Técnicas de los Sistemas Inteligentes

Práctica1: Robótica.
Sesion5. Localización y Navegación

Curso 2016-17

Localization

- **Localization** es el problema de estimar la pose de un robot relativa a un mapa conocido.
- Localization no es muy sensible a la situación exacta de objetos, por lo que puede manejar pequeños cambios en los emplazamientos de objetos.
- ROS usa el paquete **amcl** para localización

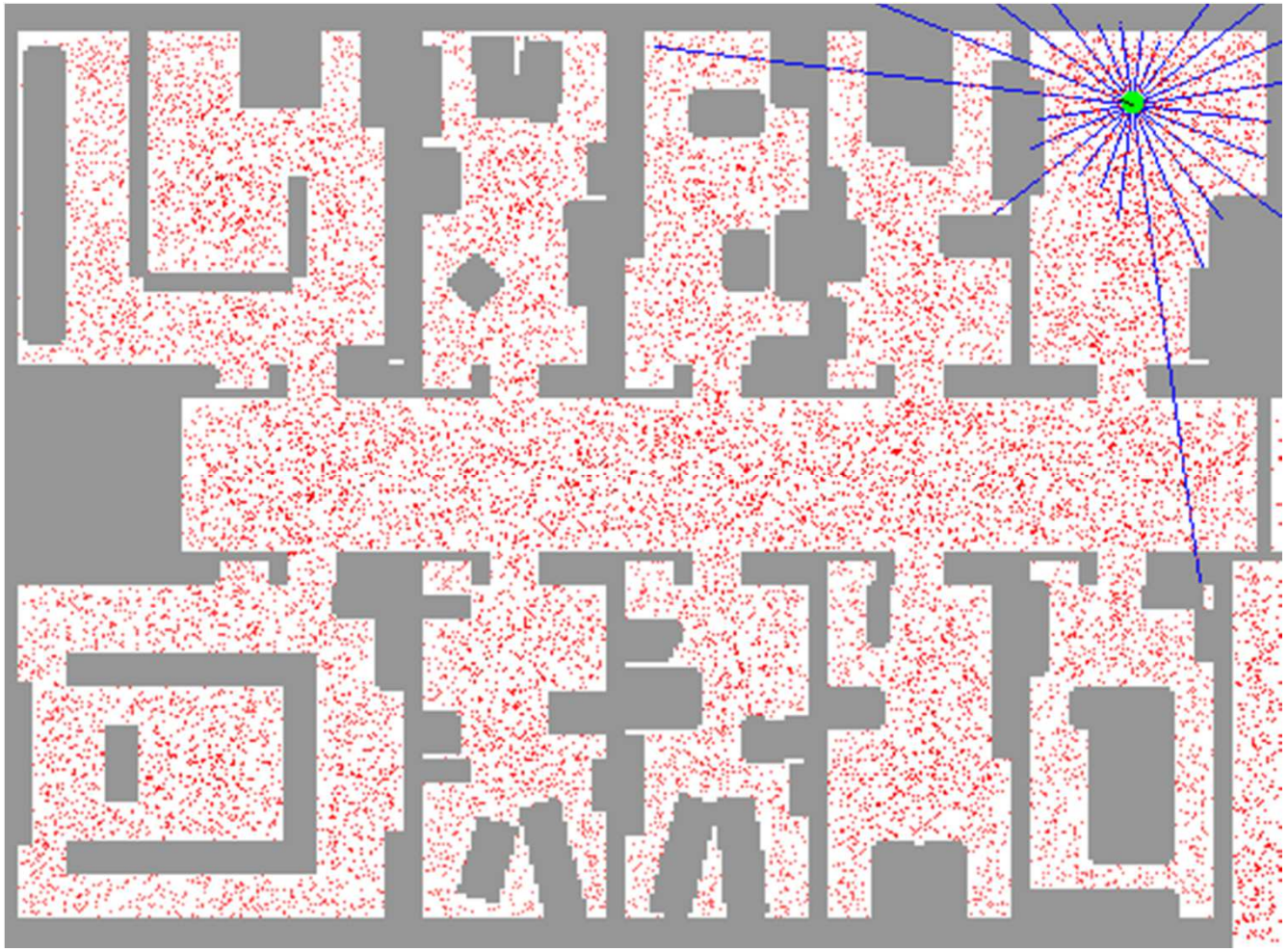
(C)2015 Roi Yehoshua

AMCL

- amcl es un sistema de localización probabilístico para un robot en 2D
- Implementa la técnica **Adaptive Monte Carlo Localization** que usa un filtro de partículas para registrar y actualizar la pose de un robot en un mapa conocido
- El algoritmo está descrito en el libro **Probabilistic Robotics** by Thrun, Burgard, and Fox (http://www.probabilistic-robotics.org/)
- amcl funciona solo con scaner láser
  - Aunque se puede extender para trabajar con otros sensores.

(C)2015 Roi Yehoshua

AMCL - Particle Filter



(C)2016 Roi Yehoshua

- amcl toma un mapa, laser scans, y devuelve estimaciones de la pose del robot
- Subscribed topics:
  - scan – Laser scans
  - tf – Transforms
  - initialpose – Mean and covariance with which to (re-) initialize the particle filter
  - map – the map used for laser-based localization
- Published topics:
  - **amcl_pose** – Robot's estimated pose in the map, with covariance.
  - Particlecloud – The set of pose estimates being maintained by the filter

(C)2015 Roi Yehoshua

Parameter	Description	Default
min_particles	Minimum allowed number of particles	100
max_particles	Maximum allowed number of particles	5000
laser_model_type	Which model to use, either beam or likelihood_field	likelihood_field
laser_likelihood_max_dist	Maximum distance to do obstacle inflation on map, for use in likelihood_field model	2.0
initial_pose_x	Initial pose mean (x), used to initialize filter with Gaussian distribution	0.0
initial_pose_y	Initial pose mean (y), used to initialize filter with Gaussian distribution	0.0
initial_pose_a	Initial pose mean (yaw), used to initialize filter with Gaussian distribution	0.0

(C)2014 Roi Yehoshua

Ejecutar el nodo amcl

- Descargar mi_mapeo_stage desde PRADO.
- Si lo has descargado antes: Actualizarlo
  - Copiar/reemplazar los directorios launch y move_base_config con los de vuestro actual paquete mi_mapeo_stage. No tocar vuestro src si habéis hecho algún ejercicio
- Sino
  - Descomprimir en el directorio <workspace>/src
  - Hacer catkin_make
  - Hacer source devel/setup.sh
  - Comprobar que el paquete se reconoce con `rospack find mi_mapeo_stage`

(C)2014 Roi Yehoshua

amcl_node.xml (1)

- Este archivo esta en mi_mapeo_stage/move_base_config

```
<launch>
<!-- ARGUMENTOS ADMITIDOS POR FICHERO LAUNCH -->
  <arg name="initial_pose_x"  default="0.0"/>
  <arg name="initial_pose_y"  default="0.0"/>
  <arg name="initial_pose_a"  default="0.0"/>
<!--
  Example amcl configuration. Descriptions of parameters, as well as a
full list of all amcl parameters, can be found at
http://www.ros.org/wiki/amcl.
-->
<node pkg="amcl" type="amcl" name="amcl" respawn="true">
  <remap from="scan" to="base_scan" />
  <param name="use_map_topic" value="false"/>
  <!-- POSICION INICIAL INTRODUCIDA POR USUARIO -->
  <param name="initial_pose_x"            value="$(arg initial_pose_x)"/>
  <param name="initial_pose_y"            value="$(arg initial_pose_y)"/>
  <param name="initial_pose_a"            value="$(arg initial_pose_a)"/>
  <!-- Publish scans from best pose at a max of 10 Hz -->
  <param name="odom_model_type" value="omni"/>
  <param name="odom_alpha5" value="0.1"/>
```

Podemos pasarle a amcl una posición inicial para facilitar el proceso de localización

amcl_node.xml (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"/>
  <!-- translation std dev, m -->
  <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_model_type" value="beam"/> -->
  <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>
```

Ejecutar el nodo amcl

- Para ejecutar amcl necesitamos ejecutar los siguientes nodos
  - map_server – para cargar y publicar el mapa
  - stageros – para el simulador Stage con un fichero world correspondiente al mapa publicado
  - amcl –para el sistema de localización
  - move_base - para que el robot navegue solo
- move_base lo veremos un poco más adelante.

(C)2014 Roi Yehoshua

mi_acml.launch

```
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>

<!-- Lanzamos move_base para navegacion -->

  <include file="$(find mi_mapeo_stage)/move_base_config/move_base.xml"/>

<!-- Lanzamos map_server con un mapa -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find mi_mapeo_stage)/stage_config/maps/willow-full-
0.05.pgm 0.05" respawn="false" />

<!-- Lanzamos stage con el mundo correspondiente al mapa -->
  <node pkg="stage_ros" type="stageros" name="stageros" args="$(find mi_mapeo_stage)/stage_config/worlds/willow-pr2-
5cm.world" respawn="false" >
    <param name="base_watchdog_timeout" value="0.2"/>
  </node>

<!-- LOCALIZACION: Lanzamos el nodo amcl -->
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find mi_mapeo_stage)/move_base_config/amcl_node.xml">
     <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
     <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
     <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
   </include>

<!-- Lanzamos rviz -->
 <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mi_mapeo_stage)/single_robot.rviz" />
</launch>
```

El nodo **map_server publica un mapa** a una resolución pasados como argumento. Los mapas están en el directorio *map* de mi_mapeo_stage
El nodo **stage_ros necesita como argumento un mundo**. Los mundos están en el directorio *worlds*.

Tener en cuenta que en el fichero **mi_gmapping.launch hay un ejemplo de como pasar otro mapa y mundo** a map_server y stage, respectivamente.

mi_amcl.launch

- Editar el fichero mi_amcl.launch y cambiar la posición inicial por 0.0 (si no está ya así puesta).
- Para ejecutar escribir:

