

# ROS Basic

Conceptos básicos de ROS

# Antecedentes

Patrones de Diseño

Patrón Observador, Publicador - Suscriptor

Ejemplos de herramientas que usan este patrón de diseño

# ¿Qué es un patrón de diseño?

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe el núcleo de la solución a ese problema, de tal manera que puedes utilizar esta solución millones de veces, sin nunca hacerlo de la misma forma dos veces.

Un patrón de diseño permite ofrecer una solución a problemas comunes en el diseño de software, describe la solución a problemas que se repiten muchas veces y que son muy similares entre ellos, en concreto, esta similitud permite diseñar una solución para un conjunto de problemas parecidos.

Los patrones de diseño actuales fueron popularizados por el libro **Design patterns: elements of reusable object-oriented software** de los autores conocidos como GoF (Gang of Four): Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

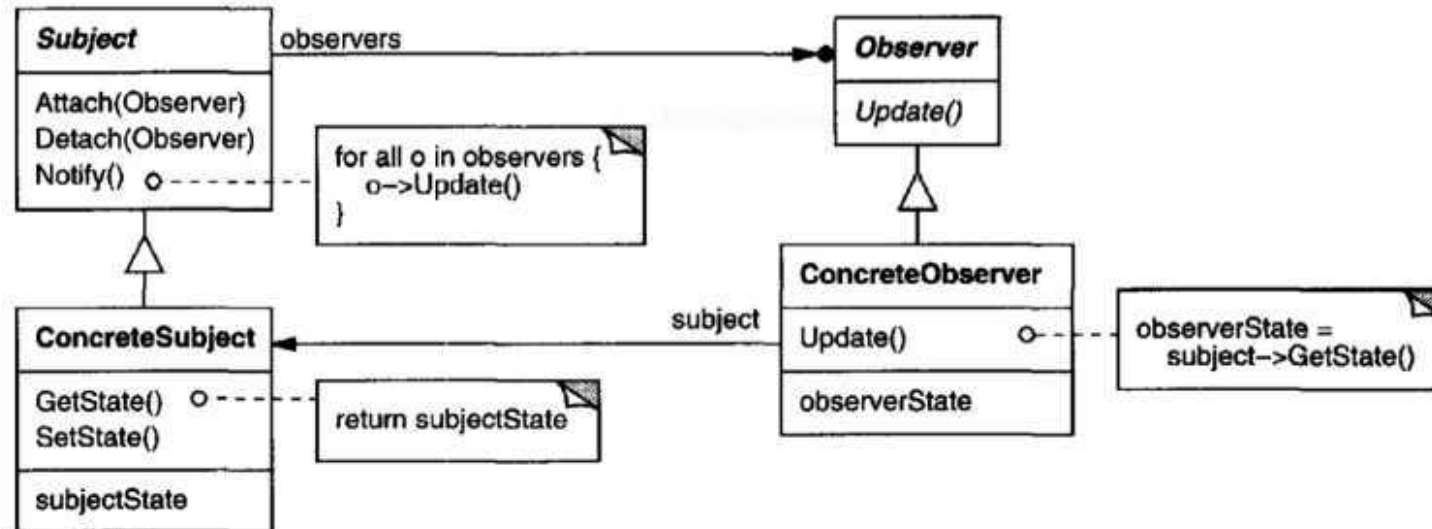
# Tipos de patrones de diseño

**Patrones de creación.** Estos patrones se utilizan cuando debemos crear objetos pero debemos tomar decisiones dinámicamente en el proceso de creación. Para ésto lo que hacemos es abstraer el proceso de creación de los objetos para realizar la decisión de qué objetos crear o cómo crearlos para el momento en que se tenga que hacer. Patrones de creación son: *Abstract Factory, Builder, Factory Method, Object Pool, Prototype y Singleton.*

**Patrones estructurales.** Nos describen como utilizar estructuras de datos complejas a partir de elementos más simples. Sirven para crear las interconexiones entre los distintos objetos y que estas relaciones no se vean afectadas por cambios en los requisitos del programa. Algunos ejemplos de patrones estructurales son: *Adapter, Bridge, Decorator, Facade, Flyweight y Proxy.*

**Patrones de comportamiento.** Fundamentalmente especifican el comportamiento entre objetos de nuestro programa. Hay varios: *Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento (o Snapshot), Observer, State, Strategy, Template Method y Visitor.*

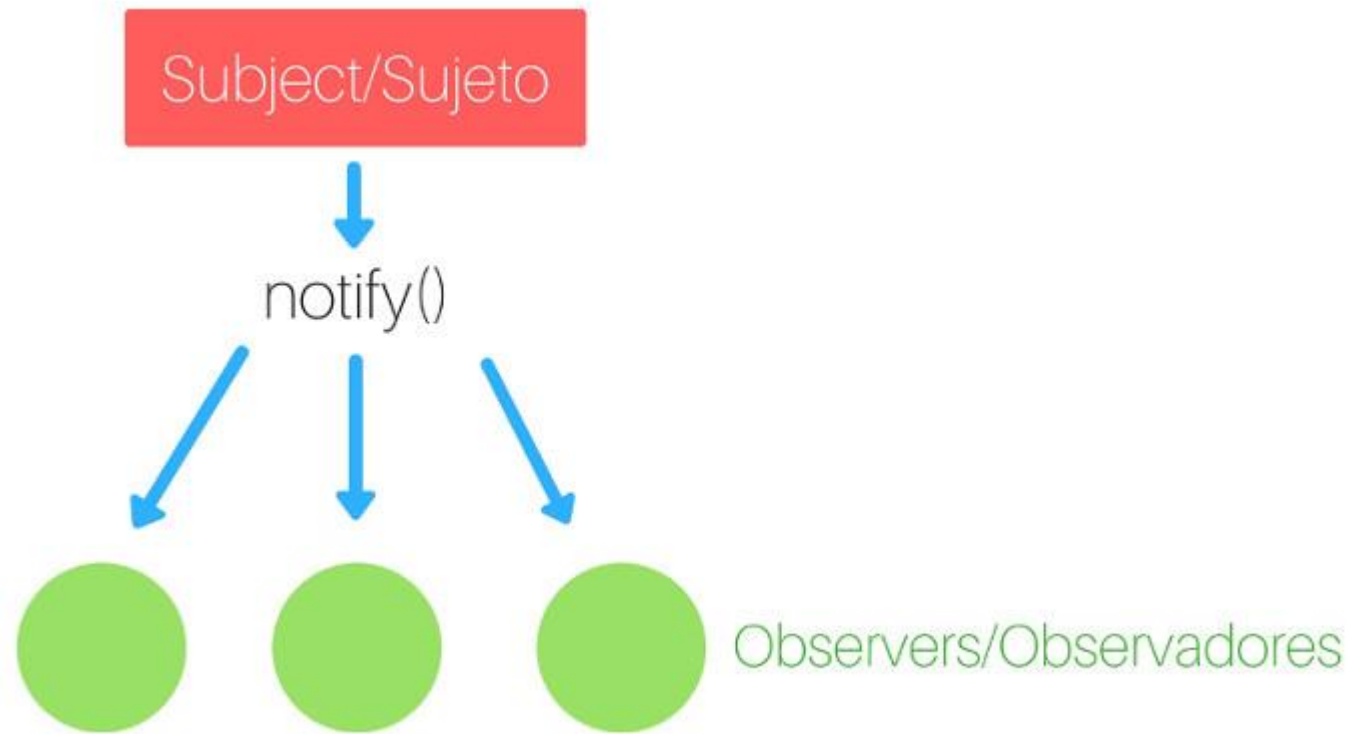
# Patrón observador



Se compone de un **Sujeto** y uno o más **Observadores**.  
El **Sujeto** está compuesto por las siguientes métodos:  
Una colección de **observers**

