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.

Lo 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

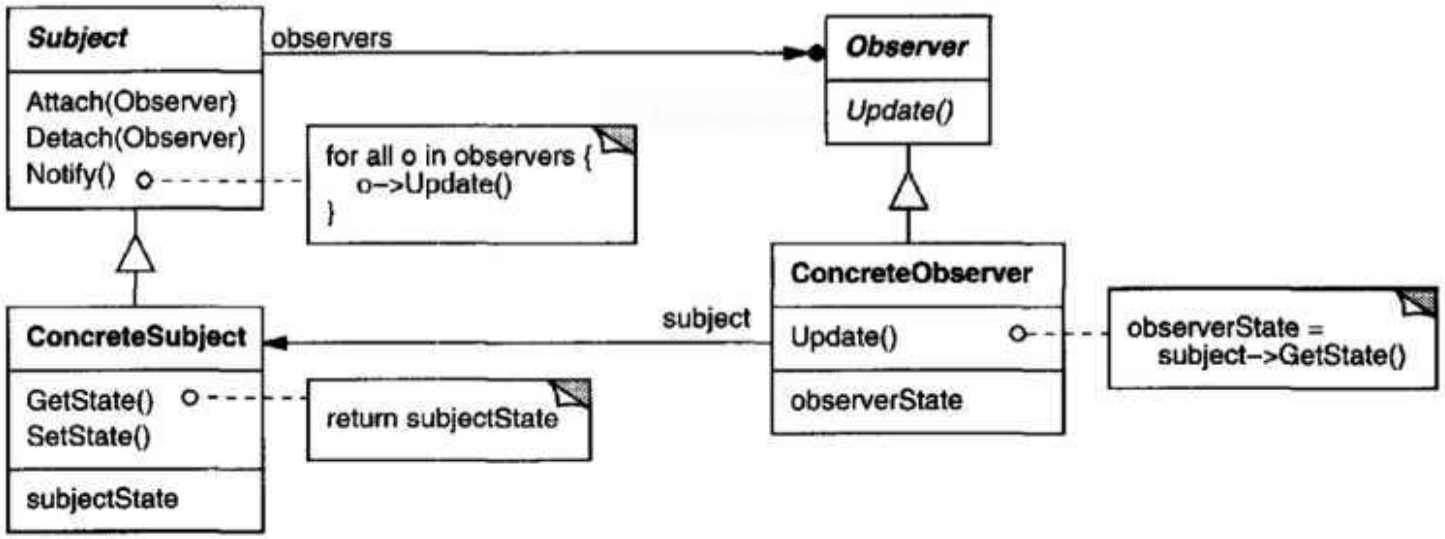
Tipos de patrones de diseño

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 Responsability, Command, Interpreter, Iterator, Mediator, Memento (o Snapshot), Observer, State, Strategy, Template Method y Visitor.*

Patrón observador



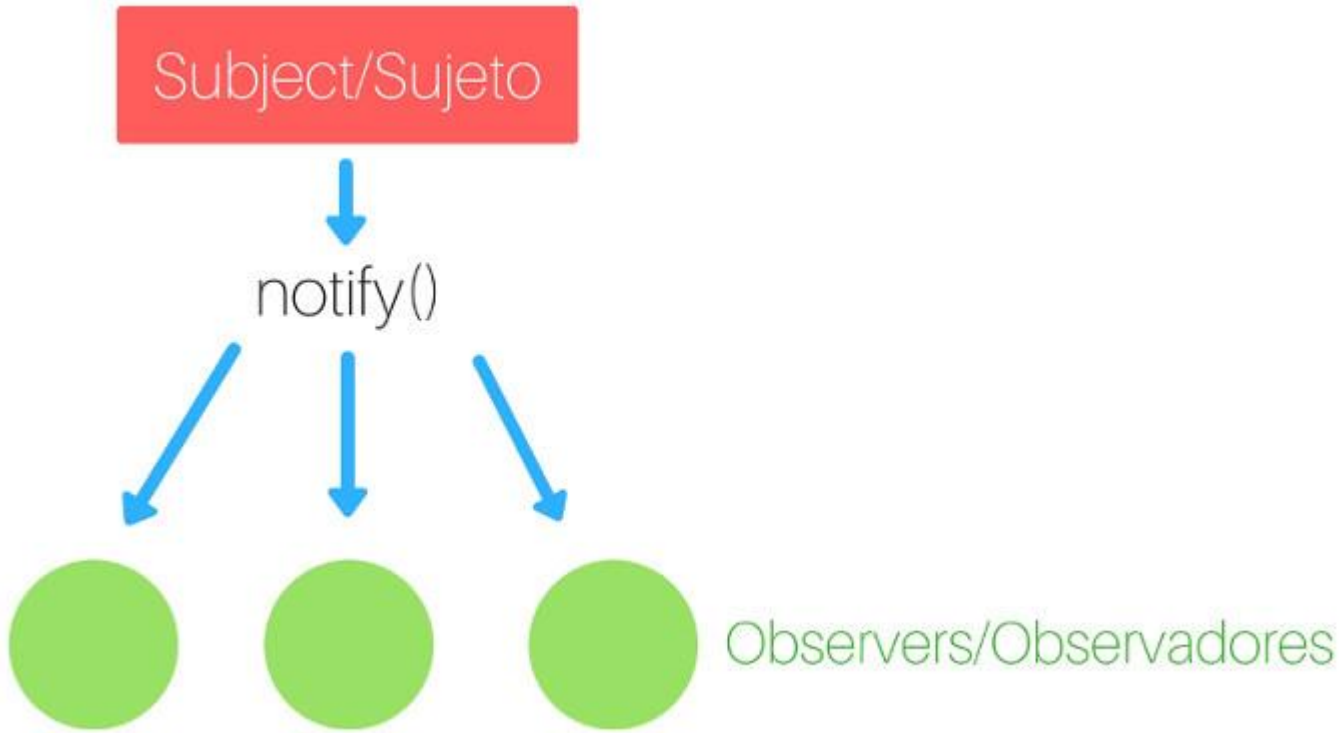
Se compone de un **Sujeto** y uno o más **Observadores**.

