los mensajes son de tipado fuerte y estático[2].

Los tipos de mensajes son definidos en archivos de texto con terminación *.msg*, y sus nombres siguen la nomenclatura de ROS, es decir, el mensaje definido en *std_msgs/msg/String.msg* es de tipo *std_msgs/String*.

Tópico

Los mensajes se envían generalmente en base a un sistema de publicación/suscripción. Un nodo envía un mensaje publicándolo bajo cierto tópico, cuyo nombre identifica el contenido del mensaje (como por ejemplo odometría). Todos los nodos que quieran recibir la odometría, se suscriben entonces a este tópico y recibirán todos los mensajes que el nodo original publica, sin que este nodo se deba preocupar de quienes están suscritos o no a él. Al desacoplar la publicación de la información de su consumo, se facilita y generaliza el proceso y simplifica el desarrollo de nodos.

Los tópicos son fuertemente tipados, es decir, el tipado del mensaje debe estar definido de antemano, y solo se permitiran mensajes dentro del tópico cuyo tipo calce con el tipo declarado en el tópico.

[2]En los lenguajes de programación, el tipado hace referencia a la manera de clasificar variables y valores en tipos. En algún momento del proceso de compilación y ejecución del programa, se deben hacer revisiones de verificación de tipos, de tal manera de revisar que las expresiones hagan sentido y verificar la pérdida de datos al hacer conversions de un tipo a otro. Cuando esta revisión se realiza al momento de compilación, se habla de un tipado estático. De la misma manera, cuando solo se permiten conversiones automáticas de tipo donde no se pierda información, se habla de tipado fuerte.

Servicio

Un sistema alternativo de comunicación entre nodos, en vez de un paradigma de publicación/suscripción presenta un sistema basado en peticiones y respuestas. Un nodo puede ofrecer un servicio, que define dos tipos de mensajes: una petición y una respuesta. Un nodo puede enviar un mensaje del tipo petición al nodo que ofrece el servicio, y este le responderá un mensaje del tipo respuesta. Al igual que en los casos anteriores, el mensaje de petición así como el de respuesta son fuertemente tipados.

El siguiente gráfico resume entonces los dos métodos de comunicación entre nodos:

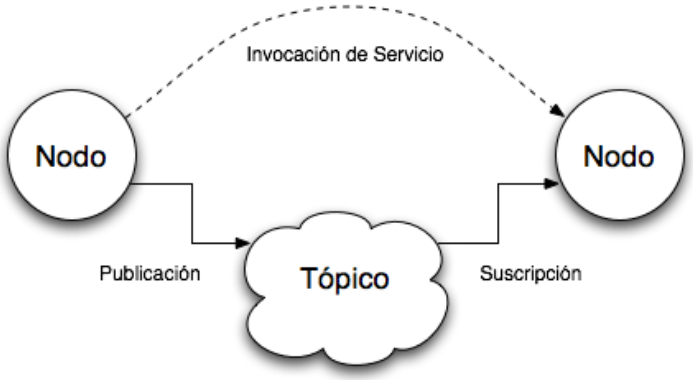


Figura 2.1: Métodos de Comunicación entre Nodos

De la figura anterior, se puede observar las dos maneras de comunicarse entre los nodos: En primer lugar, a través de un topic de comunicación, donde un nodo publica información en este tópico, y esta es recibida por todo nodo que esté suscrito al mismo topic. Por otra parte, es posible que un nodo se comunique directamente con otro nodo a través de la invocación de un servicio.

Servidor de Parámetros

En ROS, existe un servidor de parámetros, llamado *Parameter Server*. Este servidor es un diccionario multivariable accesible por red,y compartido entre todos los elementos del sistema, que permite almacenar y modificar parámetros del sistema, que son utilizados por los nodos para determinar su funcionamiento.

2.1.2. Roslaunch

Roslaunch es una utilidad del sistema ROS, que permite inicializar múltiples nodos de forma tanto local como remota (a través de SSH), así como definir parámetros en el servidor de parámetros de forma automática. Además, permite realizar diversas configuraciones de forma automática como

el balanceo de carga, y volver a lanzar de forma automática procesos que hayan terminado su ejecución.

Esta herramienta lee archivos de terminación *.launch* en formato XML, que definen los nodos a levantar, los parámetros a declarar y las características de configuración asociadas.

Funcionamiento

Para ejecutar un archivo *.launch* con la herramienta *roslaunch* es necesario especificar también el paquete al cual pertenece:

```
roslaunch nombre_paquete archivo.launch [argumentos]
```

Otra opción corresponde a especificar la ruta completa o relativa al archivo a lanzar:

```
roslaunch ruta/al/archivo.launch [argumentos]
```

Ambas opciones ejecutarán de forma correcta el archivo.

Archivos .launch

Los archivos *.launch*, llamados *launchfiles*, son archivos XML, donde cada *tag* define nodos a levantar, así como ciertos parámetros a entregarles. A continuación se muestran algunos tags importantes y su funcionamiento:

<**node**> El tag *node* se utiliza para levantar nodos dentro del sistema, ya sea de forma remota o local. Cuenta con 4 argumentos principales: *name*, el nombre para identificar el nodo; *package*, el paquete donde se encuentra el nodo; *type*, el tipo de nodo (o nombre del ejecutable) y *args*, los argumentos de ejecución a entregarle al nodo.

Modo de uso:

```
<node name="map_server" pkg="map_server" type="map_server"
args="map3.png 0.1"/>
```

<**include**> El tag *include* permite que roslaunch incluya otros archivos *.launch* en el presente, importándolos y ejecutándolos dentro del espacio del archivo actual. Esto permite organizar los *launchfiles* de forma jerárquica, simplificando la presentación y organización de estos. Su argumento principal es *file*, que define el archivo a importar.

Modo de uso:

```
<include file="hardware_controllers.launch"/>
```

<**param**> El tag *param* se utiliza para definir un parámetro en el servidor de parámetros. Este tag puede incluirse de forma anidada a un tag *node*, en cuyo caso los parámetros definidos serán considerados parámetros privados del nodo. Sus argumentos principales son: *name*, que define el nombre del parámetro en el formato *namespace/name* y *value*, que define el valor del parámetro.

Modo de uso:

```
<param name="publish_frequency" type="double" value="10.0" />
```

<**rosparam**> El tag *rosparam* se utiliza para cargar, guardar y eliminar parámetros al servidor de parámetros en lotes a través de archivos YAML[3]. Este tag extiende la funcionalidad del tag *param* permitiendo cargar muchos parámetros a la vez de forma ordenada sin declararlos en el mismo *launchfile*. Tiene 3 argumentos principales: *command*, que define si se cargarán, descargarán o eliminarán los parámetros; *file*, que define el archivo desde el cual leer los parámetros a cargar y *param*, el nombre del parámetro a eliminar.

Modo de uso para cargar parámetros:

```
<rosparam command="load" file="parametros.yaml" />
```

Modo de uso para eliminar un parámetro:

```
<rosparam command="delete" param="namespace/parametro" />
```

2.1.3. Marcos de Referencia

Un marco de referencia corresponde a una serie de convenciones utilizadas para definir diversas magnitudes físicas, como posición y orientación, entre otras; desde el punto de vista de un observador arbitrario. Dentro del sistema a desarrollar, se definen como marco de referencia un sistema de coordenadas ortogonales solidario a algún punto del entorno, o del robot.

En un sistema robótico, es común que exista una gran cantidad de marcos de referencia, y que estos además varíen en el tiempo, como por ejemplo el sistema de coordenadas asociado al mapa, el sistema asociado a la base móvil, el sistema asociado a la cabeza, y más.

En ROS, existe un paquete llamado *tf* que permite simplificar el manejo de los marcos de referencia, las relaciones entre estos, y las transformaciones de un marco de referencia a otro. El paquete *tf* se encarga de que todos los nodos del sistema tengan la información completa respecto a los marcos de referencia existentes en el sistema, y la relación entre estos.

El paquete *tf* permite a cada nodo, enviar o recibir sistemas de coordenadas y la relación entre estos, de forma fácil y automática. Además está diseñado para ser resistente a fallas, de tal forma

[3]YAML, del inglés *"YAML Ain't a Markup Language"*, es un formato de serialización de datos que permite determinar estructuras de datos de forma de ser fácilmente interpretados tanto por máquinas como por humanos

que incluso frente a la falla de algún nodo que publique un marco de referencia, sea posible acceder a la versión más reciente de este.

Las relaciones entre marcos de referencia son del tipo ”padre-hijo”, y queda definido a través de un cuaternión que describe las rotaciones, y un vector que define las traslaciones (en los ejes x, y y z).

2.1.4. Cuaterniones

Los cuaterniones son un sistema numérico hípercomplejo, que extiende a los números complejos, y que aplicados a la mecánica tridimensional, permiten representan rotaciones de cuerpos en el espacio, de forma más precisa que los ángulos de Euler [12].

Los cuaterniones, matemáticamente, corresponden a un espacio vectorial de cuatro dimensiones sobre los números reales, formado sobre la base de 3 vectores unitarios: $\hat{i}, \hat{j}, \hat{k}$, tal que $\hat{i}^2 = \hat{j}^2 = \hat{k}^2 = \hat{i}\hat{j}\hat{k} = -1$. De esta manera, un cuaternión sigue la forma $a + b\hat{i} + c\hat{j} + d\hat{k}$.

En ROS, según el estándar de unidades de medida y convenciones de coordenadas [4], la manera preferida de representar toda rotación es a través de cuaterniones. Esto se debe a que los cuaterniones no presentan singularidades como los ángulos de Euler, mientras que mantienen una representación más compacta que las matrices de rotación.

La representación de cuaterniones en el entorno ROS corresponde a un vector de cuatro dimensiones $q = [x, y, z, w]$. Para comprender la representación de rotaciones a través de cuaterniones, es necesario entender que el cuaternion que representa una rotación de α en torno al eje definido entre $[0, 0, 0]$ y $[x, y, z]$ es:

$$X = x \cdot \sin(0.5 \cdot \alpha) \tag{2.1}$$

$$Y = y \cdot \sin(0.5 \cdot \alpha) \tag{2.2}$$

$$Z = z \cdot \sin(0.5 \cdot \alpha) \tag{2.3}$$

$$W = \cos(0.5 \cdot \alpha) \tag{2.4}$$

En base a lo anterior, una rotación como las que realiza el robot Bender en el plano, en torno al eje Z ($[0, 0, 1]$) de 90° queda definida como:

$$X = 0 \cdot \sin(45^\circ) = 0 \tag{2.5}$$

$$Y = 0 \cdot \sin(45^\circ) = 0 \tag{2.6}$$

$$Z = 1 \cdot \sin(45^\circ) = 0.707\ldots \tag{2.7}$$

$$W = \cos(45^\circ) = 0.707\ldots \tag{2.8}$$

Es importante notar que en este caso, el cuaternion resultante se encuentra normalizado, pero en caso de no ser así, es necesario normalizarlo.

2.2. URBI

URBI [6] es una plataforma de software open-source, creada por la empresa francesa Gostai. Su misión es entregar una plataforma de control para robots, o sistemas complejos en general, a través de la librería de componentes basada en C++ UObject, y su lenguaje de *scripting* Urbiscript que tiene soporte incluido para paralelismo y semántica orientada a eventos.

En su estado actual el robot Bender cuenta con muchos módulos de diversas funcionalidades, como interacción con el usuario, visión, reconocimiento de voz, que actualmente están implementados como UObjects en URBI 1.0. Por otra parte, existen muchos guiones e implementaciones de pruebas Robocup hechas en Urbiscript. Por esta razón, es de interés mantener los módulos y pruebas URBI existentes, e integrarlos con el sistema ROS a implementar. Para esto, será necesario modificar las pruebas y módulos existentes a la nueva versión de URBI 2.7.4, que cuenta con soporte para ROS.

2.2.1. Urbiscript

Urbiscript es el lenguaje de programación de URBI orientado a la robótica. Una de sus grandes ventajas es el soporte directo de concurrencia de acciones y programación basada en eventos, ambas características de gran utilidad para sistémas robóticos. Dado que su implementación está íntegramente asociada a URBI, la interacción con objetos del sistema, UObjects, es directa y simple. De esta forma, se presenta como un excelente lenguaje de *scripting* para sistemas robóticos implementados en URBI.

Paralelismo

Una de las ventajas principales de Urbiscript, es la facilidad de manejo de paralelismo, concurrencia y acciones simultáneas. Urbiscipt define operadores que permiten componer declaraciones y definir de esta forma la manera en que serán ejecutados. A continuaciíon se describen las cuatro formas de componer declaraciones, asumiendo que **A** y **B** son declaraciones diferentes e independientes:

A;B El punto y coma, llamado *composición secuencial*, significa “ejecutar la declaración **A** y luego ejecutar la declaración **B**”. Es posible que otras acciones se ejecuten entre **A** y **B** (como por ejemplo debido a la llegada de algún evento).

A|B La barra vertical, llamada *composición secuencial cerrada*, significa “ejecutar la declaración **A** y de inmediato ejecutar la declaración **B**”. Ninguna otra acción puede ejecutarse entre **A** y **B**.

A,B La coma, llamada *composición en segundo plano*, significa “ejecutar la declaración **A** y en algún momento posterior ejecutar la declaración **B**, posiblemente mientras **A** aún se está ejecutando”.

A&B El signo *et*, llamado composición en paralelo, significa "comenzar a ejecutar la acción **A** exactamente al mismo tiempo que la acción **B** y ejecutarlos en paralelo". En la composición en paralelo el bloque se ejecuta sólamente cuando todos los operadores de la declaración son conocidos.

Programación Orientada a Eventos

En un sistema robótico, la ejecución de código lineal no siempre es la forma más simple de interactuar con el entorno. En particular, es deseable poder reaccionar a eventos externos, muchas veces aleatorios, de forma inmediata, algo que no es simple de hacer cuando se ejecuta el código de forma lineal.

Para solucionar este problema Urbiscript presenta programación orientada a eventos, lo que permite que en respuesta a un evento se ejecuten bloques de código en repuesta, saliendo por un momento de la ejecución lineal del código del programa.

En Urbiscript, existen diversas instrucciones que permiten trabajar con eventos, ya sea reaccionando de diversa forma frente a ellos, o emitiéndolos. Es importante notar que un evento también puede ser una condición de variable (como por ejemplo $x > 0$). Para reaccionar a un evento, se utiliza la instrucción **at**, como se muestra en el siguiente ejemplo:

```
at(x > 0)
    echo("x es mayor que 0");
```

Si insertamos el código anterior en algún programa, cada vez que la variable x sea mayor que 0, se enviará el mensaje *"x es mayor que 0"*.

Una variante del comando **at** es el comando **whenever**, donde el código dentro del bloque (en el caso anterior, *echo("x es mayor que 0")* se ejecutará no solo cuando la condición pase a ser cierta, si no que además se seguirá ejecutando de forma continua hasta que el evento deje de emitirse, o la condición deje de ser cierta.

Es posible también utilizar eventos propios, y no solo condiciones referidas a variables. Para esto, es necesario generar una variable que sea una instancia de la clase *Event*:

```
var miEvento = Event.new;
```

Este evento puede luego ser capturado por un bloque **at**, donde dentro de la condición a evaluar, se utiliza el signo *"?"*, para indicar que la condición que determina el bloque es un evento:

```
at (miEvento?)
    echo("recibido");
```

Para lanzar este evento, entonces, se utiliza el operador *(*!):

```
miEvento!;
```

Lo anterior es una breve descripción del funcionamiento orientado a eventos de Urbiscript, pero para mayor detalle, es necesario remitirse a la documentación en línea de Urbiscript [7].

2.2.2. UObjects

En URBI, UObject es una interfaz de programación de aplicaciones (API) que permite introducir objetos externos escritos en C++ al entorno URBI. Cada uno de estos objetos representa un módulo del robot, ya sea de control de hardware, o de funciones de alto nivel. Como ejemplo, los UObjects implementados para el robot Bender controlan el uso de las cámaras de visión, la síntesis de voz, el control de la cabeza, y el movimiento de los brazos, entre otras cosas.

En la API C++, cada UObject es representado por una clase que puede ser instanciada dentro de urbiscript, y cuyos métodos y variables internas pueden ser compartidos al resto del entorno URBI tanto como para leer sus valores, o como para ser accedidos como funciones.

Para que una función del UObject sea accesible por el resto del entorno URBI, es necesario que estas funciones sean declaradas como accesibles, proceso que se conoce como "*binding*". Este proceso registra la función con el servidor URBI y la declara como accesible para el resto, para lo cual se utiliza la función del API llamada *UBindFunction*. Por otra parte, en URBI 2.74, para que una variable sea accesible por el resto del entorno, no es necesario utilizar un bind, pero si es necesario declararla de clase *UVar*, a diferencia de URBI 1.0, donde si era necesaro realizar el proceso de *"binding"*.