- Método **attach()**
- Método **detach()**
- Método **notify()**

# Simplificando el observador



# Publicador/suscriptor simple

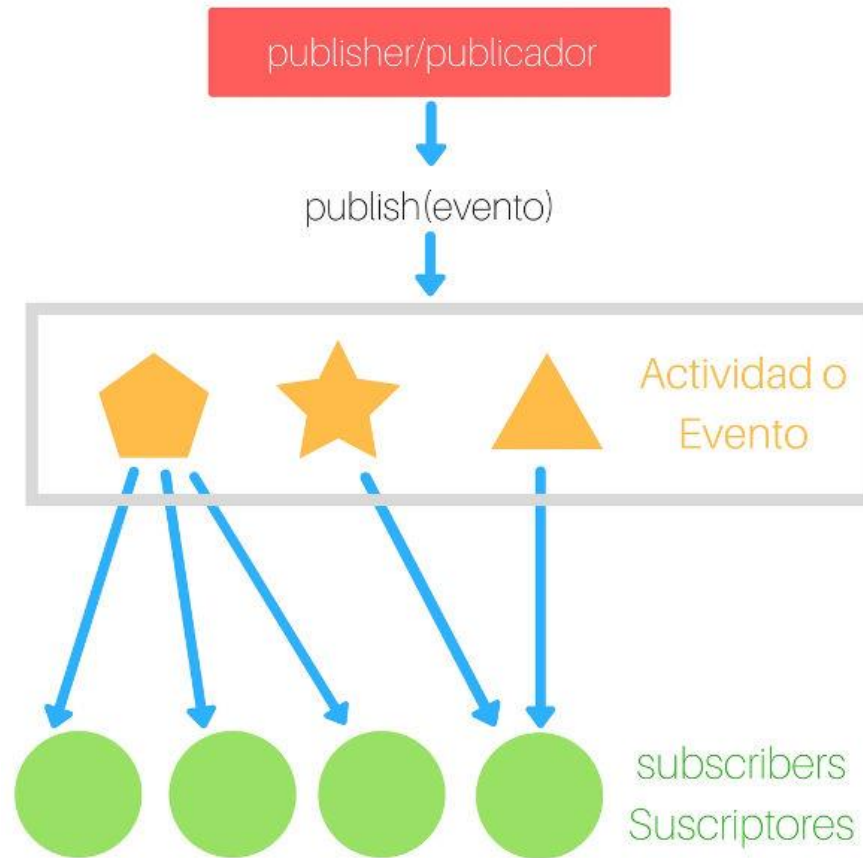
El patrón de diseño **publicador/suscriptor** es una variación del observador, en este modo el suscriptor u observador se suscribe a una actividad o evento del publicador o sujeto.

El **publicador** notifica a todos los objetos **suscritos** cuando el evento al que están interesados se dispara o **publica**.

El **publicador** está compuesto por las siguientes métodos:

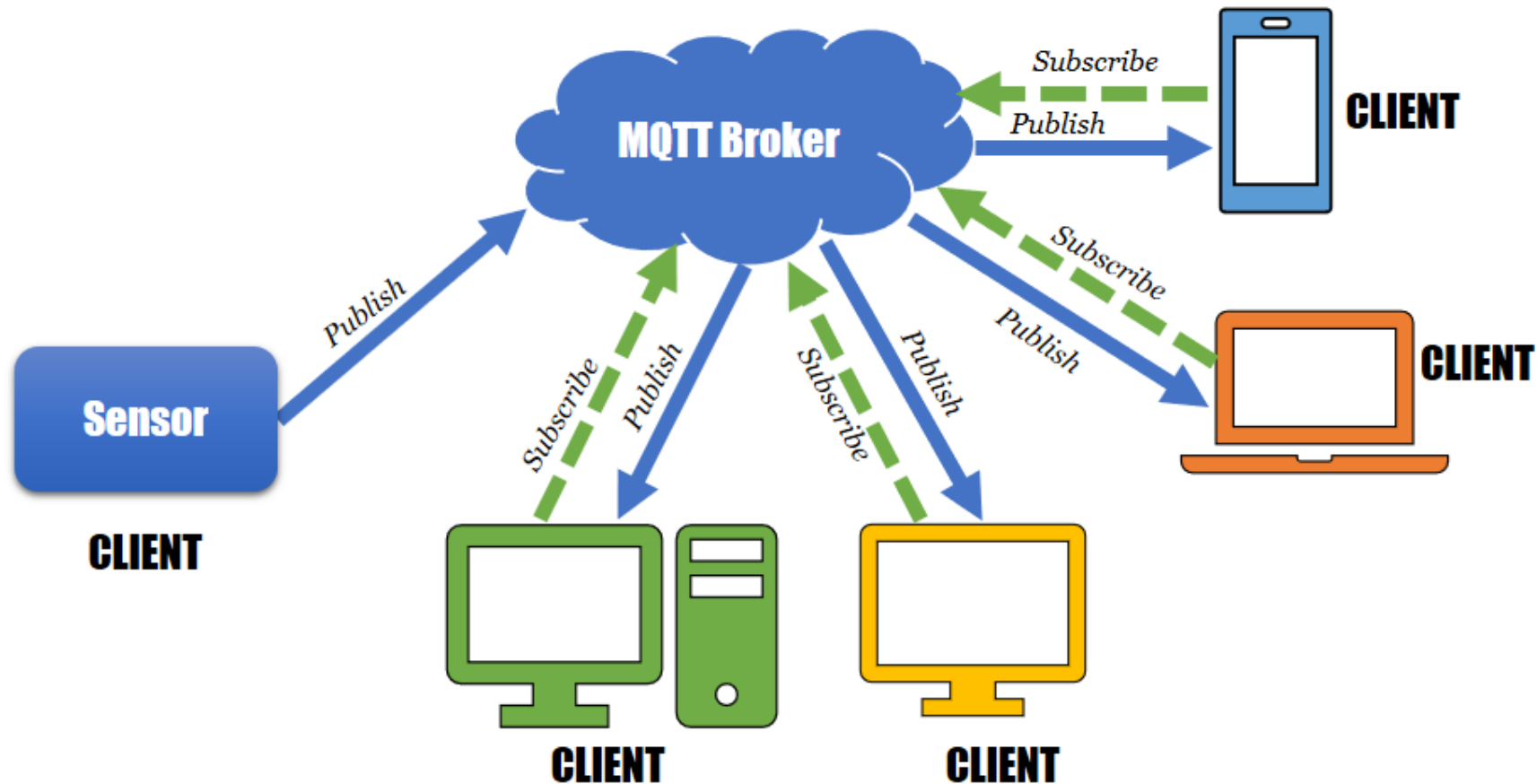
- Una colección de **observers** o **subscribers**
- Método **subscribe()** o **attach()**
- Método **unsubscribe()** o **detach()**
- Método **publish()** o **notify()**