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

Una colección de **observers**

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

Simplificando el observador



Publicador/suscriptor simple

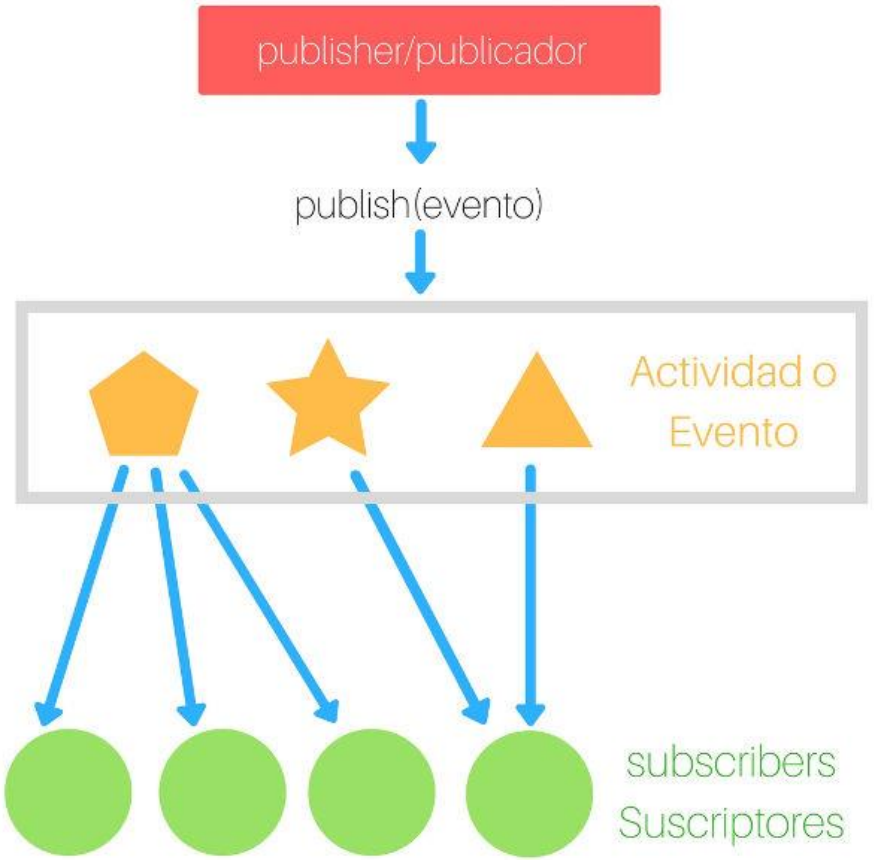
El patrón de diseño **publicador/suscriptor** es una variación del observador, en este modo el suscriptor u observador se suscribe a una actividad o evento del publicador o sujeto.