2.3. Conceptos de Navegación

Si bien la navegación es una acción simple de visualizar (moverse de un lugar del mundo a otro), cuando se trata de navegación en robots autónomos es un problema que aún no se considera resuelto en su totalidad.

La navegación robótica en ambientes dinámicos implica un ciclo constante de actuación y sensado del entorno, de forma de responder a los cambios que el ambiente presente, y requiere de varios elementos para su correcta realización: Una descripción del entorno (mapa), una localización del robot dentro del entorno, una manera de sensar los cambios del entorno y una forma de decidir que acciones tomar en respuesta a lo observado.

2.3.1. Mapas de Entorno

Un mapa de entorno corresponde a la representación interna que tiene el robot del mundo que lo rodea. Esta representación, en base a la localización, le permite al robot saber donde se encuentra, así como tambien identificar posibles obstáculos. En general existen dos tipos de mapas utilizados por robots:

Mapas Topológicos

Un mapa topológico es un mapa simplificado, en el cual solo existe información básica de cada lugar, y las conexiones entre lugares adyacentes, formando así un grafo, donde cada nodo representa una locación o lugar de interés, y cada vértice una conexión entre locaciones.

Este tipo de representación es muy eficiente, debido al bajo nivel de detalle con el que se representa el entorno, y por lo tanto permite realizar planificaciones de ruta mucho más eficientes. El gran problema de este tipo de mapas es la dificultad de generarlos y adquirirlos, puesto que normalmente es necesaria la intervención humana para su creación.

Un ejemplo de un mapa topológico de una casa típica puede observarse a continuación:

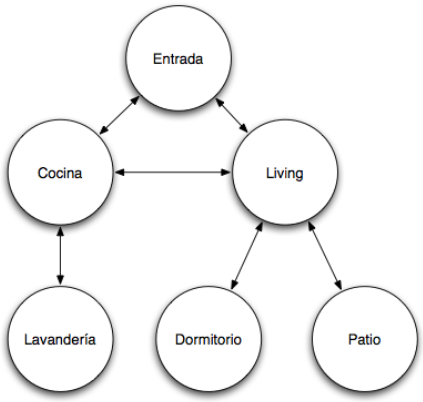


Figura 2.2: Ejemplo de Mapa Topológico

Mapas Métricos

Los mapas métricos son similares a como un ser humano imaginaría un mapa de un lugar, un plano (o espacio tridimensional) donde se colocan objetos según sus coordenadas de forma precisa [18].

La representación más común utilizada corresponde a mapas de grilla (llamados *grid maps* en inglés). Los mapas de grilla consisten en separar el plano a representar, en una serie de celdas de tamaño definido (resolución), donde cada celda representa un punto del entorno y presenta una

probabilidad de estar ocupada por un obstáculo o no.

En ROS, un mapa corresponde a una imagen mapa de bits en formato .pgm, donde cada pixel de la imagen representa una celda, y el color de cada pixel define el estado de la celda. Se pueden utilizar imágenes tanto a color como en escala de grises, siendo indiferente en términos de operación el funcionamiento de estos (no existe ventaja al utilizar imágenes a color por sobre escala de grises).

Cada celda del mapa puede tener uno de tres estados: libre, ocupada, o desconocida. Estas clasificaciones se hacen a través de umbrales. En el caso de las imágenes en escala de grises, el umbral se hace respecto al valor de intensidad de cada pixel. En el caso de las imágenes a color, cada pixel se compara en base a su valor de ocupación definido como:

$$occ = \frac{255 - color\_avg}{255} \tag{2.9}$$

Donde $color\_avg$ corresponde al promedio de intensidad de todos los canales de la imagen.

En la Figura 2.3 se presenta un mapa topológico realizado con un sensor láser. Se observa que la tonalidad de gris representa la probabilidad de ocupación del espacio, donde negro representa un obstáculo seguro, blanco espacio libre, y gris, espacio desconocido.

Figura 2.3: Ejemplo de Mapa Métrico

2.3.2. Localización

Para que un robot pueda navegar de forma satisfactoria en su entorno, es esencial que sepa donde se encuentra en el mapa. El proceso de localización basada en sensores puede dividirse en dos problemas principales: seguimiento de posición, donde el robot conoce su posición inicial y debe corregir los errores acumulados de odometría en base a medidas de sensores; y localización

global, donde el robot no tiene conocimiento alguno de su posiciíon y debe determinarla dentro de todo el mapa. Dada las condiciones de funcionamiento del robot Bender, el problema a tratar acá corresponde al primer caso, seguimiento de posición.

El problema de localización es un caso particular del problema de estimación de estados, donde se busca determinar la posición del robot dentro de un mapa global. Si bien los acercamientos originales al problema consideraban métodos basados en grilla, el enfoque predominante hoy en día es la localización basada en filtro de partículas (también llamada localización Monte Carlo) [9].

En general, un filtro de partículas es una variante de los filtros Bayesianos, donde se busca representar la probabilidad de que el robot se encuentre en un estado x_t. Para esto, el filtro de partćulas aproxima la distribución de probabilidad $p(x_t)$ a través de un conjunto S_t de muestras ponderadas, que se distribuyen en base a $p(x_t)$:

$$S_t = \{\langle x_t^{(i)}, w_t^{(i)} \rangle | i = 1, ..., n\} \tag{2.10}$$

donde $x_t^{(i)}$ es el estado en el tiempo t, y $w_t^{(i)}$ es el peso de cada partícula, normalizado de tal forma que el total sume uno, en el tiempo t.

En cada ciclo, el filtro de partículas realiza el filtrado Bayesiano en base a un procedimiento de muestreo, donde para cada partícula se predice el siguiente estado en base al modelo del sistema, se calcula su peso utilizando las mediciones de los sensores y los modelos de estos (normalizándose los pesos de todas las partiículas), y finalmente se remuestrea en base a los pesos adquiridos. Esta implementación de filtro de partículas es conocida como SISR (muestreo secuencial en importancia con remuestreo) [9].

El método de localización utilizado en este trabajo se encuentra en [15], y corresponde a una variación del filtro SISR, desarrollado por Dieter Fox. La idea de este algoritmo consiste en acotar el error introducido por la representación basada en muestras del filtro de partículas. Para esto, se utiliza una cantidad variable de partículas, determinada por la certeza que se tiene respecto a las observaciones, de tal forma de tener pocas partículas si la densidad de estas esta enfocada en una zona pequeña del espacio de estados, y tener más partículas cuando la densidad se disperse sobre una zona más amplia.

2.3.3. Mapas de Costo

Un mapa de costo, consiste en un tipo de mapa métrico donde a cada celda no se le asigna una probabilidad de ocupación, como en los mapas comunes, si no que un costo determinado asociado a atravesar esa celda. De esta forma, permiten clasificar las celdas de un mapa según el nivel de obstáculos que presentan, y son un elemento esencial dentro de lo que es la generación de rutas para navegación.

En ROS, los mapas de costo están implementados en el paquete *costmap_2d*. Este genera una grilla de ocupación que considera obstáculos inflados[4], en base a la configuración propia del robot,

[4]La inflación de obstáculos es el proceso a través del cual los obstáculos dentro de un mapa de costo se expanden

para simplificar la generación de rutas posibles. El mapa de costos generado es bidimensional, y permite operaciones de marcado y liberación de espacio ocupado en base a la información de sensores, así como la inicialización de éste en base a algún mapa de entorno previamente conocido.

El mapa de costos se suscribe automáticamente a los tópicos de sensores y se actualiza en base a sus mediciones. Cada sensor puede, ya sea marcar un espacio (insertar obstáculos en el mapa), limpiar un espacio (remover obstáculos del mapa) o ambas operaciones a la vez. Para marcar un obstáculo basta que el sensor lo detecte, y el mapa de costos actualizará el índice correspondiente a la celda de forma automática, sin embargo, para limpiar espacio libre, implica realizar *raytracing*[5] para cada celda a limpiar, hasta la próxima observación de obstáculo. En base a los obstáculos detectados por los sensores, se genera un proceso de inflación que propaga el valor del costo de cada celda hacia afuera mediante una función de decaimiento. Una imagen del mapa de costos en funcionamiento se observa en la Figura 2.4

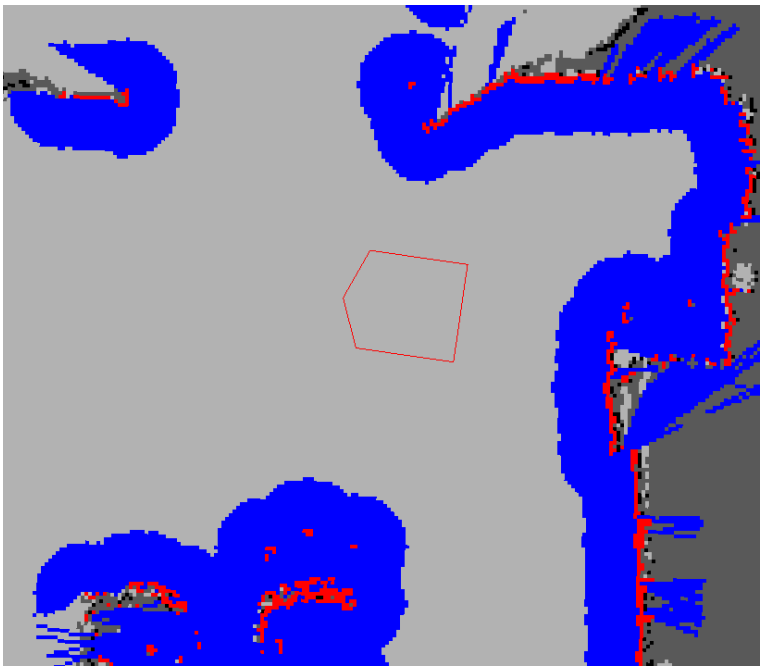


Figura 2.4: Mapa de costos en funcionamiento, mostrando la base movil, obstáculos detectados, obstáculos inflados, superpuestos al mapa del entorno. *Fuente: www.ros.org*

En la figura anterior, se observa un polígono de color rojo que representa la base móvil, los puntos de color rojo representan los obstáculos detectados, los puntos de color negro representan los obstáculos presentes en el mapa estático de entorno, las zonas azules representan los obstáculos inflados, las zonas gris claro el espacio libre, y las zonas gris oscuro el espacio desconocido.

a las celdas aledañas en función del tamaño del robot, para facilitar el proceso de determinar si existen o no colisions entre el robot y un obstáculo, para una pose determinada del robot en el mapa.

[5]El proceso de *raytracing* consiste en seguir el camino de cada rayo del sensor laser a través del mapa, para determinar el punto final donde el laser colisiona, de tal manera de asegurar que todo punto anterior que recorrió el rayo se encuentra libre de obstáculos.

El proceso de inflación y propagación de los costos de las celdas utiizado en el mapa de costos, se realiza mediante una función de decaimiento. En base a esto, se consideran 5 rangos de costo posibles para cada celda:

Letal Costo letal implica que existe una colisión efectiva, puesto que existe un obstáculo real en la celda, y por lo tanto si el robot se encuentra en ella, es obvio que hay una colisión.

Inscrito Una celda con costo 'Inscrito' significa que su distancia a una celda con obstáculo es menor que el radio inscrito del robot, y por lo tanto si el robot se encuentra en una celda con costo 'Inscrito' o mayor, seguramente está colisionando.

Posiblemente Circunscrito Es un costo similar a inscrito, solo que considera el radio circunscrito del robot. Por esta razón, si el robot se encuentra en una celda con este costo, es posible que exista una colisión dependiendo de la orientación de este.

Libre El espacio libre no tiene costo alguno y no existe razón alguna que impida que el robot se desplace por este tipo de celdas.

Desconocido El espacio desconocido es espacio que no cuenta con información alguna y su interpretación depende del usuario.

Lo anterior se resume en la Figura 2.5:

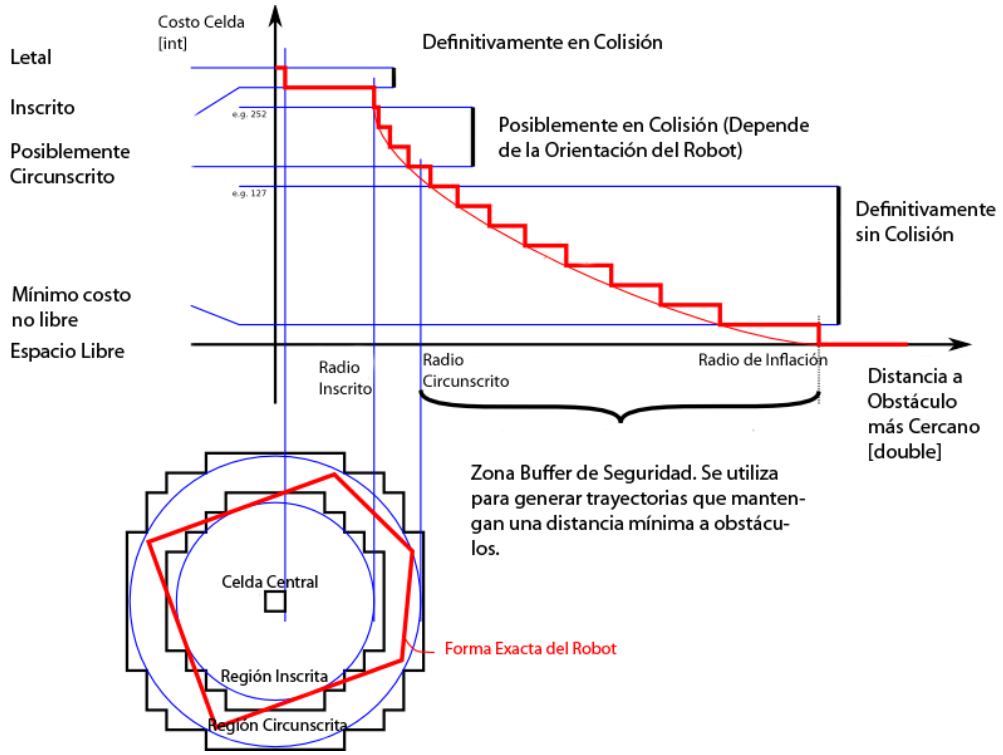


Figura 2.5: Diagrama de rangos de inflación de obstáculos.

2.3.4. SLAM

Al colocar un robot en un entorno desconocido, y buscar que este desarrolle una representación de su entorno a través de un mapa, es necesario resolver de forma simultánea dos problemas: definir en que lugar del mundo se encuentra el robot, y definir como representar las características observables del entorno. Este problema se denomina SLAM, del inglés *Simultaneous Localization And Mapping* [11].

SLAM es un problema de alta complejidad, puesto que para localizarse, un robot necesita un buen mapa del entorno; mas para generar un buen mapa, es necesario que el robot tenga una buena estimación de su posición. Esta fuerte interrelación entre la pose del robot, y las estimaciones del mapa de entorno hacen necesario que la búsqueda de la solución se realice en un espacio de muchas dimensiones.

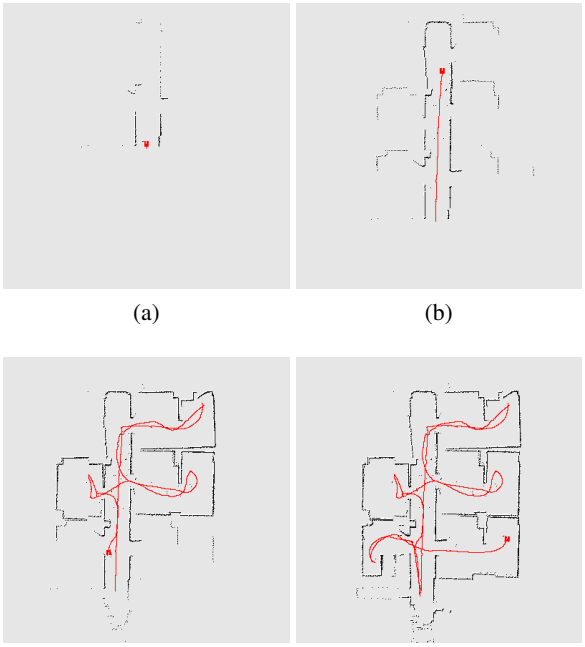
El método utilizado en el presente trabajo para resolver este problema corresponde a la implementación propuesta por Grisetti, Stachniss y Burgard en [17]. Si $x_{1:t}$ es la posición del robot desde los tiempos 1 hasta el tiempo t, m es el mapa del entorno, $z_{1:t}$ son las observaciones de los sensores desde el tiempo 1 hasta el tiempo t y $u_{1:t}$ son las mediciones de odometría en los mismos tiempos, se tiene que en general, un filtro de partículas busca aproximar $p(x_{1:t}, m|z_{1:t}, u_{1:t-1})$ a través de un conjunto de partículas, cada una de las cuales representa una posible solución, y que su distribución en el espacio asemeja la distribución de probabilidad buscada.