# Diagrama del patrón publicador/suscriptor simple

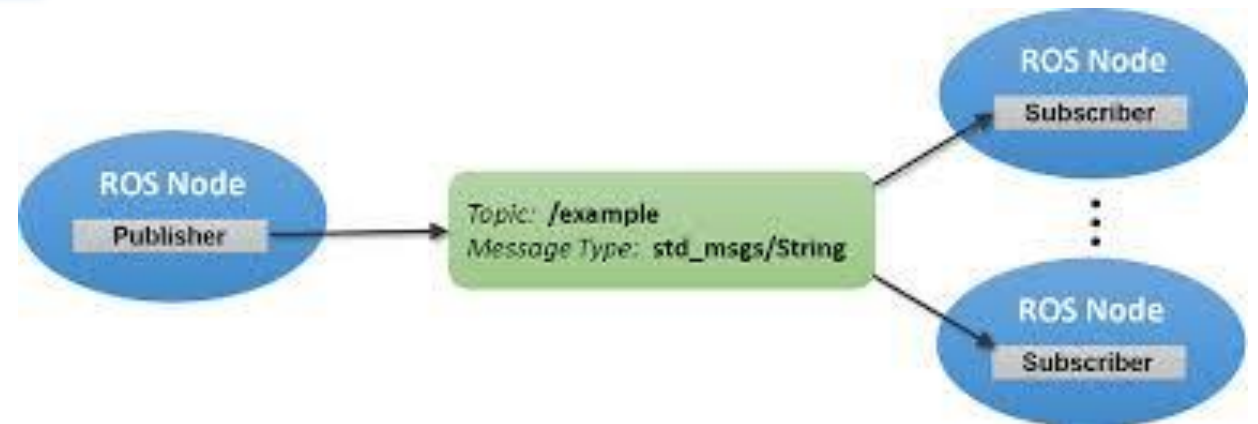
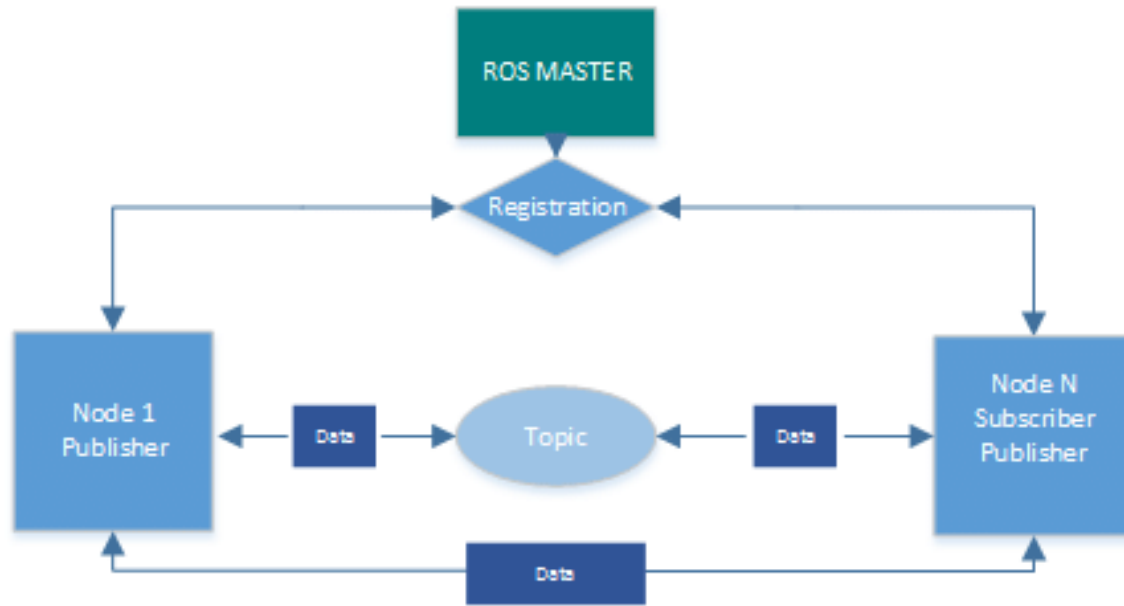




# Message Queuing Telemetry Transport - IoT

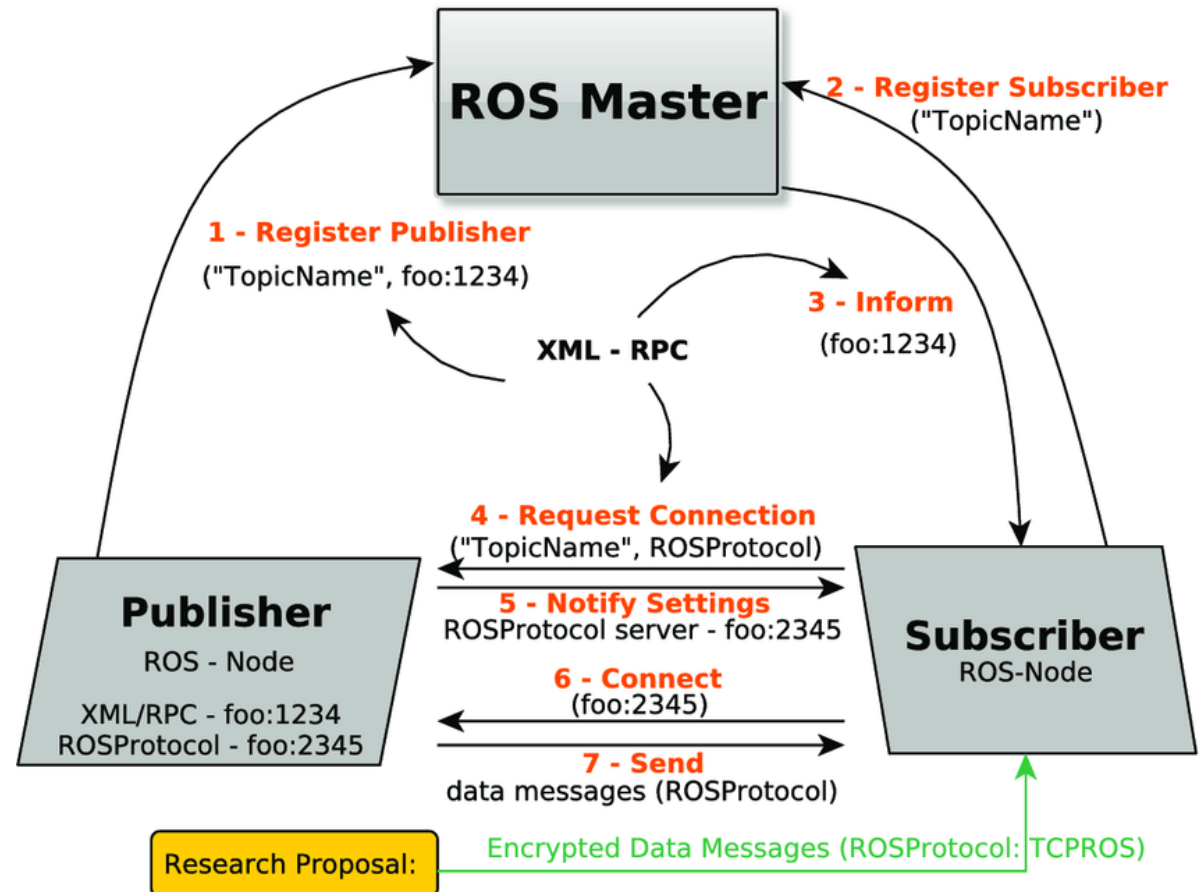


# Robot Operating System - Robótica



# ¿Qué es ROS master?

- Un servidor que mantiene un seguimiento de las direcciones de red de todos los nodos
  - Además de mantener el seguimiento de otra información como los parámetros
- Informa a los subscriptores acerca de los nodos que publican en el mismo tópico
- Publicador y subscriptor establecen una conexión **peer-to-peer (P2P)**
- Los nodos deben conocer la dirección de red de master al iniciar (**ROS\_MASTER\_URI**)
- Puede ser inicializado con **roscore** o **roslaunch** <package\_name> <launch\_file>



# ROS Packages

¿Qué son ROS Packages?

¿Cómo se crean?

Porque se necesitan, buenas prácticas y ejemplos

# ROS Package

El funcionamiento de los programas y los elementos necesarios para que interactúen dentro del entorno de **ROS**, están contenidos en carpetas denominadas **paquetes**

Para que un paquete sea considerado un paquete **catkin** debe cumplir algunos requisitos:

- El paquete debe contener un archivo **package.xml** compatible con **catkin**. Ese archivo **package.xml** proporciona metainformación sobre el paquete.
- El paquete debe contener un archivo **CMakeLists.txt** que utilice catkin. Si se trata de un **metapaquete catkin**, debe tener el archivo repetitivo **CMakeLists.txt** correspondiente.
- Cada paquete debe tener su propia carpeta. Esto significa que no hay paquetes anidados ni varios paquetes que compartan el mismo directorio.