```
$ roslaunch mi_mapeo_stage mi_amcl.launch
```

- Observar que el robot se muestra en rviz (ver el pentágono rojo representado como su footprint) en la posición origen del mapa.
- Para ver la nube de partículas:
  - Añadir un display de tipo : Pose Array
  - Asociarle el topic /particlecloud.
  - Asignarle cualquier color que no sea rojo para no confundirse con el footprint.

Particle Cloud in rviz

- **Particle Cloud** display muestra la nube de partículas usada por el sistema de localización del robot
- La dispersión de la nube representa la incertidumbre del sistema de localización sobre la pose del robot:
  - Muy dispersa = alta incertidumbre. Condensada = baja incertidumbre
  - Conforme el robot se mueve, la nube se desplaza en tamaño a medida que llegan datos adicionales del scaner, lo que permite estimar a amcl la posición y orientación del robot.
- Observar la nube de puntos alrededor del footprint del robot.
- Si hacemos “trampa” arrastrando y moviendo el robot con el ratón en Stage observaremos cómo la nube de puntos se dispersa y trata de encontrar la posición del robot.
- También podemos usar teleoperación.

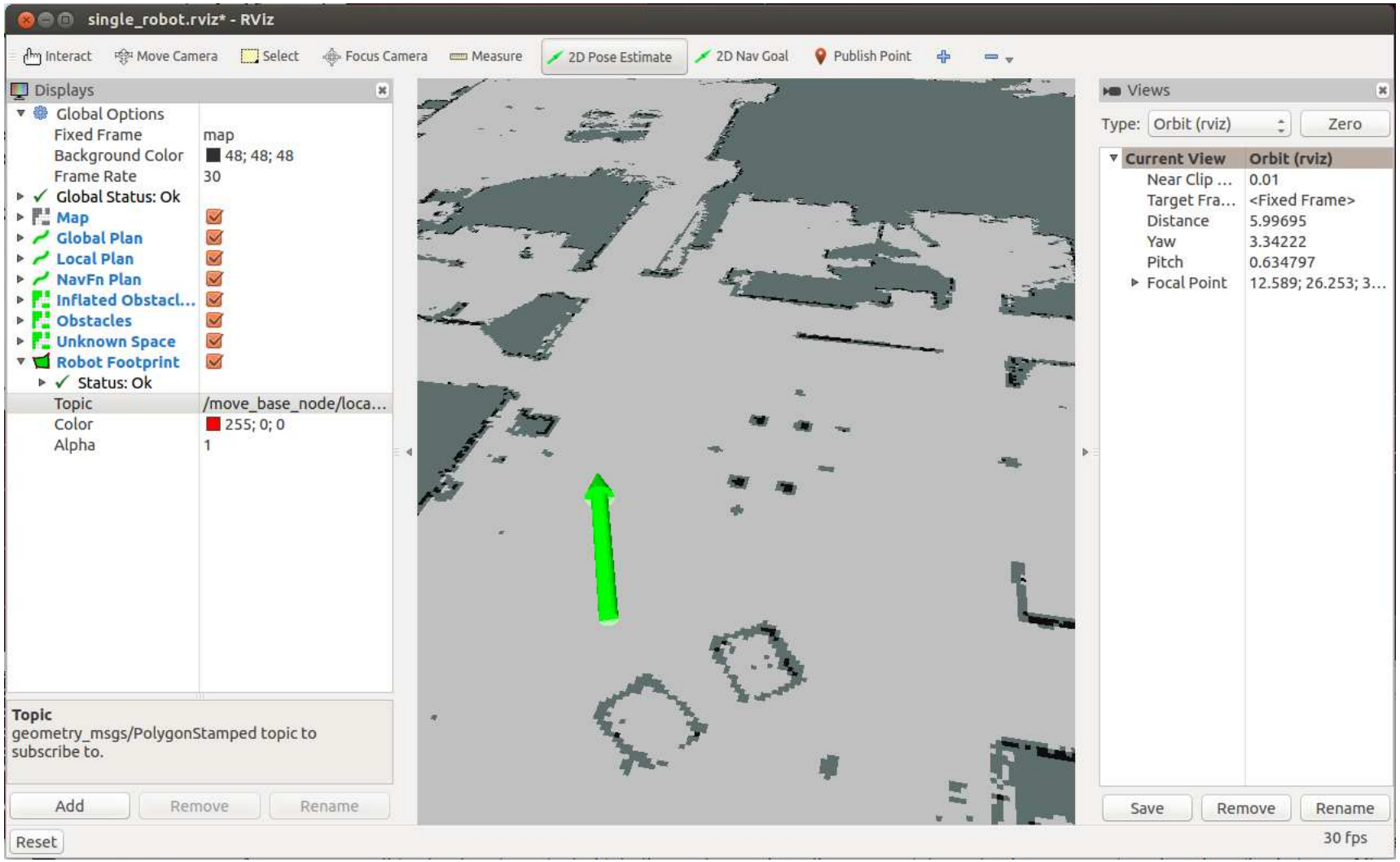
(C)2014 Roi Yehoshua

rviz 2D Pose Estimate

- Vamos a indicar de forma interactiva la posición estimada del robot, para facilitar el trabajo a amcl y que no tarde mucho en converger y autolocalizarse
- El paquete de navegación está suscrito a un topic llamado **initialpose publicado por rviz**
- El botón **2D Pose Estimate** permite al usuario inicializar la localización poniendo un valor a la posición del robot
- Hacer click en el botón y después click en el mapa para indicar la posición inicial del robot.
- Si no lo hacemos al principio, el robot tratará de autolocalizarse y durante unos momentos (que puede ser bastante tiempo) no reconocerá su posición real y no podrá navegar de forma eficiente (puede quedarse atascado).
  - Es recomendable hacer esto si queremos resolver problemas de navegación

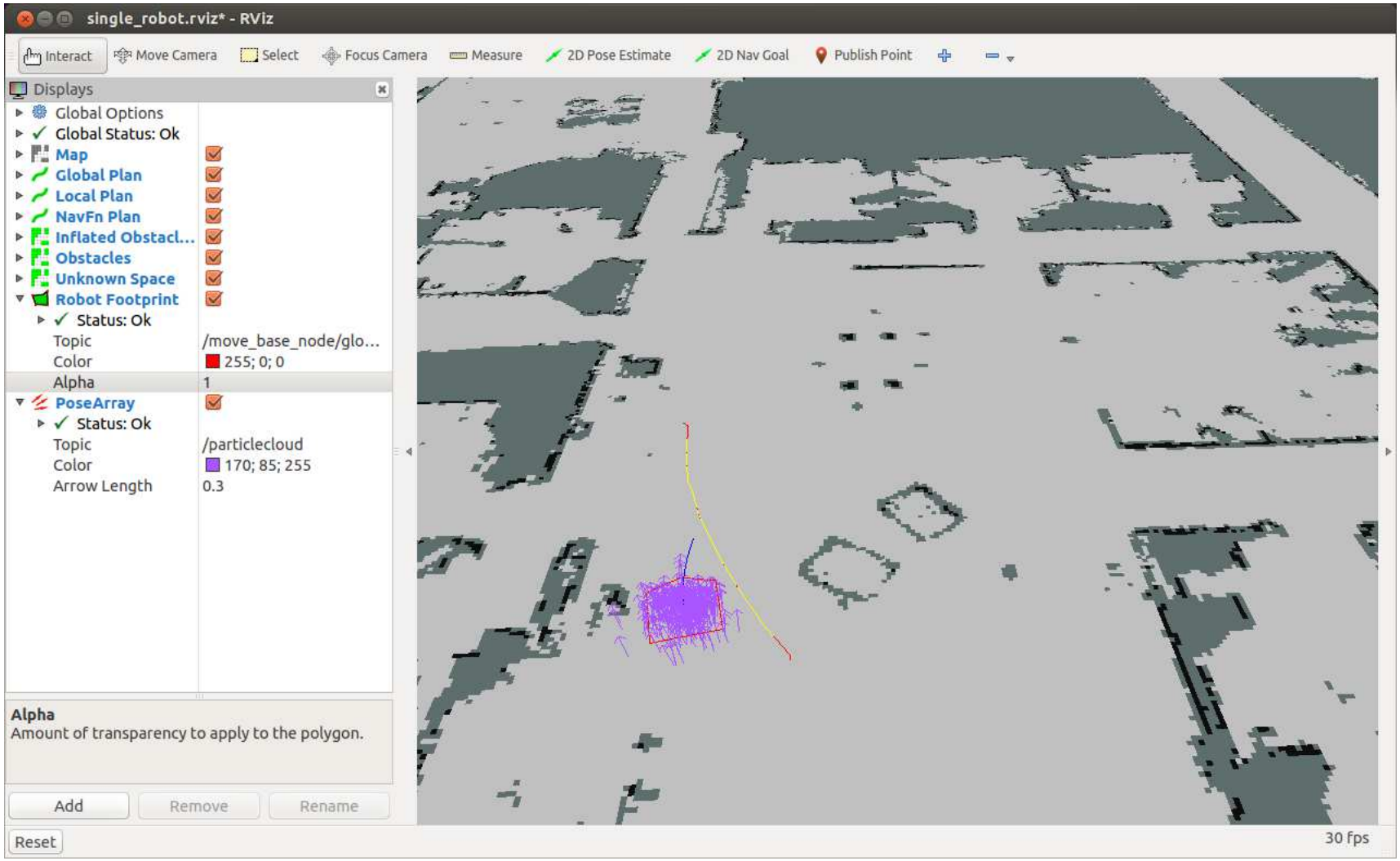
(C)2014 Roi Yehoshua

2D Pose Estimate



(C)2014 Roi Yehoshua

Particle Cloud in rviz

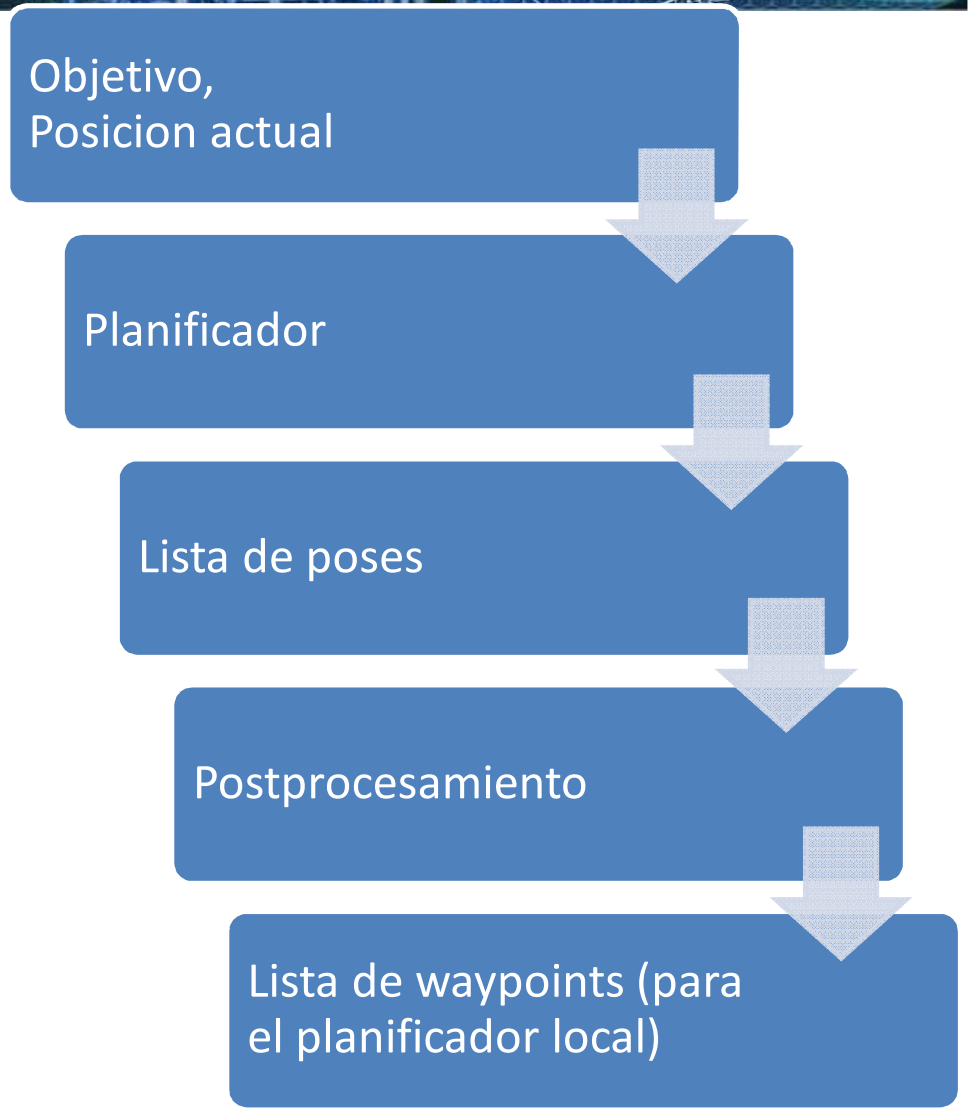


(C)2014 Roi Yehoshua

PLANIFICACIÓN GLOBAL (PLANIFICACIÓN DE CAMINOS)

Planificación de caminos

- Objetivo de la planificación de caminos:
  - Determinar un camino hacia un objetivo especificado.
  - Camino: secuencia de "poses"
  - Objetivo: una pose.
- Principales características
  - Tratan de encontrar un camino óptimo.
  - Waypoint (hito)
    - La secuencia de configuraciones se "postprocesa" y se obtiene un conjunto de subobjetivos
    - Cada punto guía (subobjetivo) es una pose (x,y, theta).
    - Ubicaciones donde el robot podría cambiar su orientación



Planificación de caminos (2)

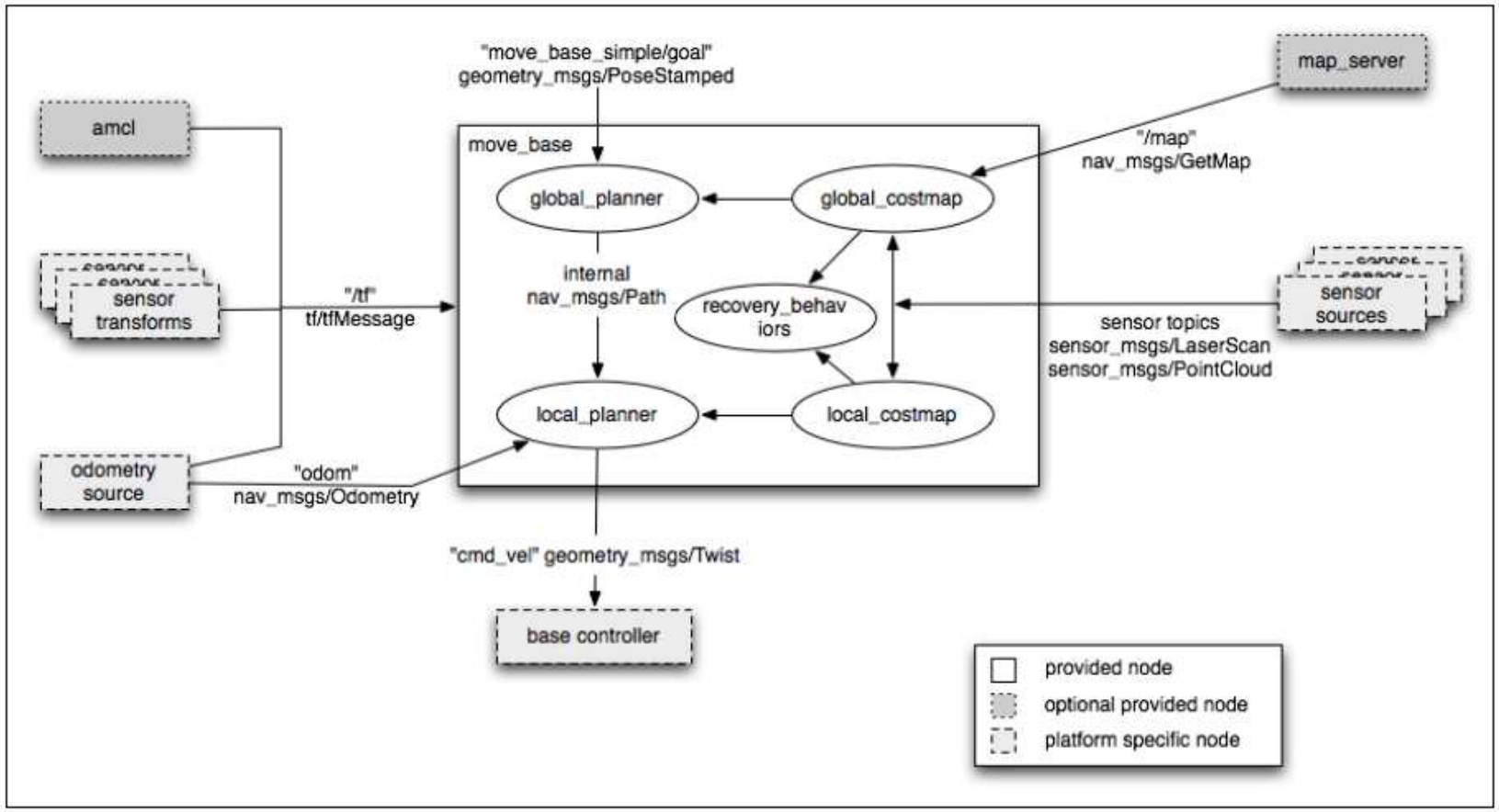
- Componentes de un planificador de caminos global
  - La representación
    - Normalmente cuadrículas regulares. Hemos visto como se puede representar un mapa del entorno en ROS como una Occupancy Grid.
    - Para Navegación no es suficiente la representación [libre, ocupada, desconocida]. Necesitamos información que nos diga si una posición del mapa está lejos, cerca o muy cerca de un obstáculo, para poder hacer navegación segura.
    - En ROS se utiliza el concepto de **costmap** para representar el mundo discretizado.
    - http://wiki.ros.org/costmap_2d
  - El algoritmo
    - Problema de búsqueda en grafos (por ejemplo A*)
    - El proceso de búsqueda hace uso directo del mapa como un costmap.
  - Un problema que siempre aparece
    - ¿Cuándo uso el planificador y cuando me muevo?
    - Entrelazado de planificación y ejecución, este problema está resuelto en ROS con el paquete move_base http://wiki.ros.org/move_base dentro de navigation stack.

ROS Navigation Stack

- http://wiki.ros.org/navigation
  - Un stack de paquetes ROS que
    - a partir de información sobre odometría, sensores y una pose objetivo,
    - devuelve comandos de velocidad enviados a una base móvil de robot
  - Diseñada para mover cualquier robot móvil sin que se quede perdido ni choque.
  - Incorpora soluciones para
    - Navegación Global: encontrar un camino entre dos puntos alejados del mapa, no se ajusta a requisitos de tiempo real.
    - Navegación Local con mapa: encontrar una trayectoria, en tiempo real, a un punto en un entorno próximo del robot.
  - **Instalar Navigation Stack (debería estar instalada):**
    - **sudo apt-get install ros-indigo-navigation**
- ROS Navigation Introductory Video

(C)2013 Roi Yehoshua

Navigation Stack



(C)2013 Roi Yehoshua

Navigation Stack Requirements

- Tres requisitos fundamentales:
  - *Navigation stack* solo maneja robots con ruedas con conducción diferencial y holonómicos.
    - Puede hacer algo más con robots bípedos, como localización, pro siempre que el robot no se mueva de lado
  - La base tiene que tener un laser montado para poder crear mapas y localizarse
    - O bien otro sensores equivalentes a los scans de un laser(como sonars o Kinect por ejemplo)
  - Funciona mejor con robots con forma aproximada cuadrada o circular.

(C)2013 Roi Yehoshua

PAQUETE MOVE_BASE

move_base

- Move_base implementa una arquitectura híbrida de tres niveles.
- Arquitecturas híbridas: combinan técnicas reactivas y deliberativas.
- Arquitectura de 3 niveles:
  - Nivel Reactivo: control a bajo nivel del robot. Bucle sensor-acción muy acoplado. Ciclo decisión en ms.
  - Nivel Ejecutivo: establece la unión entre el nivel deliberativo y el reactivo. Acepta directivas del deliberativo. Cico de decisión en 1/10 sgs.
  - Nivel Deliberativo: genera soluciones globales a tareas complejas mediante planificación (de movimientos o the tareas más abstractas). Ciclo de decisión pueden ser minutos.

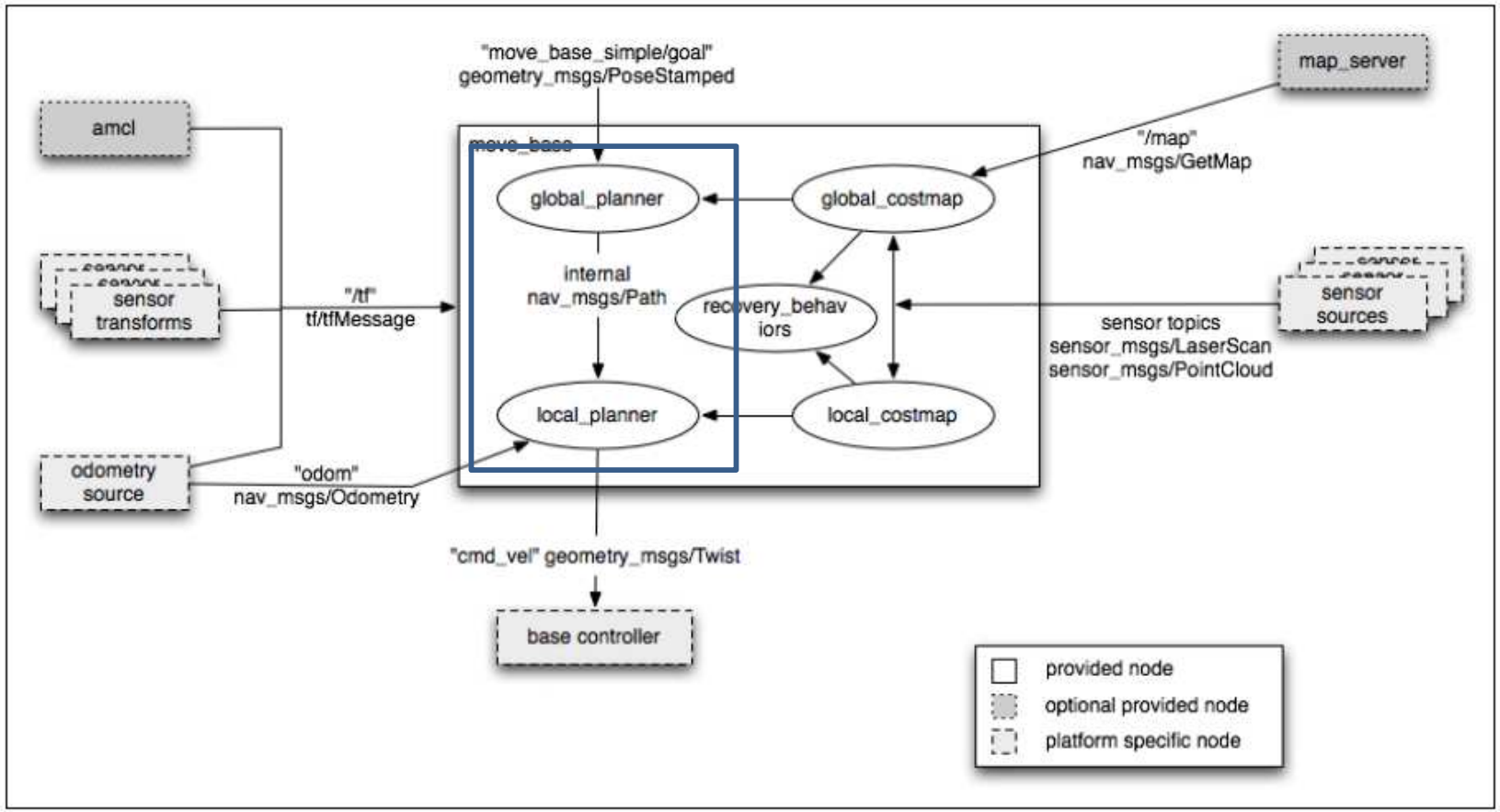
(C)2013 Roi Yehoshua

move_base

- Este paquete permite mover el robot a una posición deseada usando ***navigation_stack***
- El *nodo move_base* aúna un *global_planner (Nivel Deliberativo)* y un *local_planner (Nivel Ejecutivo)* para desempeñar la tarea de ***navegación global.***
  - ***El nivel reactivo está en general implementado como sistema de control en el propio robot.***
- El *nodo move_base* puede realizar opcionalmente *recovery behaviours* cuando el robot percibe que está atascado.
- Basta con ejecutar un fichero *launch* porque el paquete ***move_base*** viene con la instalación de ROS.

(C)2013 Roi Yehoshua

move_base package



(C)2013 Roi Yehoshua

Ejecutar ROS navigation_stack con *Stage*

- Descargar el paquete mi_mapeo_stage en Prado (si no lo has descargado antes)
  - en el directorio move_base_config hay varios ficheros de configuración para lanzar apropiadamente el conjunto de nodos que incluye el Stack move_base

(C)2013 Roi Yehoshua

Ver en src/mi_mapeo_stage/move_base_config/move_base.xml

