

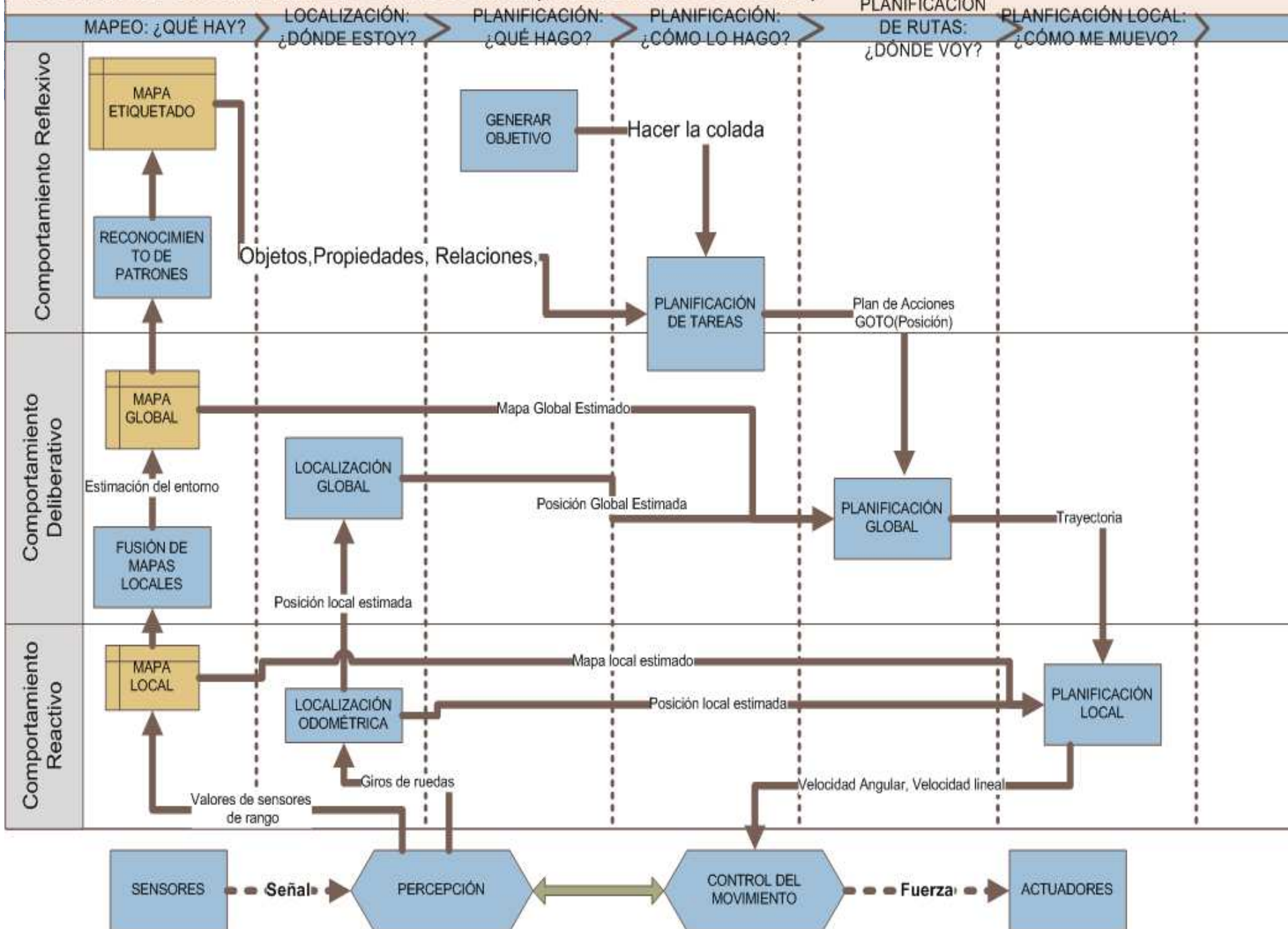


Técnicas de los Sistemas Inteligentes

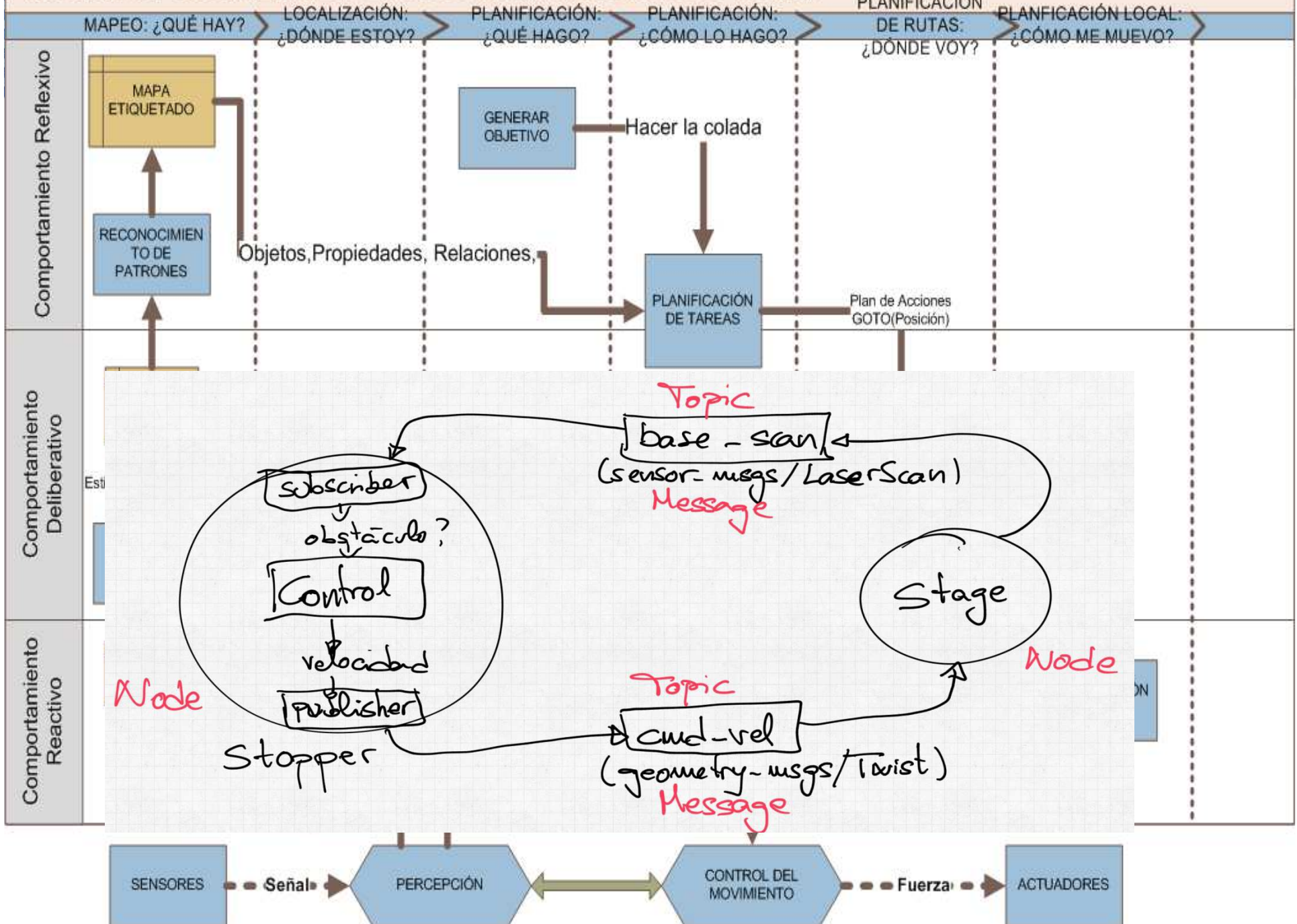
Práctica1: Robótica.

Sesión 3. Planificación local mediante campos de potencial

PROBLEMAS FUNDAMENTALES EN ROBÓTICA (PARA UN ROBOT MÓVIL)



PROBLEMAS FUNDAMENTALES EN ROBÓTICA (PARA UN ROBOT MÓVIL)



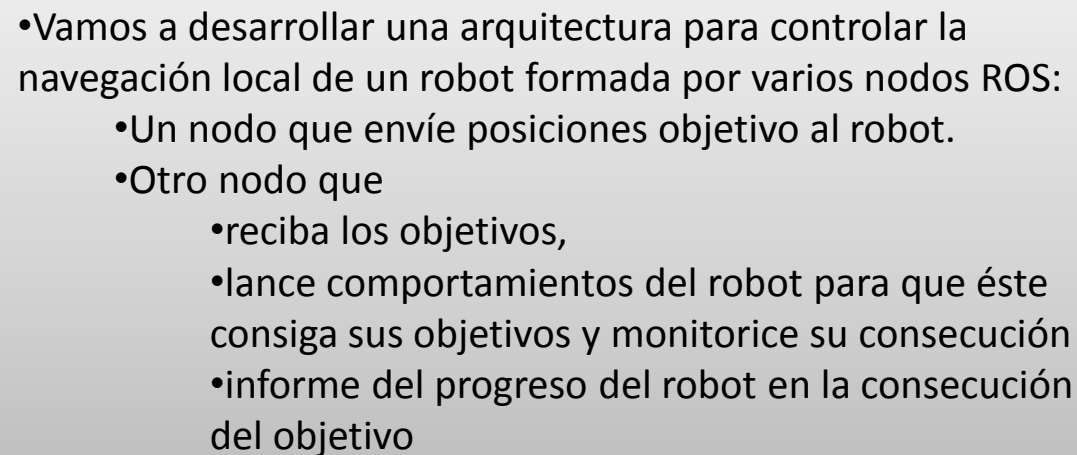


- Conceptos básicos de navegación local
- Técnicas de navegación local sin mapa: campos de potencial
- Técnicas de navegación local con mapa:
- Ejemplo de arquitectura de planificación local para un robot.
- Implementación en ROS
 - actionlib: paquete para ejecución de acciones de alto nivel
 - Cliente de acciones
 - Servidor de acciones
 - Ejemplo de ejecución
- Tareas de Entrega 2.