# ROS Package

El paquete más simple posible podría tener una estructura similar a esta:

```
my_package/  
  -CMakeLists.txt  
  -package.xml
```

# ROS Package - Contenido

El método recomendado para trabajar con paquetes **catkin** es utilizar un espacio de trabajo (**catkin workspace**), pero también puede crear paquetes **catkin** de forma independiente. Un espacio de trabajo trivial podría verse así:

```
workspace_folder/      -- WORKSPACE
  src/                  -- SOURCE SPACE
    CMakeLists.txt      -- Archivo 'Toplevel' CMake, proporcionado por catkin
  package_1/
    CMakeLists.txt      -- CMakeLists.txt file para package_1
    package.xml          -- Manifiesto del paquete para package_1
  ...
  package_n/
    CMakeLists.txt      -- CMakeLists.txt file para package_n
    package.xml          -- Manifiesto del paquete para package_n
```

# ROS Package

Para crear un paquete hacemos uso del script `catkin_create_pkg`

Sintaxis:

```
catkin_create_pkg [nombre del paquete] <dependencias>
```

Ejemplo:

```
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```



# ROS Package – Best Practices

## Utilice un prefijo para la funcionalidad específica del robot

Agregue el prefijo "**my\_robot\_**" para los paquetes diseñados específicamente para su robot.

## Utilice un sufijo para describir el propósito de un paquete

Al final del nombre de un paquete, agregue una palabra que sea lo suficientemente específica como para describir lo que hace el paquete (por ejemplo, **my\_robot\_navigation**).

## Los nombres de los paquetes deben estar en minúsculas

Los nombres de los paquetes deben seguir las convenciones comunes de ***nomenclatura de variables de C***: minúsculas, comenzar con una letra, usar separadores de guiones bajos, por ejemplo: **mi\_robot\_navegación**.

# ROS Package – Best Practices (Cont.)

## **Facilite el mantenimiento**

Debería poder realizar cambios en un paquete sin romper otros paquetes.

## **Un paquete, un propósito**

Cada paquete debe tener un solo propósito.

## **Minimizar las dependencias**

Cada paquete sólo debe utilizar las dependencias que necesita.

# ROS Package – Best Practices (Ejemplo)

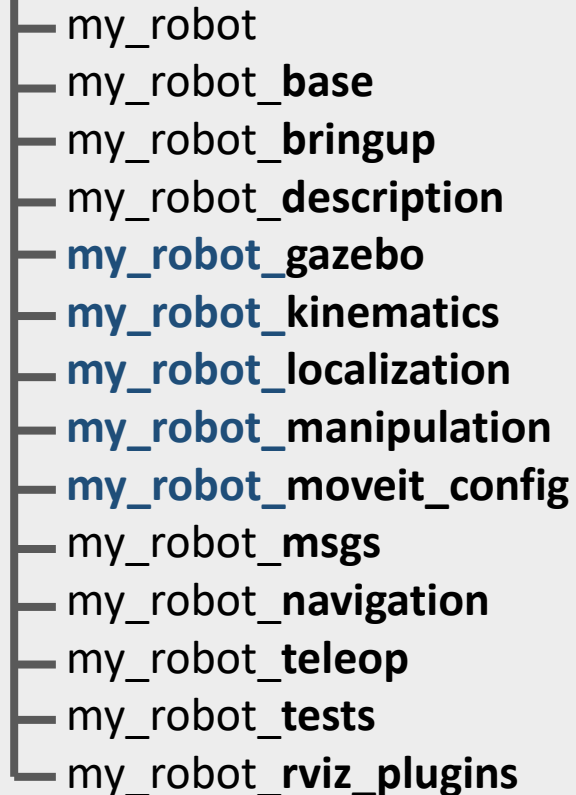
~/ros2\_ws/src/my\_robot/

```
— my_robot
— my_robot_base
— my_robot_bringup
— my_robot_description
— my_robot_gazebo
— my_robot_kinematics
— my_robot_localization
— my_robot_manipulation
— my_robot_moveit_config
— my_robot_msgs
— my_robot_navigation
— my_robot_teleop
— my_robot_tests
— my_robot_rviz_plugins
```

- **my\_robot:** Este paquete es un **metapaquete**. Un **metapaquete** no contiene nada excepto una lista de dependencias de otros paquetes. Puede usar un **metapaquete** para facilitar la instalación de varios paquetes relacionados a la vez.
- **my\_robot\_base:** Este paquete se utiliza para controlar los motores de su robot.
- **my\_robot\_bringup:** Contiene los archivos de inicio de ROS que abren el robot dentro de esta carpeta.
- **my\_robot\_description:** Contiene los archivos URDF y de malla de su robot.

# ROS Package – Best Practices (Ejemplo)

~/ros2\_ws/src/my\_robot/



- my\_robot
- my\_robot\_base
- my\_robot\_bringup
- my\_robot\_description
- my\_robot\_gazebo
- my\_robot\_kinematics
- my\_robot\_localization
- my\_robot\_manipulation
- my\_robot\_moveit\_config
- my\_robot\_msgs
- my\_robot\_navigation
- my\_robot\_teleop
- my\_robot\_tests
- my\_robot\_rviz\_plugins

- **my\_robot\_gazebo:** Archivos de configuración y lanzamiento para generar el robot en Gazebo.
- **my\_robot\_kinematics:** Los algoritmos de cinemática directa e inversa van aquí.
- **my\_robot\_localization:** Archivos para localizar al robot dentro de un entorno.
- **my\_robot\_manipulation:** Contiene algoritmos para manipular objetos en el entorno.
- **my\_robot\_moveit\_config:** Archivos de configuración utilizando **MoveIt**.

# ROS Package – Best Practices (Ejemplo)

~/ros2\_ws/src/my\_robot/