```
<launch>
<!--
  Example move_base configuration. Descriptions of parameters, as well as a full list of all amcl parameters, can be found
at http://www.ros.org/wiki/move_base.
-->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base_node" output="screen">
    <param name="footprint_padding" value="0.01" />
    <param name="controller_frequency" value="10.0" />
    <param name="controller_patience" value="3.0" />

    <param name="oscillation_timeout" value="30.0" />
    <param name="oscillation_distance" value="0.5" />

    <!--
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
    -->

    <rosparam file="$(find mi_mapeo_stage)/move_base_config/costmap_common_params.yaml" command="load" ns="global_costmap"
/>
    <rosparam file="$(find mi_mapeo_stage)/move_base_config/costmap_common_params.yaml" command="load" ns="local_costmap"
/>
    <rosparam file="$(find mi_mapeo_stage)/move_base_config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find mi_mapeo_stage)/move_base_config/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find mi_mapeo_stage)/move_base_config/base_local_planner_params.yaml" command="load" />
    <!--
    <rosparam file="$(find mi_mapeo_stage)/move_base_config/dwa_local_planner_params.yaml" command="load" />
    -->
  </node>
</launch>
```

(C)2013 Roi Yehoshua

Move_base.xml

```
<launch>
<!--
  Example move_base configuration. Descriptions of parameters, as well as a full list of all amcl parameters, can be found at
  http://www.ros.org/wiki/move_base.
-->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base_node" output="screen">
    <param name="footprint_padding" value="0.01" />
    <param name="controller_frequency" value="10.0" />
    <param name="controller_patience" value="10.0" />

    <param name="oscillation_timeout" value="30.0" />
    <param name="oscillation_distance" value="0.5" />
    <!--
    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS" />
    -->


    <param name="planner_patience" value="20.0" />
    <param name="base_global_planner" value="my_astar_planner/MyAstarPlanner"/>

    <rosparam file="$(find navigation_stage)/move_base_config/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find navigation_stage)/move_base_config/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find navigation_stage)/move_base_config/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find navigation_stage)/move_base_config/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find navigation_stage)/move_base_config/base_local_planner_params.yaml" command="load" />
    <!--
    <rosparam file="$(find navigation_stage)/move_base_config/dwa_local_planner_p
    -->
  </node>
</launch>
```

Configuración planificador local
http://wiki.ros.org/base_local_planner

Configuración planificador global
http://wiki.ros.org/move_base

Configuración costmaps

http://wiki.ros.org/costmap_2d

move_base_amcl_5cm.launch

```
<launch>
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>

<!-- Lanzamos move_base para navegacion -->

  <include file="$(find mi_mapeo_stage)/move_base_config/move_base.xml"/>

<!-- Lanzamos map_server con un mapa -->
  <node name="map_server" pkg="map_server" type="map_server"
        args="$(find mi_mapeo_stage)/stage_config/maps/willow-full-0.05.pgm 0.05"
respawn="false" />

<!-- Lanzamos stage con el mundo correspondiente al mapa -->
  <node pkg="stage_ros" type="stageros" name="stageros"
        args="$(find mi_mapeo_stage)/stage_config/worlds/willow-pr2-5cm.world"
respawn="false" >
    <param name="base_watchdog_timeout" value="0.2"/>
  </node>

<!-- Lanzamos el nodo amcl -->
  <include file="$(find mi_mapeo_stage)/move_base_config/amcl_node.xml"/>