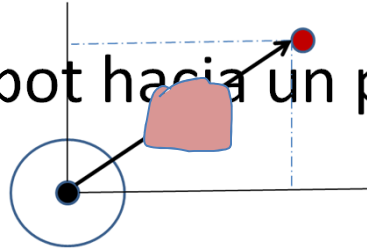
Arquitecturas Integradas

- El desarrollo de un sistema inteligente puede dar lugar a:
 - Un agente inteligente autónomo
 - Una herramienta inteligente
- En ambos casos necesitan tener capacidad de:
 - Recibir un objetivo (o autoplantearse un objetivo)
 - Elaborar un plan de acción para alcanzarlo
 - Ejecutar el plan de acción
 - Monitorizar el plan de acción para comprobar fallos.
 - Responder de alguna forma a los fallos
- La solución consiste en el desarrollo de una arquitectura que integre estas funcionalidades como módulos independientes.



Planificación local de movimientos

- Ejemplo con ROS.
- Cómo dirigir el robot hacia un punto dado?
 - conocidos
 - pose actual
 - pose objetivo
 - qué valores de v (velocidad lineal) y w (angular) enviar al robot?
 - considerar que tiene que ir a la posición objetivo pero tiene que evitar obstáculos
 - considerar que los robots tienen inercia y no pueden ejecutar rutas arbitrarias a velocidades arbitrarias (como la aspiradora del año pasado)
 - nos servirán los conceptos de
 - representación del estado dinámico como un vector de fuerza
 - Técnica de campos de potencial para controlar un robot sin mapa



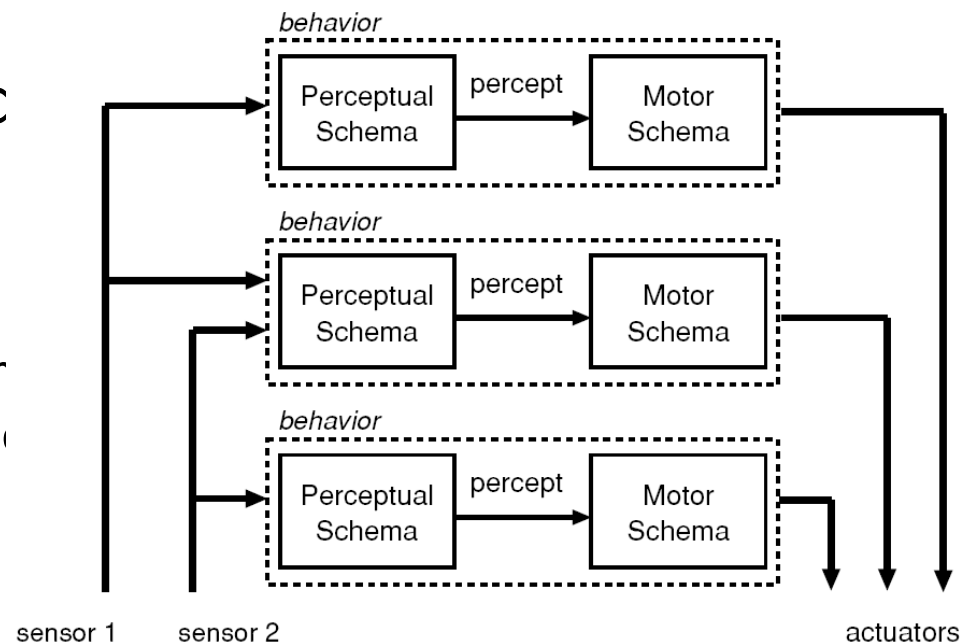
- El programa que guía al robot está basado en una arquitectura de bloques
- Cada bloque representa un comportamiento

– Comportamiento

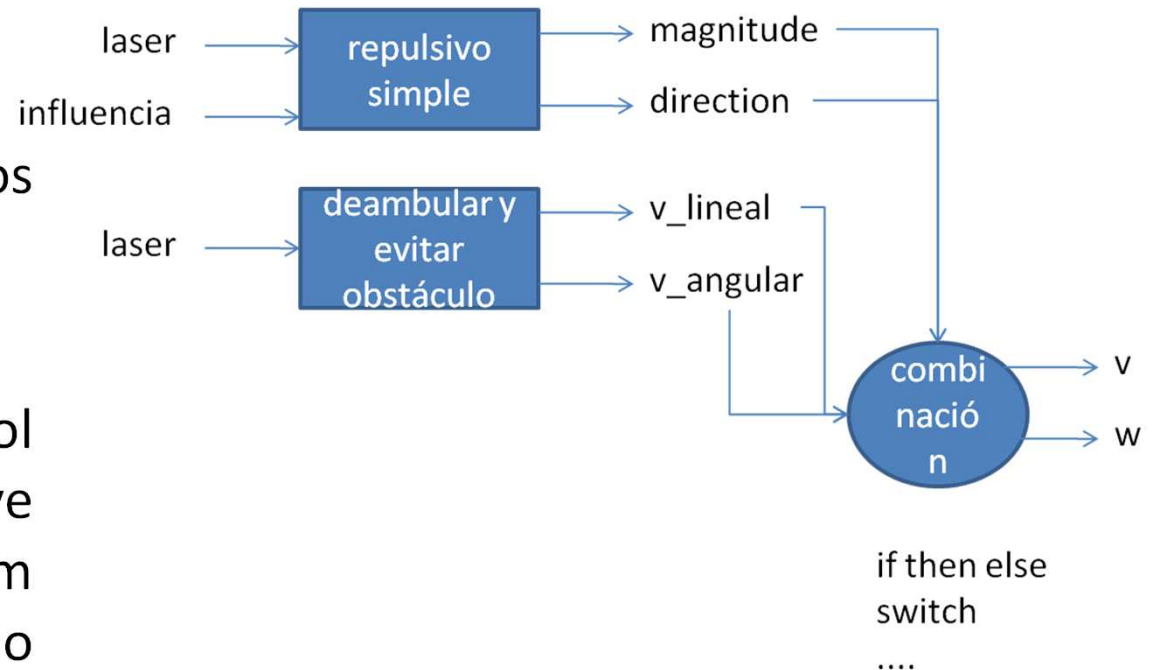
- Una función que transforme entradas de sensores en valores que son pasados a acciones de motores



c



- Por ejemplo:
 - Comportamiento repulsivo simple
 - Evitar obstáculos simple
- Idea básica:
 - el software de control de robots se construye de abajo arriba (bottom up) desarrollando comportamientos adaptados ecológicamente (adaptados al entorno)





Modelo cinemático

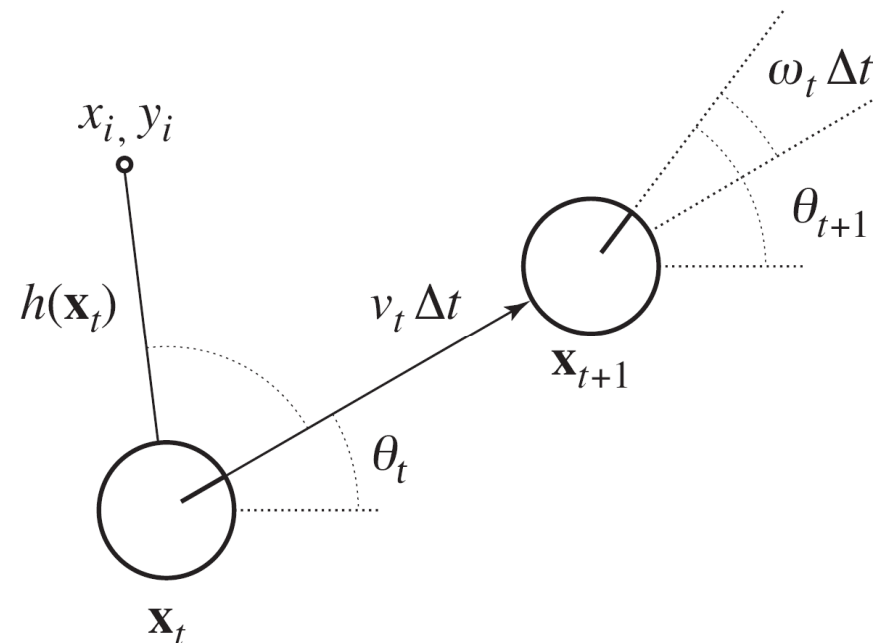
- Debemos tener alguna forma de representar cómo cambia la posición y orientación del robot.
- Considerando que tratamos de controlar el movimiento del robot mediante la especificación de dos velocidades en un instante de tiempo t :
 - v_t (lineal o de traslación) y
 - w_t (angular o rotacional)

Modelo cinemático

- Esta representación nos la ofrece un modelo cinemático (de movimiento del robot)
- Para cada incremento de t , Δt , podemos saber la pose del robot siguiendo un modelo cinemático determinista a partir de
 - La velocidad lineal v del robot
 - La velocidad angular w del robot
- Este modelo determinista puede usarse para implementar un modelo odométrico determinista (en el que no hay incertidumbre).
 - algo por el estilo es lo que implementa Stage para saber la posición exacta del robot, dado un punto actual y las velocidades.