```
— my_robot
— my_robot_base
— my_robot_bringup
— my_robot_description
— my_robot_gazebo
— my_robot_kinematics
— my_robot_localization
— my_robot_manipulation
— my_robot_moveit_config
— my_robot_msgs
— my_robot_navigation
— my_robot_teleop
— my_robot_tests
— my_robot_rviz_plugins
```

- **my\_robot\_msgs:** Contiene mensajes, servicios y acciones personalizados.
- **my\_robot\_navigation** Contiene archivos de configuración y de inicio para la pila de navegación ROS.
- **my\_robot\_teleop:** Un nodo para teleoperar manualmente un robot usando un teclado, joystick, controlador de consola de juegos, etc.
- **my\_robot\_tests:** Un paquete utilizado para pruebas del sistema.
- **my\_robot\_rviz\_plugins:** Los complementos específicos de **RViz** van aquí.

# ROS Nodes, ROS Topics y ROS Messages

¿Qué son?

Publicadores, subscriptores y tipos de mensajes

Ejemplos y consideraciones

# ROS Publishers y subscribers (nodos)



Node 1

Odometría  
Rueda

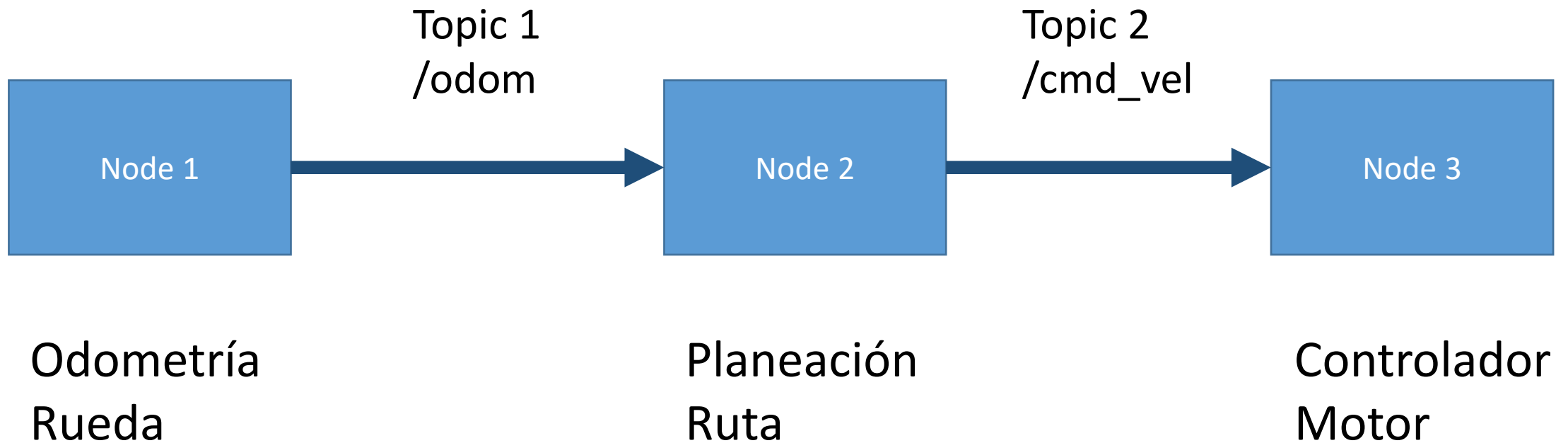
Node 2

Planeación  
Ruta

Node 3

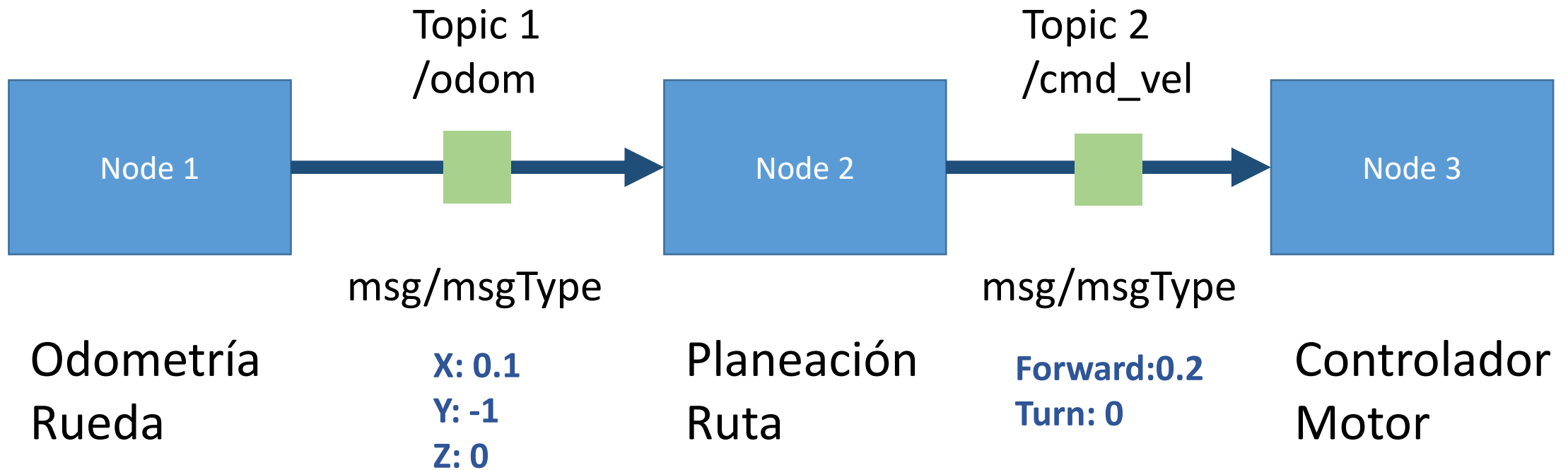
Controlador  
Motor

# ROS Publishers y subscribers (Tópicos)





# ROS Publishers y subscribers (mensajes)



# Reglas para publicar y suscribirse

- ☐ Cualquier nodo puede publicar (enviar) un mensaje a cualquier tópico
- ☐ Cualquier nodo puede suscribirse (recibir) a cualquier tópico
- ☐ Múltiple nodos pueden publicar al mismo tópico
- ☐ Múltiples nodos pueden suscribirse a el mismo tópico
- ☐ Un nodo puede publicar a múltiples tópicos.
- ☐ Un nodo puede suscribirse a múltiples tópicos

# Publish/subscribers – **node** *command tools*

- **roscpp** **list** lista los nodos activos
- **roscpp** **info** /<nombre\_nodo> muestra la información del nodo
- **roscpp** **ping** /<nombre\_nodo> prueba la conectividad al nodo
- **roscpp** **kill** /<nombre\_nodo> detiene la ejecución de un nodo
- **roscpp** **cleanup** purga la información de registro de nodos inalcanzables

# Publish/subscribers – **topic** *command tools*

- **rostopic list** lista los tópicos activos
- **rostopic info** /<nombre\_topic> muestra la información sobre un tópico activo
- **rostopic echo** /<nombre\_topic> imprime los mensajes en pantalla
- **rostopic pub** /<nombre\_topic> msg/MessageType “data: value” publica datos (message data) al tópico

# Mensajes

Mensaje.msg