Los filtros de partículas Rao-Blackwellized, utilizados en [17] utilizan la siguiente factorización:

$$p(x_{1:t}, m|z_{1:t}, u_{1:t-1}) = p(m|x_{1:t}, z_{1:t}) \cdot p(x_{1:t}|z_{1:t}, u_{1:t-1}) \tag{2.11}$$

Esta factorización separa el problema en dos: la estimación de $p(m|x_{1:t}, z_{1:t})$, simple de hacer en base a *generación de mapas con poses conocidas* [21]; y la estimación de la trayectoria del robot: $p(x_{1:t}|z_{1:t}, u_{1:t-1})$ que se resuelve en base a un filtro de partículas, donde cada partícula representa una posible trayectoria con un mapa posible asociado. Finalmente, se elige como solución la mejor partícula de trayectoria, es decir la que tiene un mayor peso asociado por el proceso de remuestreo de filtro SIR [9].

A continuación, en la Figura 2.6 se observa el proceso SLAM con un robot móvil:



(a) (b)

(c) (d)

Figura 2.6: Proceso SLAM

Se puede observar como en la Figura 2.6a el robot, representado como el cuadrado rojo, se encuentra en el principio del pasillo sin tener mucha información de su entorno, conociendo solamente con cierta seguridad la posición de las paredes del pasillo cercanas a él. En la Figura 2.6b, el robot avanza hasta el final del pasillo (su trayectoria es representada por la línea roja), y la seguridad respecto a la posición de las paredes del pasillo aumenta, así como parte de las paredes de los cuartos conectados al pasillo, que se observan desde afuera. En las Figuras 2.6c y 2.6d el robot termina el mapa ingresando a los diversos cuartos, obteniéndose un mapa completo del entorno del robot.

2.3.5. Generación de Trayectorias

La generación de trayectorias es el problema en el cual, existiendo un mapa del entorno, dada una pose inicial del robot y una pose final, se busca generar una secuencia de acciones que lleven al robot desde la pose inicial a la pose final, sin colisionar con ningún obstáculo, y de la forma más eficiente posible.

En ROS se divide este problema en la generación de dos trayectorias diferentes. Una es la trayectoria global, que en base al mapa genera una trayectoria posible, dadas las dimensiones del robot, desde la pose inicial hasta la final. La segunda es la generación de una trayectoria cinemática que en base al mapa de costos del entorno, y el plan global anterior, intenta llevar al robot desde el

origen hasta el objetivo en base a la generación de una secuencia de comandos de velocidad a ser enviados a la base móvil.

La primera trayectoria es implementada por un módulo ROS llamado **Navfn**, que en base a la utilización del algoritmo de Dijkstra, genera una trayectoria de costo mínimo entre el punto inicial y final, considerando las restricciones dimensionales del robot.

La segunda trayectoria, se genera utiliizando el algoritmo *Trajectory Rollout* [16], integrado en ROS, que basado en el mapa de costos del entorno, y el plan global, genera una trayectoria cinemática que lleva al robot desde el origen al objetivo. A medida que el robot se desplaza, se va generando además un mapa de costos local que es utilizado para generar los comandos de velocidad que se envían a la base, de forma de llegar al objetivo, al mismo tiempo que se navega de forma reactiva para esquivar obstáculos.

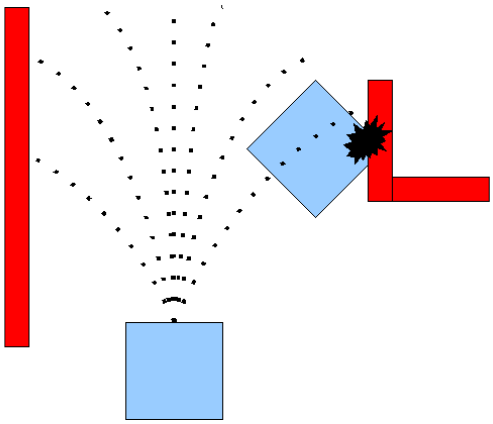


Figura 2.7: Generación de Trayectorias

El proceso para generar los comandos de velocidad del algoritmo es como se indica a continuación:

1. Muestrear de forma discreta el espacio de control del robot $(dx, dy, d\theta)$.
2. Para cada muestra de velocidad, se simula el comportamiento del robot desde la pose y estado actual N pasos hacia adelante (periodo de simulación) como si se aplicase la velocidad muestreada durante un periodo de tiempo definido por el tamaño del paso.
3. Se evalúan las trayectorias generadas en el paso anterior en base a una métrica programable que incluye la cercanía a obstáculos, la cercanía al plan original, y la velocidad de desplazamiento. En este mismo paso, se eliminan las trayectorias inválidas, es decir, aquellas que resultan en la colisión del robot con algún obstáculo.
4. Se elige la trayectoria con mejor puntuación, y se envía el comando de velocidad asociado a la trayectoria
5. Se actualiza la pose y el estado del robot, y se comienza de nuevo el algoritmo.

Capítulo 3

Implementación de Middleware ROS para Robot de Servicio Bender

3.1. Instalación de Programas Base

3.1.1. Sistema Operativo

En primer lugar, fue necesario acondicionar los computadores a utilizar para la función que se les daría. Esto significó instalar en el computador Alienware M11x R3 un sistema operativo basado en Linux, los drivers específicos asociados al hardware del computador, el middleware ROS en su versión Diamondback con sus librerías respectivas, y la plataforma de software URBI 2.7.4.

El sistema operativo elegido fue Linux Ubuntu en su versión 11.04. Esta elección fue debido a que a diferencia de versiones anteriores, incluye en el kernel drivers para la tarjeta de red AR8132, presente en el computador Alienware M11x R3. Por otra parte, existen paquetes de instalación ROS para esta versión de Ubuntu, que incluyen binarios precompilados de ROS, por lo que la instalación del middleware se simplifica notablemente, siendo innecesario compilar de cero todo el código fuente.

La instalación del sistema operativo se realizó según las instrucciones para la versión 11.04 encontradas en la página oficial de la distribución Ubuntu Linux[1], sin ninguna modificación adicional al proceso de instalación.

[1] *Sitio oficial: www.ubuntu.org*

3.1.2. ROS Diamondback y Paquetes Extra

Para instalar ROS en su versión Diamondback, es necesario agregar primero los repositorios *"restricted", "universe"* y *"multiverse"* al sistema operativo. Para esto, es necesario abrir *"Ubuntu Software Center"*, y del menú *"Edit"*, seleccionar *"Software Sources"*. En la ventana resultante, es posible activar los repositorios antes dichos, tal como muestra la Figura 3.1 a continuación:

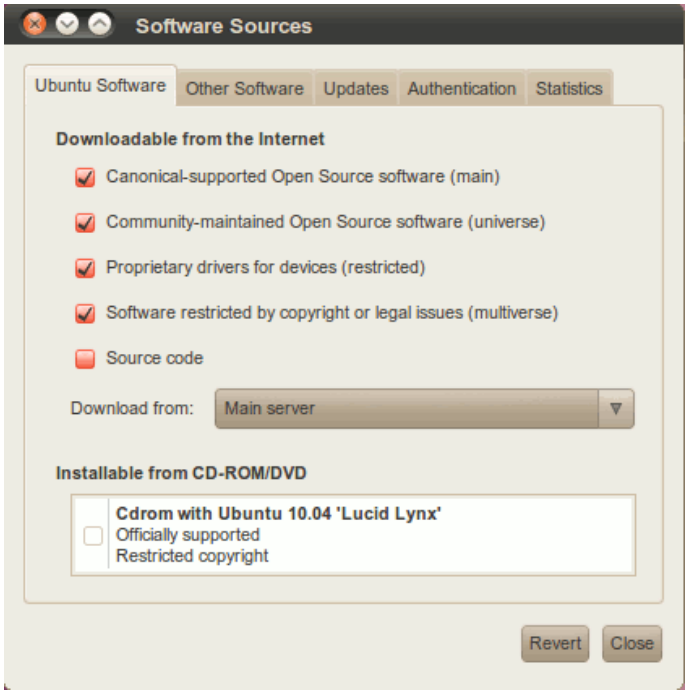


Figura 3.1: Selección de Repositorios en Ubuntu

Es importante recalcar que esta interfaz gráfica solo es un intermedio que produce cambios en el archivo de configuración de **apt**, ubicado en */etc/apt/sources.list*.

A continuación, es necesario agregar los repositorios propios de ROS a la lista de repositorios a usar por **apt**. Para esto, es necesario ejecutar el siguiente comando, que agrega el repositorio correspondiente a Ubuntu 11.04 al archivo *sources.list*.

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

Hecho lo anterior, es necesario agregar las llaves de identificación del servidor de ROS.org, de manera de poder autentificar el contenido al momento de descargarlo via **apt-get**:

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

Finalmente, actualizamos el listado de paquetes, de forma que **apt** incluya los paquetes ubicados en los repositorios que recién agregamos:

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

Una vez que se termina la actualización de las listas de software, podemos comenzar a bajar los archivos e instalar ROS.

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

El paso anterior puede resultar lento, ya que es necesario bajar alrededor de 1.5 GBytes de información.

Una vez terminada la instalación, solo queda actualizar el entorno del sistema, para que las variables de entorno ligadas a ROS se incluyan en la sesión **bash** cada vez que se abra una nueva línea de comandos:

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

Instalación de Paquete de control RosAria

El paquete de control **RosAria**, es una solución de software desarrollada por la Facultad de Ingeniería Eléctrica y Computación de la Universidad de Zagreb, que permite enviar y recibir mensajes a plataformas robóticas que utilicen la librería **ARIA**. En particular, permite comunicarse de forma bidireccional con la plataforma móvil Pioneer 3 AT, que utiliza Bender, permitiendo por una parte enviarle comandos de movimiento (velocidad), y por otra recibiendo información de la base como son el estado de la batería, odometría y estado de encoders entre otros.

Para instalar el paquete, es necesario bajar el código fuente de **RosAria**, utilizando el software de control de versiones **Mercurial**. Si este no esta instalado, se instala fácilmente:

```
$sudo apt-get install mercurial
```

Terminada la instalación de **Mercurial**, es necesario bajar el código fuente de **RosAria**:

```
$roscd
$cd stacks
$hg clone https://code.google.com/p/amor-ros-pkg/
```

A continuación, es necesario asegurarse de que la librería **ARIA 2.7.2** esté instalada en el sistema. Si no esta instalada, es necesario bajarla desde el sitio web oficial de MobileRobots (http://robots.mobilerobots.com/wiki/ARIA). Es importante bajar la versión 2.7.2 en formato *tar* (con terminación *.tgz*). Una vez bajado el archivo, debe ser colocado en la carpeta *ARIA/build* dentro del directorio que contiene el código fuente de **RosAria**. Hecho esto, se procede a compilar y configurar:

```
$rosmake ROSARIA RosAria
```

Instalación de Paquete de Teleoperación PR2_Teleop

El paquete **pr2_teleop** es un paquete de teleoperación que permite enviar mensajes de movimiento a un robot genérico, utilizando un teclado, o un joystick de PlayStation 3. Esto permitirá controlar la base móvil Pioneer utilizando el teclado o joystick, a través de los mensajes enviados por **pr2_teleop** e interpretados por **RosAria**. La instalación del paquete se hace directamente desde apt-get:

```
$sudo apt-get install ros-diamondback-pr2_teleop
```

Instalación de Paquete de Visualización Rviz

Rviz es un paquete de visualización que permite representar en una interfaz gráfica la distinta información que se tiene del robot como: lectura de sensores laser, mapa del entorno, mapas de costos, odometría, localización y marcos de referencia. El paquete viene por defecto con la versión ROS-desktop-full que fue instalada, por lo que solo es necesario compilar el código:

```
$rosmake rviz
```

Para verificar su correcto funcionamiento, ejecutamos el siguiente comando para iniciar **Rviz**:

```
$rosrun rviz rviz
```

Después de lo cual deberíamos obtener la ventana de **Rviz** en blanco, como muetra la Figura 3.2:

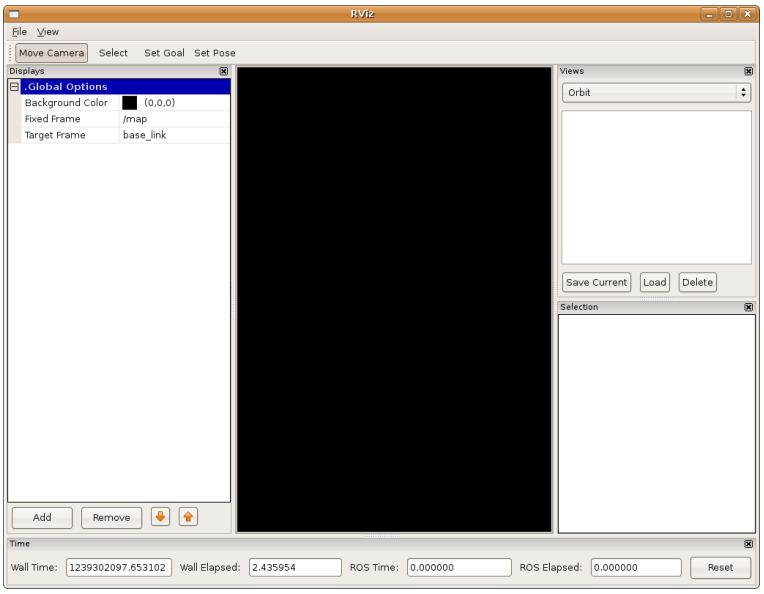


Figura 3.2: Ventana principal de Rviz

Instalación de Paquete de Control Sensores Laser

Para poder utilizar el sensor laser Hokuyo URG 04LX de Bender, es necesario instalar el paquete de drivers de sensores laser para ROS. La instalación es directa, al igual que para el paquete anterior, y se realiza a través de **apt-get**:

```
$sudo apt-get install ros-diamondback-laser-drivers
```

Instalación de Paquete SLAM GMapping

El paquete **GMapping** integra el algoritmo SLAM GMapping, de [17], al entorno ROS, y permite la creación de mapas de forma online y offline. Para instalarlo, debido a que se encuentra incluido en la versión de ROS instalado, solo hace falta su compilación:

```
$rosmake gmapping
```

3.1.3. Instalación de Urbi 2.7.4

Ubuntu Linux

La instalación de Urbi en Linux es bastante directa, pues existen paquetes que incluyen los binarios ya compilados para la arquitectura que se usa, lo que hace innecesario bajar y compilar el código fuente. Para poder utilizar Urbi, solo es necesario bajar del sitio oficial de Gostai el archivo con los binarios comprimidos, y luego descomprimirlo en una carpeta a elección, lugar donde quedará instalado Urbi. Es desde esta locación donde se deberán incluir las librerías y cabeceras a utilizar en proyectos Urbi.

Windows XP

La instalación de Urbi en Windows se realiza en base al instalador oficial que es necesario descargar desde la página oficial de Gostai. Es importante bajar la versión a ser utilizada con Visual Studio 2008, puesto que de esta manera se contará con un asistente para crear proyectos, que automáticamente incluye los archivos necesarios y cambia los parámetros de preprocesamiento para poder utilizar Urbi en un proyecto de C++.

3.2. Implementación de Sistema de Navegación

3.2.1. Control de base Pioneer 3 AT y Sensor Láser

Base Pioneer 3 AT

El primer paso a realizar para implementar un sistema de navegación, corresponde a ser capaz de controlar la base Pioneer 3 AT desde un módulo ROS, así como poder leer los mensajes que ésta envíe, como odometría, estado de batería y posibles mensajes de error. Para esto, se utiliza el paquete **RosAria**, instalado anteriormente, en conjunto con un módulo de teleoperación, y un nodo escrito a medida para recibir y transmitir la información de odometría proveniente de la base móvil.