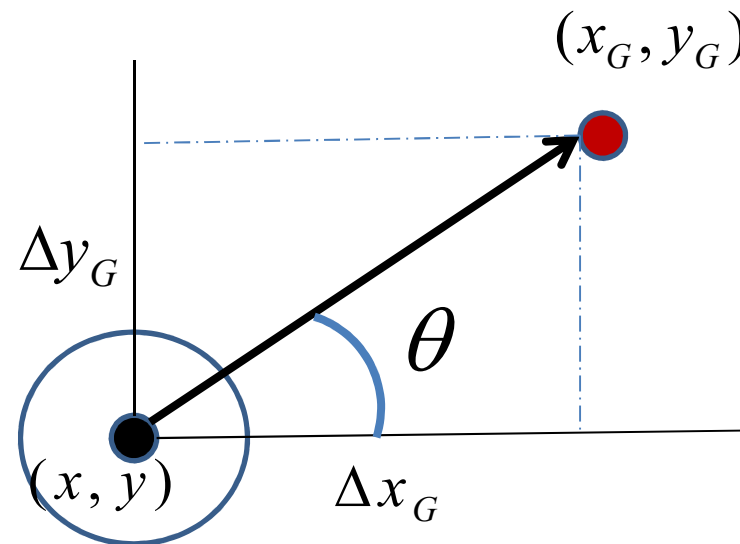
$$X_t = (x_t, y_t, \theta_t)^T$$

$$X_{t+1} = f(X_t, v_t, \omega_t) = X_t + \begin{pmatrix} v_t \Delta t \cos \theta_t \\ v_t \Delta t \sin \theta_t \\ \omega_t \Delta t \end{pmatrix}$$



Modelo dinámico como un vector

- ¿Cómo podemos calcular la velocidad lineal y la angular?
- Representación como un vector
 - la tupla (v,w) se puede calcular a partir de la representación como un vector de la posición actual y la objetivo
 - El módulo del vector es el valor de la velocidad
 - El ángulo del vector es la orientación objetivo del robot
 - La orientación del robot cambia de acuerdo a la velocidad angular en incrementos del tiempo
 - La posición del robot cambia de acuerdo al módulo del vector en incrementos del tiempo

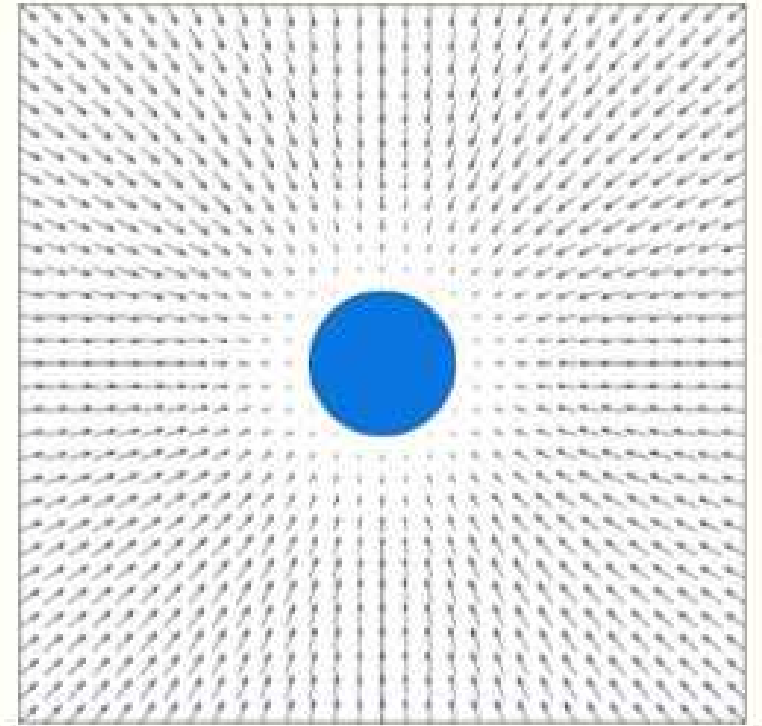




- Idea
 - Imaginar el robot como una partícula con carga navegando por un campo magnético o una bola rodando descendiendo por una colina
 - [Artificial Potential Field Approach in Robot Motion Planning - YouTube](#)
 - [Demonstration-Guided Motion Planning for Robotic Assistance - YouTube](#)
 - Metodología a grandes rasgos
 - Bloque básico: Vector de acción: (velocidad, orientación)
 - Cada comportamiento obtiene un vector acción de salida
 - diseñar comportamientos, cada uno para una tarea (ir hacia el objetivo, evitar obstáculo,)
 - representar cada comportamiento como un campo de potencial
 - combinar los comportamientos mediante la combinación de los campos de potencial

Campos de potencial: IrAObjetivo

- IrAObjetivo
 - Tarea que debe implementar el comportamiento
 - Hacer que el robot se oriente y desplace hacia un objetivo identificado.
 - Salida: un vector que apunta hacia el objetivo.
 - Campo de potencial
 - Colección de vectores para todos los puntos del espacio bidimensional.
 - Cada vector representa la energía potencial que empuja al robot en cada punto
 - Campo atractivo
 - el campo causa que el robot sea atraído por el objetivo (todos los vectores apuntan al objetivo)

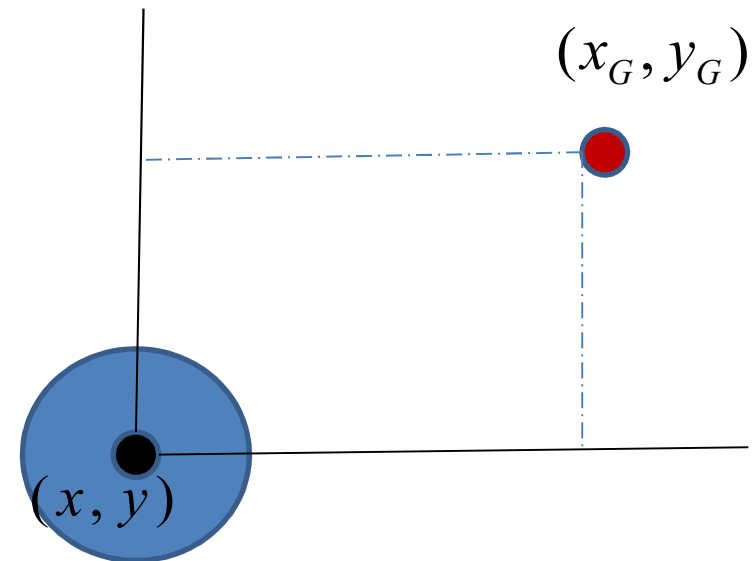


- Entradas:

- posición del objetivo y posición del vector.

(x_G, y_G) la posición del objetivo

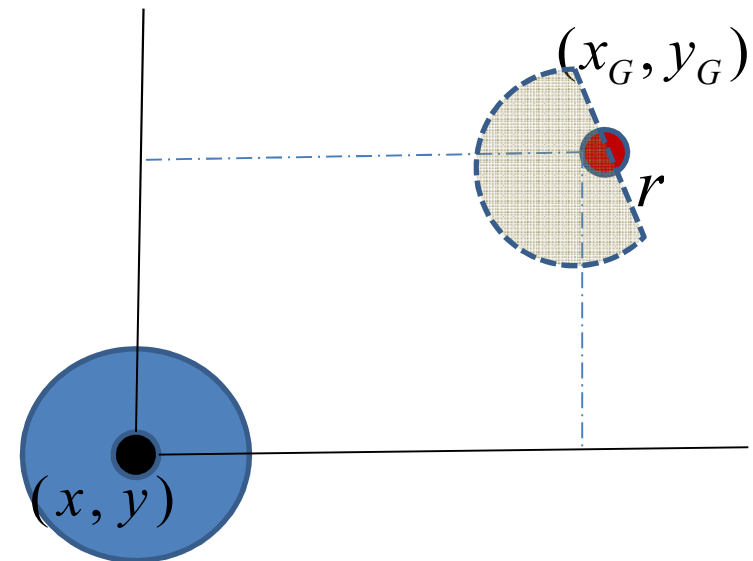
(x, y) la posición del robot



- Entradas:

- posición del objetivo y posición del vector.
- radio del objetivo (parámetro configurable)

(x_G, y_G) la posición del objetivo
 (x, y) la posición del robot
 r el radio del objetivo.