<!-- Lanzamos rviz -->
 <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mi_mapeo_stage)/single_robot.rviz"
/>
</launch>
```

(C)2013 Roi Yehoshua

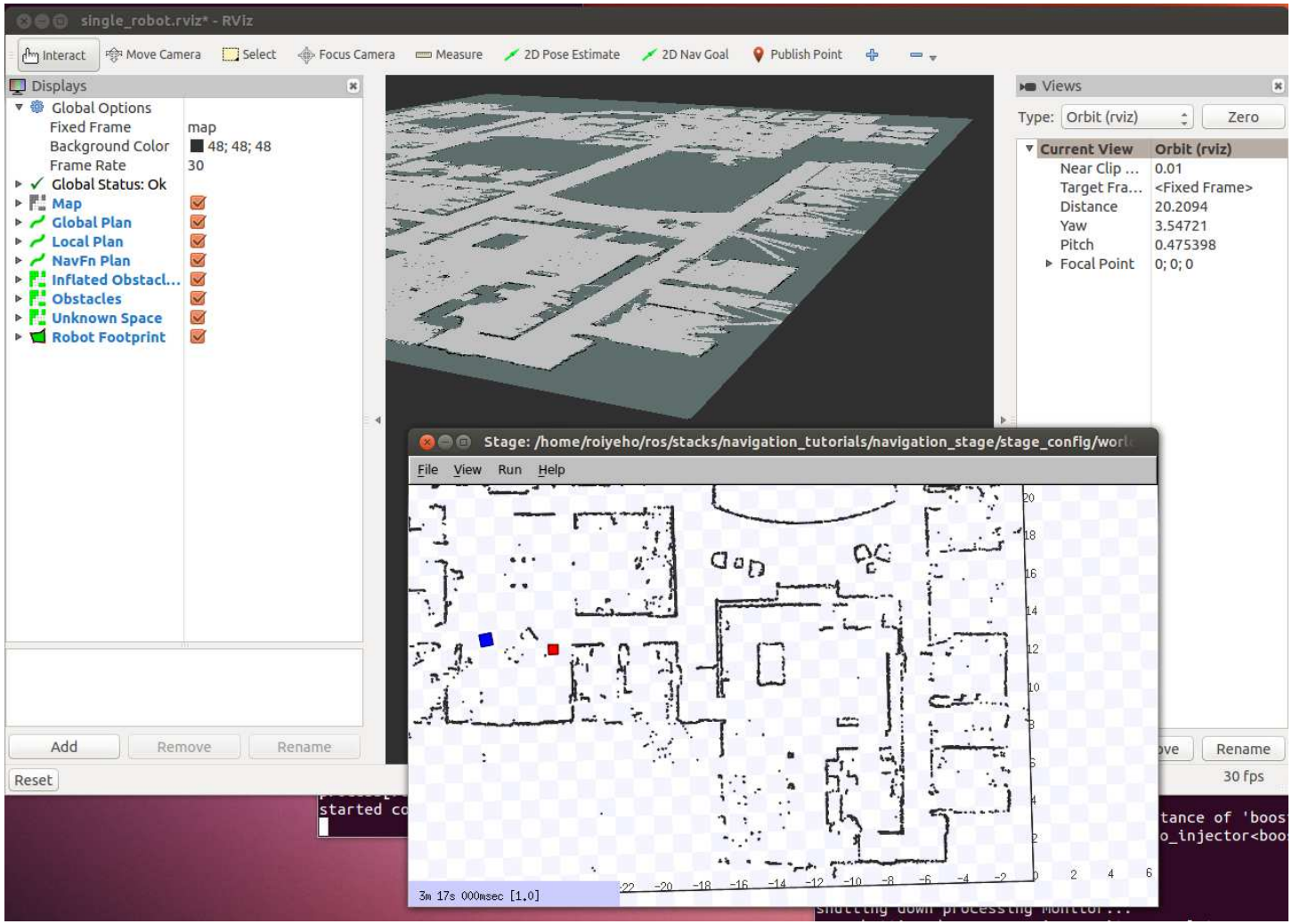
move_base_amcl_5cm.launch

- Ejecutar:
  - comprobar antes que mi_mapeo_stage está reconocido como un paquete ROS

```
$ roslaunch mi_mapeo_stage move_base_amcl_5cm.launch
```

(C)2013 Roi Yehoshua

Ejecutando fichero launch



(C)2013 Roi Yehoshua

Rviz con *Navigation Stack*

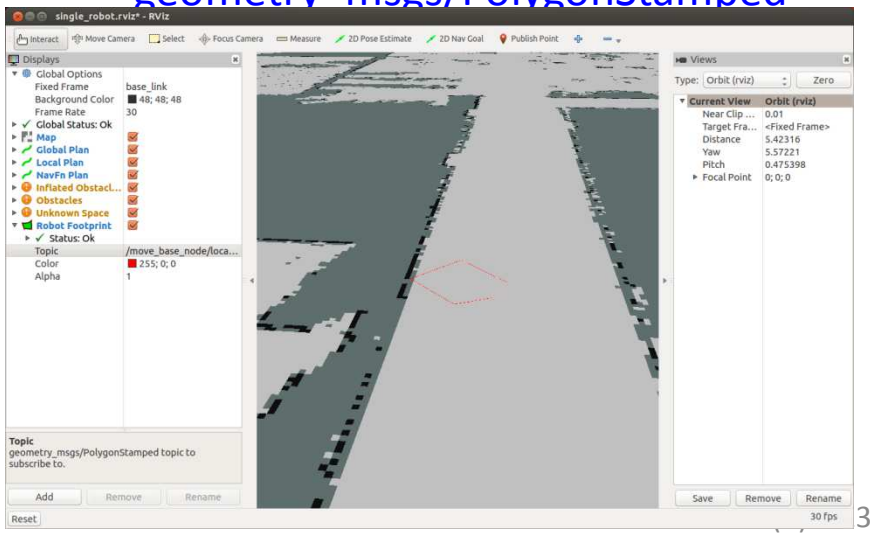
- Configurar *rviz* con *navigation stack*:
  - Asignar la pose del robot para el sistema de localización.
  - Visualizar toda la información que suministra ***navigation stack***.
  - Enviar goals desde rviz.

(C)2013 Roi Yehoshua

Robot Footprint

- Muestra la “huella” **(footprint)** del robot
  - En nuestro caso un pentágono
  - Parámetro configurado en el fichero *costmap_common_params.yaml*
- Topic:
  - move_base_node/local_costmap/footprint_layer/footprint_stamped
- Type:
  - geometry_msgs/PolygonStamped