En primer lugar, conectamos la base al notebook a través de un adaptador USB – RS232, y ejecutamos el nodo **RosAria** para verificar que se contacte correctamente con la base, y determinar los tópicos de comunicación que suscribe y publica mediante la herramienta **rxgraph**:

```
$roscore
$rosrun ROSARIA RosAria
$rxgraph
```

De donde se determina en primer lugar que el nodo se conecta correctamente con la base, que publica odometría en el marco de referencia “/odom” y que se suscribe a mensajes de control de

velocidad en el tópico "/RosAria/cmd_vel". En base a una revisión de código fuente, se observa que la odometría que se publica desde la base es en milímetros, por lo que se escribe un nodo que transforma la odometría a metros, unidad usada por ROS, y la retransmite generando una transformada TF entre el marco "/odom" y el marco "/base_link", que representa a la base. Este nodo fue implementado en el archivo Pioneer_tf.cpp.

Para simplificar y estandarizar los tópicos de comunicación, se modifica la suscripción de mensajes de control de velocidad de "/RosAria/cmd_vel" a "/cmd_vel", lo que es consistente con el resto del stack de navegación ROS, y el módulo de teleoperación **pr2_teleop**.

Lectura de Laser Hokuyo URG-04LX

Para la lectura del sensor Laser Hokuyo URG-04LX, se utiliza el paquete **laser_drivers**, instalado anteriormente. Este programa, permite la interaccón bidireccional con cualquier laser compatible con el protocolo SCIP 2.0 de Hokuyo. De esta manera, es posible no sólo leer los datos provenientes del laser, si no también configurar su modo de operación o requerir el envío de información adicional (ID del dispositivo, configuración actual, errores, etc.).

En primer lugar, se conecta el sensor laser al computador a través de un cable USB, y se lanza el nodo de forma normal:

```
$ rosrun hokuyo_node hokuyo_node
```

Obteniéndose la siguiente respuesta por parte del nodo:

```
[ INFO] 1256687975.743438000: Connected to device with ID: H0807344
```

Que verifica el normal funcionamiento del nodo, y la conexión correcta con el sensor laser. A continuación, se verifica la correcta lectura e interpretación de los datos. Para esto, utilizaremos el programa de visualización **Rviz** en conjunto con una configuración tipo para **Rviz** que viene incluida por defecto en el paquete de **hokuyo_node**, llamada *hokuyo_test.vcg*:

```
$ rosrun rviz rviz -d `rospack find hokuyo_node`/hokuyo_test.vcg
```

Este comando, ejecuta **Rviz** entregándole el archivo de configuración *hokuyo_test.vcg* que define el entorno a representar.

En la Figura 3.3 a continuación se muestra la ventana resultante al inicializar la prueba de lectura del láser:

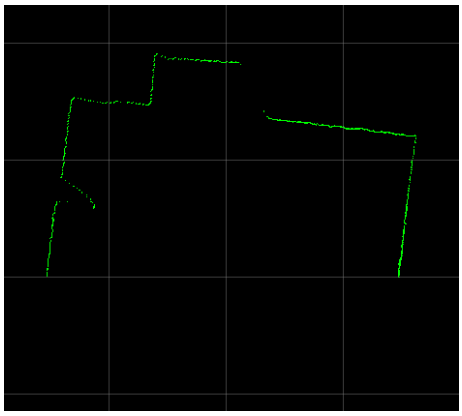


Figura 3.3: Ventana de Rviz con lecturas del sensor láser

En verde se observan las lecturas del láser, que corresponden con las paredes del laboratorio donde se realizó la prueba. Se observa que al mover la base, las medidas del laser se adecuan correctamente, por lo que este funciona bien.

Revisión de Consistencia de Odometría

La odometría es una parte esencial de un sistema de navegación, y su calidad determina el buen funcionamiento de los algoritmos de generación de mapas, navegación y localización. Teniéndose ya el control de la base por teleoperación funcionando, y la posibilidad de leer el sensor laser, se procede a verificar la calidad y consistencia de la odometría entregada por la base móvil.

En primer lugar, se revisa la odometría rotacional de la base móvil, para lo cual se abre **Rviz**, y se selecciona como marco de referencia a */odom*, y para el tópico */laser* se selecciona un tiempo de decaimiento alto, cercano a 20 segundos. Esto provocará que las medidas que se reciban del laser no desaparezcan de inmediato, si no que cada medición se mantenga en pantalla durante 20 segundos.

Hecho esto, se hace girar el robot en su lugar y se observan las medidas del laser. Frente a una odometría perfecta, se esperaría que cada lectura nueva calce con las anteriores, pero siempre es aceptable un pequeño desplazamiento de las medidas.

Los resultados de la prueba se observan en la Figura 3.4 a continuación:

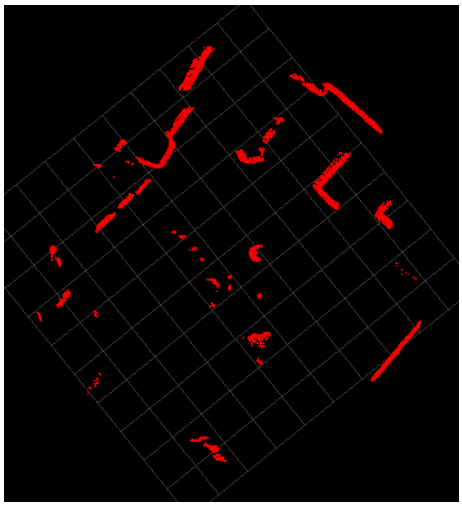


Figura 3.4: Revisión Consistencia Odometría Rotacional

Para la base móvil utilizada, se observa que existe una buena odometría rotacional, y que existe un desplazamiento muy pequeño en las medidas de laser subsecuentes, cercanas a los 5$[deg]$, según la diferencia entre medidas secuenciales.

A continuación se verifica la odometría de desplazamiento lineal. Con la misma configuración de **Rviz** anterior, se coloca el robot a 3 metros de una pared y apuntando hacia ella. A continuación, se hace avanzar el robot en línea recta hacia la pared, y se observan las medidas del laser, que en el caso ideal deberían moverse en conjunto a medida que el robot se acerca a la pared, y generar una línea delgada que represente la pared. El resultado de esta acción se observa a continuación:

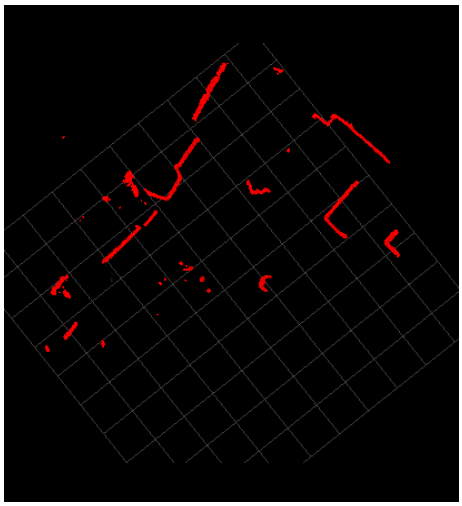


Figura 3.5: Revisión Consistencia Odometría Lineal

Al igual que en el caso anterior, se observa que la odometría es muy buena, con un error máximo de $4[cm]$ según el calce de las mediciones del sensor laser.

3.2.2. Generación de Mapas

Con los elementos ya existentes de movimiento de la base y sensado laser, es posible hacer un mapa del entorno, punto necesario para localización y navegación. Para esto se utiliza el paquete **GMapping**, instalado anteriormente, en conjunto con la teleoperación del robot.

Para simplificar el lanzamiento de los nodos necesarios, se creó un *launchfile* que se encarga de levantar todos los nodos necesarios para realizar el mapeo. Este *launchfile* levanta los nodos **Pioneer_tf, teleop_keyboard, RosAria, hokuyo_node** y **gmapping**, además de cambiar algunos parámetros de **GMapping** para mejorar su funcionamiento con la base móvil y el sensor utilizados. La implementación del *launchfile* se encuentra en el archivo *complete_map.launch*[2].

Una vez lanzados los nodos a través del *launchfile*, se mueve el robot de forma teleoperada a través de la habitación en la cual se realizará el mapa, en este caso el Laboratorio de Mecatronica de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile, mostrado en la Figura 5.1, teniendo cuidado de que el sensor laser cubra todas las áreas y paredes a incluir en el mapa. Una vez terminado, se detiene el robot, sin detener ningún proceso, y se guarda el mapa con el siguiente comando:

```
$rosrun map_server map_saver
```

Ejecutado este comando, se generara una pareja de archivos: *map.pgm* y *map.yaml*. El primero es la representación del mapa en un archivo de bits y el segundo es un archivo que describe la configuración del mapa, su resolución y el nombre de archivo asociado. Hecho esto, se pueden finalizar los procesos anteriores.

3.2.3. Configuración del Módulo de Localización

El primer paso a realizar para lograr que el robot navegue, es verificar el funcionamiento del sistema de localización. Para esto, se carga el mapa hecho anteriormente, se inicializa el sistema de localización **AMCL** y se mueve el robot por la pieza, mientras se verifica en **Rviz** que las mediciones del laser calcen con los obstáculos del mapa a medida que el robot se desplaza. De esta manera, se ve que el sistema de localización funcione correctamente.

Para realizar lo anterior, en primer lugar es necesario lanzar el nodo **map_server** que se encarga de leer el mapa a utilizar y entregárselo al resto del sistema. Para esto se ejecuta el nodo entregándole como argumento el archivo yaml creado en el punto anterior, que define las características del mapa.

[2]Este archivo, junto a los demás implementados, pueden encontrarse en el Apéndice A al final del presente documento.

Luego, se ejecutan los nodos **Pioneer_tf, teleop_keyboard, RosAria** y **hokuyo_node** que permiten mover el robot, recibir su odometría y recibir las mediciones del sensor laser. Finalmente, se lanza el nodo de localización **AMCL**. Para poder modificar ciertos parámetros de operación del sistema de localización, se creó el *launchfile amcl_diff.launch*, que modifica los parámetros de operación del sistema de localización, para adecuarse más a la geometría y funcionamiento del robot utilizado.

Los parámetros que fueron modificados se observan en la Tabla 3.1.

Tabla 3.1: Configuración Parámetros de Localización

Cantidad de partículas mínima	100
Cantidad de partículas máxima	3000
Distancia de Actualización [m]	0.05
Ángulo de Actualización [rad]	0.5

Esto permite tener una respuesta más rápida y precisa por parte del sistema de localización, a cambio de tener un costo computacional más alto. Dado que la capacidad de procesamiento del sistema es elevada, es posible entonces buscar una precisión mayor utilizando mayor cantidad de ciclos de procesador.

3.2.4. Configuración de Mapas de Costos

Para la configuración de navegación, un paso esencial corresponde a configurar los mapas de costos. Para el sistema actual, se decidió utilizar dos mapas de costos, uno global, que se utiliza para generar trayectorias de navegación de largo plazo, incluyendo todos los elementos del entorno; y un mapa local, que se utiliza para planes a locales de pequeña escala y evasión de obstáculos.

En base a la disposición anterior, es necesario entonces generar tres archivos de configuración: uno para los parámetros del mapa global, uno para los del mapa local, y finalmente uno que contenga los parámetros comunes a todo el sistema.

Configuraciones Comunes

El primer archivo de configuración editado corresponde al de los parámetros comunes, y se encuentra en *costmap_common_params.yaml*:

```
obstacle_range: 3.5
raytrace_range: 0.30


#---pioneer AT footprint:---
#---(in meters)---
footprint: [ [0.254, -0.2], [0.254, 0.2], [-0.254, 0.2], [-0.254, -0.2]]
```

```
inflation_radius: 0.5

transform_tolerance: 0.2
map_type: costmap

observation_sources: laser_scan_sensor
laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan, topic: scan,
     marking: true, clearing: true, expected_update_rate: 0.2}
```

Las primeras dos líneas determinan la distancia a la cual un obstáculo comenzará a ser considerado, y la distancia a la cual estos obstáculos seran eliminados del mapa en base a *raytracing*.

A continuación, se definen en metros los puntos que definen el contorno del robot, para ser considerados dentro de la visualización de su posición. Estos puntos están medidos en metros desde el centro del robot, y pueden definirse en sentido horario o antihorario.

La línea siguiente define el radio de inflación del robot, y por tanto es esencial que sea medida de forma precisa, puesto que un valor menor al real produciría posibles colisiones con obstáculos al momento de navegar.

Finalmente, las últimas dos líneas determinan las fuentes de observación con las que se cuentan. En particular, se define que existirá solo una fuente que será un sensor laser, y se entregan a continuación los parámetros del sensor: su marco de refencia, tipo de datos que entrega, tópico sobre el cual se comunica, y si es capaz de marcar y limpiar obstáculos. Finalmente, se indica la frecuencia esperada de actualización, es decir, cada cuanto tiempo enviará sus datos (medido en segundos).

Mapa de Costo Global

El segundo archivo de configuración, *global_costmap_params.yaml*, describe los parámetros de operación exclusivos al mapa de costos global, que funcionará en base al mapa de entorno completo y servirá para hacer la planificación de la ruta global de navegación. Su contenido es el siguiente:

```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_link
  update_frequency: 2.0
  publish_frequency: 10.0
  static_map: true
```

En primer lugar, se define el namespace sobre el cual aplicarán las configuraciones: *global_costmap*. A continuación, se determina que el marco de referencia sobre el cual trabajará el

mapa de costos es el marco de referencia asociado al mapa del entorno */map*, y que el marco que define la posición del robot es */base_link*.

Finalmente, se determina la frecuencia de actualización del mapa de costos en Hertz (cada cuanto tiempo se actualizarán se ejecutará el loop interno del programa), frecuencia de publicación en Hertz (cada cuanto tiempo se enviará el mapa de costos al resto del sistema); y la condición de estaticidad del mapa, que determina que el mapa se incialice en base a un mapa publicado por el *map_server*. Dado que el mapa de costos global es un mapa estático, es posible tener una frecuencia de actualización baja (2 Hertz) sin que esto presente problemas al sistema, y de esta forma ahorrar recursos de cómputo.

Mapa de Costo Local

El mapa de costo local recibe su configuración del archivo *local_costmap_params.yaml*, que será utilizado para generar un mapa de costos en torno a la posición actual del robot, que considere los obstáculos dinámicos del entorno y permita una navegación reactiva. El contenido del archivo de configuración es el siguiente:

```
local_costmap:
  global_frame: /odom
  robot_base_frame: /base_link
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
  rolling_window: true
  width: 5.0
  height: 5.0
  resolution: 0.02
```

En primer lugar, se define que el marco de referencia sobre el cual se trabajará será el de la odometría, */odom*, y que el marco de referencia del robo corresponde a */base_link*. Dado que este mapa de costos se utiliza para la evasión de obstáculos dinámicos, se aumenta la frecuencia de actualización del mapa a 5 Hertz, y se define como un mapa no estático.

El parametro siguiente, *rolling_window*, indica que el mapa de costos será una ventana móvil que se desplazará junto con el robot, de tal forma de siempre representar los obstáculos que esten en torno a éste. Finalmente, se definen las dimensiones del mapa de costos en metros, y la resolución de la grilla (es decir, que tamaño tendrá cada celda del mapa, en metros). Se considera una dimension de 5 x 5 metros para el mapa de costos local, puesto que de esta manera acomoda el rango máximo de lectura del sensor láser. Además, como se necesita que el robot sea capaz de desplazarse a través de espacios pequeños, así como pasar a través de umbrales de puertas se elige una resolución detallada, de forma de poder representar con mayor detalle los obstáculos, al costo de un mayor uso de procesador.

3.2.5. Configuración de Planificadores de Rutas

Dada la configuración del sistema, se utilizan dos planificadores de ruta, uno local y uno global, cada uno asociado a un mapa de costos. El planificador global genera una trayectoria en el mapa que lleva el robot desde su posición actual hasta el objetivo de navegación, mientras que el planificador local genera comandos de velocidad que envía al robot para intentar seguir el plan global generado con anterioridad.

El planificador global a utilizar es el planificador **Navfn**, parte del paquete **move_base**, que en base al mapa de costos global, y utilizando el algoritmo de Dijkstra [10], genera una trayectoria que lleva al robot al objetivo de navegación sin que éste colisione con ningún obstáculo del mapa. Se utilizan las configuraciones por defecto del paquete, por lo que no es necesario crear ningún archivo de configuración.