```
# Esquema (comentario)
[tipo de dato] [nombre campo]
```

```
# Ejemplos
```

```
string    campo1
float32    campo2
boolean    campo3
int32      campo4
```

- ❑ Es un formato serializado para estructuras de datos.
- ❑ Permite a los nodos programados en C++ y Python comunicarse entre ellos.
- ❑ Está definido en un archivo .msg
- ❑ Las clases deben estar compiladas en C++/Python antes de usarlas.

# ROS Parameters

¿Qué son y por qué se necesitan?

ROS Parameter Server

# ¿Qué son y por qué se necesitan?

## Problema:

Supongamos que tenemos una aplicación para un robot, compuesta de muchos paquetes y nodos en cada uno de ellos.

Ahora, necesitamos crear configuraciones globales, por ejemplo:

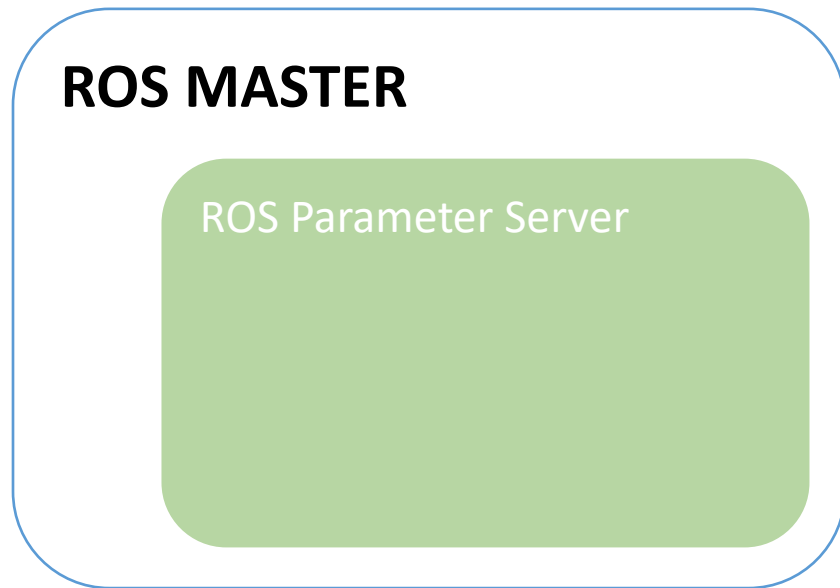
- ☐ Nombre del robot.
- ☐ Frecuencia a la que se lee algunos sensors.
- ☐ Una bandera de simulación que indique a todos los nodos si el robot se está ejecutando en modo real o modo simulación.

Es Seguro que no se quiere poner en “**código duro**” (*hardcoding*) éstos valores y tampoco se quiere tener demasiadas dependencias inútiles entre los nodos.

Además, es deseable que un nodo pueda iniciar con diferentes parámetros de entrada, sin tener que cambiar el código del nodo y, aún peor, recompilar los códigos de los nodos en C++.

Es fácil darse cuenta en éste momento, que necesitamos un **diccionario global** para las configuraciones compartidas en la aplicación, que se pueda recuperar en **tiempo de ejecución** al iniciar los nodos.

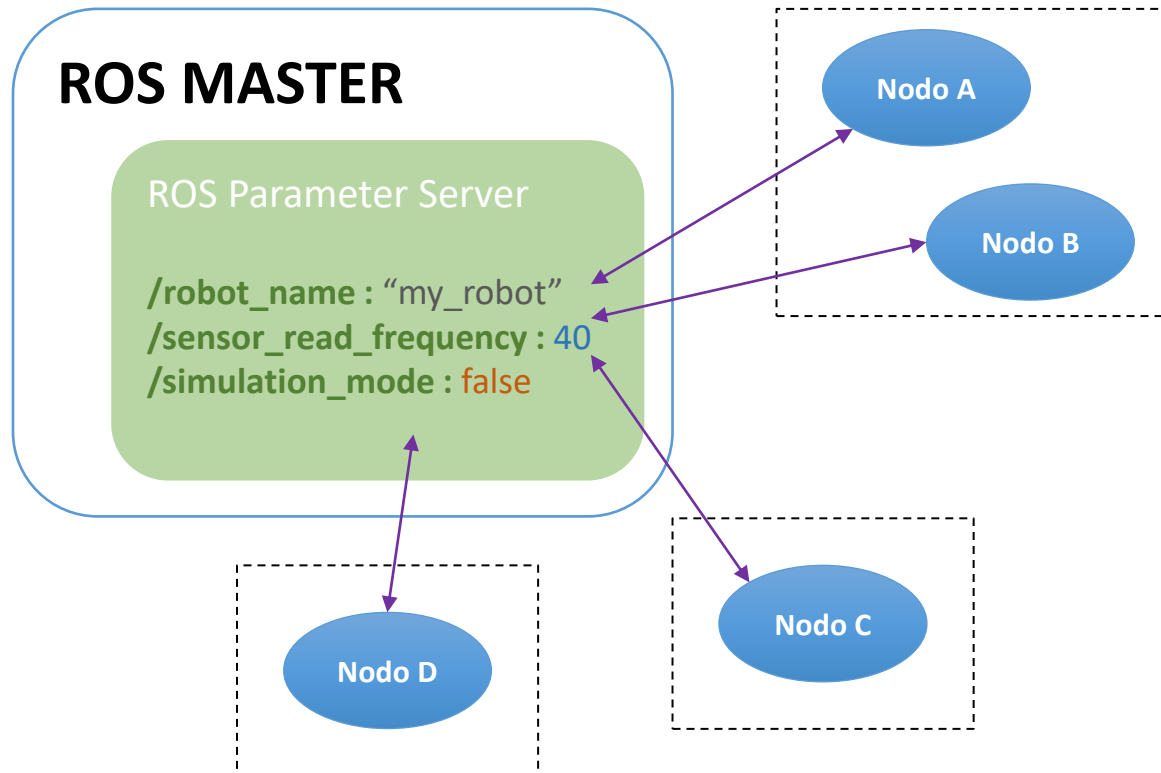
# ROS Parameter Server



- Después de iniciar ROS Master, el servidor de parámetros se crea automáticamente dentro del ROS Master.
- El **ROS Parameter Server** es básicamente **un diccionario que contiene variables globales** a las que se puede acceder desde cualquier lugar del entorno ROS actual.
- Esas **variables globales** se denominan **ROS Parameters**.



# ROS Parameter Server



En la figura se muestran tres parámetros:

- **robot\_name** (tipo string)
- **sensor\_read\_frequency** (tipo entero)
- **simulation\_mode** (tipo booleano)

En cualquier momento, **un nodo puede leer un parámetro, modificar un parámetro y crear otros nuevos.**

En este ejemplo, se muestran 4 nodos en 3 paquetes diferentes. Como se aprecia, cualquier nodo de cualquier paquete tiene acceso al **ROS Parameter Server**.

Cuando el **nodo A** es iniciado, puede agregar nuevos parámetros en el servidor de parámetros. Si el **nodo B** inicia después del **nodo A**, el **nodo B** tendrá acceso a éste nuevo parámetro.

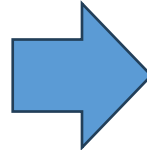
# ROS Parameter Server – command tools

- **rosparam list** lista los nombres de los parámetros
- **rosparam set** /<nombre\_parametro> <valor> coloca el valor del parámetro
- **rosparam get** /<nombre\_parametro> obtiene el valor del parámetro
- **rosparam load** </ruta/archivo.yaml> carga los parámetros desde un archivo
- **rosparam dump** </ruta/archivo.yaml> arroja los parámetros del servidor de parámetros a un archivo
- **rosparam delete** /<nombre\_parametro> elimina un parámetro

# ROS Parameter Server – archivo YAML

## Parametros.yaml

```
# Esquema (comentario)
campo1: 'foo' # string
campo2: 1234 # integer
campo3: 1234.5 # float
campo4: true # boolean
campo5: # list
- 1.0
- "mixed list"
campo6: # dictionary
  a: 'b'
  c: 'd'
```



## ROS Parameter Server

```
/campo1 'foo'
/campo2 1234
/campo3 1234.5
/campo4 true
/campo5: "[1.0, 'mixed list']"
/campo6/ "a: 'b' c: 'd' "
```

# ROS Services

¿Qué son?

¿Cuándo usar servicios?

Consideraciones al usar servicios

# ¿Qué son ROS services?

- ❑ Son llamadas a procesos remotos de ROS (RPC)
- ❑ Un nodo puede implementar uno o más servicios (server)
- ❑ Cualquier nodo puede llamar a un servicio (client)
- ❑ Las llamadas son **síncronas / blocking**
  - ❑ Acciones (Actions) se usan preferentemente para ejecutar tareas largas



# Service messages

- ❑ Deben ser definidos en un archivo **.srv** describiendo los parámetros del mensaje de la petición (**request message**) y el mensaje de respuesta (**response message**)

Ejemplo: **GetDistance.srv**