```
roiyeho@ubuntu: ~
roiyeho@ubuntu:~$ rostopic echo move_base_node/local_costmap/footprint_layer/foo
tprint_stamped -n1
header:
  seq: 2175
  stamp:
    secs: 438
    nsecs: 0
  frame_id: odom
polygon:
  points:
    -
      x: -0.334999978542
      y: -0.334999978542
      z: 0.0
    -
      x: -0.334999978542
      y: 0.334999978542
      z: 0.0
    -
      x: 0.334999978542
      y: 0.334999978542
      z: 0.0
    -
      x: 0.469999998808
      y: 0.0
      z: 0.0
    -
      x: 0.334999978542
      y: -0.334999978542
      z: 0.0
---
roiyeho@ubuntu:~$
```

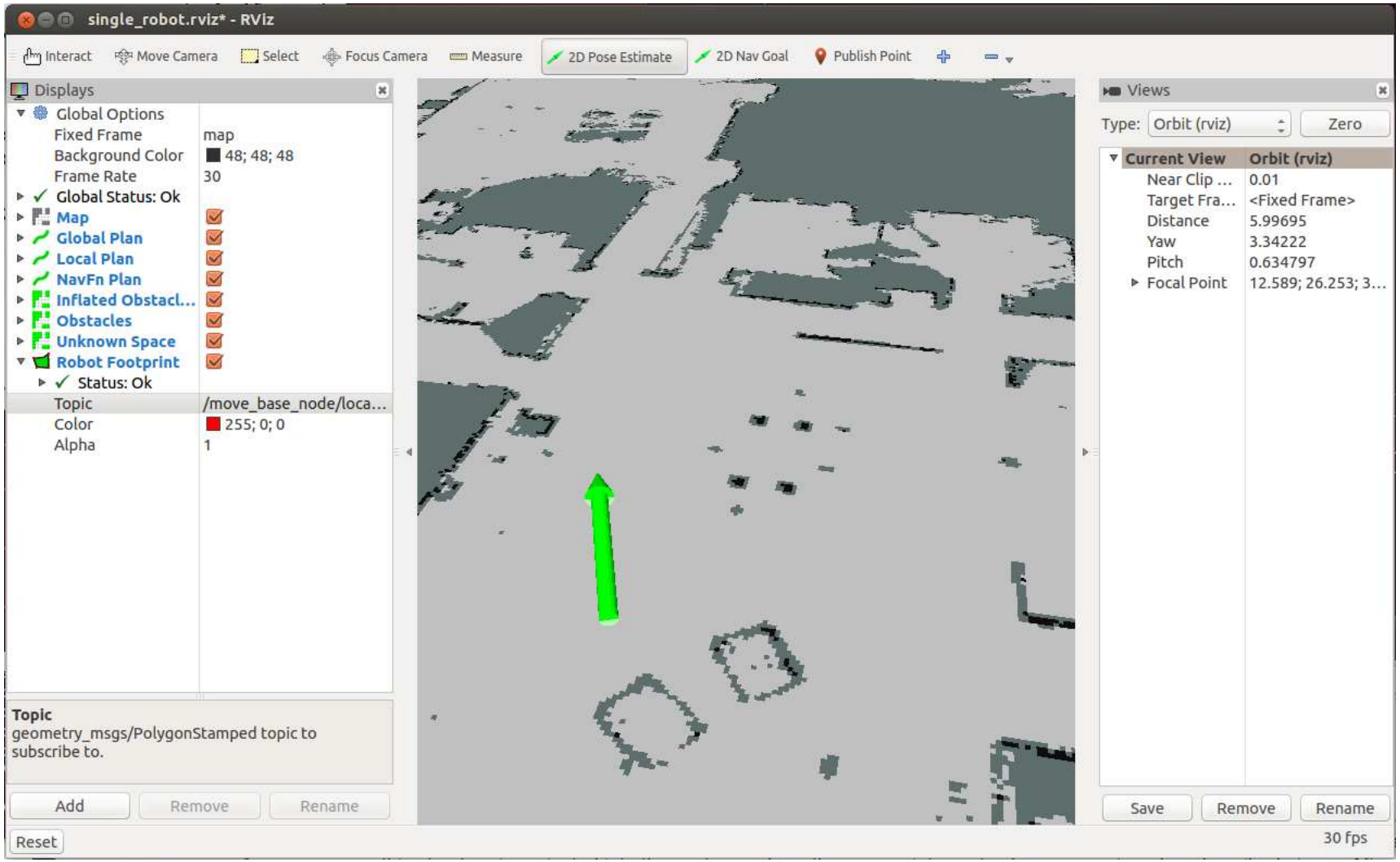


2D Pose Estimate

- La estimación de la pose en 2D (P shortcut) nos permite inicializar el sistema de localización usado por *navigation* stack mediante la asignación de la *pose* del robot.
- *Navigation* stack **espera** a que se le asigne esta nueva pose en un *topic* llamado **initialpose**.
- Para asignar la *pose*
  - Click on the **2D Pose Estimate** button
  - Then click on the map to indicate the initial position of your robot.
  - If you don't do this at the beginning, the robot will start the auto-localization process and try to set an initial pose.
- ***Note: For the "2d Nav Goal" and "2D Pose Estimate" buttons to work, the Fixed Frame must be set to "map".***

(C)2013 Roi Yehoshua

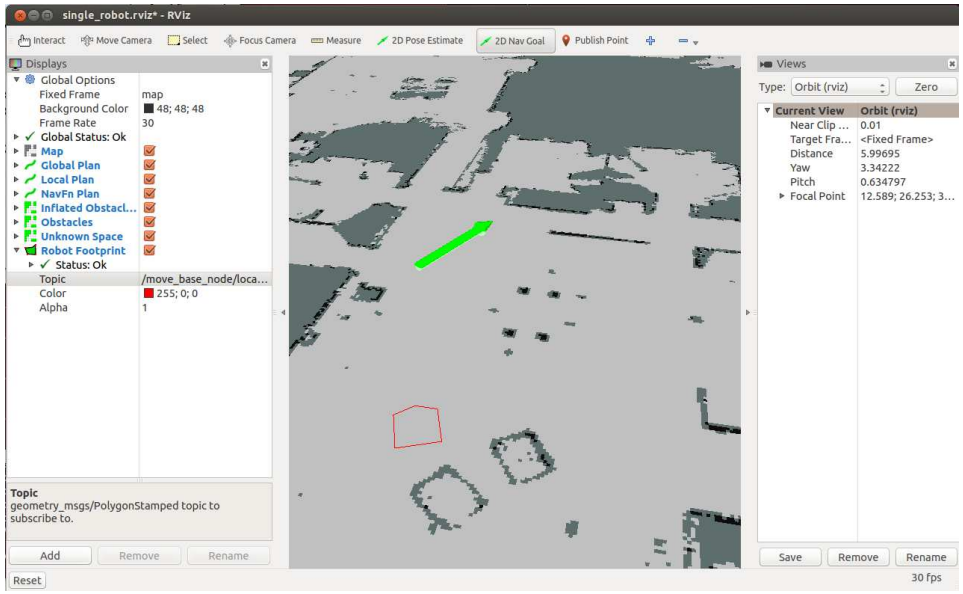
2D Pose Estimate



(C)2013 Roi Yehoshua

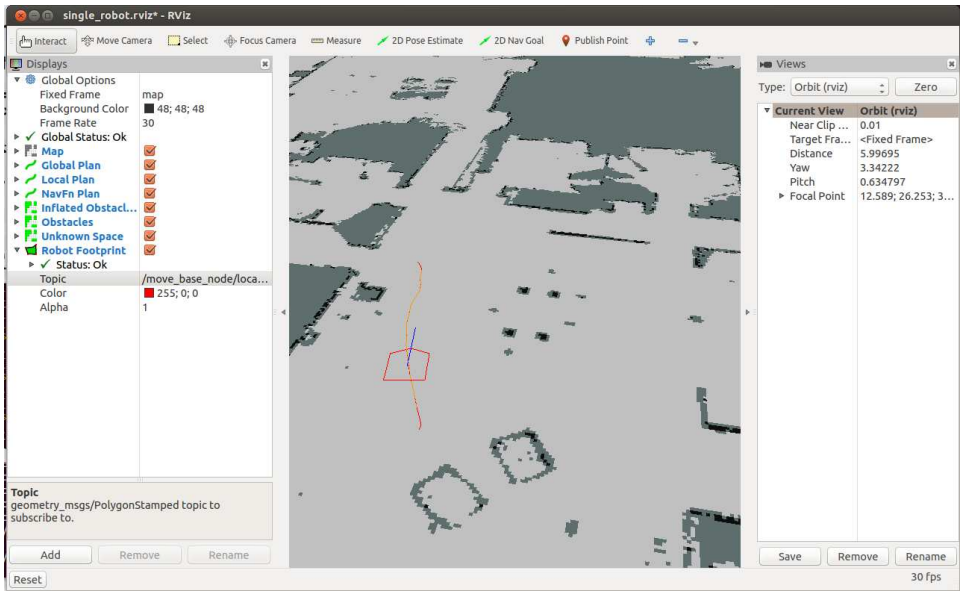
2D Nav Goal

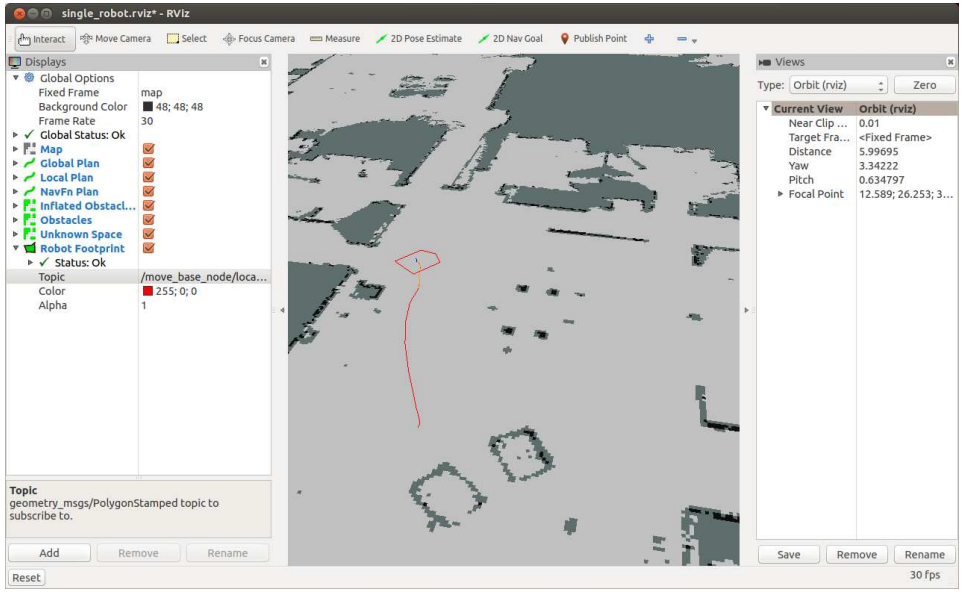
- 2D nav goal (G shortcut) nos permite enviar un goal a *navigation stack.*
- Click on the **2D Nav Goal** button and select the map and the goal for your robot.
- Podemos seleccionar la posición (x,y) y la orientación (theta) del robot deseados.



(C)2013 Roi Yehoshua

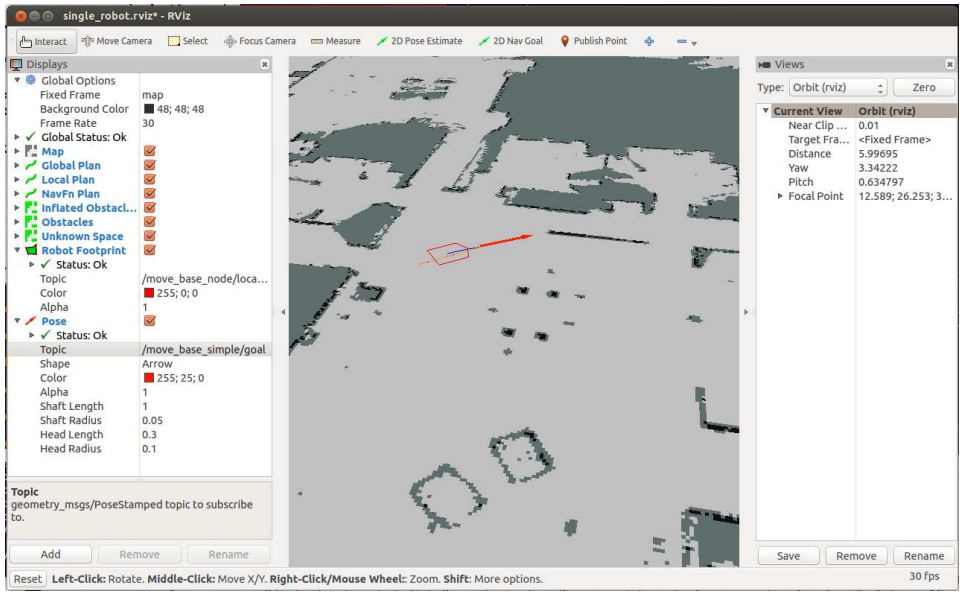
El robot se mueve al destino





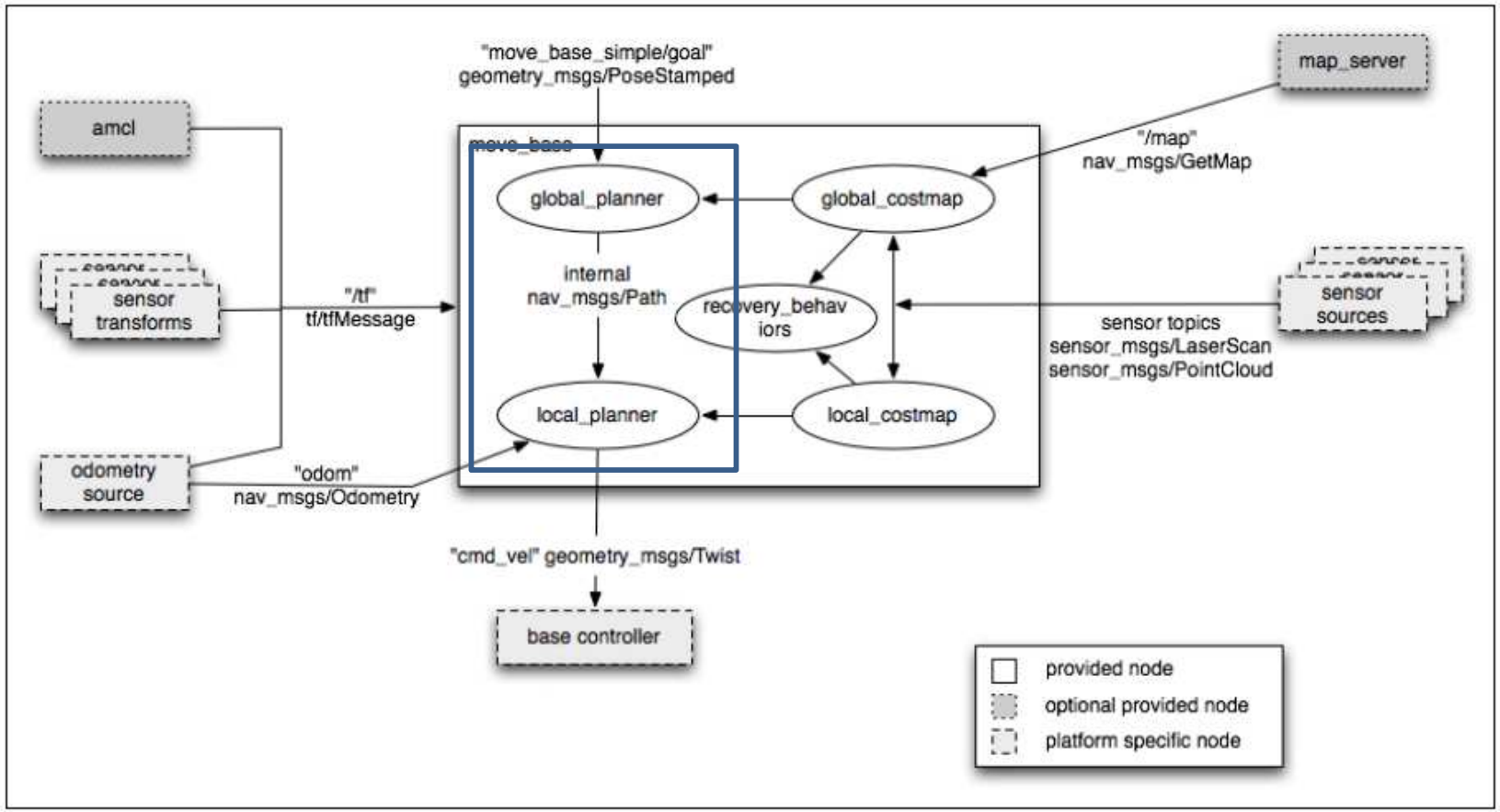
Current Goal

- Para mostrar el goal actual que *navigation stack* trata de alcanzar añadir un ***Pose Display.***
- Poner su topic a **/move_base_simple/goal**
- El goal se visualiza como una flecha roja.
- Puede usarse para conocer la posición final del robot.



(C)2013 Roi Yehoshua

Navigation Stack



(C)2013 Roi Yehoshua

Global and Local Planner

Global planner: *navfn*

http://wiki.ros.org/navfn

- este paquete provee funciones para calcular planes globales (búsqueda en grafos)

- El plan global se calcula antes de que el robot se mueva a la próxima posición.
- Asume un robot circular y opera en un ***costmap global*** para encontrar un camino de coste mínimo en una cuadrícula.
- La función de navegación (por eso ***navfn)*** se calcula según el algoritmo de Dijkstra's
  - aun no tienen soporte para A*.

Local planner: *base_local_planner*

http://wiki.ros.org/base_local_planner

- este paquete provee funciones para calcular trayectorias como planes locales.

- Monitoriza datos de sensores y selecciona las velocidades angular y lineal apropiadas para que el robot atraviese un segmento del plan global.
- Combina datos de odometría con los mapas de coste global y local para seleccionar el camino que debe seguir el robot.
- Puede recalcular el camino sobre la marcha para mantener al robot lejos de zonas de colisión, garantizando alcanzar el destino.

(C)2013 Roi Yehoshua

Global and Local Planner

Global planner: *navfn*

- Hay ficheros de configuración para definir el comportamiento del global planner por defecto (Dijkstra).
- Ver: http://wiki.ros.org/global_planner?distro=indigo
- Es posible implementar global planners diferentes, siguiendo las recomendaciones e interfaces provistas por ROS.

Ver documentación en: http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS

Local Planner: *base_local_planner*

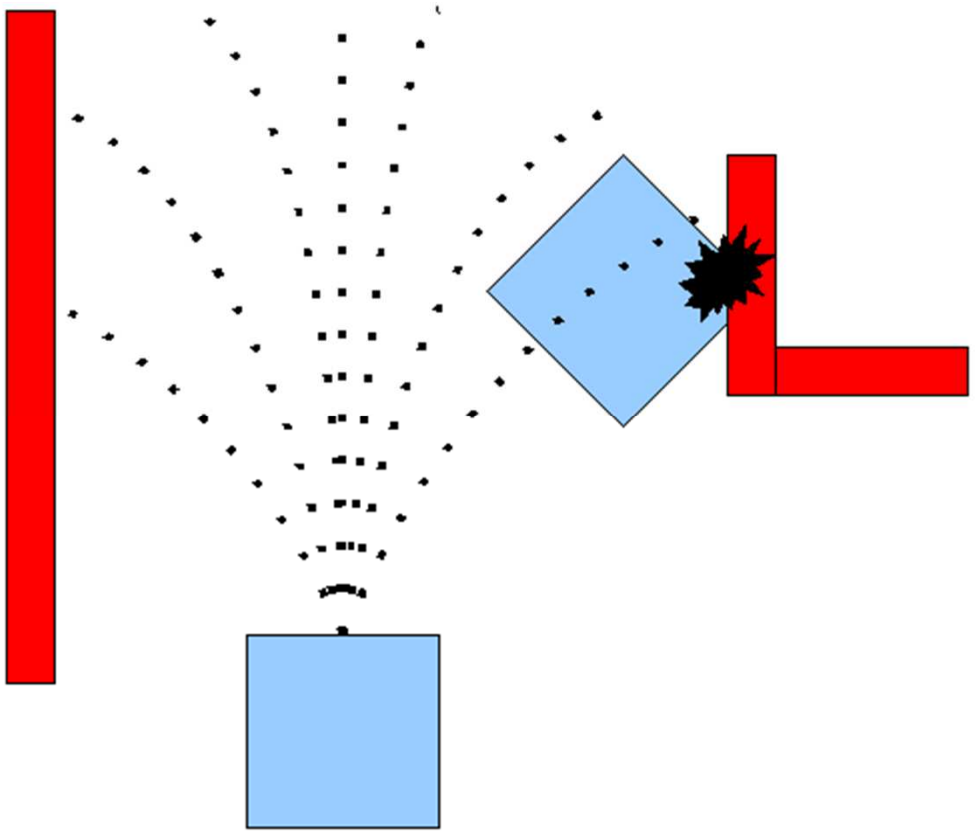
- Implementa dos tipos distintos de técnicas de navegación local:
  - **Trajectory Rollout.** Brian P. Gerkey and Kurt Konolige. "Planning and Control in Unstructured Terrain". Discussion of the Trajectory Rollout algorithm in use on the LAGR robot
  - **Dynamic Window.** D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". The Dynamic Window Approach to local control.
- Puede reimplementarse también. Ver base_local_planner::TrajectoryPlannerROS.

Trajectory Rollout Algorithm

1. Hacer un muestreo inicial de las posibles velocidades del robot (dx,dy,dθ)
2. Por cada velocidad muestreada, obtener una trayectoria.
    1. Haciendo simulación hacia adelante desde la posición actual del robot para predecir qué pasaría si la velocidad muestreada se aplicara un corto período de tiempo
3. Evaluar cada trayectoria, usando metricas como: proximidad a obstáculos, al goal, al camino global y velocidad
4. Descartar trayectorias ilegales (aquellas en las que se colisione con obstáculos)
5. Escoger la trayectoria mejor evaluada y enviar su velocidad asociada a la base móvil
6. Rinse and repeat

(C)2014 Roi Yehoshua

Trajectory Rollout Algorithm



Taken from ROS Wiki

(C)2014 Roi Yehoshua

Local Planner Parameters

- **base_local_planner.yaml** contiene un gran número de parámetros que pueden configurarse para personalizar el comportamiento del planificador local
- These parameters are grouped into several categories:
  - robot configuration
  - goal tolerance
  - forward simulation
  - trajectory scoring
  - oscillation prevention
  - global plan

(C)2014 Roi Yehoshua

base_local_planner.yaml (1)