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

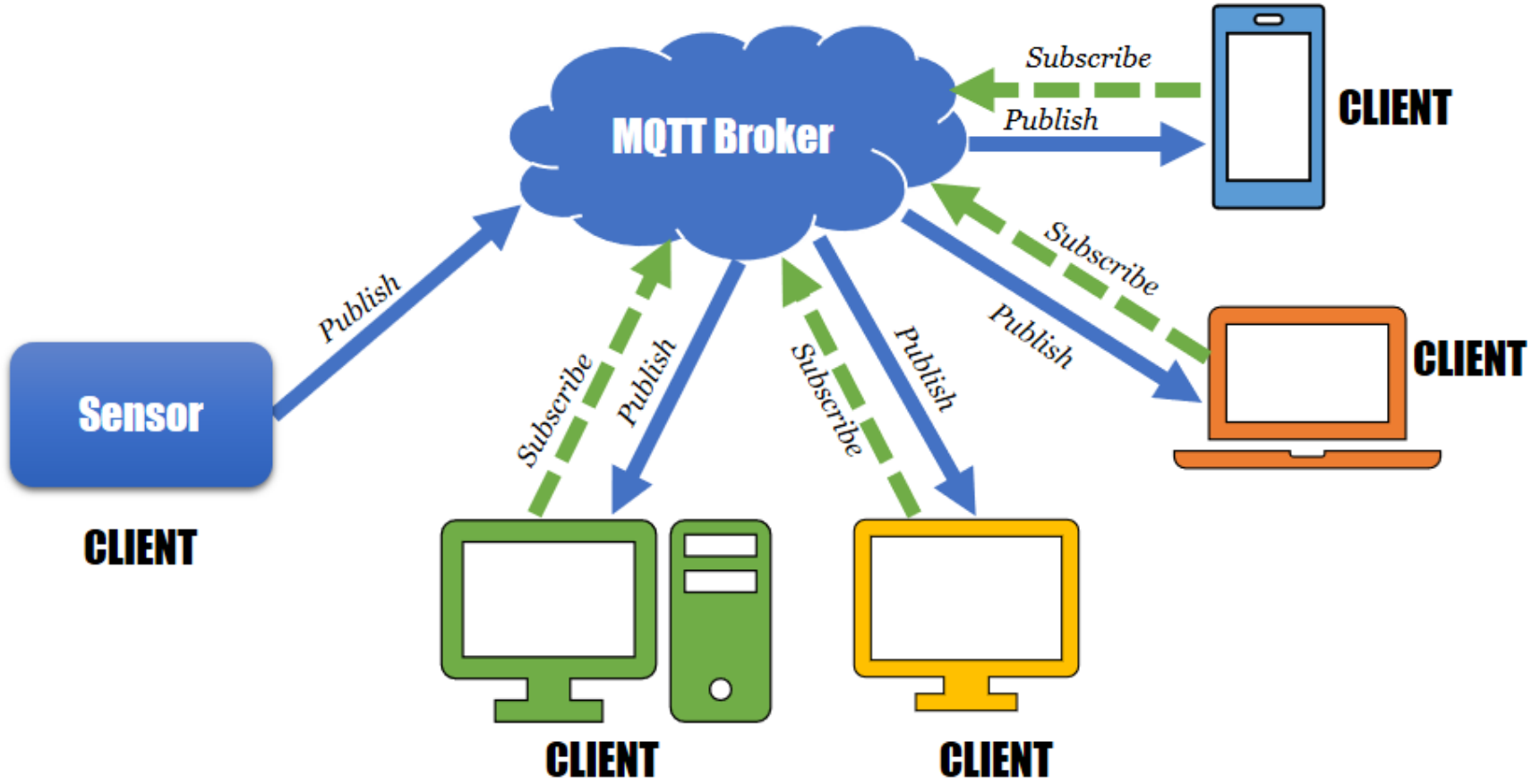
El **publicador** está compuesto por las siguientes métodos:

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

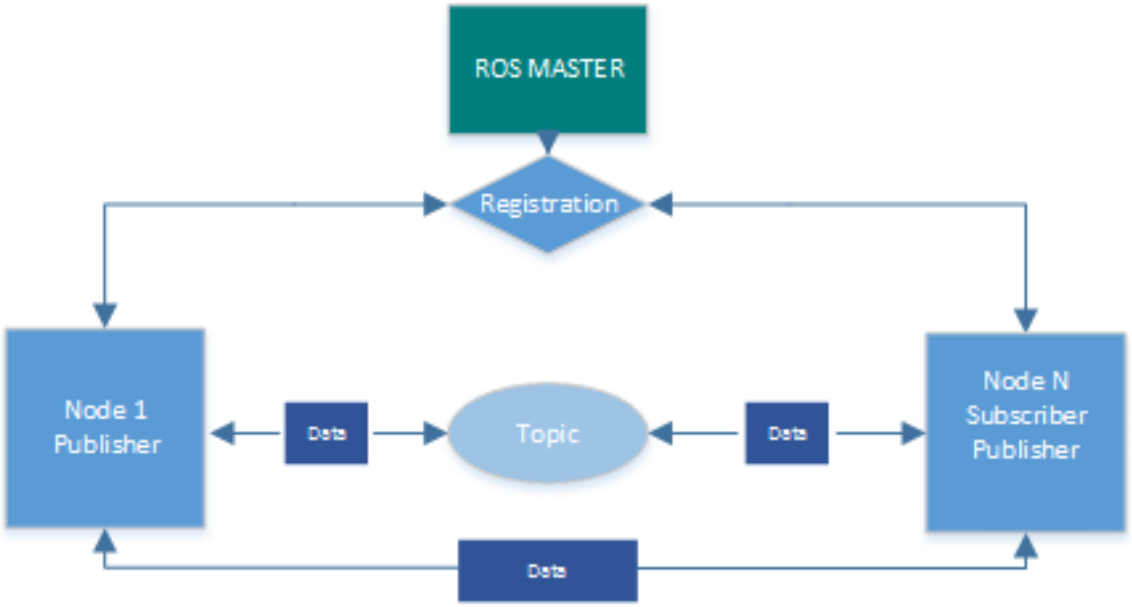
Diagrama del patrón publicador/suscriptor simple