```
# request message
string name

---

# response message
float64 distance
```

# ¿Cuándo usar servicios?

- ❑ En muchos de los casos, es mejor escribir una librería de funciones
- ❑ Operaciones rápidas
  - ❑ Para funciones largas es mejor utilizar **Actions**
- ❑ Interoperatibilidad entre C++ / Python
- ❑ Cuando la computadora está conectada al hardware real

# Service tools

- **rosservice list**
- **rosservice info** /<some\_service>
- **rosservice call** /<some\_service> “param1: value”
- **rosservice type** /<some\_service>
- **rossrv show** my\_msg/ServiceName



# ROS Bags

¿Qué son?

# ROS bags

- ❑ Un archivo **.bag** es usado para almacenar los datos de los mensajes generados por los tópicos y servicios
- ❑ Se crean usando el commando **roscap**, el cual puede subscribirse a uno o más tópicos almacenándolos en el archivo.
- ❑ Este archivo puede volver a ejecutar los mismos tópicos en el orden en que fueron.
- ❑ La función principal de los **roscap** es llevar un registro de los datos generados durante el proceso para posteriormente visualizarlos y procesarlos fuera de línea.

# ROS bag tools

- **rosvag record** [topic\_1] [topic\_2] --duration=[value] --output-name=[bag\_name]
- **rosvag play** [bag\_name]
- **rosvag info** [bag\_name]
- **rosvag check** [bag\_name]
- Descripción más detallada de los comandos y acciones de los archivos bag pueden ser encontradas en:

<http://wiki.ros.org/rosvag/Commandline>

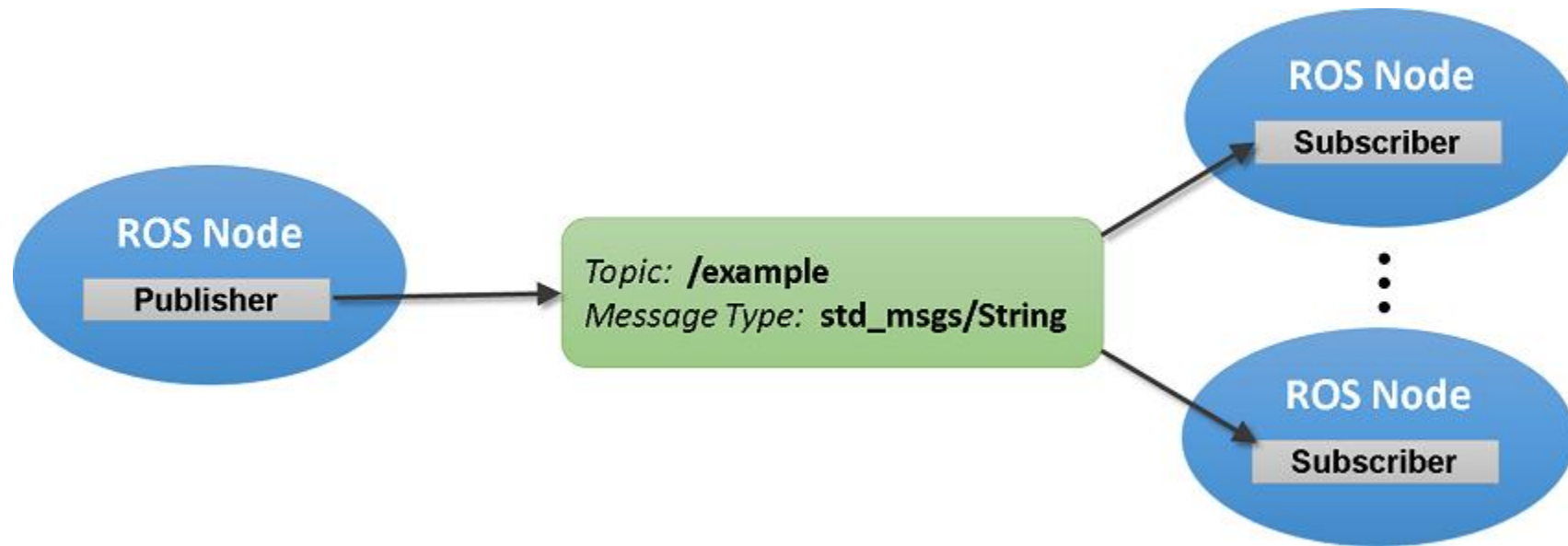
# ROS Actions

¿Por qué se necesitan?

¿Qué son ROS Actions?

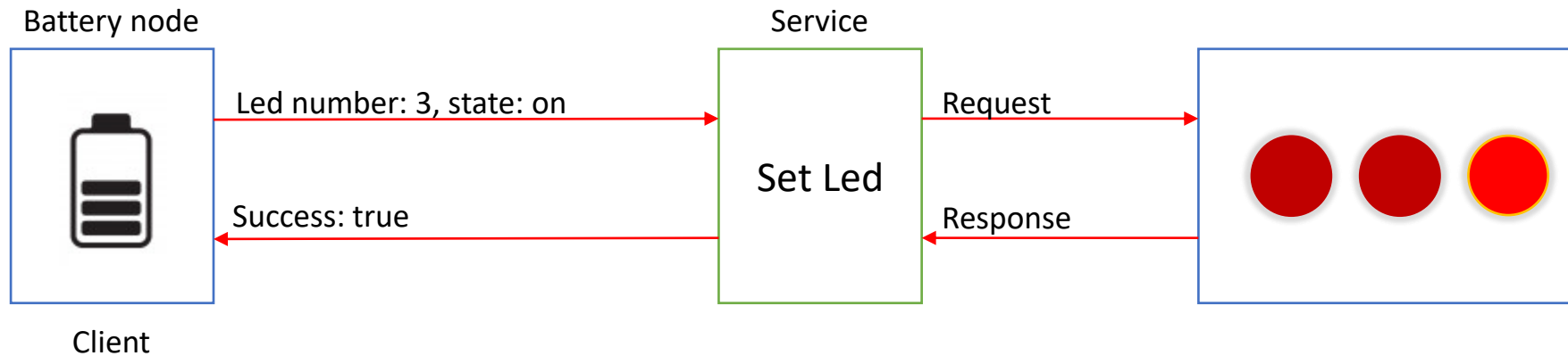
# ROS Actions, ¿Por qué se necesitan?

- Herramientas de comunicación de ROS
  - Topics

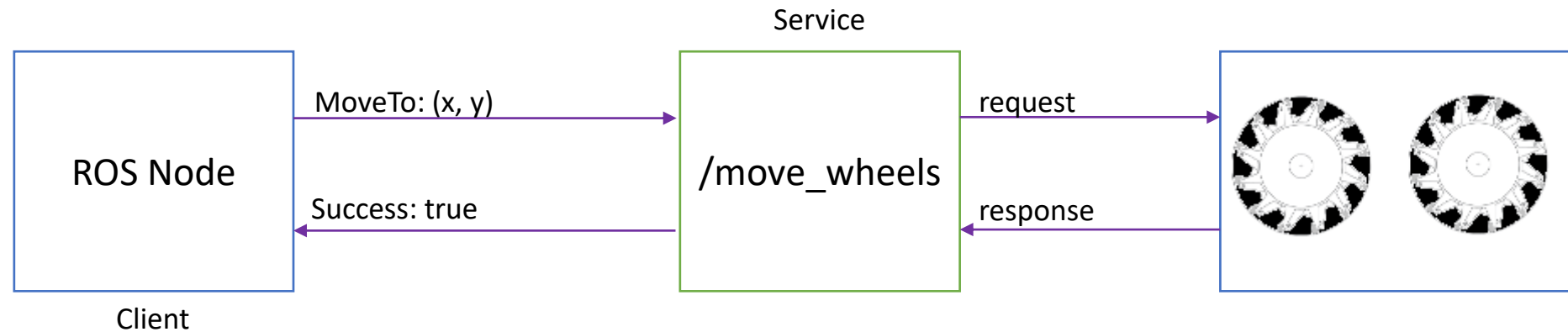


# ROS Actions, ¿Por qué se necesitan?

- Herramientas de comunicación de ROS
  - Topics
  - Services



# ROS Actions - Problema



→ El tiempo de ejecución puede ser prolongado, el cliente esta “atorado” esperando.

- ¿Es posible cancelar (**cancel**) la ejecución?
- ¿Cómo tener retroalimentación (**feedback**) de el servidor?
- ¿Cómo puede el servidor manejar múltiples metas (**goals**)?

# ROS Actions - Problema

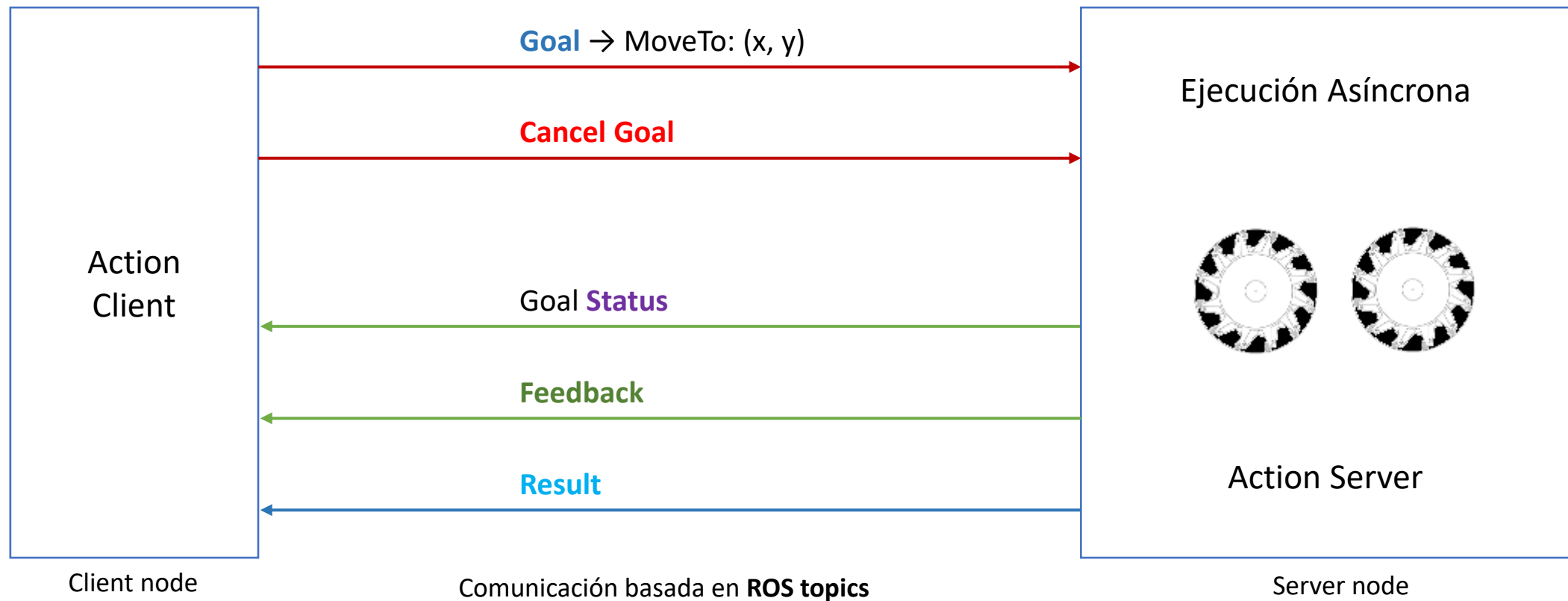
Problemas con Servicios

→ **Síncronos**

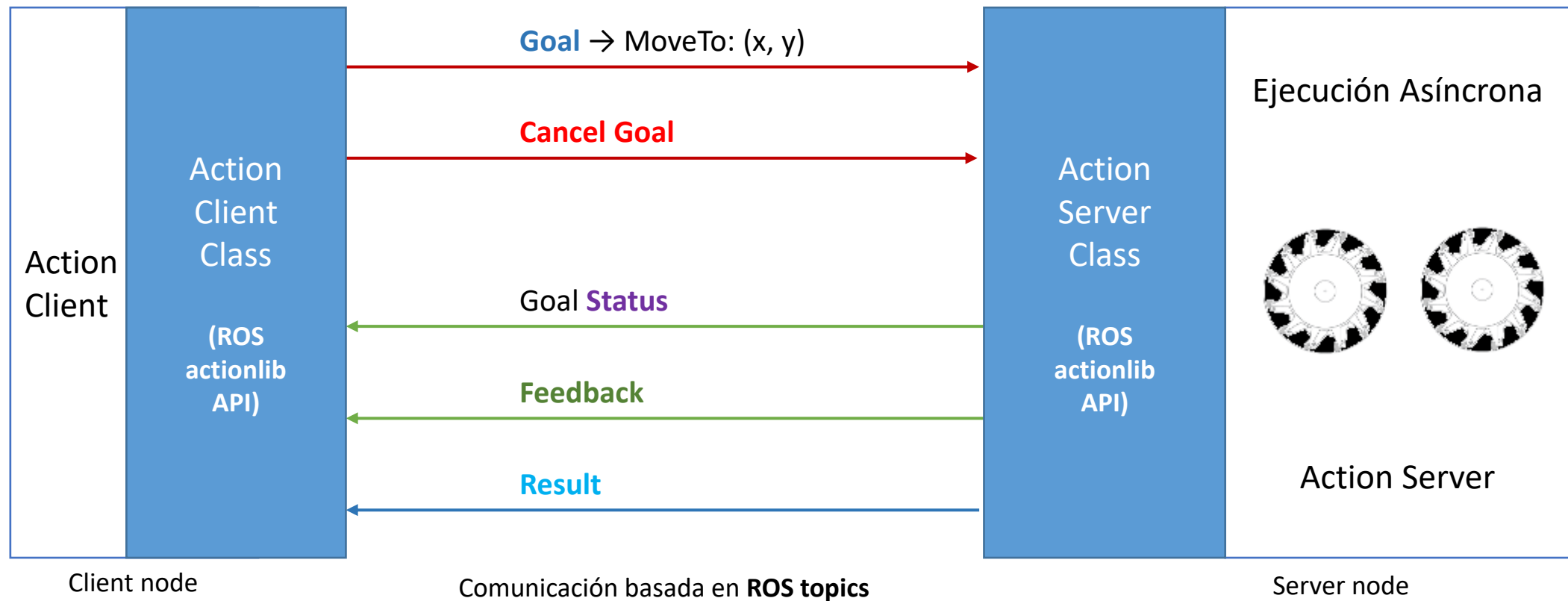
→ Están diseñados sólo para **acciones o cálculos rápidos**



# ROS Actions, ¿Qué son?



# ROS Actions, ¿Qué son?



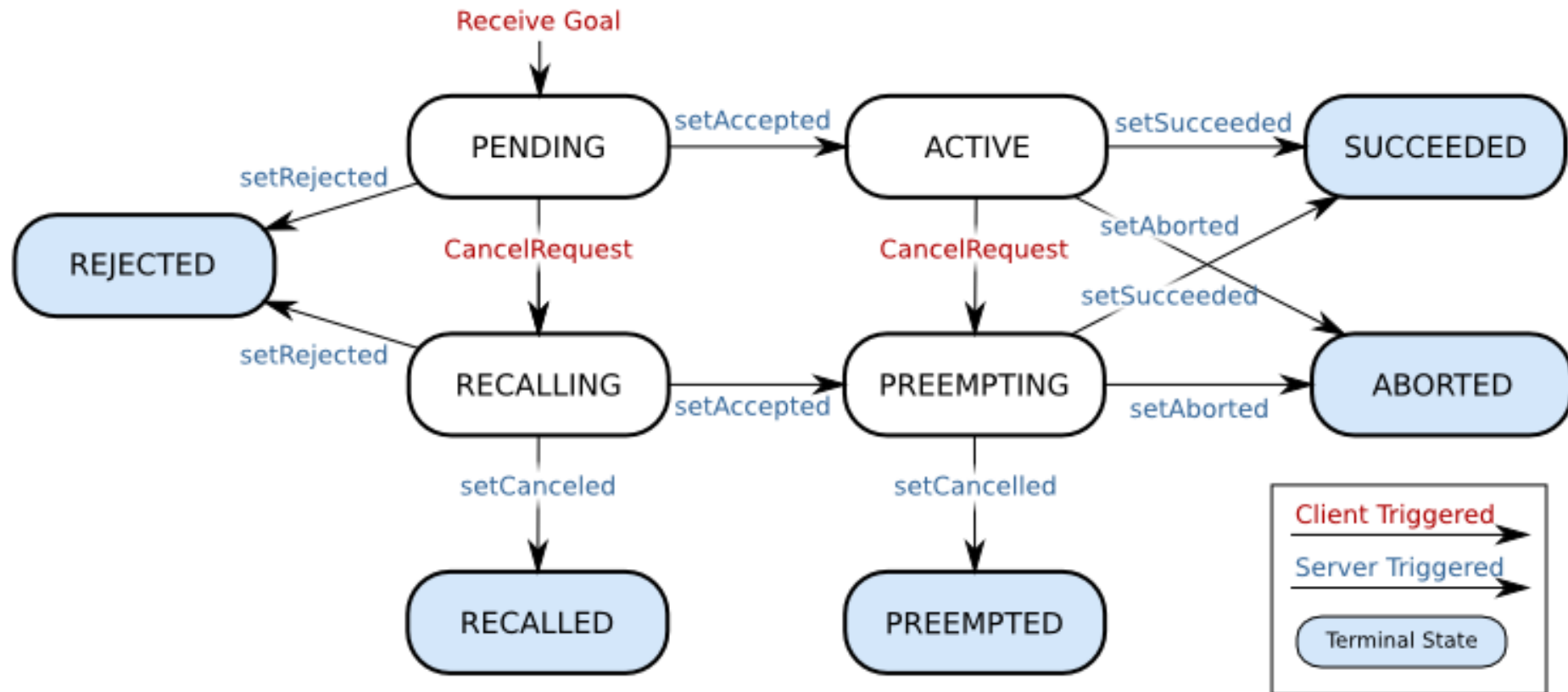
# ROS Actions

Con **ROS Actions** es posible:

- Ejecutar tareas de manera asíncrona
- Obtener el **status** (estado) y **feedback** (retroalimentación) del **goal** (meta) ejecutándose actualmente.
- **Cancel** (cancelar) un **goal** (meta).

# Actionlib - Server State Transitions

## Server State Transitions



## Client State Transitions

# Actionlib - Client State Transitions