```
#For full documentation of the parameters in this file, and a list of all the
#parameters available for TrajectoryPlannerROS, please see
#http://www.ros.org/wiki/base_local_planner
TrajectoryPlannerROS:
 #Set the acceleration limits of the robot
 acc_lim_th: 3.2
 acc_lim_x: 2.5
 acc_lim_y: 2.5

 #Set the velocity limits of the robot
 max_vel_x: 0.65
 min_vel_x: 0.1
 max_rotational_vel: 1.0
 min_in_place_rotational_vel: 0.4

 #The velocity the robot will command when trying to escape from a stuck situation
 escape_vel: -0.1

 #For this example, we'll use a holonomic robot
 holonomic_robot: true

 #Since we're using a holonomic robot, we'll set the set of y velocities it will sample
 y_vels: [-0.3, -0.1, 0.1, -0.3]
```

(C)2014 Roi Yehoshua

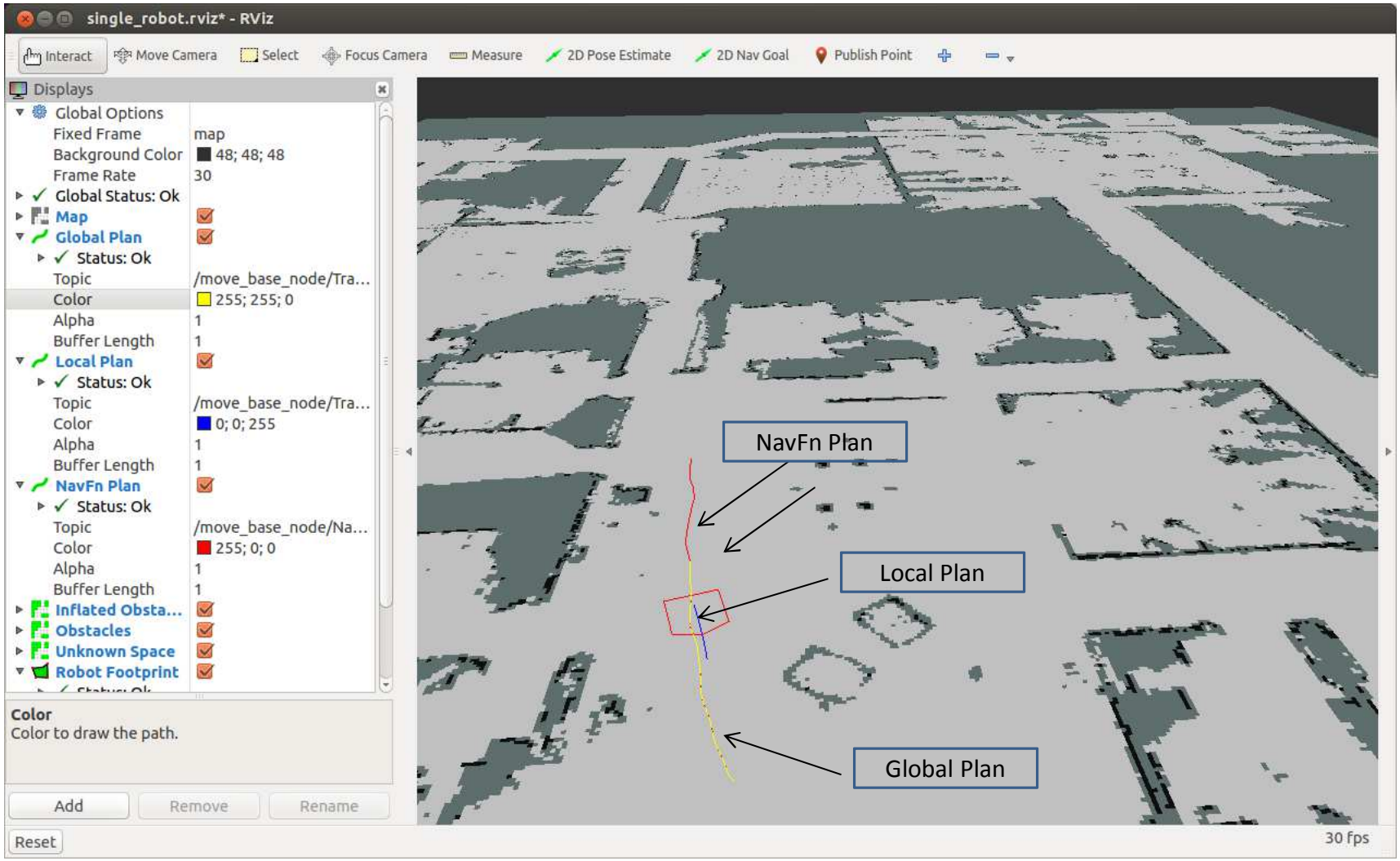

```
#Set the tolerance on achieving a goal
xy_goal_tolerance: 0.1
yaw_goal_tolerance: 0.05

#We'll configure how long and with what granularity we'll forward simulate trajectories
sim_time: 1.7
sim_granularity: 0.025
vx_samples: 3
vtheta_samples: 20

#Parameters for scoring trajectories
goal_distance_bias: 0.8
path_distance_bias: 0.6
occdist_scale: 0.01
heading_lookahead: 0.325

#We'll use the Dynamic Window Approach to control instead of Trajectory Rollout for this example
dwa: true

#How far the robot must travel before oscillation flags are reset
oscillation_reset_dist: 0.05