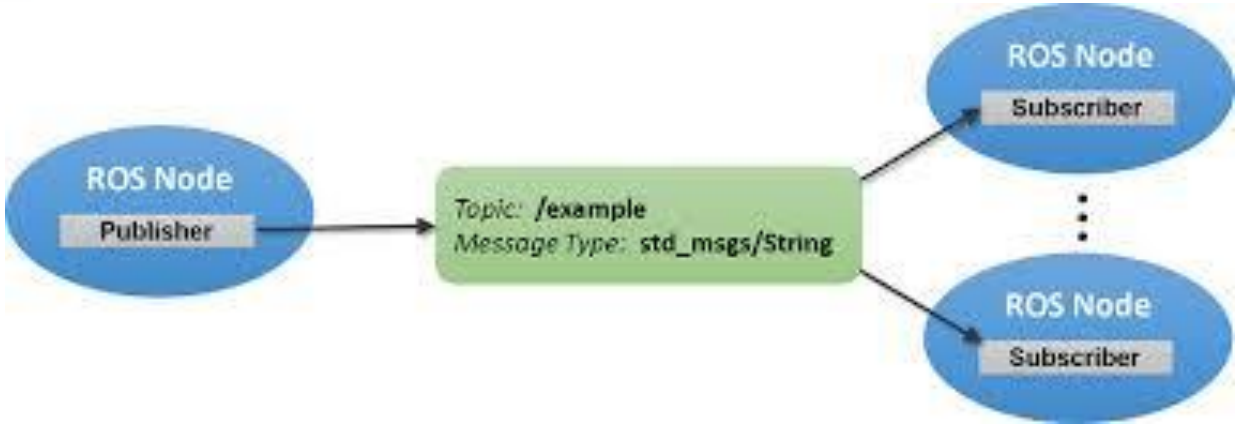


Message Queuing Telemetry Transport - IoT



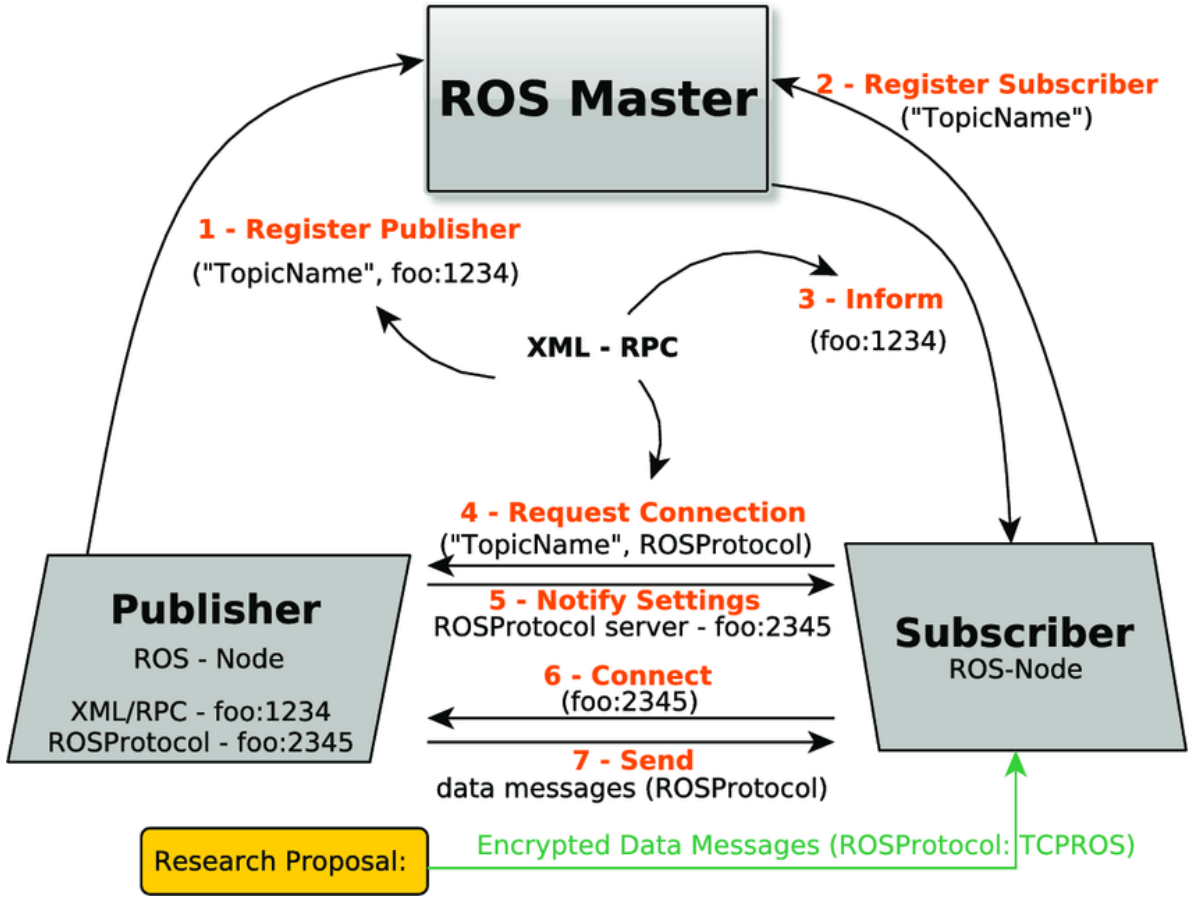
Robot **O**perating **S**ystem - Robótica





¿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.