El planificador local a utilizar es **base_local_planner**, también parte del paquete **move_base**. Este planificador presenta implementaciones de dos algoritmos de navegación local en el plano: *Trajectory Rollout* y *Dynamic Window*. El archivo de configuración asociado a este planificador es *base_local_planner_params.yaml* cuyo contenido es el siguiente:

```
TrajectoryPlannerROS:
  holonomic_robot: false
  max_vel_x: 1.5
  min_vel_x: 0.1
  max_rotational_vel: 0.8
  min_in_place_rotational_vel: 0.3
  escape_vel: -0.0
  acc_lim_th: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 0

  sim_time: 2.0
  sim_granualarity: 0.02

  path_distance_bias: 0.75
  goal_distance_bias: 0.5

  yaw_goal_tolerance: 0.09
  xy_goal_tolerance: 0.14
  latch_xy_goal_tolerance: true

  dwa: false
```

La primera sección del archivo determina las características de movimiento del robot. En primer lugar, se define que el robot es no-holonómico[3], cosa a tomar en cuenta al considerar los

[3]Un robot se llama holonómico cuando la cantidad de grados de libertad del robot es igual a la cantidad de grados

comandos de movimento posibles para el robot. A continuación se fijan las velocidades máximas y mínimas tanto de desplazamiento como de rotación, en $\frac{m}{s}$ y $\frac{rad}{s}$, así como la velocidad de escape del robot, que es la velocidad a la cual el robot se desplaza en reversa en caso de encontrarse atascado en algún lugar.

Dado que el robot Bender no cuenta con sensores que permitan la detección de obstáculos hacia atrás, esta velocidad se define como cero, de forma de no permitir el desplazamiento en reversa y la posible colisión con obstáculos que se encuentren atrás del robot.

A continuación se definen los límites de aceleración del robot de desplazamiento lineal y rotacional en $\frac{m}{s^2}$ y $\frac{rad}{s^2}$. Dado que el robot no es omnidireccional, y cuenta con restricciones no holonómicas, es necesario fijar su aceleración en el eje Y como cero.

La siguiente sección contempla el seteo de dos parámetros: *sim_time* y *sim_granularity*. El primero es la ventana de tiempo, en segundos, que se utilizará para la simulación de movimiento futuro. El segundo determina la distancia a tomar entre cada punto de una trayectoria al momento de simular, en metros. Aprovechando la gran capacidad de cómputo del sistema sobre el cual se ejecutan los procesos, se aumenta el tiempo de simulación, así como también se minimiza la distancia entre puntos de la trayectoria a simular, mejorando la resolución.

Los siguientes dos parámetros representan cuanto toma en cuenta el planificador la distancia al plan global y la distancia al objetivo. En base a prueba y error, se determinó que los parámetros que mejor resultado producen en términos de flexibilidad de movimiento son los arriba descritos.

Los parámetros que siguen, describen cuan tolerante es el planificador respecto a alcanzar el objetivo. Si se configura una tolerancia muy baja, es posible que el robot nunca considere haber llegado al objetivo, y por lo tanto que gire en torno al punto de llegada de forma constante. De forma análoga, si se elige una tolerancia muy amplia, el robot se detendrá a una distancia elevada del punto o con una orientación errada. Dado que la tolerancia ideal depende de las capacidades de movimiento del robot, así como de su localización, los valores seleccionados fueron determinados en base a prueba y error.

El tercer parámetro del grupo, *latch_xy_goal_tolerance*, indica que una vez que el robot considere que llegó a su objetivo, no volverá a desplazarse, aún cuando al orientarse rotando se desplace de forma de quedar a una distancia mayor a la tolerancia. Esto evita que ciclos en los cuales el robot llega a un punto, se orienta rotando y queda muy lejos, dando entonces una vuelta, llegando de nuevo al punto y repitiendo el ciclo.

El parámetro final del archivo de configuración determina que no se utilizará el algoritmo *Dynamic Window (DWA)* si no que se usará *Trajectory Rollout*. Si bien *DWA* es más eficiente que *Trajectory Rollout*, se comprobó de forma empírica que su funcionamiento en robots no holonómicos no es óptimo, presentando soluciones más complejas de movimiento, y velocidades de operación más bajas.

---

de libertad del controlador del robot. Cuando la cantidad de grados de libertad de control es menor que la cantidad de grados de libertad en el espacio, se habla de un robot no-holonómico. En el caso actual, el robot cuenta con 3 grados de libertad en el espacio (posición en x, y y su orientación en θ), pero solamente 2 grados de libertad en el control (velocidad de la rueda derecha y velocidad de la rueda izquierda), por lo que es un robot no-holonómico.

3.2.6. Grafo Computacional del Sistema

El grafo computacional corresponde a la red de todos los elementos que componen el sistema implementado en ROS, así como las interconexiones entre estos. En particular, en las Figuras 3.6 y 3.7 a continuación, se representa cada nodo del sistema como un vértice, y las conexiones entre estos representan los tópicos de comunicación que los comunican. La Figura 3.6 muestra el sistema bajo operación de navegación normal, y la Figura 3.7 muestra el sistema bajo operación SLAM Online:

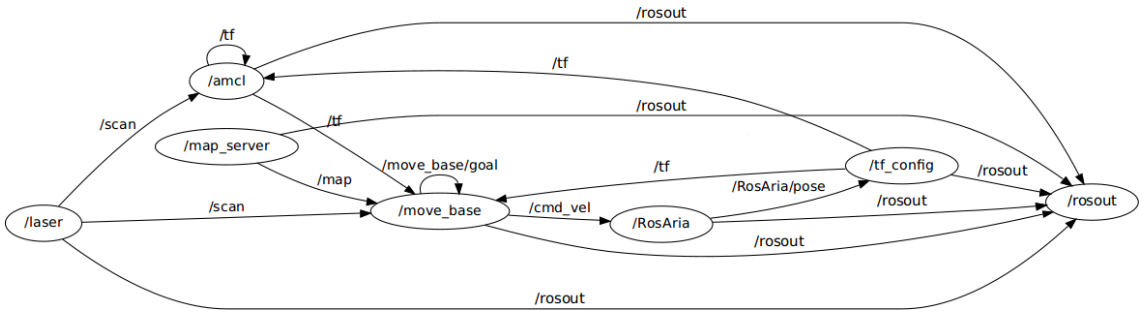


Figura 3.6: Grafo Computacional del Sistema: Navegación Normal

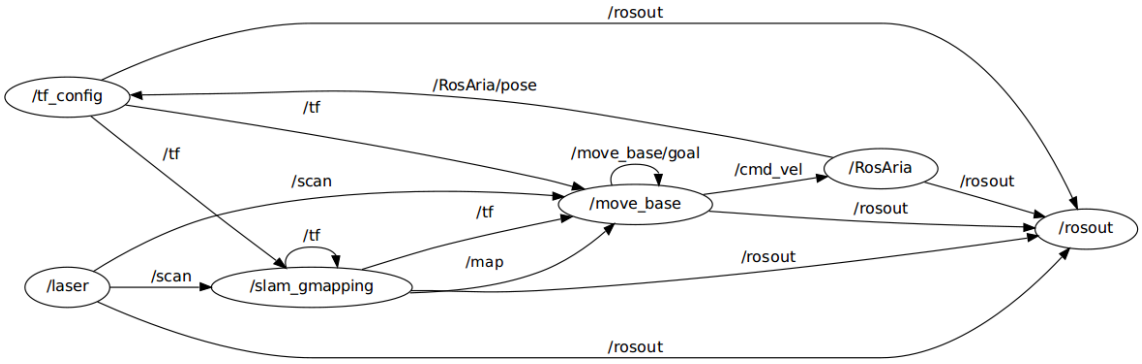


Figura 3.7: Grafo Computacional del Sistema: SLAM Online

Se observa en primer lugar, para el sistema bajo navegación normal, como el nodo **laser**, que corresponde al nodo que realiza las lecturas del sensor laser entrega las mediciones a través del tópico */scan* al nodo de localización **amcl**, y al nodo de navegación **move_base**.

El nodo **RosAria**, a su vez, recibe comandos de velocidad del nodo **move_base** a través del tópico */cmd_vel*, y publica la odometría de la base Pioneer a través del tópico */RosAria/pose*. Este mensaje de odometría es recibido por el nodo **tf_config**, que genera en base a la odometría el marco de referencia */odom* publicado bajo el tópico *tf*.

Se observa también el nodo **map_server**,que publica el mapa de entorno a través del tópico */map*, que es recibido por el nodo **move_base**. Este nodo, **move_base**, recibe desde el nodo de localizacion **amcl** los marcos de referencia completos y localizados del robot, y en base a esto,

genera los comandos de velocidad para entregarle al nodo **RosAria** de modo de navegar hacia el objetivo.

Finalmente se observa que todos los nodos publican en el tópico */rosout*, que corresponde al tópico que recolecta los mensajes informativos, de debug, avisos y errores del sistema, y que son recopilados por el nodo homónimo.

Para el sistema utilizando SLAM Online, se observa que el nodo de localización **amcl** y el servidor de mapas **map_server** desaparecen, siendo ambos reemplazados por el nodo SLAM **gmapping**. Esto pues el nodo SLAM genera tanto la localización como el mapa del entorno, y los entrega al resto del sistema, que se mantiene sin ningún cambio. Este es un claro ejemplo de la transparencia y modularidad del sistema, dado que se observa como se puede cambiar dos elementos del sistema (en este caso, localización y servidor de mapas) por otro completamente distinto, sin que el resto del sistema se vea afectado.

3.2.7. Marcos de Referencia

A continuación, se presentan todos los marcos de referencia utilizados en el sistema, con una descripción de ellos y sus conexiones. Los marcos considerados son los implementados a través de la herramienta *tf*.

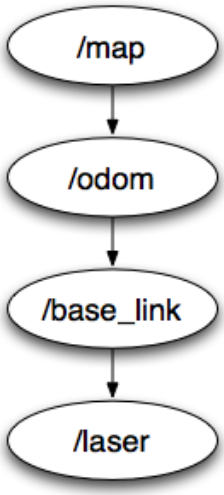


Figura 3.8: Grafo de Marcos de Referencia del Sistemal

Se observa de la figura 3.8 que existen 4 marcos de referencia principales: El primero corresponde a */map* y representa el marco de referencia fijo al mapa del entorno. El segundo, es el marco */odom*, hijo de */map*, y que correspone a la posición del robot según la odometría del sistema (corregido según el algoritmo de localización). La relación entre el mapa y el marco de odometría viene dado por el nodo de localización **amcl**. El tercer nodo es el nodo */base_link*, y corresponde a la relación entre el marco de referencia de la odometría y el centro de la base móvil. En el caso del

sistema actual, el origen es el mismo, pero aún así es necesario explicitarlo, lo que se hace a través del nodo **tf_config**. El último marco de referencia utilizado corresponde al marco */laser*, y que corresponde al marco de referencia donde se encuentra montado el sensor laser Hokuyo del robot. La relación entre este marco, y el marco */base_link* (es decir el centro de la base), es publicado por el nodo **tf_config**.

En base a la configuración utilizada al crear los nodos, se tiene que la velocidad de actualización de cada relación entre marcos de referencia es de 10 [Hz], excepto entre los marcos */base_link* y */laser*, que es de 20 [Hz], para igualar el ciclo de funcionamiento al ciclo de actualización del sensor láser.

3.2.8. Configuración Herramienta Rviz para Navegación

La herramienta de visualización **Rviz** puede ser configurada para permitir controlar la navegación del robot, y de esta forma poder ver el mapa utilizado, los datos que entregan los sensores, la pose que entrega el sistema de localización, las rutas generadas; así como también poder enviar instrucciones de navegación, como son nuevos objetivos de navegación, o pose de localización inicial.

Para esto, es necesario realizar algunos cambios. En primer lugar es necesario definir como *Fixed Frame* el marco de referencia */map*, y como *Target Frame*, el marco */base_link*. Luego se deben agregar las siguientes visualizaciones mediante el botón *Add*, según se describe en la Tabla 3.2:

Tabla 3.2: Configuración Visualizaciones para Navegación Rviz

Nombre	**Tipo**	**Tópico**
Floor Scan	Laser Scan	/laser
Map	Map	/map
Footprint	Polygon	/move_base/local_costmap/robot_footprint
Obstacles	Grid Cells	/move_base/local_costmap/obstacles
Inflated Obstacles	Grid Cells	/move_base/local_costmap/inflated_obstacles
Pose Array	Pose Array	/particlecloud
Global Plan	Path	/move_base/NavfnROS/plan
Goal	Pose	/move_base/current_goal

Finalmente, en la sección *Tool Properties*, es necesario definir que el tópico de *2D Nav Goal* sea */move_base_simple/goal*, y que el tópico de *2D Pose Estimate* sea */initialpose*.

Capítulo 4

Integración del Sistema

4.1. Paso de módulos a URBI 2.7.4

El primer paso para integrar el sistema de navegación desarrollado en ROS, con el resto del robot, consiste en pasar los módulos ya existentes, implementados en URBI 1.0, a URBI 2.7.4. Para esto, es necesario recompilar por completo el código fuente de cada módulo incluyendo los encabezados y el código de la nueva versión de URBI.

Los módulos a ser actualizados son:

UExec Módulo que permite ejecutar llamadas a sistema.

UHead Módulo de control de la cabeza del robot.

ULaserTracker Módulo que implementa seguimiento general a través de sensor láser.

UPersonTracker Módulo que implementa seguimiento de personas a través de sensor láser.

USpeech Módulo de síntesis y reconocimiento de voz.

UVision2 Módulo de visión computacional.

UMainGui Módulo que provee la interfaz gráfica de control del robot.

Existen dos módulos que no serán modificados: *UVisionThermalVisible* y *UArm2*. Esto debido a que ambos módulos deben ser reescritos casi por completo por razones ajenas al proyecto, por lo que el traspaso del código en su forma actual no presenta una mejora ni aporte al proyecto.

4.1.1. Cambios Generales

El primer paso de la actualización corresponde a traspasar el proyecto Visual Studio 2005 de cada módulo a la versión Visual Studio 2008. Para esto, es necesario abrir los proyectos antiguos

con Visual Studio 2008, y seguir las instrucciones de actualización que aparezcan.

Terminado el traspaso del proyecto a Visual Studio 2008, se deben modificar las librerías de URBI a incluir. Para esto, se abren las opciones de compilación del proyecto, se selecciona como objetivo de compilación el modo *Release*, y se realizan los siguientes cambios:

En primer lugar, es necesario modificar la inclusión de los encabezados de URBI para que considere el código fuente de la nueva versión. Para esto, dentro de la categoría *Additional Include Directories* se eliminan todos los directorios relacionados con URBI y se agrega el nuevo directorio *include* de la carpeta donde se instaló URBI 2.7.4 (en este caso *C:\Program Files\Gostai Engine Runtime\2.7.4\include*). Luego, en la categoría *Additional Library Directories* se eliminan nuevamente las referencias a URBI y se agregan los directorios *bin* y *engine* de la nueva versión de URBI. En la misma línea, dentro de la categoría *Additional Dependencies* se eliminan todas las dependencias asociadas a URBI y se agregan las siguientes: *libjpeg4urbi-vc90.lib libport-vc90.lib libsched-vc90.lib libuobject-vc90.lib*.

Dado que las librerías Boost utilizadas por URBI requieren ser compiladas de forma dinámica, es necesario agregar (en la sección *Preprocessor Definitions*) la siguiente directiva de preprocesador: *BOOST_ALL_DYN_LINK;* que fuerza la compilación de las librerías Boost en modo dinámico. A continuación, se modifica el tipo de información de debug a utilizar, cambiando *Debug Information Format* a *\Z7* para facilitar la compilación de los proyectos grandes (como Uvision2 y UVisionThermalVisible).

Dado que desde URBI 2.7.4 en adelante los proyectos deben ser compilados como una librería dinámica y no como un ejecutable, es necesario modificar en el menú *General* la opción *Configuration Type*, y seleccionar *Dynamic Library (.dll)* como la configuración objetivo. Finalmente, para posibilitar la ejecución de los módulos desde Visual Studio a modo de debug, es necesario modificar dentro del mismo menú anterior el campo *Command* y seleccionar el ejecutable *urbi-launch.exe*, ubicado en: *C:\Program Files\Gostai Engine Runtime\2.7.4\bin\urbi-launch.exe*; y cambiar el campo *Command Arguments* por:

```
-s "$(targetDir)\$(targetFileName)" -- --host 0.0.0.0 --port 54000
```

Lo anterior le indica a Visual Studio que al momento de ejecutar la solución, es necesario invocar al programa *urbi-launch.exe* que levantará un servidor URBI, y cargará el módulo compilado a través de la librería generada. Los argumentos entregados en *Command Arguments* indican en primer lugar la ubicación de la librería .dll generada por la solución, y además permiten la conexión de clientes de cualquier origen al puerto 54000.

4.1.2. Cambios Específicos

UMainGui El módulo UMainGui utiliza la librería QT para generar una interfaz gráfica. Por esta razón, para poder compilar el módulo es necesario instalar e incluir los encabezados QT necesarios.