- Entradas:

- posición del objetivo y posición del vector.
- radio del objetivo
- configuración del campo:
 - extensión del campo
 - fuerza del campo

(x_G, y_G) la posición del objetivo

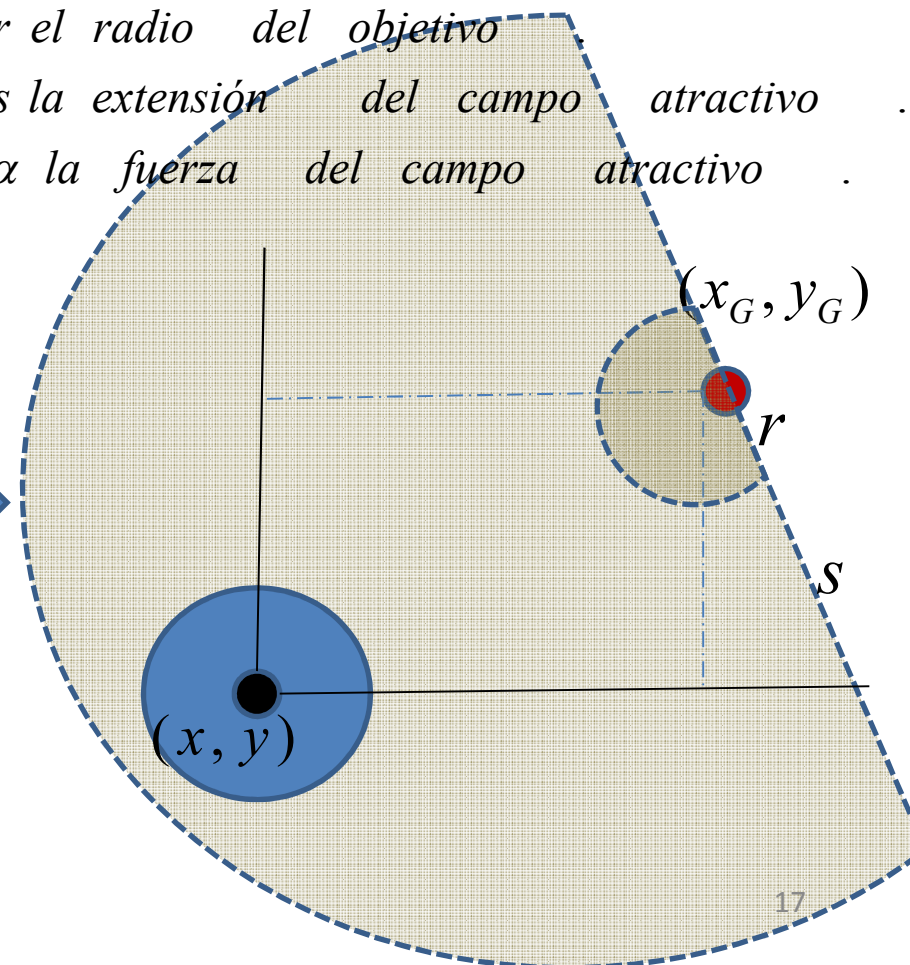
(x, y) la posición del robot

r el radio del objetivo

s la extensión del campo atractivo

α la fuerza del campo atractivo

Parámetros de configuración del campo de potencial. Son constantes que se estiman mediante experimentación (prueba y error)



Campos de potencial: Ir A Objetivo

- Entradas:
 - posición del objetivo y posición del vector.
 - radio del objetivo
 - configuración del campo:
 - extensión del campo
 - fuerza del campo
- Salidas
 - componentes del vector fuerza atractivo del objetivo
- Proceso:
 - en cada iteración, se obtiene la posición del robot y se calculan las componentes del vector y se transforman en velocidades.
 - Obtenemos así una trayectoria hacia el objetivo

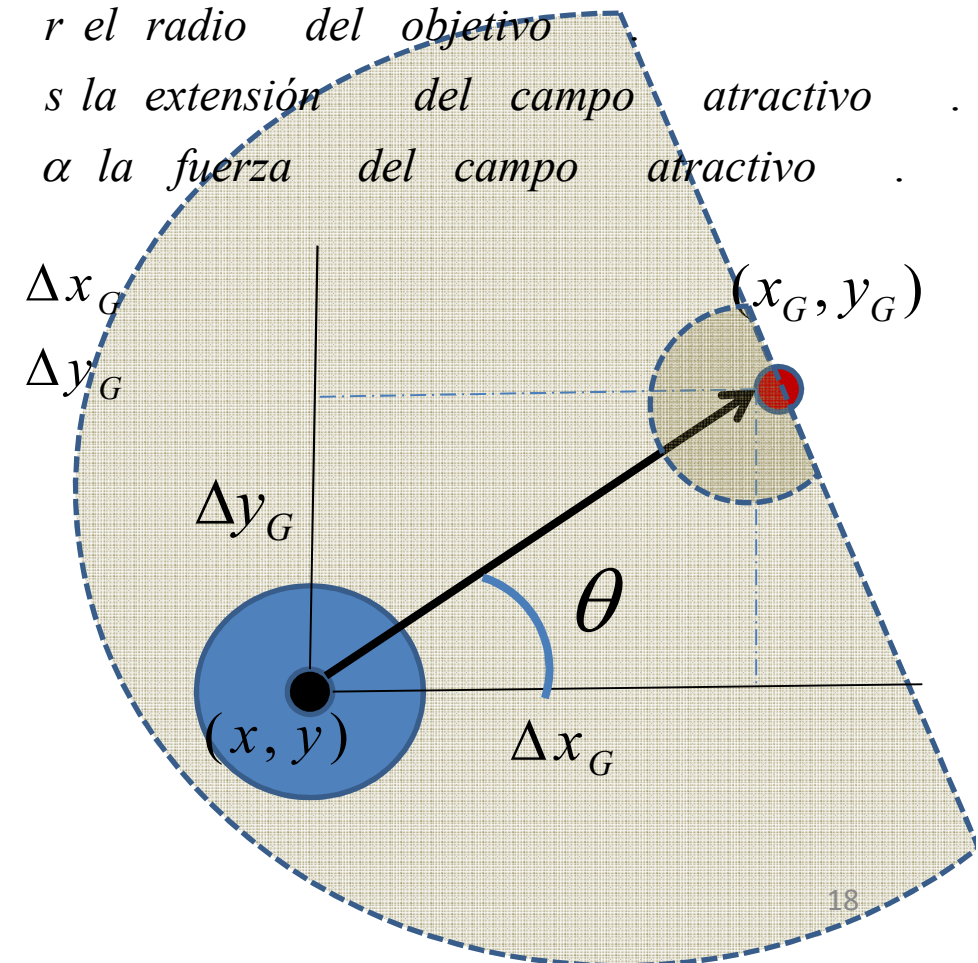
(x_G, y_G) la posición del objetivo

(x, y) la posición del robot

r el radio del objetivo

s la extensión del campo atractivo

α la fuerza del campo atractivo





Campos de potencial: Ir A Objetivo

¿Cómo calcular las componentes del vector?

1. Calcular la distancia entre el objetivo y el robot

$$d = \sqrt{(x_G - x)^2 + (y_G - y)^2}$$

2. Calcular el ángulo entre el robot y el objetivo

$$\theta = \arctan \left(\frac{y_G - y}{x_G - x} \right)$$

3. Obtener las componentes a partir de **distancia**, **ángulo** y **parámetros de configuración** del campo.

$$\text{if } d < r$$

$$\Delta x = \Delta y = 0$$

$$\text{if } r \leq d \leq s + r$$

$$\Delta x = \alpha(d - r) \cos(\theta)$$

$$\Delta y = \alpha(d - r) \sin(\theta)$$

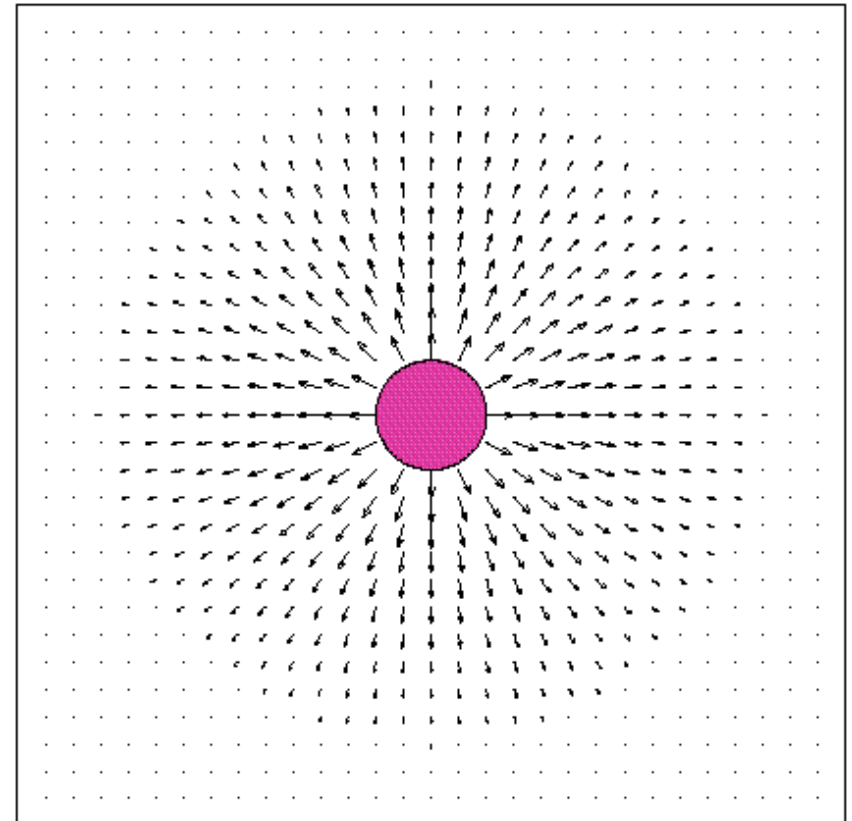
$$\text{if } d > s + r$$

$$\Delta x = \alpha s \cos(\theta)$$

$$\Delta y = \alpha s \sin(\theta)$$

Campos de potencial: Evitar Obstáculo

- Evitar Obstaculo
 - Tarea que debe implementar el comportamiento
 - Hacer que el robot se oriente y desplace **repeliendo un obstáculo**.
 - Salida: un vector que apunta fuera del obstáculo
 - Campo de potencial
 - Colección de vectores para todos los puntos del espacio bidimensional.
 - Cada vector representa la energía potencial que empuja al robot en cada punto
 - Campo repulsivo
 - el campo causa que el robot sea repelido por el obstáculo (todos los vectores apuntan fuera del obstáculo).