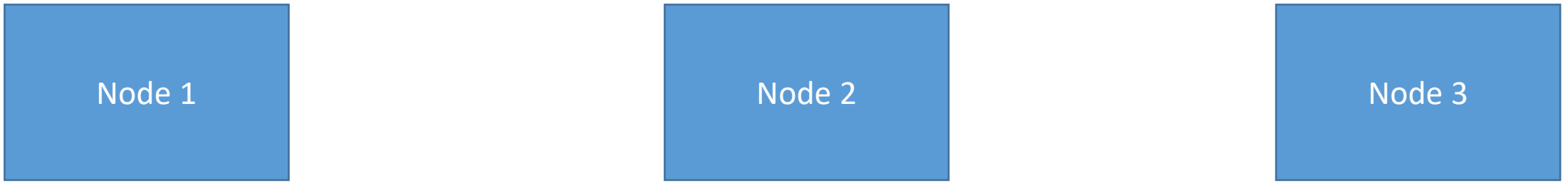
ROS Nodes, ROS Topics y ROS Messages

¿Qué son?

Publicadores, subscriptores y tipos de mensajes

Ejemplos y consideraciones

ROS Publishers y subscribers (nodos)

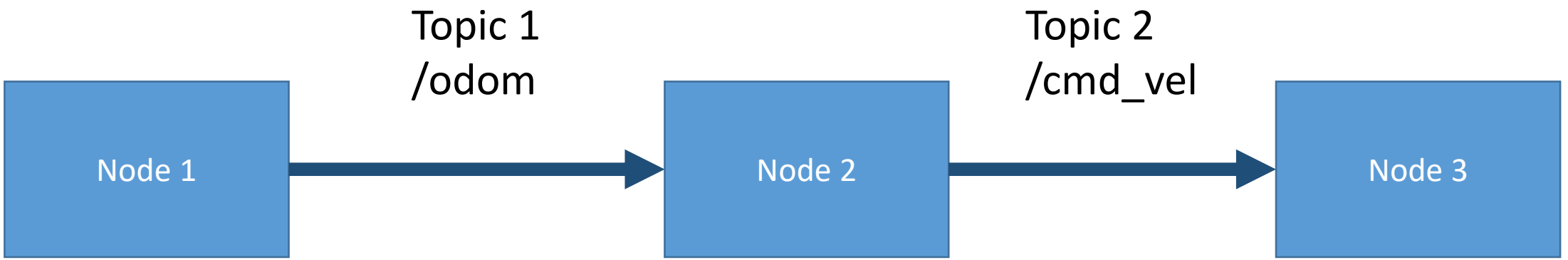


Odometría
Rueda

Planeación
Ruta

Controlador
Motor

ROS Publishers y subscribers (Tópicos)

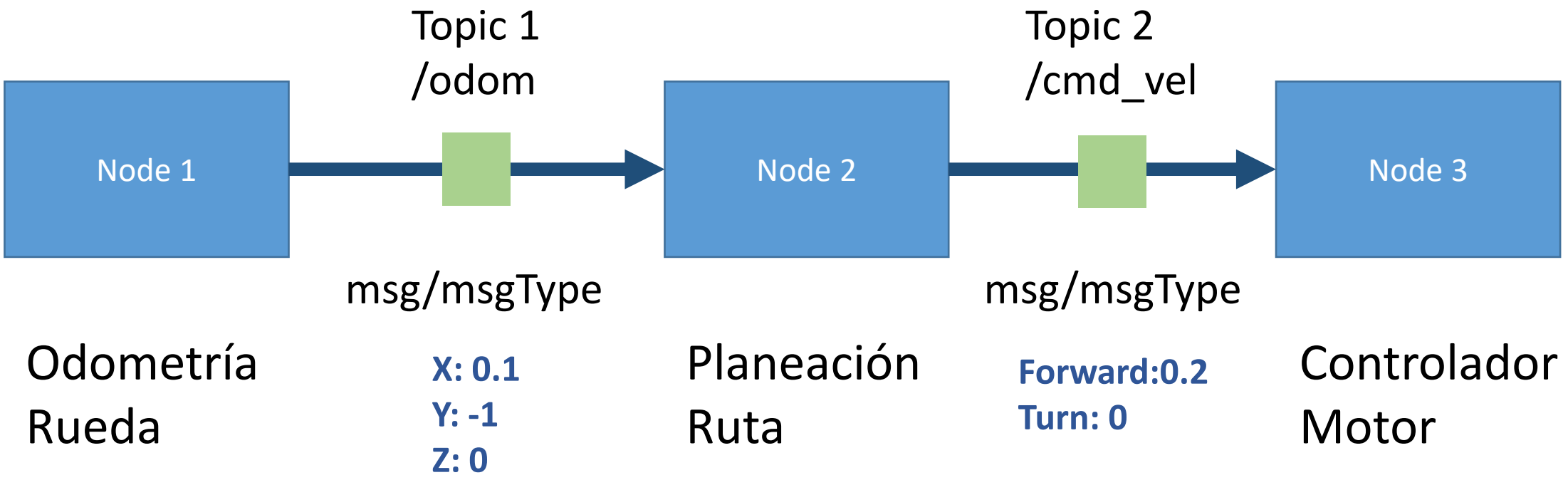


Odometría
Rueda

Planeación
Ruta

Controlador
Motor

ROS Publishers y subscribers (mensajes)



Reglas para publicar y subscribirse

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

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

- **rosnode list** — lista los nodos activos
- **rosnode info** /<nombre_nodo> — muestra la información del nodo
- **rosnode ping** /<nombre_nodo> — prueba la conectividad al nodo
- **rosnode kill** /<nombre_nodo> — detiene la ejecusión de un nodo
- **rosnode 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 temenos una aplicación para un robot, compuesta de muchos paquetes y nodos en cada uno de ellos.