#Eat up the plan as the robot moves along it
prune_plan: true
```

(C)2014 Roi Yehoshua

- **NavFn Plan**
  - Displays the full plan for the robot computed by the global planner
  - Topic: /move_base_node/NavfnROS/plan
- **Global Plan**
  - It shows the portion of the global plan that the local planner is currently pursuing.
  - Topic: /move_base_node/TrajectoryPlannerROS/global_plan
- **Local Plan**
  - It shows the trajectory associated with the velocity commands currently being commanded to the base by the local planner
  - Topic: /move_base_node/TrajectoryPlannerROS/local_plan

(C)2013 Roi Yehoshua

Navigation Plans in rviz



(C)2013 Roi Yehoshua

Changing Trajectory Scoring

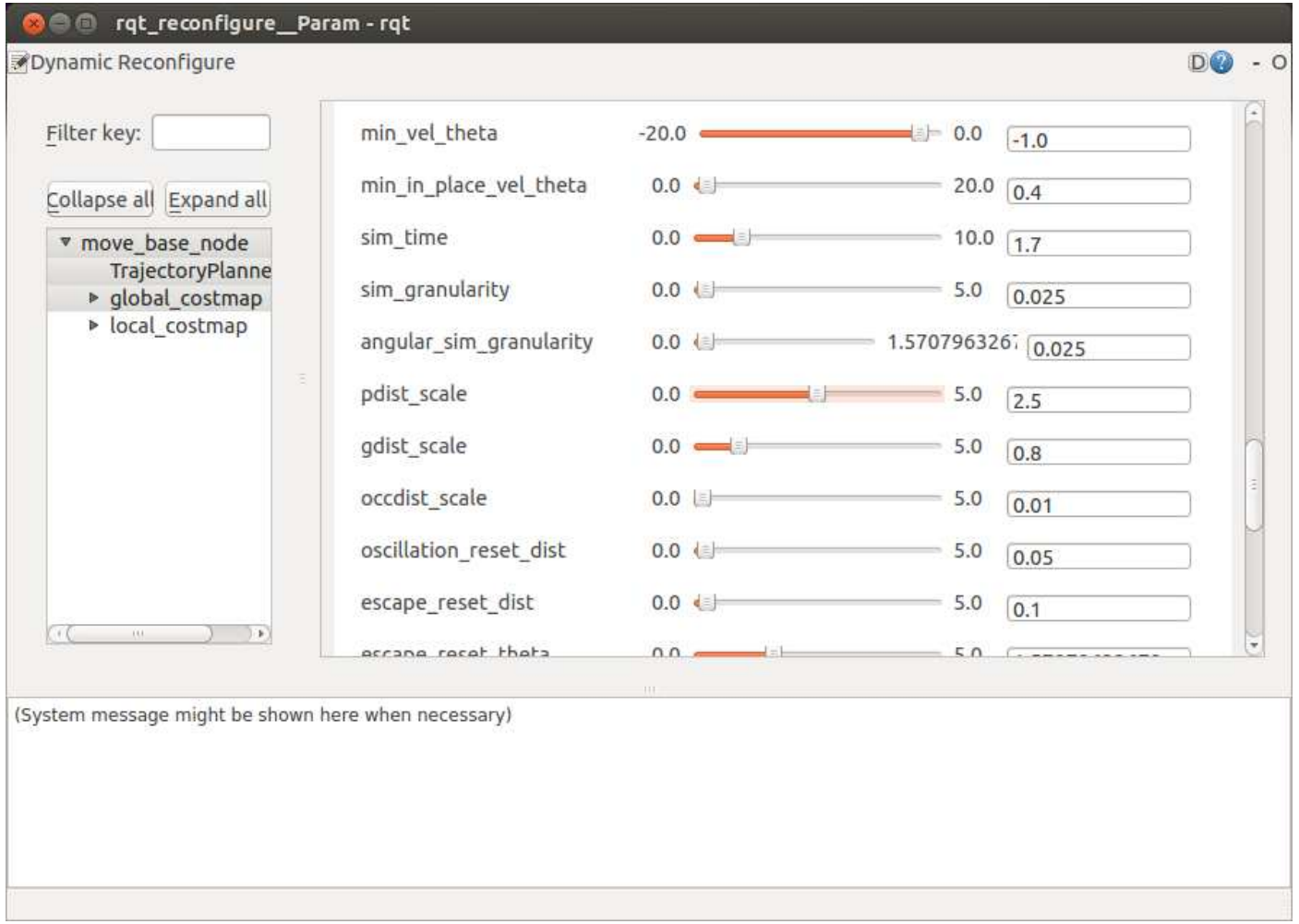
- Run the rqt_reconfigure tool

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

  - This tool allows changing dynamic configuration values
- Open the move_base group
- Select the TrajectoryPlannerROS node
- Then set the pdist_scale parameter to something high like 2.5
- After that, you should see that the local path (blue) now more closely follows the global path (yellow).

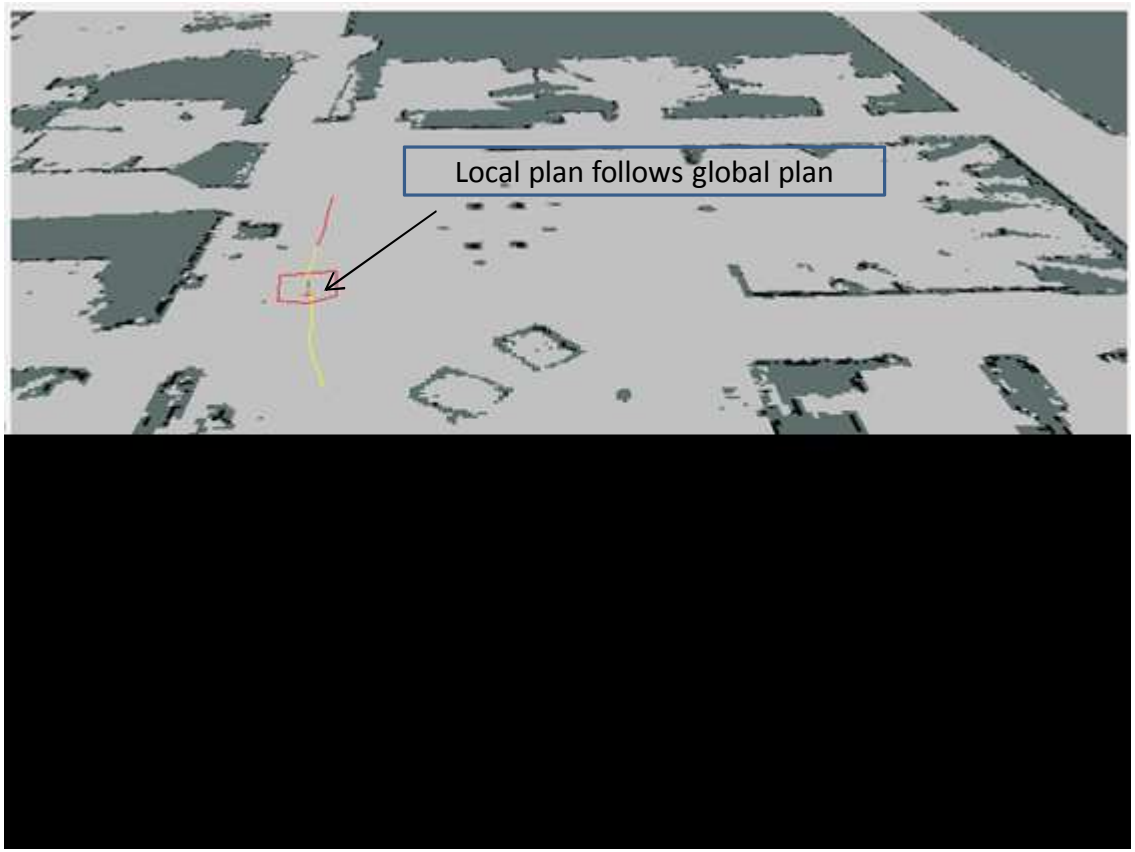
(C)2013 Roi Yehoshua

Changing Trajectory Scoring



(C)2013 Roi Yehoshua

Changing Trajectory Scoring

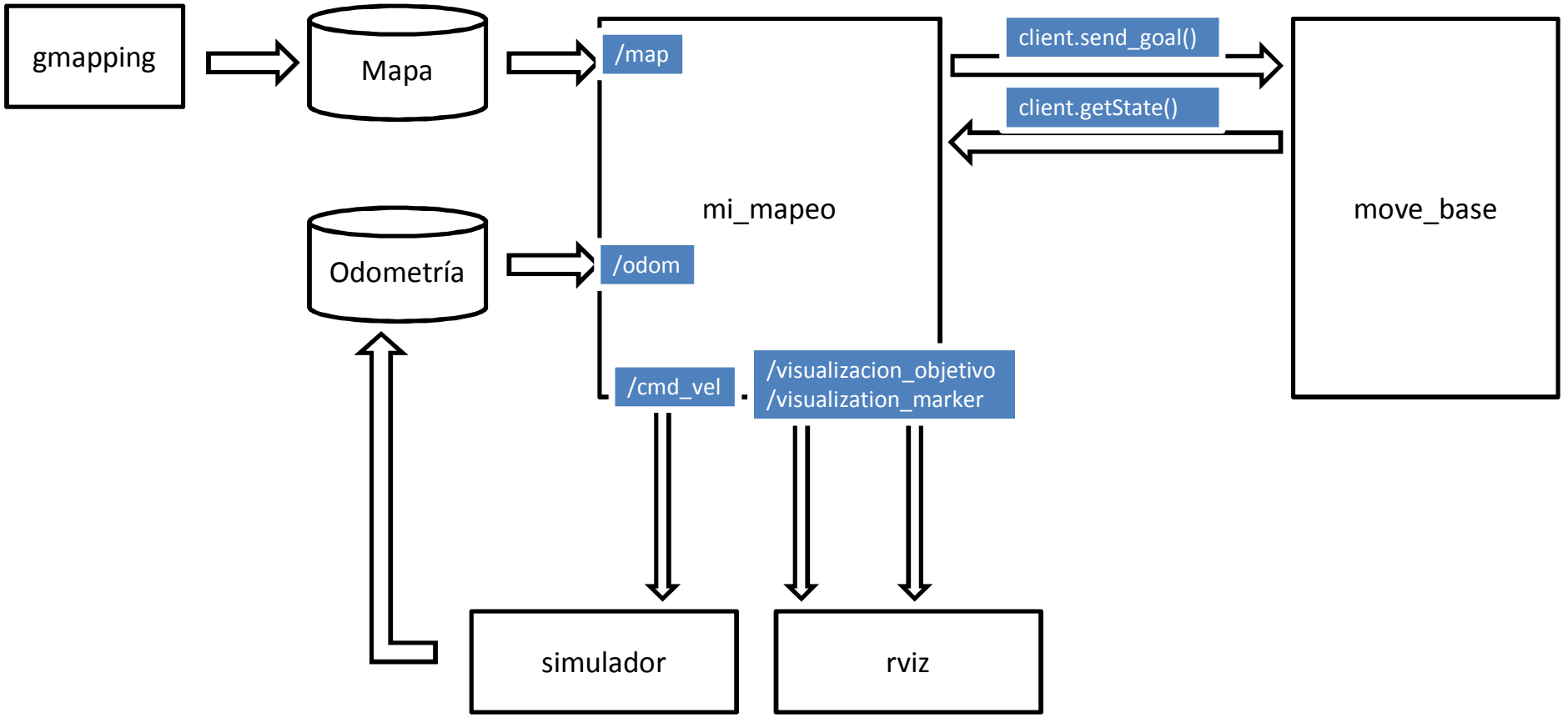


(C)2013 Roi Yehoshua

Objetivo más concreto

- Ya sabemos cómo detectar frontera y cómo seleccionar un punto de la frontera.
- ¿Cómo le decimos al robot que navegue hasta ese punto?
- ROS contiene un paquete que ayuda a hacer parte de estas tareas: **actionlib**
  - http://wiki.ros.org/actionlib

Arquitectura



Paquete *actionlib* de ROS

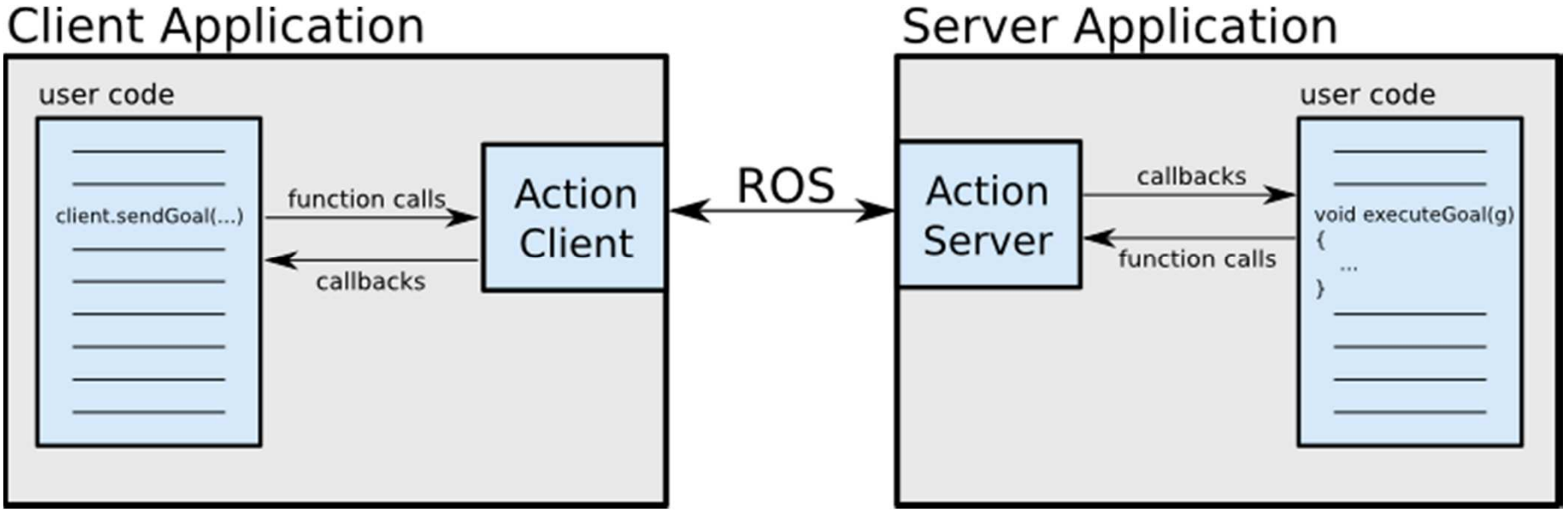
- Objetivo de actionlib.
  - Proveer una interfaz estándar para poder gestionar tareas de largo plazo (no instantáneas)
  - Node A envía una petición a Node B para que éste realice alguna tarea:
    - Moverse hasta un punto objetivo.
    - Hacer un escaneo láser de un objeto
    - Detectar el pomo de una puerta
    - ...

Por qué actionlib

- ROS provee
  - **Services,** más apropiados para tareas instantáneas, aunque requieran comunicación síncrona, como petición de información.
    - Por ejemplo: solicitar al nodo map_saver que nos guarde en un fichero el mapa adquirido con gmapping
  - **Actions,** más apropiadas cuando la tarea solicitada toma más tiempo y además
    - queremos **monitorizar** su progreso
    - obtener **feedback contínuo**
    - poder **cancelar** la petición durante la ejecución.

- es un paquete que provee herramientas para:
  - crear servidores que ejecutan tareas de largo plazo (y que pueden ser aplazadas - preempted).
  - crear clientes que interactúan con los servidores.
- Referencias:
  - http://wiki.ros.org/actionlib
  - http://wiki.ros.org/actionlib/DetailedDescription
  - http://wiki.ros.org/actionlib/Tutorials

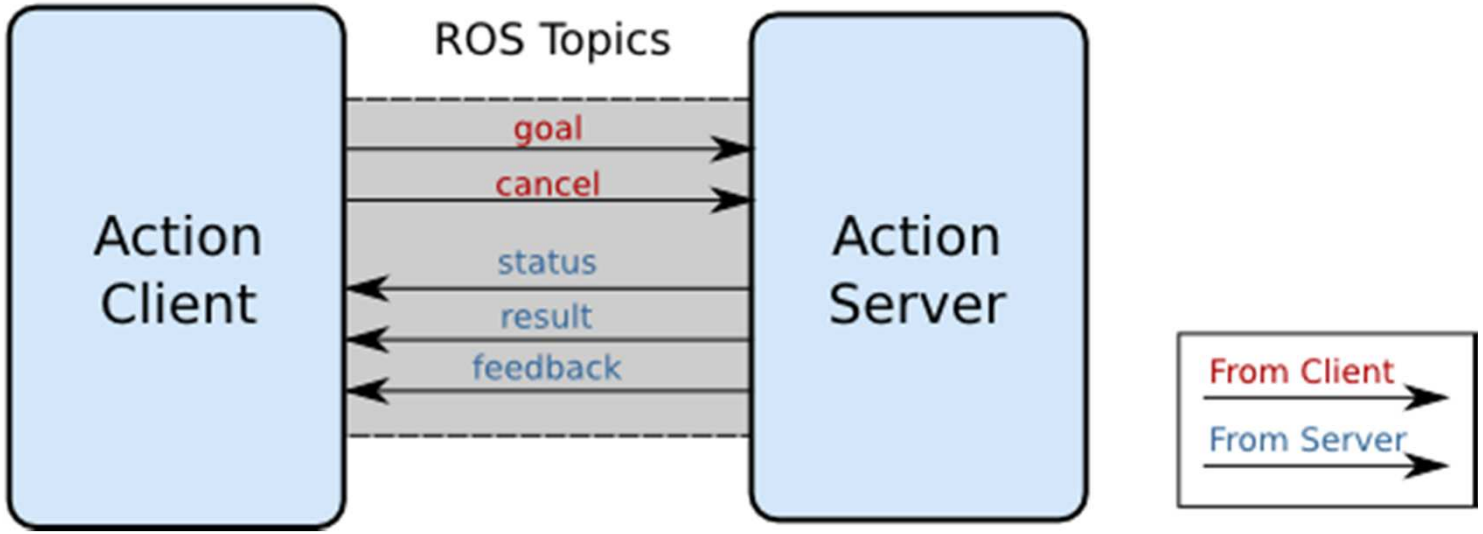
actionlib: Interacción client-server usando ROS Action protocol



- Con la librería actionlib podemos levantar un cliente asociado a nuestro nodo (o un servidor) que opera como una hebra lanzada desde dentro de nuestro nodo.
- En el nodo cliente se programa una petición de acción (por ejemplo: “ir a posición objetivo (x,y)”) como un envío de mensaje (mediante una llamada a función) y en el nodo servidor se programa la recepción de mensajes mediante callbacks.
- En el nodo cliente se reciben mensajes de progreso, enviados por el servidor, y se gestionan mediante callbacks. En el nodo servidor se programa con llamadas a función el envío de mensajes de progreso de una acción.
- El nodo cliente tiene asociado un tipo de objeto llamado “Action Client” que es el que se encarga de gestionar la comunicación a bajo nivel (un programador sólo se centra en determinar cuándo hacer el envío de mensaje)
- El nodo servidor tiene asociado un tipo de objeto llamado “Action Server” que se encarga de comunicación a bajo nivel y el programador se centra en implementar las callbacks functions asociadas a cada tipo de mensaje que se pueda recibir.

actionlib: ROS Action protocol

Action Interface



- El *action client* y el *action server* se comunican enviando mensajes mediante unos *topics* concretos (dentro de un *namespace* definido por los tipos de mensajes).
- El action client publica mensajes en los topics “goal” y “cancel”.
- El action server publica mensajes en los topics “status”, “result” y “feedback”
- Esta comunicación es absolutamente transparente para el programador (pero tenemos que saber que se está realizando así).

actionlib: Interacción Client-Server

- **¿Qué podemos hacer con actionlib?**
- **ROS Topics**
  - **PUBLICADOS POR EL CLIENTE** (de forma transparente a nuestro programa)
  - **goal** - Used to **send new goals to server**
  - **cancel** - Used to send **cancel requests** to server

  - **PUBLICADOS POR EL SERVIDOR** (también de forma transparente a nuestro programa)
  - **status** - Used to **notify clients on the current state** of every goal in the system.
  - **feedback** - Used to **send clients periodic auxiliary information** for a goal
  - **result** - Used to **send clients one-time auxiliary information upon completion** of a goal

actionlib: estructura de los mensajes

- Los mensajes usados para comunicarse entre cliente/servidor pueden autogenerarse, a partir de ficheros de especificación de acciones, dependiendo del tipo de aplicación.
- La especificación de una acción se hace en ficheros con extensión **.action**
- Vamos a utilizar ficheros y mensajes **que ya están generados** en el paquete move_base_msgs

actionlib:estructura de mensajes

- **Fichero MoveBase.action**: fichero de texto para la especificación de la estructura de los mensajes enviados/recibidos para comunicarse con move_base