Campos de potencial: Evitar Obstáculo

- Entradas:

- posición del obstáculo y (x_o, y_o) la posición del objetivo
- posición del robot (x, y) la posición del robot
- radio del obstáculo r el radio del obstáculo.
- configuración del campo:
 - extensión del campo s la extensión del campo repulsivo.
 - fuerza del campo β la fuerza del campo repulsivo.

- Salidas

- componentes del vector Δx_o
- fuerza atractivo del Δy_o
- objetivo

1. Calcular la distancia entre el obstáculo y el robot

$$d = \sqrt{(x_o - x)^2 + (y_o - y)^2}$$

2. Calcular el ángulo entre el robot y el obstáculo

$$\theta = \arctan \left(\frac{y_o - y}{x_o - x} \right)$$

3. Calcular el gradiente de x y de y (componentes del vector resultante)

if $d < r$

$$\Delta x = -\text{sign}(\cos(\theta))\infty$$

$$\Delta y = -\text{sign}(\sin(\theta))\infty$$

if $r \leq d \leq s + r$

$$\Delta x = -\beta(s + r - d)\cos(\theta)$$

$$\Delta y = -\beta(s + r - d)\sin(\theta)$$

if $d > s + r$

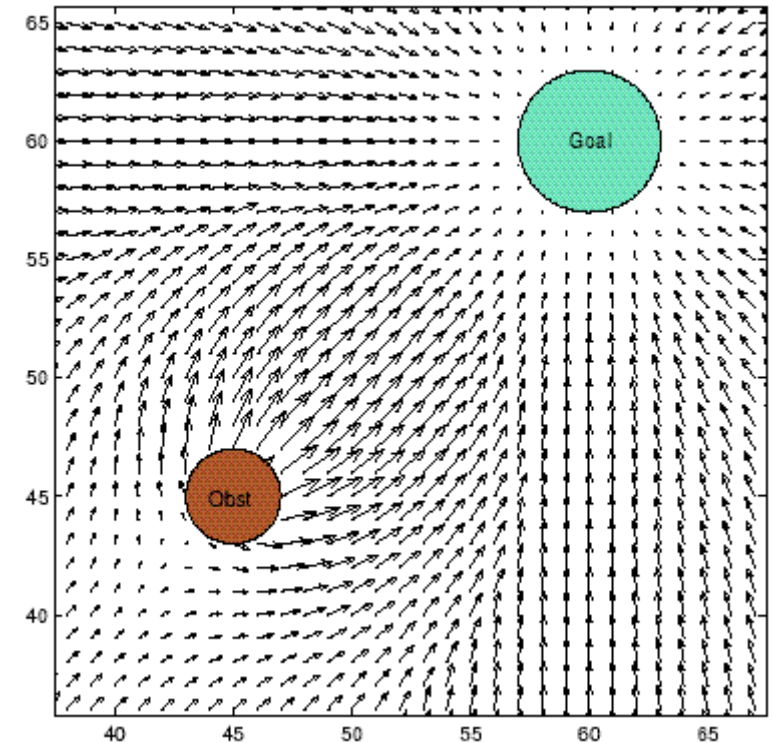
$$\Delta x = \Delta y = 0$$

Combinación de ambos comportamientos

- Para obtener el vector resultante, sumamos ambos vectores

$$\Delta x = \Delta x_G + \Delta x_O$$

$$\Delta y = \Delta y_G + \Delta y_O$$



Campo de potencial resultante



¿Cómo determinar el comportamiento final?

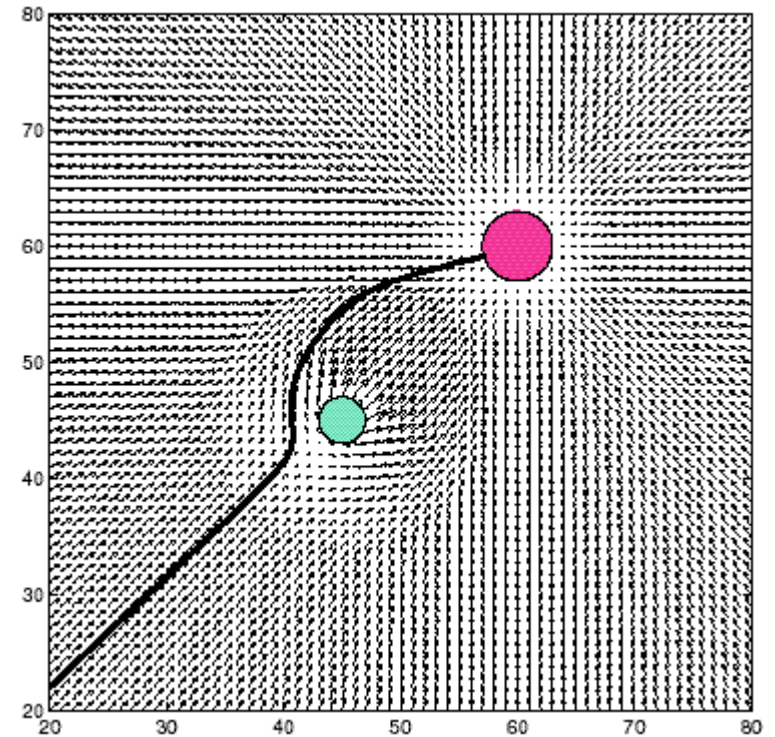
1. Calcular la suma de los vectores, como hemos visto.
2. velocidad lineal = módulo del vector

$$v = \sqrt{\Delta x^2 + \Delta y^2}$$

¿y la velocidad angular?

1. Primero, calcular el ángulo

$$\theta = \arctan \left(\frac{\Delta y}{\Delta x} \right)$$

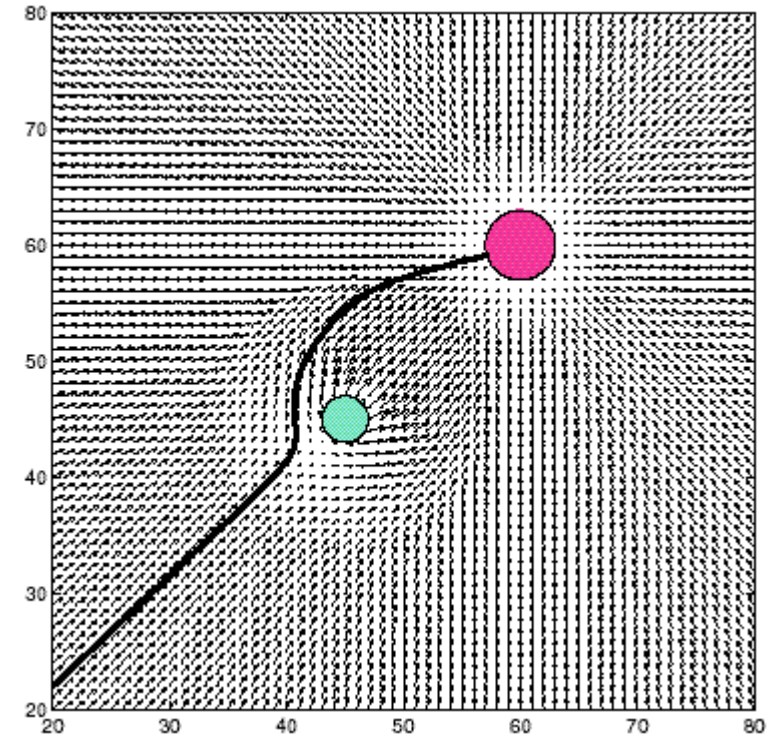


Trayectoria Resultante.

¿Cómo determinar el comportamiento final?

4. Después, calcular la velocidad angular (por casos)

- Si $\text{Yaw} - \theta$ es grande
 - Usar un valor constante de v.angular
- Si $\text{Yaw} - \theta$ es pequeña
 - Usar esa diferencia como velocidad angular



Trayectoria Resultante.