Ahora, necesitamos crear configuraciones globales, por ejemplo:

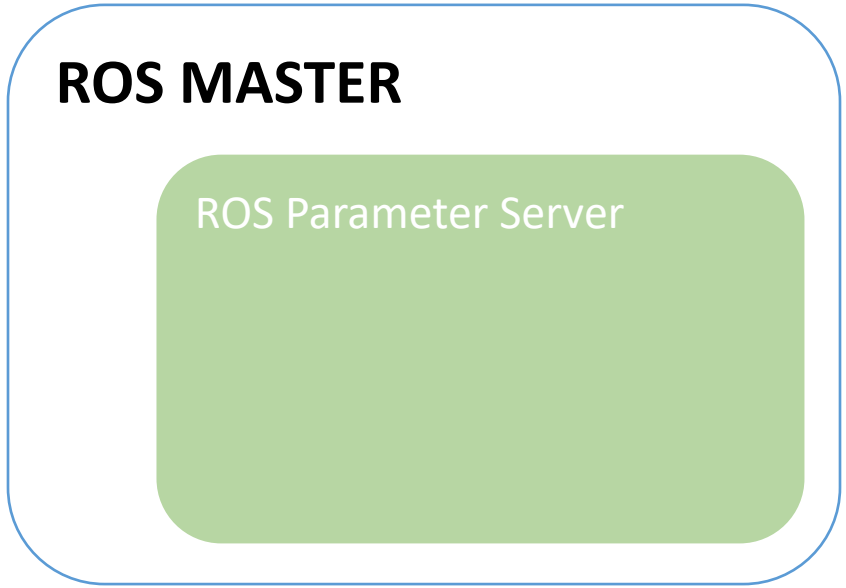
- Nombre del robot.
- Frecuencia a la que se lee algunos sensors.
- Una bandera de simulación que indique a todos los nodos si el robot se está ejecutando en modo real o modo simulación.

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++.

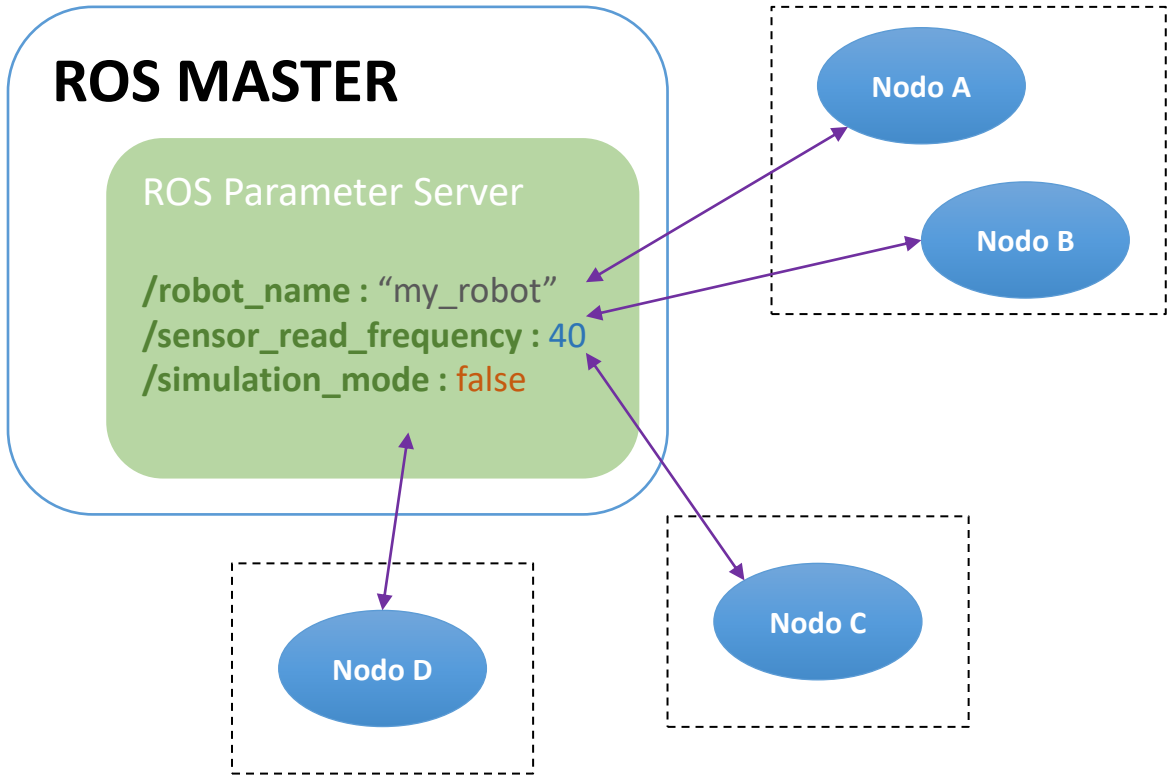
Es fácil darse cuenta en éste momento, que necesitamos un **diccionario global** para las configuraciones compartidas en la aplicación, que se pueda recuperar en **tiempo de ejecución** al iniciar los nodos.