Para esto, es necesario instalar QT 4.5.1, y asegurarse que la variable de entorno *$QTDIR*, que define el directorio de instalación de QT, esté bien definida. Luego, es necesario agregar a la variable de entorno *$PATH* los directorios *$QTDIR* y *$QTDIR/bin*.

A continuación, en el proyecto de Visual Studio asociado al módulo, se agrega el directorio *$QTDIR/bin* a la lista de directorios de librerías adicionales, y agregar las siguientes librerías de QT a la lista de dependencias adicionales: *qtmaind.lib, QtCored4.lib, QtGuid4.lib*.

UVision2 Para compilar el módulo UVision2 es necesario utilizar la versión más nueva de la librería SIFT realizada por Patricio Loncomilla [19,20], recompilada con Visual Studio 2008, OpenCV 2.1 y sin uso de ATL. Una vez recompilada esta librería, se reemplazan los archivos de la carpeta *UVision2/SIFT* por la versión nueva de la librería recién compilada.

4.2. Integración Sistema ROS con URBI 2.7.4

La integración del sistema ROS con URBI 2.7.4 busca lograr la comunicación bidireccional entre los módulos implementados en ROS (navegación, sensado laser, control de la base móvil, localización, generación de mapas) y los ya existentes implementados en URBI.

Para esto, es necesario aprovechar el soporte nato que tiene URBI 2.7.4 con ROS, lo que permite que URBI se conecte al nodo maestro de ROS y sea capaz de publicar y suscribirse a tópicos, invocar servicios, y obtener información del sistema, como cantidad de nodos conectados, servicios y tópicos disponibles, y más.

Dados los objetivos del presente trabajo, existen tres elementos necesarios que requieren de la comunicación entre ROS y URBI: En primer lugar, es necesario que la información de localización, es decir la pose actual del robot, sea conocida por el resto del sistema. En segundo lugar, es necesario poder enviar objetivos al sistema de navegación, de forma de poder mover el robot. Finalmente, es necesario ser capaz de recibir retroalimentación por parte del sistema de navegación de manera de saber si se logró llegar al objetivo, y si no fue posible, conocer las razones.

Además de los tres elementos anteriormente descritos, es necesario generar ciertas funciones administrativas que permitan comunicarse entre ROS y URBI con el objetivo de asegurar la buena calidad de funcionamiento del sistema. Estas funciones serán: verificar la correcta conexión entre ROS y URBI; verificar la cantidad, nombre y tipo de nodos conectados y registrados al maestro; obtener una lista de topics disponibles; obtener una lista de servicios disponibles.

4.2.1. Conexión entre ROS y URBI

Para realizar la comunicación entre ROS y URBI, se utilizan las funciones internas de URBI 2.7.4 que tienen soporte para ROS, y que facilitan la interacción entre ambos sistemas.

En primer lugar, es necesario tener las variables de entorno asociadas a ROS, como por ejemplo *$ROS_ROOT*, bien definidas. A continuación, se levanta un nodo maestro de ROS y una sesión URBI:

```
$ roscore &
$ rlwrap urbi -i
```

Una vez en la sesión de URBI, se puede verificar que la conexión entre el servidor URBI y el nodo maestro ROS se haya montado correctamente a través del siguiente comando:

```
Global.hasLocalSlot("Ros");
```

Frente a lo cual el servidor debería responder *true*. Si esto no ocurre, es posible que el módulo ROS no esté cargado en el sistema URBI (probablemente porque las variables de entorno de ROS no han sido definidas correctamente), por lo que es necesario levantar el módulo de forma manual:

```
loadModule("urbi/ros");
```

Una vez creada la conexión entre ROS y URBI, es posible utilizar ciertas construcciones propias de URBI para verificar la existencia de nodos ROS, y recibir información de ellos, así como interactuar de forma bidireccional.

Revisión de Nodos y Tópicos

Hecha la conexión, se genera un diccionario llamado *Ros.nodes*, que contiene como llaves los nombres de los nodos cargados, y como valores los tópicos suscritos y los publicados. Para poder interactuar con este diccionario, primero es necesario guardar una referencia a el:

```
var nodos = Ros.nodes;
```

Teniendo esta referencia, es posible entonces ver la lista de nodos existentes con el siguiente comando:

```
nodos.keys;
```

Que entregará una lista de los nodos registrados. Una respuesta de ejemplo se muestra a continuación:

```
nodos.keys;
[00000002] ["/rosout", "/urbi_12730604222952507O3", "/talker"]
```

A partir de esta lista, es posible obtener la lista de tópicos publicados, por ejemplo, de la siguiente manera:

```
nodos["talker"]["publish"];
[00000004]["/logger", "/msg"]
```

Suscripción de Tópicos

Para suscribirse a un tópico, es necesario primero generar un objeto que represente el tópico dentro del sistema. A continuación, es necesario suscribirse a través del objeto a los mensajes del tópico, y finalmente definir un evento que se lanzará cada vez que un nuevo mensaje llegue, y que definirá la acción a ejecutar.

El objeto que representará el tópico es una instancia de Ros.Topic, a la cual se le entrega el nombre del tópico (en este caso, *"/pose"*):

```
var pose = Ros.Topic.new("/pose");
```

Este objeto es luego suscrito al tópico:

```
pose.suscribe;
```

Y finalmente, se define el evento que se ejecutará al recibir un nuevo mensaje:

```
var tagPose = Tag.new;
tagPose: at (pose.onMessage?(var e))
    //Ejecutar este código
    echo(e);
```

El código anterior ejecutará el comando *echo(e)* cada vez que se reciba un mensaje nuevo en el tópico pose. La variable *e* representa al mensaje recibido, y su contenido depende de la estructura del mensaje asociado al tópico.

Publicación de Tópicos

Para publicar mensajes en un tópico, el procedimiento es similar. En primer lugar, se inicializa un objeto que representará el tópico. Luego, se realiza un *advertise*, es decir, se anuncia que se comenzará a publicar en ese tópico y se indica de que tipo es el mensaje. Finalmente, se genera una plantilla con la estructura del mensaje, se rellena con los datos a enviar, y se publica.

Supongamos que se desea publicar un comando de velocidad del tipo */Velocity* en el tópico */command_velocity*. Es importante notar la estrucutra del mensaje del tipo */Velocity*, que contiene dos campos, uno llamado *'linear'* que representa la velocidad lineal, y uno llamado *'angular'* que representa la velocidad de rotación.

Primero, generamos el objeto que representa el tópico, y realizamos el aviso que determina que se publicará en ese tópico y con el tipo de mensaje */Velocity*:

```
var velocidad = Ros.Topic.new("/command\_velocity");
velocidad.advertise("/Velocity");
```

A continuación, se obtiene la estructura del mensaje y se rellenan los campos asociados:

```
var msg = velocidad.structure.new;
msg["linear"] = 1;
msg["angular"] = 0;
```

Finalmente, el mensaje se publica y es enviado a todos los nodos suscritos a él:

```
velocidad << msg;
```

4.2.2. Funciones de Navegación

En base a la información anterior, se generaron las 4 funcionalidades necesarias para controlar el sistema de navegación basado en ROS desde URBI.

Iniciar Localización

El primer paso corresponde a entregarle al sistema de localización (*AMCL*) la posición inicial del robot y la covarianza asociada, que da una medida de la confianza que se tiene de la localización inicial.

Se muestra a continuación el código que implementa la funcionalidad:

```
function iniciarLocalizacion(x_ini, y_ini, t_ini, cov_x, cov_y, cov_t){
    var iniLoc = Ros.Topic.new("/initialpose");
    iniLoc.advertise("geometry_msgs/PoseWithCovarianceStamped");
    var initialPose = iniLoc.structure.new;

    initialPose["pose"]["pose"]["position"]["x"]=x_ini;
    initialPose["pose"]["pose"]["position"]["y"]=y_ini;
    initialPose["pose"]["pose"]["orientation"]["z"]=cos(t_ini/2);
    initialPose["pose"]["pose"]["orientation"]["w"]=sin(t_ini/2);
    initialPose["pose"]["covariance"] =   \\
        [cov_x, 0.0, 0.0, 0.0, 0.0,  0.0, \\
        0.0, cov_y, 0.0, 0.0, 0.0, 0.0,   \\
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,      \\
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,      \\
        0.0, 0.0, 0.0, 0.0, 0.0, 0.0,      \\
```

```
        0.0, 0.0, 0.0, 0.0, 0.0, cov_t];
    initialPose["header"]["frame_id"] = "/map";

    iniLoc << initialPose;

};
```

La función recibe como parámetros la pose inicial del robot a través de 3 parámetros de posición, *x_ini, y_ini y t_ini*, que representan la posición inicial en los ejes x e y y su orientación en θ respectivamente; y 3 parámetros que representan la varianza de la posición en cada eje y su orientación en θ (se asume que la matriz de covarianza es diagonal), todos con respecto al sistema de referencia solidario al mapa (*/map*).

En primer lugar, se genera un objeto *iniLoc* que representa el tópico */initialpose* sobre el cual se publicará la pose inicial y al cual está suscrito el sistema de localización *AMCL*. A continuación, se anuncia la publicación de mensajes sobre este tópico y se define que serán de tipo *geometry_msgs/PoseWithCovarianceStamped*.

Hecho el anuncio de publicación sobre el tópico, se procede a generar un objeto *initialPose* que instancia la estructura del mensaje a ser publicado, y sus campos son llenados de acuerdo a los parámetros recibidos como argumentos de la función. Finalmente, se publica el mensaje al tópico.

Leer Pose Actual

Para leer la pose actual, se toma provecho de la funcionalidad orientada a eventos que ofrece Urbiscript, y se implementa código que permite que la actualización de la pose se realice de forma constante y automática cada vez que el sistema de navegación la actualice. La implementación se basa en la declaración de variables globales *x_actual, y_actual* y *t_actual*, declaradas en el archivo *variables.u*, y se muestra a continuación:

```
var topicPose = Ros.Topic.new("/amcl_pose");
topicPose.subscribe;

var poseTag = Tag.new|;
poseTag: at(topicPose.onMessage?(var msg)){
    Global.x_actual = msg["pose"]["pose"]["position"]["x"];
    Global.y_actual = msg["pose"]["pose"]["position"]["y"];
    Global.t_actual = 2*acos(msg["pose"]["pose"]["orientation"]["z"]);
};
```

En primer lugar, se genera un objeto *topicPose* que representa el tópico de comunicación */amcl_pose*, y se suscribe al tópico publicado por **AMCL**. A continuación, se genera un *tag* llamado poseTag que contiene el código a ser ejecutado al recibirse un mensaje nuevo en el tópico

anteriormente descrito. De esta forma, cada vez que llegue un nuevo mensaje, se actualizarán las variables globales *x_actual, y_actual* y *t_actual* con la información de pose más reciente.

Enviar Objetivo de Navegación

El envío de un objetivo de navegación al sistema es un proceso similar a enviar la posición inicial, solo que más simple dado que no considera la covarianza asociada a la pose. Por esta razón, la implementación mostrada a continuación es muy similar a la solución creada para la inicialización de la localización:

```
function objetivoNav(x_obj, y_obj, t_obj){
    var objNav = Ros.Topic.new("/move_base_simple/goal");
    objNav.advertise("geometry_msgs/PoseStamped");
    var goal = objNav.structure.new;

    goal["pose"]["pose"]["position"]["x"]=x_obj;
    goal["pose"]["pose"]["position"]["y"]=y_obj;
    goal["pose"]["pose"]["orientation"]["z"]=cos(t_obj/2);
    goal["pose"]["pose"]["orientation"]["w"]=sin(t_obj/2);
    goal["header"]["frame_id"] = "/map";

    objNav << goal;

};
```

El método recibe como argumentos la pose deseada del robot respecto al sistema de referencia del mapa (*/map*), en los ejes *x, y* y su orientación en θ en radianes. Luego, se genera un objeto *objNav* que representa el tópico sobre el cual se publicará, y se anuncia que se publicarán mensajes de tipo */move_base_simple/goal*. A continuación, se genera un objeto *goal* que representa la estructura del mensaje a ser enviado. Este objeto se rellena con la pose objetivo deseada y se publica en el tópico correspondiente.

Guardar Información Semántica del Mapa

Como se explicitó en los objetivos del trabajo, es necesario que exista una manera simple de guardar información semántica respecto al mapa. En particular, es necesario poder asignarle nombres a diversos puntos de este, de forma de poder referenciarlos en base al nombre en vez de sus coordenadas, como por ejemplo: “sofá”.

Para esto, se utilizan un tipo de construcción en Urbiscript llamada *Dictionary* (diccionario). Un diccionario es una arreglo asociativo, es decir un arreglo donde el índice (llave) de cada objeto es otro objeto de cualquier tipo.

De esta forma, se puede crear un diccionario con 3 campos, x, y y θ, para cada punto a almacenar, e incluso puede almacenarse un arreglo de diccionarios, que contenga el conjunto completo de puntos guardados, con sus nombres.

Para crear un diccionario, se utiliza el constructor de la clase, entregando pares llave-elemento como se muestra a continuación:

```
var sofa = Dictionary.new("x", 1, "y", 0, "theta", pi);
[00000005] ["x" => 1, "y" => 0, "theta" => 3.14159265]
```

Creado este diccionario es posible entonces referenciarse a cada elemento por separado, en base a su llave, o al diccionario completo:

```
sofa["x"];
[00000006] 1
sofa["y"];
[00000007] 0
sofa;
[00000009] ["x" => 1, "y" => 0, "theta" => 3.14159265]
```

4.2.3. Conexión entre Windows y Linux

Dado que se tendrán elementos de URBI corriendo en Linux y en Windows al mismo tiempo, y en computadores diferentes, es necesario determinar como será la estructura del sistema. Para esto, es importante tener en cuenta que tanto el servidor de ROS como el de URBI correrán en Linux, y que este último debe recibir conexiones externas. Esto se logra levantando el servidor en URBI como sigue:

```
$ urbi -H 0.0.0.0 -P 54000
```

De esta forma, cuando se lance un UObject desde el computador Windows, debe hacerse a través de **urbi-launch** definiendo que la conexión se hará al computador con Linux (de IP: 192.168.4.29):

```
urbi-launch -r -H 192.168.4.29 -P 54000 NombreUObject
```

Finalmente, es importante tener siempre en cuenta que el servidor de ROS debe ejecutarse antes de lanzar el servidor URBI. Para facilitar este proceso se creó un script en Linux de lanzamiento que automatiza el levantar ambos servidores con su configuración y orden correcto, en el archivo: *urbi-ros-server.sh*. De forma análoga, se crearon archivos de procesamiento por lotes que permiten levantar de forma simple cada módulo del sistema de forma remota. Estos archivos de procesamiento por lotes se encuentran en la carpeta *URBI2.0/launchfiles*.

4.2.4. Diagrama de Conexiones del Sistema

Para facilitar la comprensión del sistema, se presenta a continuación en la Figura 4.1, un diagrama que representa el sistema completo, y las conexiones entre ambos, así como los computadores donde se ejecuta cada módulo:

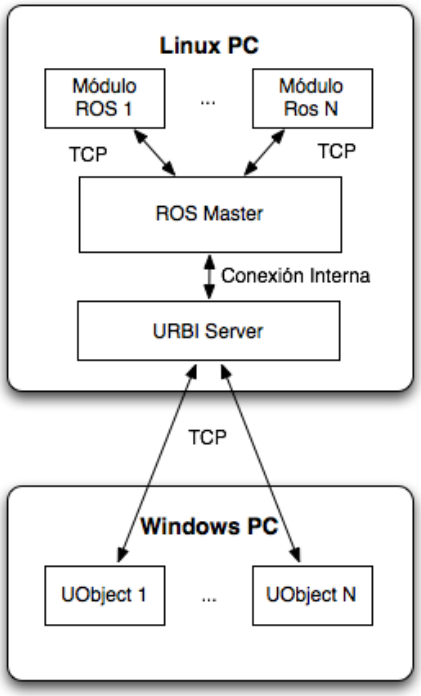


Figura 4.1: Diagrama de Conexiones del Sistema Completo

En la figura anterior se observan dos recuadros que representan los computadores ejecutando Linux y Windows, respectivamente, con rectángulos internos que representan componentes de software ejecutándose dentro de ellos. Se observan también líneas continuas que indican las conexiones entre elementos del sistema.

De esta forma, se clarifica como en Linux existen módulos que se comunican con el nodo ROS Master, y como a su vez la comunicación con URBI se realiza de forma interna, mientras que los UObjects residentes en Windows, se comunican con el servidor URBI vía TCP.

Capítulo 5

Pruebas del Sistema

En este capítulo se presentan algunas pruebas y resultados respecto al sistema implementado en los capítulos anteriores, verificando la funcionalidad de lo realizado, y de la integración de los sistemas.