Algoritmo en bucle cerrado

1. Obtener la posición goal GoalX, GoalY

1. While !objetivoAlcanzado()

1. Obtener Odometría:

1. Obtener posición actual PosX, PosY

2. Obtener ángulo actual Yaw

2. Obtener Scan Láser

1. Calcular posición actual obstáculo ObsX,ObsY

3. Calcular componente Atractivo

4. Calcular componente Repulsivo

5. Calcular la resultante total

6. Calcular velocidad lineal

7. Calcular velocidad angular

8. Enviar las velocidades al robot.

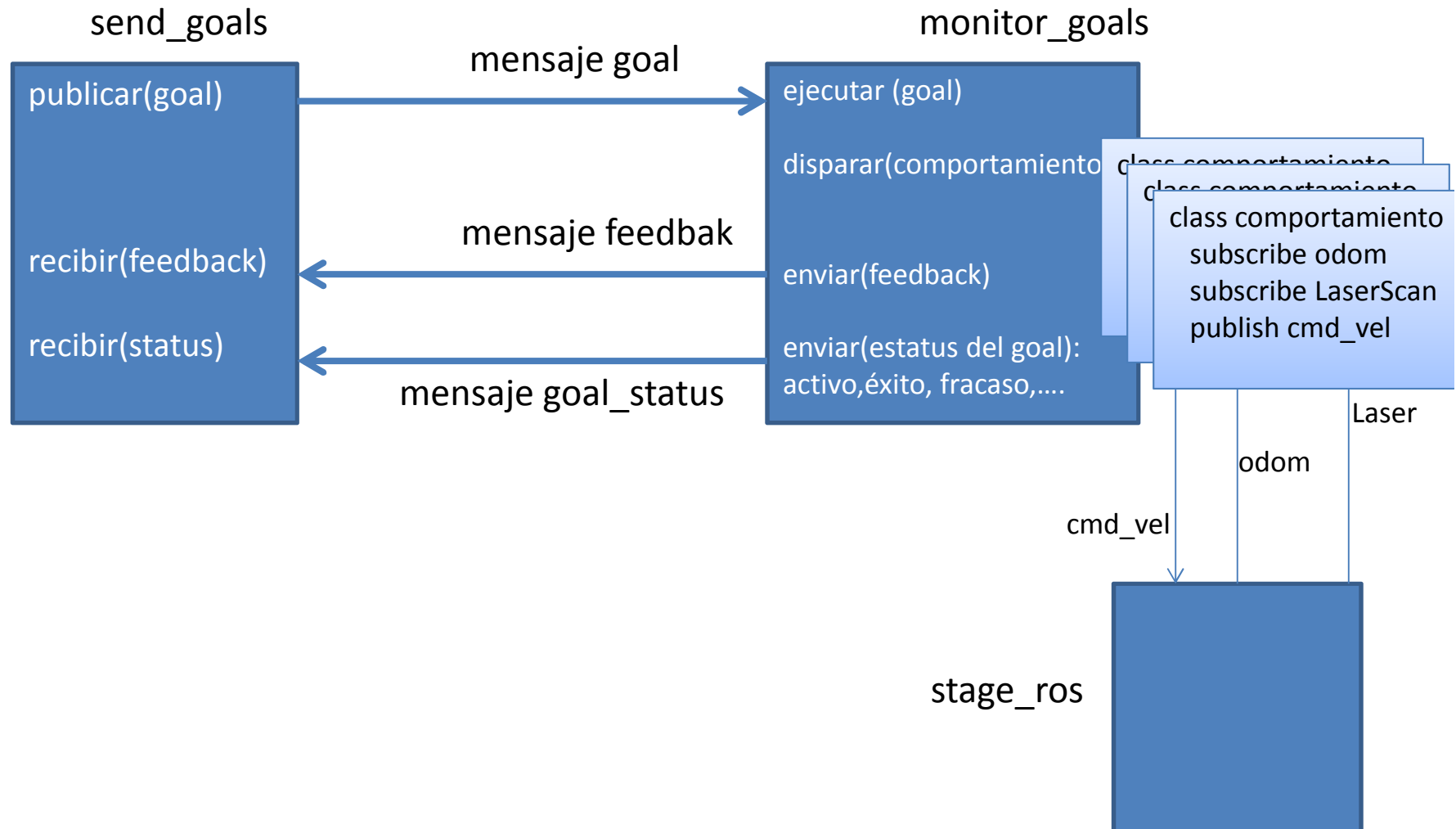


Objetivo más concreto

- Construir un planificador de movimientos locales que permita
 - recibir una posición objetivo
 - disparar un comportamiento basado en campos de potencial para alcanzar el objetivo
 - informar del progreso y del éxito/fracaso en la consecución del objetivo.
- ROS contiene un paquete que ayuda a hacer parte de estas tareas: **actionlib**
 - <http://wiki.ros.org/actionlib>



Recordemos la 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.
 - **Actions**, más apropiadas cuando la tarea solicitada toma más tiempo y además
 - queremos **monitorizar** su progreso
 - obtener **feedback continuo**
 - poder **cancelar** la petición durante la ejecución.