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.

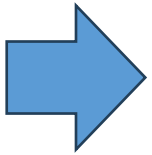
ROS Parameter Server – command tools

○ **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 Launch Files

¿Qué es un archivo **launch**?

¿Qué es un archivo **launch**?

- Un archivo **launch** es un documento **XML** en cual especifica:
  - Cuáles nodos ejecutar
  - Sus parámetros
  - Qué otros archivos **launch** incluir
- Los archivos **launch** deben tienen la extensión **.launch**
- Por convención, se deben crear en la carpeta **launch**
- **roslaunch** es el programa para ejecutar archivos **launch**
  - **roslaunch** <paquete> <archivo.launch> <argumentos**>

↓

```
<?xml version="1.0"?>     ← especificación de la versión del archivo XML
<launch>                  ← etiqueta de inicio del contenido del archivo launch
</launch>                 ← etiqueta de cierre del contenido del archivo launch
```

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Node launching** – Ejecuta los nodos especificados. No es necesario una terminal por cada nodo, un solo archivo launch puede lanzar todos los nodos que se requieran.

```
<?xml version="1.0"?>   ← especificación de la versión del archivo XML
<launch> ← etiqueta de inicio del contenido del archivo launch
   <node pkg="package_name" type="executable" name="node_name">
atributos: ↑ nombre paquete        ↑ nombre ejecutable ↑ nombre|alias nodo
   </node>
        ⋮    'n' nodos a ejecutar
   <node pkg="package_name" type="executable" name="node_name">
   </node>
</launch> ← etiqueta de cierre del contenido del archivo launch
```

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Atributos de la etiqueta <node>:**
  - **pkg**="package_name"- Paquete del nodo.
  - **type**="nodetype" - Node type. Debe haber un ejecutable correspondiente con el mismo nombre.
  - **name**="node_name" - Node name. Nombre con el que será publicado el nodo. NOTA: El nombre no puede contener un espacio de nombres. Utilice el atributo ns en su lugar.
  - **output**="**log|screen**" (opcional) - Si es **'screen'**, stdout/stderr del nodo se enviará a la pantalla. Si es **'log'**, la salida stdout/stderr se enviará a un archivo de registro en **$ROS_HOME/log** y stderr continuará enviándose a la pantalla. El valor predeterminado es **"log"**.
  - **ns**="**foo**" (opcional) - Inicia el nodo en el espacio de nombres 'foo'.

Ver otros atributos e información relacionada en ROS Docs <node> tag , 1.2 Atributes

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

Ejemplo:

```
<?xml version="1.0"?>
<launch>
   <node pkg="my_robot" type="robot_ctrl" name="move_node">
   </node>
   <node pkg="my_robot" type="teleop_key" name="teleop_node" output="screen">
                        atributo 'output' - salida del programa pantalla (screen) ↑
   </node>
</launch>
```

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Arguments <arg>** – Especifica valores que se pasan a través de la línea de comandos, a través de un **<include>** o declarados para archivos de nivel superior. Los argumentos no son globales. Una declaración arg es específica de un único archivo de inicio, muy parecida a un parámetro local en un método. Debe pasar explícitamente los valores arg a un archivo incluido, de forma muy similar a como lo haría en una llamada a un método.
- **<arg>** se puede utilizar de tres maneras:
  - **<arg name="foo" />** Declara la existencia de foo. foo debe pasarse como argumento de línea de comandos (si es de nivel superior) o mediante **<include>** (si está incluido).
  - **<arg name="foo" default="1" />** Declara foo con un valor predeterminado. foo puede anularse mediante un argumento de línea de comandos (si es de nivel superior) o mediante el paso **<include>** (si está incluido).
  - **<arg name="foo" value="bar" />** Declara foo con valor constante. El valor de foo no se puede anular. Este uso permite la parametrización interna de un archivo de inicio sin exponer esa parametrización en niveles superiores.

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Atributos de la etiqueta <arg>:**
  - **pkg=**“package_name”- Paquete del nodo.
  - **name=**"arg_name“ - Nombre del argumento.
  - **default=**"valor predeterminado" **(opcional)** - Valor predeterminado del argumento. No se puede combinar con el atributo **value**.
  - **value=**"valor" **(opcional)** - Valor del argumento. No se puede combinar con el atributo **default**.
  - **doc=**"descripción de este argumento" **(opcional) -** Descripción del argumento. Puede obtener esto a través del argumento *--ros-args* del comando **roslaunch**.

```
roslaunch tb3_ctrl_pkg bringup_robot.launch a:=1 b:=5
```

Ver otros atributos e información relacionada en ROS Docs <node> tag , 1.2 Atributes

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Parameters <param>** – Define un parámetro que se establecerá en el servidor de parámetros. En lugar de valor, se puede especificar un archivo de texto, un archivo bin o un atributo de comando para establecer el valor de un parámetro. La etiqueta **<param>** se puede colocar dentro de una etiqueta **<node>**, en cuyo caso el parámetro se trata como un **parámetro privado**.