5.1. Pruebas de Navegación

Se presentan a continuación algunas pruebas realizadas para

En primer lugar, se describirá el entorno utilizado para realizar las pruebas, a continuación se mostrarán las pruebas realizadas con sus resultados, y finalmente se presentan algunas conclusiones de lo hecho.

5.1.1. Entorno de Pruebas

Las pruebas de navegación fueron realizadas en el Laboratorio de Mecatrónica del Departamento de Ingeniería Eléctrica de la Universidad de Chile. Dentro de este laboratorio, se asignó un sector de pruebas que simula un entorno casero típico, con un par de sillones, un estante y una mesa de centro.

Una fotografía entorno de pruebas se muestra en la Figura 5.1 a continuación:

Figura 5.1: Entorno de Pruebas

Este espacio es de $20[m^2]$, y contiene algunos pasajes estrechos, con un ancho mínimo de $80[cm]$, dejando un márgen de $11[cm]$ para el paso del robot, que permiten verificar la capacidad de navegación en espacios reducidos, proceso que requiere un excelente desempeño de todos los sistemas que componen la navegacón (sensado, actuación, localización).

5.1.2. Pruebas Realizadas

Se describen a continuación las pruebas de navegación realizadas y sus resultados:

Prueba de Localización

En esta prueba, se busca verificar la calidad del sistema de localización. Para esto, se coloca el robot en modo teleoperación, y se levanta el sistema de navegación sin enviar un objetivo, de tal forma que solo funcione la localización. A continuación, se le entrega la pose inicial del robot de forma aproximada (con un error de posición de $0.5[m]$ y de orientación de 20° aproximadamente).

Al momento de entregar la pose inicial, se activa la visualización de las partículas del filtro utilizado para localizar el robot, y se observa que la distribución de estas es amplia en el espacio, como muestra la siguiente figura, y que la posición del robot no calza con su ubicación real (lo que se nota debido a que las mediciones del sensor láser no calzan con los obstáculos del mapa),

como muestra la Figura 5.2 a continuación:

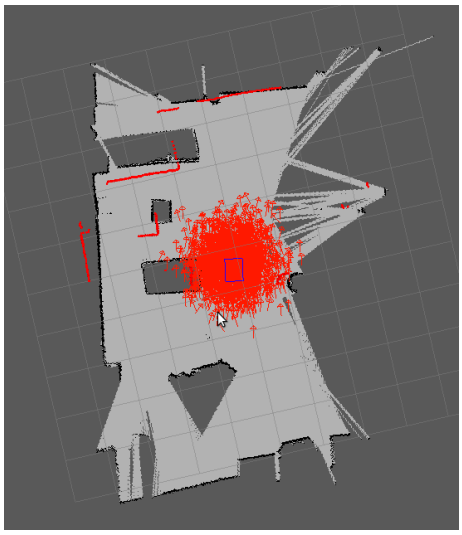


Figura 5.2: Localización Inicial

A continuación, el robot se mueve de forma manual alrededor del mapa, y se observa como en base a la odometría acumulada y las observaciones del sensor láser las partículas comienzan a acercarse a la posición del real del robot, de la misma forma, se puede ver como la estimación de la posición del robot se ajusta de mejor manera mientras más este se mueve, puesto que mientras el robot continua en movimiento, las marcas que corresponden a los obstáculos detectados por el sensor láser se ajustan a las lineas que definen estos obstáculos en el mapa, como muestra la Figura 5.3.

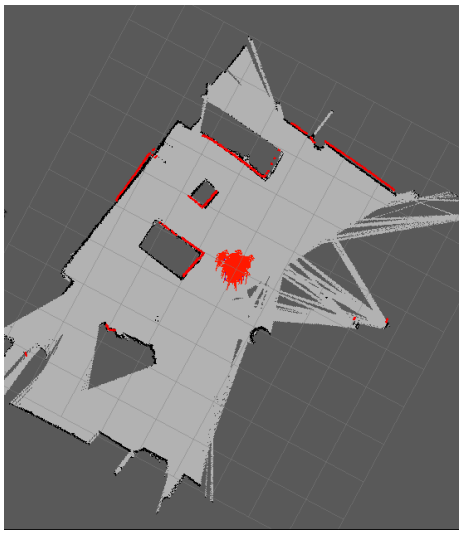


Figura 5.3: Localización Mejorada en base a Odometría y Sensado Laser

En base a la medición de la posición de la base para establecer una pose como pose verdadera base (*ground truth*), se determinó que el error de localización era menor a $1[cm]$ de desplazamiento, y menor a $5[deg]$ de rotación, luego de mover la base cerca de $1.5[m]$, para entregarle información al algoritmo de localización.

Prueba de Navegación Simple

La prueba de navegación simple corresponde a verificar la capacidad del robot de recibir objetivos de navegación simples de alcanzar (que impliquen sólamente girar y avanzar sin esquivar obstáculos) y desplazarse hasta estos de forma correcta. Para esto se definió un punto a 2 metros del robot y se le hizo avanzar hacia él, como muestra la Figura 5.4.

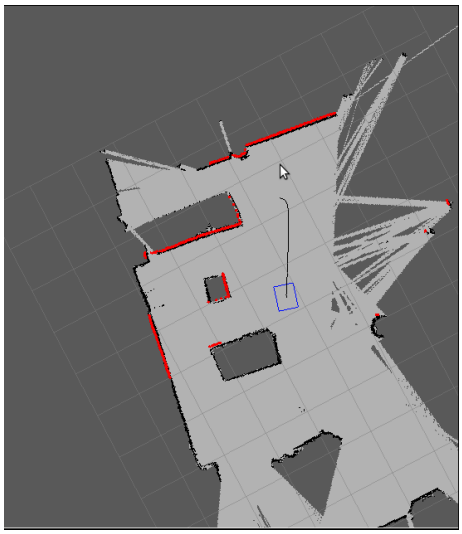


Figura 5.4: Navegación Simple

Se observó que el robot correctamente alcanzó el objetivo planteado de forma rápida y a través de una ruta eficiente, sin tener problemas al momento de alcanzar la orientación final en el objetivo seleccionado. Este proceso se repitió varias veces, con objetivos en distintas direcciones y orientaciones finales diferentes, obteniéndose siempre resultados satisfactorios.

El hecho de que el robot alcance el objetivo se evalúa, para esta prueba y las demás, en base a los parámetros de navegación, de tal forma que el robot alcance la posición objetivo, con un margen de error menor al especificado en el algoritmo, en este caso, de $10[cm]$ de distancia al punto objetivo y $0.05[rad](2.9[deg])$ grados respecto a la orientación.

Prueba de Navegación Espacios Reducidos

En este punto, se extiende el trabajo realizado en la prueba anterior, a objetivos no fáciles de ser alcanzados. Esto significa que son objetivos que se encuentran ubicados de tal forma que el

robot debe girar mas de una vez para alcanzarlos, además de pasar entre diversos obstáculos que no otorgan demasiado espacio de maniobra. Un ejemplo de objetivo utilizado, y la ruta planeada de forma automática para alcanzarlo, se muestra en la Figura 5.5:

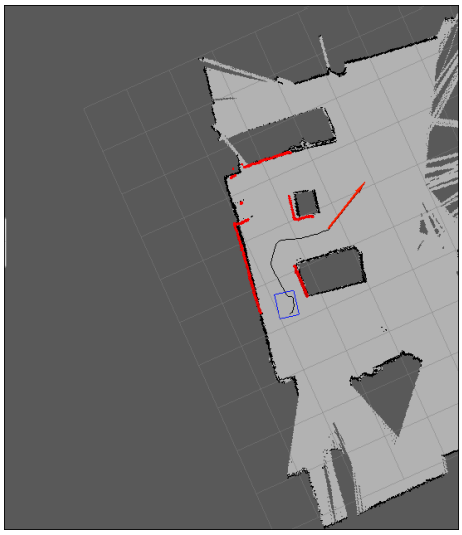


Figura 5.5: Navegación en Espacios Reducidos

Se observa que el robot alcanza de forma satisfactoria el objetivo, manteniéndose siempre a un márgen razonable de los obstáculos (cuando es posible), y siguiendo rutas satisfactorias. En ningun momento el robot colisionó ni rozó algun obstáculo. Esta prueba se repitió varias veces, obteniéndose similares resultados.

Prueba de Navegación con Obstáculos Dinámicos

Esta prueba corresponde a la prueba final, y de mayor dificultad, a la que se somete el robot. Consiste en extender la prueba anterior de navegación en espacios reducidos, incluyendo además obstáculos dinámicos no incluidos en el mapa original. Esto significa que mientras el robot se desplaza, existirá una persona en el entorno, que se interpondrá en el camino del robot. Se espera que el robot sea capaz de detectar a la persona, detenerse y/o esquivarla, y finalmente continuar su camino hacia el objetivo a través de la ruta original si la persona desocupa el paso, o de una ruta alternativa si la persona continúa impidiendo el desplazamiento a través de la ruta original.

Como se observa en la Figura 5.6, al pasar una persona caminando frente al robot, genera obstáculos dinámicos observables como puntos azules en torno a la pose del robot:

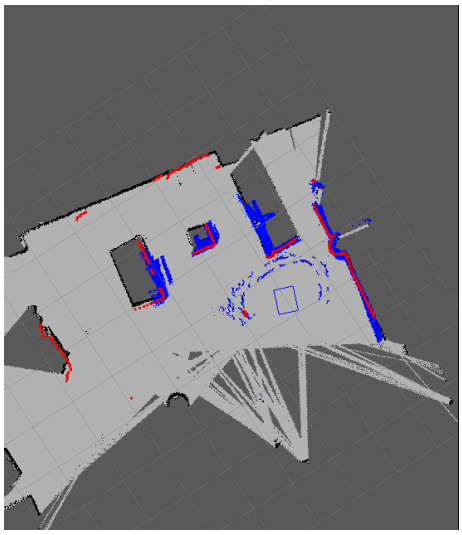


Figura 5.6: Obstáculos Dinámicos

Sin embargo, al recibir el robot nuevas medidas del sensor laser, y detectar que la persona se ha desplazado, estos obstáculos desaparecen, como muestra la Figura 5.7. Es importante notar que los obstáculos que se encuentran detrás del robot no son limpiados, debido a que el sensor láser no cubre esa zona, y por lo tanto es imposible saber si el espacio sigue desocupado o no.

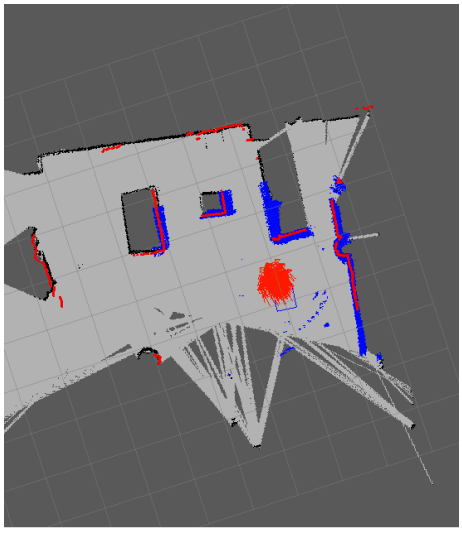


Figura 5.7: Obstáculos Dinámicos Eliminados

Esta prueba se realizó en repetidas ocasiones, y con diversos objetivos finales, consiguiéndose

siempre un correcto funcionamiento del sistema: los obstáculos fueron detectados y añadidos al mapa de costos local de forma correcta, siendo después eliminados cuando correspondía, lo que significó que el robot fue capaz de llegar al objetivo en toda las ocasiones en que esto era factible, sin colisionar en ninguna ocasión con obstáculos ya sea dinámicos o estáticos.

5.1.3. Prueba SLAM

En esta prueba, se realiza una serie de tareas de forma secuencial que sirve para probar las capacidades de SLAM, navegación en mapas dinámicos y almacenamiento de información semántica dentro del mapa, por parte del sistema.

En primer lugar, se ubica al robot en un entorno desconocido y se inicia el módulo de SLAM GMapping. Al observar el mapa existente, es posible apreciar que el robot no cuenta con información alguna de su entorno, como muestra la Figura 5.8. Sin detener el módulo SLAM, se marca la pose actual del robot en un diccionario URBI asignándole un nombre arbitrario, en este caso '*inicio*'.

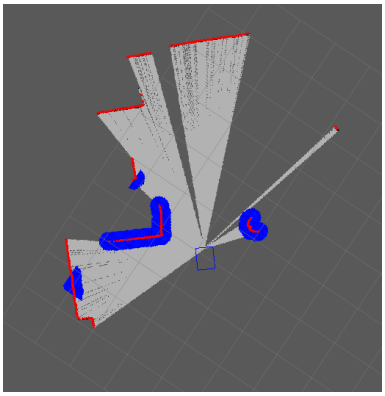


Figura 5.8: Posición inicial Prueba Slam

A continuación, mediante teleoperación se mueve al robot por el sector de pruebas, hasta que el mapa termina de completarse, como se observa en la Figura 5.9:

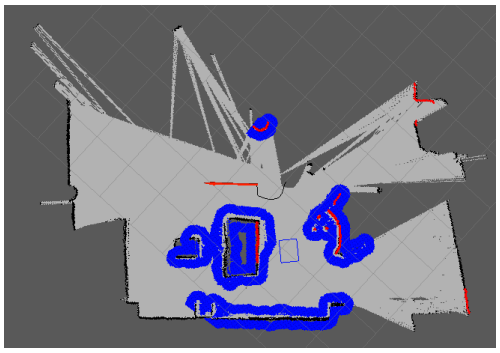


Figura 5.9: Mapa de Entorno Final

Finalmente, sin modificar el modo de operación ni detener el módulo SLAM, se le indica al robot que se desplace hacia el punto guardado como '*inicio*'. Como muestra la Figura 5.10, el sistema genera una ruta hasta el punto, después de lo cual el robot se desplaza sin problema hasta el lugar marcado.

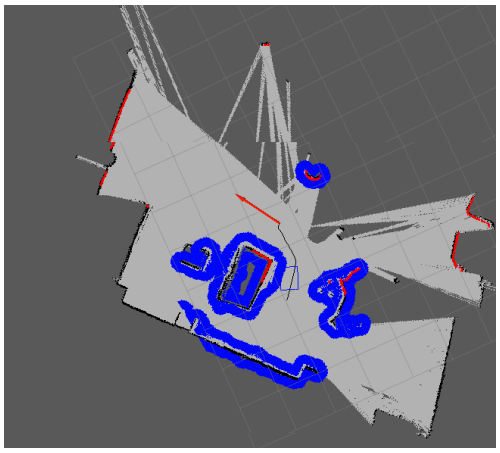


Figura 5.10: Retorno a Posición Guardada

El proceso anterior puede coninuarse más de una vez, guardando tantos puntos arbitrarios como se desee, y modificando o no el mapa entre cada punto guardado, sin que sea necesario detener el módulo SLAM para realizar la navegación a alguno de los puntos, o para el almacenado de estos.

De esta prueba se observa que el robot puede realizar el mapa de entorno con suficiente precisión de tal manera que no se produzcan desviaciones importantes respecto al punto guardado a medida que el mapa se completa. Esto se produce debido a la buena calidad de la odometría, así como a algoritmos de calzado de líneas utilizados dentro del sistema SLAM. Dada la buena

precisión del mapa generado, y la poca deformación de este en el tiempo, se observa que el robot navega sin problemas hasta el punto guardado.

5.2. Prueba de Módulos UObject

En esta sección se prueba el funcionamiento de los módulos UObject traspasados a URBI 2.7.4, y su correcta integración al sistema. En general, todos los módulos traspasados funcionan correctamente, ejecutándo su código de forma completa y registrando sus funciones en el servidor URBI, por tanto siendo accesibles desde toda la red. A pesar de esto, se decidió ejecutar algunas pruebas con los dos módulos más grandes para verificar su funcionalidad por completo.

5.2.1. USpeech

El módulo **USpeech** implementa las funcionalidades de síntesis y reconocimiento de voz. Para probar la funcionalidad de síntesis, se probó la funcionalidad *Text-to-Speech*, transformando textos en inglés y español a lenguaje hablado de forma correcta, tanto en modo de alta velocidad, como de baja velocidad.

A continuación se realizaron pruebas de reconocimiento de voz, donde se pudo comprobar que dicho módulo funcionaba de forma correcta, reconociendo nombres y diversas palabras clave, tanto en español como en inglés.

5.2.2. UVision2

El módulo **UVision2** es el encargado de la detección de rostros, reconocimiento de rostros, y algoritmos de visión en general.