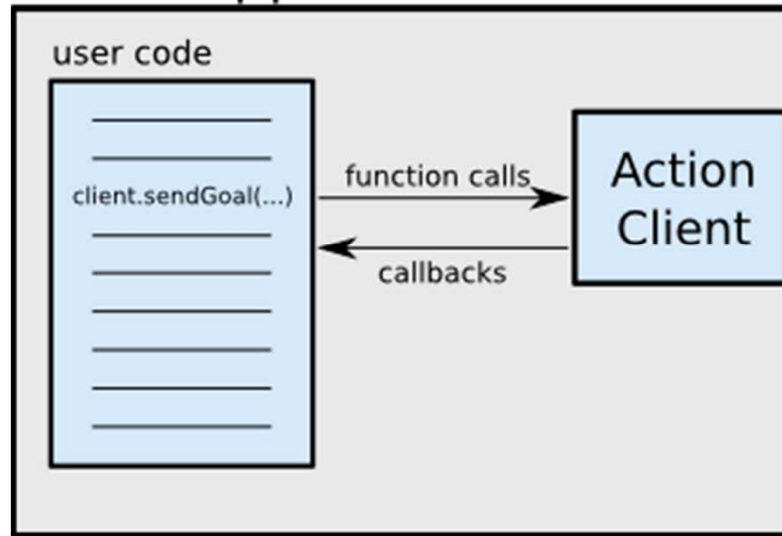
Qué es actionlib

- 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>

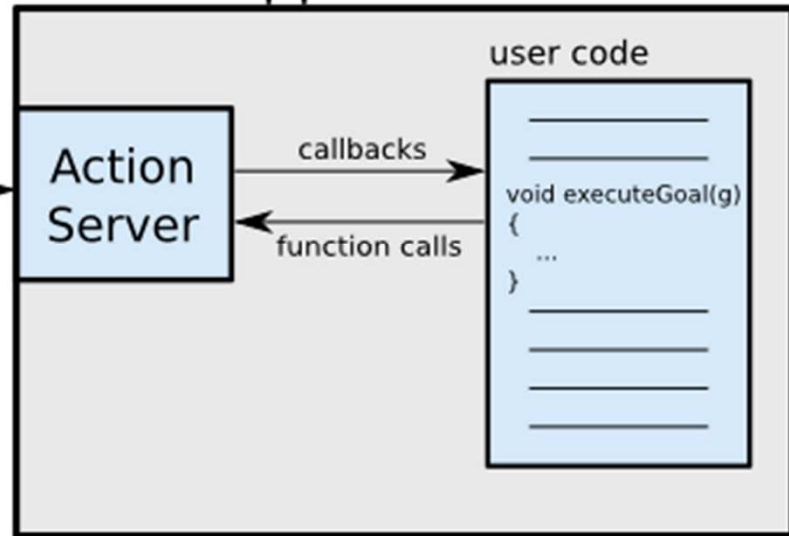


actionlib: Interacción client-server usando ROS Action protocol

Client Application



Server Application



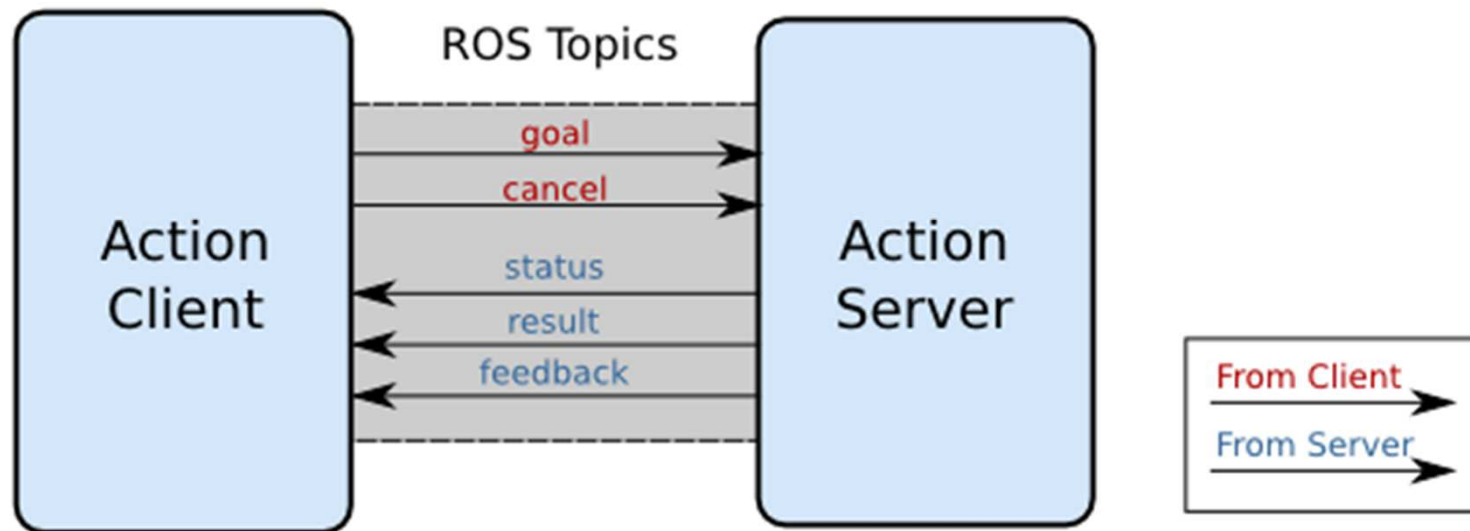
ROS

- 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

- **ROS Topics**

- **goal** - Used to send new goals to server
- **cancel** - Used to send cancel requests to server
- **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 tienen una estructura que se define a partir de ficheros de texto con extensión *.action*
- 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

```
geometry_msgs/PoseStamped target_pose
---
---
geometry_msgs/PoseStamped base_position
```

- Tipo y campo del mensaje usado para definir un goal
- Tipo y campo del mensaje para definir el resultado (en este caso es vacío).
- Tipo y campo del mensaje para definir el feedback.



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 internamente 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.



- Compilar
 - Descargar el fichero ejemplo_actionlib.zip del material de prácticas.
 - Descomprimirlo en `~/<vuestro_work_space>/src`
 - Comprobar que hay un directorio adecuado de paquete ros.
 - Hacer `catkin_make` en `~/catkin_ws`



Ejecución ejemplo

- Ejecutar
 - roscore
 - rosrun stage_ros stageros `rospack find stage_ros`/world/willow-erratic.world
 - rosrun action_lib_scb server_node
 - rosrun action_lib_scb cliente_node
- Observar comportamiento en stage



Implementación Action Client: *myClientLite.cpp*

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

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

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

    // create the action client
    // true causes the client to spin its own thread
    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");
```



Implementación Action Client: *myClientLite.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 = 18.174;
goal.target_pose.pose.position.y = 28.876;
goal.target_pose.pose.orientation.w = 1;

ROS_INFO("Sending goal");
ac.sendGoal(goal);

// 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;
}
```



Implementación Action Client: myClientLite.cpp

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

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

```
int main(int argc, char** argv) {
    ros::init(argc, argv, "send_goals_
```

```
    // create the action client
    // true causes the client to spin
    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)



Implementación *Action Client*: *MyClientLite.cpp*

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

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

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

```
    // create the action client
    // true causes the client to wait for the server to start
    MoveBaseClient ac("move_base", true);
```

```
    // Wait 60 seconds for the action server to start
    ROS_INFO("Waiting for the move base action server");
    ac.waitForServer(ros::Duration(60));
```

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

CASO DE USO SIN UTILIZAR CLASES

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*: *MvClientLite.cpp*

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

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

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

    // create the action client
    // true causes the client to spin its own thread
    MoveBaseClient ac("move_base", true);

    // Wait 60 seconds for the action server to start
    ROS_INFO("Waiting for the move base 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*: *MyClientLite.cpp*

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

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

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

    // create the action client
    // true causes the client to spin its own thread
    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("Conne
```

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*.



Implementación *Action Client*: *MyClientLite.cpp*

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

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

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

    // create the action client
    // true causes the client to spin its own thread
    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");
}
```

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



Implementación *Action Client: MyClientLite.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 = 18.174;
goal.target_pose.pose.position.y = 28.876;
goal.target_pose.pose.orientation.w = 1;

ROS_INFO("Sending goal");
ac.sendGoal(goal);
```

```
// Wait for the a
```

Rellenamos el contenido el mensaje *MoveBaseGoal*.

```
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;
}
```



Implementación *Action Client*: *MyClientLite.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 = 18.174;
goal.target_pose.pose.position.y = 0.0;
goal.target_pose.pose.orientation = ros::Quaternion::getIdentity();
```

```
ROS_INFO("Sending goal to move_base");
ac.sendGoal(goal);
```

Detenemos el proceso hasta que el *action server* nos devuelve un resultado.

Y actuamos en consecuencia.

```
// 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");
```

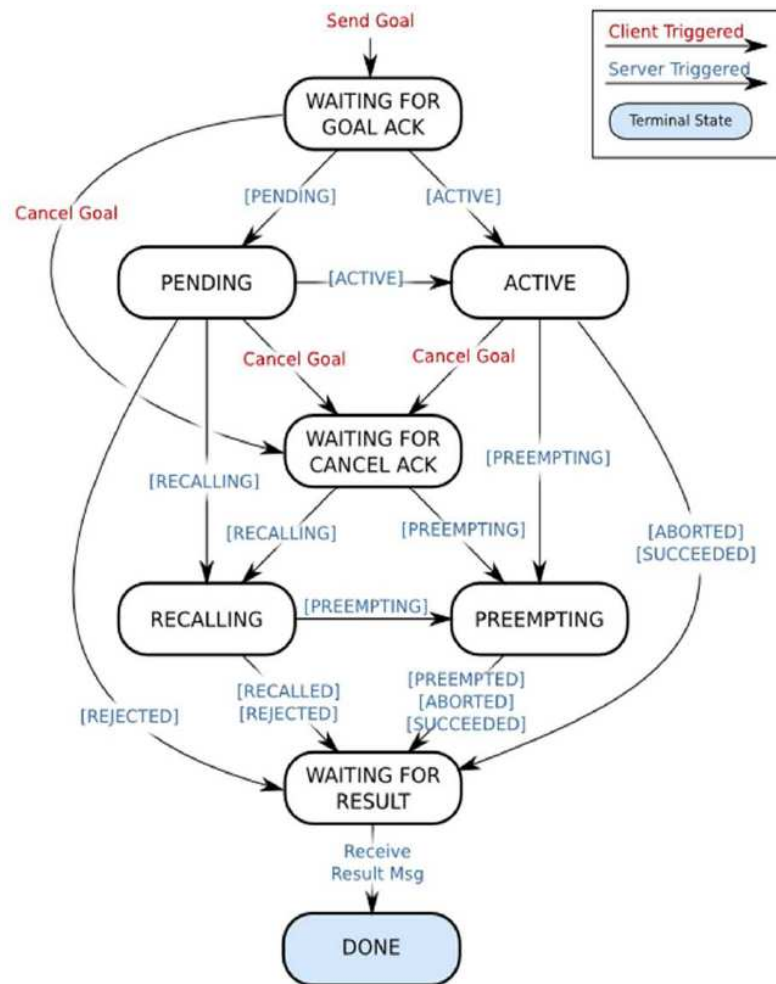
```
return 0;
```

```
}
```



Autómata de estados *ActionClient*

- Transiciones de estados



- Ejemplo detallado:
<http://robot.wpi.edu/wiki/index.php/Actionlib>
- Callbacks asociados a estados:
 - done
 - active
 - feedback.
- Un goal puede
 - cancelarse
 - aplazarse por otro más prioritario (preempting)



Implementación *Action Server*

- Implementación como una clase
- Definir métodos callbacks para cada uno de los envíos/peticiones del *Action Client*:
 - *Enviar goal -> Callback para ejecutar goal.*
 - *Solicitar aplazamiento de goal (preempted) -> Callback para cambiar el estado de un goal a Preempted*
 - *Solicitar cancelación de goal -> Callback para cambiar el estado de un goal a Cancelled*
 - ...



Implementación *Action Server*

- ¿Cómo asociar un comportamiento a *Action Server*?
 - Observar que la clase *MyActionServer* incluye un objeto de tipo *LocalPlanner* que implementa un comportamiento basado en campos de potencial
 - Tendréis que modificar la implementación de *LocalPlanner* para mejorar el comportamiento del robot en la consecución del goal.



Implementación de un Action Server

MyServerLite.cpp

```
#include <ros/ros.h>
#include "myPlannerLite.h"
#include <nav_msgs/Odometry.h> // manejar tb. datos de odometría.
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/server/simple_action_server.h>

//Clase que contiene el action server
class MyActionServer
{
protected:

    ros::NodeHandle n;
    actionlib::SimpleActionServer<move_base_msgs::MoveBaseAction> as;
    move_base_msgs::MoveBaseFeedback feedback;
    move_base_msgs::MoveBaseResult result;
    move_base_msgs::MoveBaseGoal goal;
    std::string action_name;

    LocalPlanner planner; //el server tiene asociado un objeto planner, es el que
                          //desarrolla el comportamiento de navegar hasta el bjetivo.
```

// Adaptado de <http://robot.wpi.edu/wiki/index.php/Actionlib>



Implementación de un *Action Server*