```
geometry_msgs/PoseStamped target_pose
---
Aquí podría venir otro tipo de mensaje pero se ha decidido dejarlo vacío
---
geometry_msgs/PoseStamped base_position
```

Tipo del mensaje para enviar un goal

Tipo del mensaje para recibir feedback

- Aquí se especifica lo siguiente: hay tres tipos de mensajes
  - Mensaje usado para enviar/recibir un goal: tiene un campo llamado "target_pose" que es de tipo *geometry_msgs/PoseStamped*
  - No se contemplan mensajes para enviar/recibir resultados (el espacio destinado a definirlo se deja vacío)
  - Mensaje usado para enviar/recibir feedback: tiene un campo llamado "base_position" que es de tipo *geometry_msgs/PoseStamped*

actionlib: estructura de mensajes

- Desde el fichero MoveBase.action se generan:
  - MoveBaseAction.msg
  - MovebaseActionGoal.msg
  - MoveBaseActionResult.msg
  - MoveBaseActionFeedback.msg
  - MoveBaseGoal.msg
  - MoveBaseResult.msg (vacío para MoveBase)
  - MoveBaseFeedback.msg
- Estos mensajes son usados internament por **actionlib** para que *ActionClient* y *ActionServer* se comuniquen.

Implementación de un *Action Client*

- Implementación de un cliente simple que soporta sólo un *goal* a la vez.
  - Tutorial de ROS para crear un *action client* simple.
  - Tutorial de RiverLab, más detallado, para crear un *action client.*
- La implementación de un *action client* depende de los tipos de *action messages* usados.
- Nosotros implementaremos un *action client* basado en los mensajes ***move_base_msgs/**** que hemos visto antes.

Implementación *Action Client*

- El siguiente código es un ejemplo de un nodo que implemente un *action client* para enviar un *goal* para que se mueva un robot.
- En este caso el *goal* incluye un mensaje de tipo *PoseStamped* que contiene información sobre dónde debería moverse el robot en el mundo.
- Documentación en
- http://docs.ros.org/api/geometry_msgs/html/msg/PoseStamped.html

Ejecución ejemplo

- Compilar
  - Descargar el fichero mi_send_goals.zip desde PRADO
  - Descomprimirlo en <workspace>/src
  - Hacer catkin_make en <workspace>
  - Hacer source devel/setup.sh
  - Comprobar que se reconoce el paquete con rospack find mi_send_goals

Ejecución ejemplo

- Ejecutar
  - roslaunch mi_send_goals mi_send_goals.launch
- Observar comportamiento en stage. Debe desplazarse automáticamente hasta la posición proporcionada en el fichero mi_send_goals.launch.

Ejecución ejemplo (mi_send_goals.launch)

```
<launch>
  <param name="goal_x" value="18.75" />
  <param name="goal_y" value="29.64" />
  <param name="goal_theta" value="180" />
  <master auto="start"/>
  <param name="/use_sim_time" value="true"/>

<!-- Lanzamos move_base para navegacion -->
   <include file="$(find mi_mapeo_stage)/move_base_config/move_base.xml"/>

<!-- Lanzamos map_server con un mapa -->
  <node name="map_server" pkg="map_server" type="map_server"
        args="$(find mi_mapeo_stage)/stage_config/maps/willow-full-0.05.pgm 0.05" respawn="false" />

<!-- Lanzamos stage con el mundo correspondiente al mapa -->
  <node pkg="stage_ros" type="stageros" name="stageros"
        args="$(find mi_mapeo_stage)/stage_config/worlds/willow-pr2-5cm.world" respawn="false" >
    <param name="base_watchdog_timeout" value="0.2"/>
  </node>
```

Representamos como parámetros del launch la posición y orientación objetivo (en grados). Luego son capturados por el nodo.

Ejecución ejemplo (mi_send_goals.launch)

```
<!-- LOCALIZACION: Lanzamos el nodo amcl -->
  <arg name="initial_pose_x" default="13.56"/>
  <arg name="initial_pose_y" default="28.61"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find mi_mapeo_stage)/move_base_config/amcl_node.xml">
     <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
     <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
     <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
   </include>

<!-- Lanzamos rviz -->
 <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mi_mapeo_stage)/single_robot.rviz" />

 <!-- Launch send goals node -->
  <node name="mi_send_goals_node" pkg="mi_send_goals" type="mi_send_goals_node" output="screen"/>
</launch>
```

Indicamos a amcl cual es la posición inicial del robot.
Atención: por motivos de creación de la imagen del mapa en Stage es necesario realizar esta transformación (manualmente).

rviz.x = stage.y
rviz.y = -stage.x

Implementación *Action Client:* *sendGoals.cpp*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_datatypes.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_node");

    // Get the goal's x, y and angle from the launch file
    double x, y, theta;
    ros::NodeHandle nh;
    nh.getParam("goal_x", x);
    nh.getParam("goal_y", y);
    nh.getParam("goal_theta", theta);

    // create the action client
    MoveBaseClient ac("move_base", true);

    // Wait 60 seconds for the action server to become available
    ROS_INFO("Waiting for the move_base action server");
    ac.waitForServer(ros::Duration(60));

    ROS_INFO("Connected to move base server");
```

Necesitamos:
La librería de roscpp
Usar algún tipo de mensaje de MoveBaseAction
Usar la librería actionlib (para el lado del cliente)
Usar la librería de transformaciones entre marcos de referencia (solo saber que existe, no es necesario más)

Implementación *Action Client: sendGoals.cpp*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_datatypes.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_node");

    // Get the goal's x, y and angle from the launch file
    double x, y, theta;
    ros::NodeHandle nh;
    nh.getParam("goal_x", x);
    nh.getParam("goal_y", y);
    nh.getParam("goal_theta", theta);

    // create the action client
    MoveBaseClient ac("move_base", true);

    // Wait 60 seconds for the action server to become available
    ROS_INFO("Waiting for the move_base action server");
    ac.waitForServer(ros::Duration(60));

    ROS_INFO("Connected to move base server");
```

Definimos el tipo ***MoveBaseClient.*** Luego lo usaremos para crear un objeto *action client.*
Observar que aquí está la clave en la dependencia de los mensajes usado.

Implementación *Action Client:* *sendGoals.cpp*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_datatypes.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_node");

    // Get the goal's x, y and angle from the launch file
    double x, y, theta;
    ros::NodeHandle nh;
    nh.getParam("goal_x", x);
    nh.getParam("goal_y", y);
    nh.getParam("goal_theta", thet

    // create the action client
    MoveBaseClient ac("move_base",

    // Wait 60 seconds for the action server to become available
    ROS_INFO("Waiting for the move_base action server");
    ac.waitForServer(ros::Duration(60));

    ROS_INFO("Connected to move base server");
```

Iniciamos el nodo. El nombre del nodo tiene que ser único, pero no afecta al *action client* que es una hebra dependiente, pero con su propio nombre , de este nodo.

Implementación *Action Client:* *sendGoals.cpp*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_datatypes.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_node");

    // Get the goal's x, y and angle from the l
    double x, y, theta;
    ros::NodeHandle nh;
    nh.getParam("goal_x", x);
    nh.getParam("goal_y", y);
    nh.getParam("goal_theta", theta);

    // create the action client
    MoveBaseClient ac("move_base", true);

    // Wait 60 seconds for the action server to become available
    ROS_INFO("Waiting for the move_base action server");
    ac.waitForServer(ros::Duration(60));

    ROS_INFO("Connected to move base server");
```

Capturamos los parámetros que se nos han enviado desde el launch

Implementación *Action Client:* *sendGoals.cpp*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_datatypes.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_node");

    // Get the goal's x, y and angle from the launch file
    double x, y, theta;
    ros::NodeHandle nh;
    nh.getParam("goal_x", x);
    nh.getParam("goal_y", y);
    nh.getParam("goal_theta", theta);

    // create the action client
    MoveBaseClient ac("move_base", true);

    // Wait 60 seconds for the action server to become available
    ROS_INFO("Waiting for the move_base action server");
    ac.waitForServer(ros::Duration(60));

    ROS_INFO("Connected to move ba
```

Creamos el *action client* asociado a este nodo.
La cadena que pasamos tenemos que verla como un *topic*.
El cliente enviará mensajes a un servidor que esté escuchando mensajes de ese topic, en nuestro caso es “move_base”

Implementación *Action Client:* *sendGoals.cpp*

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <tf/transform_datatypes.h>

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_node");

    // Get the goal's x, y and angle from the launch file
    double x, y, theta;
    ros::NodeHandle nh;
    nh.getParam("goal_x", x);
    nh.getParam("goal_y", y);
    nh.getParam("goal_theta", theta);

    // create the action client
    MoveBaseClient ac("move_base", tr
```

Timeout para comprobar si el *action server* está levantado.

```
// Wait 60 seconds for the action server to become available
ROS_INFO("Waiting for the move_base action server");
ac.waitForServer(ros::Duration(60));

ROS_INFO("Connected to move base server");
```

Implementación *Action Client: sendGoals.cpp*

```
// Send a goal to move_base
move_base_msgs::MoveBaseGoal goal;
goal.target_pose.header.frame_id = "map";
goal.target_pose.header.stamp = ros::Time::now();

goal.target_pose.pose.position.x = x;
goal.target_pose.pose.position.y = y;
```

Rellenamos el contenido el mensaje *MoveBaseGoal con la posición del objetivo.*

```
// Convert
double radi
tf::Quaternion quaternion;
quaternion = tf::createQuaternionFromYaw(radians);

geometry_msgs::Quaternion qMsg;
tf::quaternionTFToMsg(quaternion, qMsg);

goal.target_pose.pose.orientation = qMsg;
```

(C)2013 Roi Yehoshua

Implementación *Action Client: sendGoals.cpp*

Rellenamos el contenido el mensaje *MoveBaseGoal con la orientación.*

Antes transformamos el ángulo a radianes y lo transformamos a su vez en una representación basada en quaternions (un mecanismo para representar de forma compacta la orientación). La transformación entre marcos de referencia y la representación de rotaciones como quaternions queda fuera de los contenidos de la práctica)

```
// Send a
move_base_
goal.targe
goal.targe

goal.targe
goal.targe
```

```
// Convert the Euler angle to quaternion
double radians = theta * (M_PI/180);
tf::Quaternion quaternion;
quaternion = tf::createQuaternionFromYaw(radians);

geometry_msgs::Quaternion qMsg;
tf::quaternionTFToMsg(quaternion, qMsg);

goal.target_pose.pose.orientation = qMsg;
```

(C)2013 Roi Yehoshua

Implementación *Action Client: sendGoals.cpp*

```
    // Send the goal command
    ROS_INFO("Sending robot to: x = %f, y = %f, theta = %f", x, y,
theta);
    ac.sendGoal(goal);

    // Wait for
    ac.waitForRe

    if (ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO("You have reached the goal!");
    else
    ROS_INFO("The base failed for some reason");

    return 0;
}
```

ENVIAMOS EL OBJETIVO!!!
Usando el método sendGoal del cliente “ac”.

(C)2013 Roi Yehoshua

Implementación *Action Client:*

Detenemos el proceso hasta que el *action server* nos devuelve un resultado. Una vez recibido un resultado, podemos acceder al resultado de la acción con el método ac.getState(). La constante SUCCEEDED representa que el objetivo enviado con sendGoals() se alcanzó. Si el resultado de getState() es otro, se entiende que el objetivo falló.

Y actuamos en consecuencia.

```
      // Send the g
    ROS_INFO("Send
theta);
    ac.sendGoal(go

    // Wait for the action to return
    ac.waitForResult();

    if (ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED)
    ROS_INFO("You have reached the goal!");
    else
    ROS_INFO("The base failed for some reason");

    return 0;
}
```

(C)2013 Roi Yehoshua

Ejercicio propuesto

- Modificar la función main del código fuente donde hayáis implementado los ejercicios de mapeo para
  - Enviar el punto seleccionado por la función selectNode a move_base.
  - Integrando la idea de cómo enviar objetivos implementada en el fichero ejemplo SendGoals.cpp.

Estrategia básica de detección de fronteras

1. Repetir sin fin
   1. Girar 360º (quizá recomendable un par de veces)
   2. Detectar la frontera
   3. Seleccionar un punto de la frontera
   4. Enviar al robot a ese punto objetivo.
   5. Adoptar una estrategia si no se alcanza el objetivo.
2. Cuando se haya terminado el proceso (frontera vacía), escribir el mapa en un fichero de formato imagen

- Tener en cuenta que es una propuesta básica, pueden hacerse las modificaciones que se consideren oportunas para mejorar la eficiencia y efectividad de la exploración basada en fronteras.