Para probar la detección de rostros, se utiliza **UVision2** para leer las imágenes de una cámara web *Philips SPC 900NC*. Se encuadra un rostro humano en la imágen de la cámara y se utiliza la funcion *UVision2.frDetect()* para intentar detectar un rostro en la imagen.

Como muestra la Figura 5.11 a continuación, la detección es exitosa, marcándose el rostro y los ojos de la cara detectada.

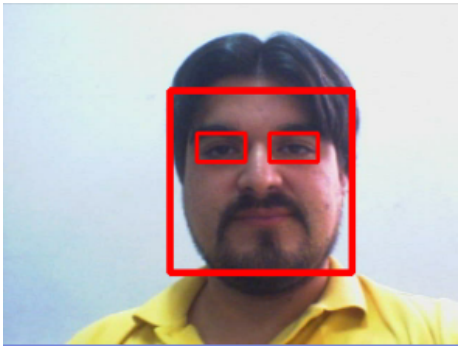


Figura 5.11: Detección de Rostro

Respecto al reconocimiento de rostros, posterior a la detección, se guarda a través de la funcion *frStoreFace* la cara detectada con un número de ID igual a 1. Luego, se encuadra un rostro de otra persona en la imagen de la cámara, se detecta, y se intenta identificar. De forma satisfactoria, el sistema indica que la cara detectada no se encuentra en la base de datos. Finalmente, se encuadra nuevamente la cara original, se detecta, y al intentar reconocerla a través de la función *frRecognize*, el sistema indica que la cara detectada correspone a la cara con ID 1. Al detectar y reconocer una cada perteneciente a otra persona, el sistema indica de forma correcta que la persona no se encuentra en la base de datos.

Capítulo 6

Conclusiones

En el presente trabajo se implementó de forma correcta el middleware ROS para el robot de servicio Bender, así como se utilizaron las librerías del sistema para la navegación del robot, y se migraron los módulos existentes en URBI a la nueva versión URBI 2.7.4. En particular se cumplieron todos y cada uno de los objetivos propuestos: Se determinó tanto la estructura de módulos como los tópicos de comunicación entre estos y se instalaron los sistemas operativos y programas necesarios para el desarrollo e implementación del sistema en ambos computadores.

Respecto al sistema de navegación, se configuraron e implementaron todos los módulos necesarios para la navegación autónoma del robot: Sensado a través de un sensor láser, localización del robot, control de la base móvil, generación de trayectorias, evasión de obstáculos y generación de mapas en base a SLAM, de forma online y offline. Además, se generaron funciones que permiten comunicar el sistema de navegación en ROS con el sistema en URBI. Es esencial notar que el sistema de navegación implementado, en conjunto con el sistema URBI, permite el almacenamiento de información semántica del mapa, de esta forma satisfaciendo el requerimiento respectivo.

Al momento de evaluar el desempeño del sistema de navegación se observa un funcionamiento superior en calidad y velocidad al sistema antiguo. En particular, se mantiene la capacidad de operación en espacios reducidos, a la vez que las trayectorias generadas son más eficientes y de giro más suave que las generadas por el sistema anterior, a la vez que siempre evade obstáculos de forma dinámica. Es importante notar que se implementó la funcionalidad de SLAM online, que permite navegar en un entorno desconocido y generar un mapa del mismo, una necesidad para el sistema, y que no había sido implementada anteriormente, lo que significa una adición de importancia al robot, sobre todo orientado a pruebas RoboCup que requieren de esta funcionalidad.

La migración de los módulos existentes a URBI 2.7.4 permite utilizar los módulos de visión, síntesis y reconocimiento de voz con el nuevo sistema, así como aprovechar las funcionalidades nuevas en Urbiscript, así como más eficiencia y estabilidad en el procesamiento, y la comunicación directa con ROS.

El diseño modular del sistema permite la abstracción e independencia de funcionalidades, de esta manera abriendo el camino a la reutilización de código abierto de terceros, y además de habilitar el reemplazo de módulos de forma transparente, abstrayéndose del resto del sistema. De esta manera, es posible modificar o reemplazar ciertos módulos del sistema, de forma que el resto del sistema no note el cambio.

Finalmente, la integración de todo el sistema al lenguaje de control y scripting Urbiscipt, permite utilizar los módulos de navegación ROS y los módulos hechos en URBI, de forma sincronizada aprovechando todas las ventajas que otorga Urbiscript al momento de determinar conductas de altmo nivel, como son el procesamiento paralelo, sincronismo y programación orientada a eventos.

6.1. Trabajos Futuros

Como trabajo futuro, se propone utilizar todas las capacidades de Urbiscript para la generación de libretos de prueba para la Robocup, aprovechando la capacidad de programación orientada a eventos que tiene el sistema.

Respecto a navegación, existen muchas posibilidades. Es de particular interés la posibilidad de incluir el sensor Kinect al sistema de navegación, y de esta manera poder detectar obstáculos que se encuentren no solo cercanos al nivel del piso, si no que de cualquier altura. Es de particular interés para la navegación autónoma debido a la existencia de obstáculos en un entorno casero, como mesas y sillas, que vistos solo con un sensor laser de baja altura, no registran como obstáculos impasables.

Bibliografía

[1] *Robocup@Home League*. www.robocupathome.org.

[2] *ROS*. www.ros.org.

[3] *ROS Boxturtle*. http://ros.org/wiki/boxturtle.

[4] *Standard Units of Measure and Coordinate Conventions*. http://www.ros.org/reps/rep-0103.html.

[5] *UChile HomeBreakers Robocup@Home Team*. www.robocup.cl.

[6] *URBI Robot Framework*. http://www.urbiforge.org/.

[7] *Urbiscript Language Reference Manual*. http://www.gostai.com/downloads/urbi/doc/urbiscript-language-reference-manual.html.

[8] *WillowGarage*, 2012. http://www.willowgarage.com/.

[9] Arulampalam, M.S., S. Maskell, N. Gordon y T. Clapp: *A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking*. Signal Processing, IEEE Transactions on, 50(2):174–188, 2002.

[10] Dijkstra, E.W.: *A note on two problems in connexion with graphs*. Numerische Mathematik, 1:269–271, 1959.

[11] Dissanayake, M.W.M.G., P. Newman, S. Clark, H.F. Durrant-Whyte y M. Csorba: *A solution to the simultaneous localization and map building (SLAM) problem*. Robotics and Automation, IEEE Transactions on, 17(3):229–241, 2001.

[12] DPRIAFRAS, H.: *II. On quaternions; or on a new system of imaginaries in algebra*. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 25(163):10–13, 1844.

[13] Fox, D., W. Burgard, F. Dellaert y S. Thrun: *Monte carlo localization: Efficient position estimation for mobile robots*. En *Proceedings of the National Conference on Artificial Intelligence*, páginas 343–349. JOHN WILEY & SONS LTD, 1999.

[14] Fox, D., W. Burgard y S. Thrun: *The dynamic window approach to collision avoidance*. Robotics & Automation Magazine, IEEE, 4(1):23–33, 1997.

[15] Fox, Dieter: *KLD-Sampling: Adaptive Particle Filters and Mobile Robot Localization*. Informe técnico, University of Washington, 2001.

[16] Gerkey, Brian P. y Kurt Konolige: *Planning and Control in Unstructured Terrain*. Informe técnico, Willow Garage.

[17] Grisetti, G., C. Stachniss y W. Burgard: *Improved techniques for grid mapping with rao-blackwellized particle filters*. Robotics, IEEE Transactions on, 23(1):34–46, 2007.

[18] Leonard, J.J., H.F. Durrant-Whyte y I.J. Cox: *Dynamic map building for an autonomous mobile robot*. The International Journal of Robotics Research, 11(4):286–298, 1992.

[19] Loncomilla, P. y J. Ruiz-del Solar: *Improving SIFT-based object recognition for robot applications*. Image Analysis and Processing–ICIAP 2005, páginas 1084–1092, 2005.

[20] Loncomilla, P. y J. Ruiz-del Solar: *A fast probabilistic model for hypothesis rejection in SIFT-based object recognition*. Progress in Pattern Recognition, Image Analysis and Applications, páginas 696–705, 2006.

[21] Moravec, H.P.: *Sensor fusion in certainty grids for mobile robots*. AI magazine, 9(2):61, 1988.

Apéndice A

Apendice A: Códigos Fuente

Nota de Diagramación: Por limitaciones de diagramación, algunas líneas de código debieron ser separadas en dos distintas. Para indicar esta modificación se utiliza el indicador "\\", continuando la línea original de programa en la línea siguiente del próximo documento. Poner atención especial a la existencia o no de espacios antes del indicador.

A.1. Programas

A.1.1. pioneer_tf.cpp

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>

//Funcion callback que se llama al recibir un mensaje de odometria 'pose'
void poseCallback(const nav_msgs::Odometry::ConstPtr& odomsg)
{
   //Generar Transformacion TF => base_link
    static tf::TransformBroadcaster odom_broadcaster;
    odom_broadcaster.sendTransform(
    tf::StampedTransform(
        tf::Transform(tf::Quaternion(odomsg->pose.pose.orientation.x,
            odomsg->pose.pose.orientation.y,
            odomsg->pose.pose.orientation.z,
            odomsg->pose.pose.orientation.w),
        tf::Vector3(odomsg->pose.pose.position.x/0001.0,
            odomsg->pose.pose.position.y/0001.0,
            odomsg->pose.pose.position.z/0001.0)),
            odomsg->header.stamp, "/odom", "/base_link"));
```

```
        ROS_DEBUG("odometry frame sent");
}

int main(int argc, char** argv){
    ros::init(argc, argv, "pioneer_tf_publisher");
    ros::NodeHandle n;

    //Definir frecuencia de actualizacion
    ros::Rate r(20);

    tf::TransformBroadcaster broadcaster;
    //Suscribirse a la odometria RosAria en el topic pose
    ros::Subscriber pose_sub = n.subscribe<nav_msgs::Odometry> \\
            ("RosAria/pose", 1, poseCallback);

    while(n.ok()){
    //Generar transformacion fija entre base_link y laser (posicion del laser)
        broadcaster.sendTransform(
            tf::StampedTransform(
             tf::Transform(tf::Quaternion(0, 0, 0), tf::Vector3(0.13, -0.04, 0.294)),
             ros::Time::now(), "/base_link", "/laser"));
    ros::spinOnce();
    r.sleep();
}
}
```

A.2. Archivos de Configuración

A.2.1. base_local_planner.yaml

```
TrajectoryPlannerROS:
  max_vel_x: 1.5
  min_vel_x: 0.1
  max_rotational_vel: 0.8
  min_in_place_rotational_vel: 0.3
  escape_vel: -0.1

  sim_time: 2.0
  sim_granualarity: 0.02
  path_distance_bias: 0.75
  goal_distance_bias: 0.5

  acc_lim_th: 3.2
```

```
  acc_lim_x: 2.5
  acc_lim_y: 0

  holonomic_robot: false

  yaw_goal_tolerance: 0.09
  xy_goal_tolerance: 0.14
  latch_xy_goal_tolerance: true

  dwa: false
```

A.2.2. costmap_common_params.yaml

```
obstacle_range: 3.5
raytrace_range: 0.30
inflation_radius: 0.5
#---pioneer AT footprint:---
#---(in meters)---
footprint: [ [0.254, -0.2], [0.254, 0.2], [-0.254, 0.2], [-0.254, -0.2]]

transform_tolerance: 0.2
map_type: costmap

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan, topic: scan, \\
marking: true, clearing: true, expected_update_rate: 0.2}
```

A.2.3. global_costmap_param.yaml

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 2.0
  publish_frequency: 10.0
  static_map: true
```

A.2.4. local_costmap_param.yaml

```
local_costmap:
  global_frame: /odom
```

```
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
  rolling_window: true
  width: 5.0
  height: 5.0
  resolution: 0.02
```

A.3. Launchfiles

A.3.1. amcl_diff.launch

```
<launch>
<node pkg="amcl" type="amcl" name="amcl" output="screen">
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0.1"/>
  <param name="transform_tolerance" value="0.2" />
  <param name="gui_publish_rate" value="10.0"/>
  <param name="laser_max_beams" value="30"/>
  <param name="min_particles" value="500"/>
  <param name="max_particles" value="5000"/>
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="odom_alpha1" value="0.2"/>
  <param name="odom_alpha2" value="0.2"/>
  <param name="odom_alpha3" value="0.8"/>
  <param name="odom_alpha4" value="0.2"/>
  <param name="laser_z_hit" value="0.5"/>
  <param name="laser_z_short" value="0.05"/>
  <param name="laser_z_max" value="0.05"/>
  <param name="laser_z_rand" value="0.5"/>
  <param name="laser_sigma_hit" value="0.2"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_model_type" value="likelihood_field"/>
  <param name="laser_likelihood_max_dist" value="2.0"/>
  <param name="update_min_d" value="0.2"/>
  <param name="update_min_a" value="0.5"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="resample_interval" value="1"/>
  <param name="transform_tolerance" value="0.1"/>
  <param name="recovery_alpha_slow" value="0.0"/>
```

```
  <param name="recovery_alpha_fast" value="0.0"/>
</node>
</launch>
```

A.3.2. bender_configuration.launch

```
<launch>
    <include file="$(find bender_nav)/launchfiles/rosaria_tf.launch" />
    <include file="$(find bender_nav)/launchfiles/hokuyo.launch"/>
</launch>
```

A.3.3. complete_map.launch

```
<launch>
    <include file="$(find bender_nav)/launchfiles/bender_configuration.launch" />
    <include file="$(find bender_nav)/launchfiles/gmapping.launch" />
    <include file="$(find bender_nav)/launchfiles/teleop_keyboard.launch" />
</launch>
```

A.3.4. complete_nav.launch

```
<launch>
    <include file="$(find bender_nav)/launchfiles/bender_configuration.launch" />
    <include file="$(find bender_nav)/launchfiles/move_base.launch"/>
    <include file="$(find bender_nav)/launchfiles/map_server.launch"/>
    <include file="$(find bender_nav)/launchfiles/amcl_diff.launch"/>
</launch>
```

A.3.5. gmapping.launch

```
<launch>
     <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" args="/scan">
          <param name="delta" type="double" value="0.02" />
          <param name="temporalUpdate" type="double" value="2.5" />
          <param name="xmin" type="double" value="-20" />
          <param name="xmax" type="double" value="20" />
          <param name="ymin" type="double" value="-20" />
          <param name="ymax" type="double" value="20" />
          <param name="map_update_interval" value="3" />
     </node>
</launch>
```

A.3.6. hokuyo.launch

```
<launch>
     <node pkg="hokuyo_node" type="hokuyo_node" name="laser" output="screen">
          <param name="min_ang" type="double" value="-2.09" />
          <param name="max_ang" type="double" value="2.09" />
     </node>
</launch>
```

A.3.7. map_server.launch

```
<launch>
     <node pkg="map_server" type="map_server" name="map_server" \\
          args="$(find bender_nav)/maps/map.yaml">
     </node>
</launch>
```

A.3.8. move_base.launch

```
<launch>
     <master auto="start"/>

     <!--- Run AMCL -->
     <include file="$(find bender_nav)/launchfiles/amcl_diff.launch" />

     <!--- Run Move_Base -->
     <node pkg="move_base" type="move_base" respawn="false" \\
               name="move_base" \\output="screen">
          <rosparam file="$(find bender_nav)/config_files/\\
               costmap_common_params.yaml" command="load" ns="global_costmap" />
          <rosparam file="$(find bender_nav)/config_files/\\
               costmap_common_params.yaml" command="load" ns="local_costmap" />
          <rosparam file="$(find bender_nav)/config_files/\\
               local_costmap_params.yaml" command="load" />
          <rosparam file="$(find bender_nav)/config_files/\\
               global_costmap_params.yaml" command="load" />
          <rosparam file="$(find bender_nav)/config_files/\\
               base_local_planner_params.yaml" command="load" />
     </node>
</launch>
```

A.3.9. rosaria_tf.launch

```
<launch>
     <node pkg="ROSARIA" type="RosAria" name="RosAria" output="screen">
          <remap from="/RosAria/cmd_vel" to="/cmd_vel" />
     </node>
     <node pkg="bender_nav" type="pioneer_tf" name="tf_config" output="screen">
     </node>
</launch>
```

A.3.10. teleop_keyboard.launch

```
<launch>
     <node pkg="pr2_teleop" type="teleop_pr2_keyboard" \\
               name="spawn_teleop_keyboard"  output="screen">
          <param name="walk_vel" value="0.25" />
          <param name="run_vel" value="0.5" />
          <param name="yaw_rate" value="0.5" />
          <param name="yaw_run_rate" value="0.75" />
     </node>
</launch>
```