```
public:
    MyActionServer(std::string name):
        //Crea y configura el action server. Necesita una referencia al manejador del
nodo ros, n
        //Se subscribe al topic "mi_move_base",
        // Y registra una callback mediante el puntero a la función ejecutaCB
        // la especificación de la callback es con la sintaxis boost:: porque
estamos dentro
        // de una especificación de server como una clase C++
        as(n, "mi_move_base",
            boost::bind(&MyActionServer::ejecutaCB, this, _1), false),
        action_name(name)
    {
        //Registra Callbacks.
        as.registerPreemptCallback(boost::bind(&MyActionServer::preemptCB, this));
        //Inicia el server

        as.start();
        ROS_INFO("Action Server Lanzado");
    }
```



Implementación de un *Action Server*

//Callback para manejar aplazamiento

```
void preemptCB()  
{  
    ROS_INFO("%s está aplazado", action_name.c_str());  
    as.setPreempted(result, "Goal en estado preempted");  
}
```




Implementación de un *Action Server*

//Callback para procesar el goal

```
void ejecutaCB(const move_base_msgs::MoveBaseGoalConstPtr& goal)
{
```

//Si el servidor está muerto, no procesar

```
if (!as.isActive() || as.isPreemptRequested())
```

```
{
```

```
    ROS_INFO("El servidor está muerto.");
```

```
    return;
```

```
}
```

//Configurar la ejecución del procesamiento del goal a 5Hz

```
ros::Rate rate(5);
```

```
ROS_INFO("Procesando el goal. Enviando goal al navegador local");
```

//Le pasamos al planner la información el objetivo recibido!!!

```
planner.setGoal(goal);
```



Implementación de un *Action Server*

```
//procesamiento del goal
bool success = true;
while (true)
{
    //Comprobar si ROS está vivo
    if (!ros::ok())
    {
        success = false;
        ROS_INFO("%s apagando ", action_name.c_str());
        break;
    }

    //Si el action server está muerto o preempted, detener procesamiento
    if(!as.isActive() || as.isPreemptRequested())
        return;


    //Informar del goal , obteniendo los datos del goal enviado
    ROS_INFO("...yendo hacia el goal (%f,%f,%f)",
            goal->target_pose.pose.position.x,
            goal->target_pose.pose.position.y,
            goal->target_pose.pose.orientation.w );
}
```



Implementación de un *Action Server*

```
planner.setDeltaAtractivo();
//*****
//aquí habrá que mejorar el navegador local para que evite obstáculos
//con el componente repulsivo
//*****
planner.setv_Angular();
planner.setv_Lineal();
if (planner.goalAchieved())
    break;
planner.setSpeed();
ROS_INFO("Enviando velocidad (%f,%f)", planner.v_lineal,
        planner.v_angular);

//Informar del feedback al action client
feedback.base_position.pose.position.x =
    planner.odometria.pose.pose.position.x;
feedback.base_position.pose.position.y =
    planner.odometria.pose.pose.position.y;
feedback.base_position.pose.orientation.w =
    planner.odometria.pose.pose.orientation.w;
as.publishFeedback(feedback);
rate.sleep();
} //while(true)
```



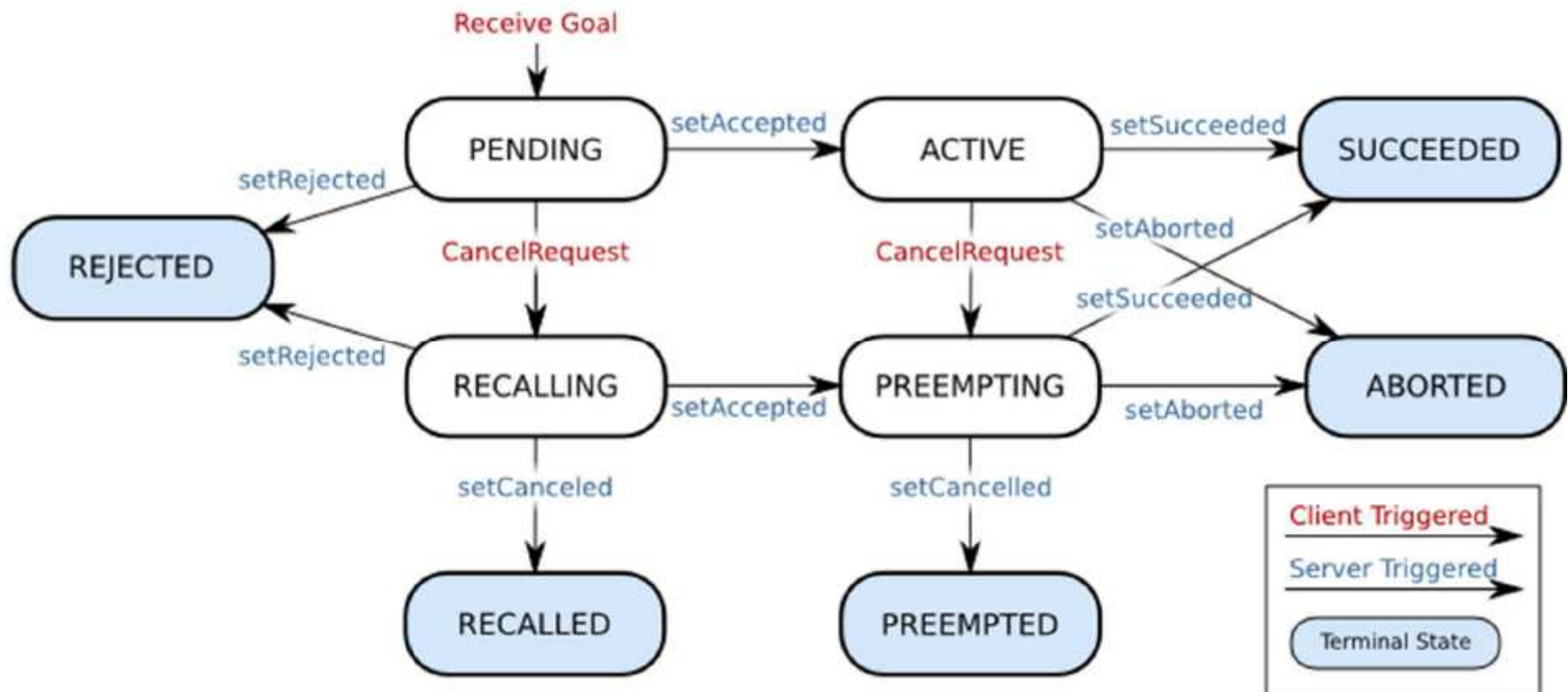
Integración
Planner



Implementación de un *Action Server*

```
//Publicar el resultado si el goal no está preempted
if (success)
{
    ROS_INFO("%s tuvo éxito al llegar al goal (%f %f %f)",
        action_name.c_str(),
        goal->target_pose.pose.position.x,
        goal->target_pose.pose.position.y,
        goal->target_pose.pose.orientation.w );
    as.setSucceeded();
}
else
    as.setAborted();
};
```


Server State Machine





Tareas para la segunda entrega

- Obligatorio (mínimo exigido):
 - Comprender la implementación de campos de potencial en *myPlannerLite.h* y *myPlannerLite.cpp*.
 - Mejorar esta implementación para que durante la navegación se eviten obstáculos de acuerdo a la técnica de campos de potencial.
 - Mejorar la implementación del *Action Client* para que, desde un fichero *.launch* se pueda
 - configurar los parámetros de los campos de potencial atractivo y repulsivo
 - configurar la posición objetivo como parámetro.



ROS Parameters

- Se pueden enviar desde terminal objetivos
- Se pueden poner valores a parámetros usando el tag `<param>` en el ROS launch file:
- Más información en <http://wiki.ros.org/roslaunch>

```
<launch>
  <param name="goal_x" value="18.5" />
  <param name="goal_y" value="27.5" />
  <param name="goal_theta" value="45" />
  <node name="send_goals_node" pkg="send_goals" type="send_goals_node"
output="screen"/>
</launch>
```



Recuperar Parameters

- Hay dos métodos para recuperar los parámetros del .launch en la clase NodeHandle:
 - `getParam(key, output_value)`
 - `param()` is similar to `getParam()`, but allows you to specify a default value in the case that the parameter could not be retrieved
- Ejemplo:

```
// Read x, y and angle params
ros::NodeHandle nh;
double x, y, theta;
nh.getParam("goal_x", x);
nh.getParam("goal_y", y);
nh.getParam("goal_theta", theta);
```

- Más información en
 - <http://wiki.ros.org/roscpp/Overview/Parameter%20Server>



Tarea para la segunda entrega

- Opcional
 - Mejorar la implementación del *Action Client* para poder gestionar más estados posibles de un goal (aplazarlo, cancelarlo) y/o para poder gestionar mejor la comunicación con el servidor.
 - Por ejemplo, poder informar de que se ha recibido feedback, o poder configurar un timeout para la consecución del goal,...
 - Consultar cómo hacerlo en las referencias que aparecen en esta presentación
 - Mejorar la implementación del Action Server /Action Client para poder gestionar más estados posibles o más comportamientos, por ejemplo, detectar cuando el robot está atrapado en un mínimo local y tratar de sacarlo de ahí.



- GrupoLunes: 28 de Marzo a las 14:00
- GrupoMartes: 29 de Marzo a las 14:00