- **name=**“namespace/name“ - Nombre del parámetro. Se pueden incluir espacios de nombres (namespaces) en el nombre del parámetro, pero se deben evitar los nombres especificados globalmente.
- **value=**“value"**(opcional)** - Define el valor del parámetro. Si se omite este atributo, se debe especificar *binfile, textfile o command*.
- **type=**“string|int|doble|bool|yaml"(opcional) - Especifica el tipo de parámetro. Si no especifica el tipo, **roslaunch** intentará determinar automáticamente el tipo. Estas reglas son muy básicas:
  - los números con '.' son de punto flotante, en caso contrario son enteros;
  - “true" y “false" son booleanos (no distinguen entre mayúsculas y minúsculas).
  - todos los demás valores son cadenas

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **textfile="$(find package-name)**/path/archivo.txt“ **(opcional)** El contenido del archivo se leerá y almacenará como una cadena. El archivo debe ser accesible localmente, aunque se recomienda utilizar la sintaxis **$(find)/file.txt** relativa al paquete para especificar la ubicación.
- **binfile="$(find package-name)**/path/archivo“ **(opcional)** - El contenido del archivo se leerá y almacenará como un objeto binario **XML-RPC** codificado en **base64**. El archivo debe ser accesible localmente, aunque se recomienda utilizar la sintaxis **$(find)/file.txt** relativa al paquete para especificar la ubicación.
- **command="$(find package-name)**/exe ‘**$(find package-name)**/arg.txt’” (opcional) - La salida del comando se leerá y almacenará como una cadena. Se recomienda utilizar la sintaxis **$(find)/file.txt** relativa al paquete para especificar los argumentos del archivo. También debes citar los argumentos del archivo usando comillas simples debido a los requisitos de escape XML.
- **Ejemplo:**

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

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Group <group>** – facilita la aplicación de configuraciones a un grupo de nodos. Tiene un atributo **ns** que le permite enviar el grupo de nodos a un espacio de nombres separado. También puede utilizar la etiqueta **<remap>** para aplicar la configuración de reasignación en todo el grupo.
- **name=**“namespace/name“ - Nombre del parámetro. Se pueden incluir espacios de nombres (namespaces) en el nombre del parámetro, pero se deben evitar los
- **ns=**"namespace“ **(opcional)** - Asigna el grupo de nodos al espacio de nombres especificado. El espacio de nombres puede ser global o relativo, aunque no se recomiendan los espacios de nombres globales.
- **clear_params=**“true|false" **(opcional)** - Elimina todos los parámetros en el espacio de nombres del grupo antes del lanzamiento. Esta característica es muy peligrosa y debe usarse con precaución.

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **namespace <ns>** – ROS admite nombres relativos, que utilizan el concepto de espacio de nombres predeterminado. La forma habitual de establecer el espacio de nombres predeterminado para un nodo-un proceso a menudo llamado empujar hacia abajo **(pushing down)** en un espacio de nombres- es asignar el atributo **ns** al elemento de nodo:

ns=“*namespace*”

Ejemplo:

```
<param name="publish_frequency" type="double" value="10.0" />
 <node pkg="tutlebot3“ type="tutlebot3_node“ name="tutlebot3“ respawn="true"
ns="sim1"/>
 <node pkg="tutlebot3" type=" tutlebot3_teleop_key“ name="teleop_key“ required="true"
launch-prefix="xterm -e" ns="sim1"/>
 <node pkg="tutlebot3" type="tutlebot3_node“ name="tutlebot3“ respawn="true“
ns="sim2"/>
 <node pkg="tutlebot3" type=" tutlebot3_teleop_key" name="teleop_key“ required="true"
launch-prefix="xterm -e" ns="sim2"/>
```

Subcomponentes de los archivos **launch**

- ✓ Nodes launching
- ✓ Arguments
- ✓ Parameters
- ✓ Grouping
- ✓ Namespaces
- ✓ If conditions

- **Atributos <if>|<unless>** – Todas las etiquetas admiten atributos ***if*** y ***unless***, que incluyen o excluyen a una etiqueta según la evaluación de un valor. **"1"** y **“true"** se consideran valores **verdaderos**. **"0"** y **“false"** se consideran valores **falsos**. Otros valores producirán errores.
  - ***if*** = valor (opcional). **“Si”** el valor se evalúa como verdadero, se incluye la etiqueta y su contenido.
  - ***unless*** = valor (opcional) **“A menos que”** el valor se evalúe como verdadero (lo que significa que si el valor se evalúa como falso), se incluye la etiqueta y su contenido.

Ejemplo:

```
<group if="$(arg foo)">
  <!-- cosas que sólo serán evaluadas si foo es cierto-->
</group>

<param name="foo" value="bar" unless="$(arg foo)" />
<!-- Este parámetro no se establecerá cuando se cumpla la condición “unless" -->
```

Subcomponentes de los archivos **launch**

- **<include>** – Permite importar otro archivo **XML** de **roslaunch** al archivo actual. Se importará dentro del alcance actual de su documento, incluidas las etiquetas **<group>** y **<remap>**.
- **Atributos:**
  - **file=**"$(find name-pkg)/path/nombredearchivo.xml" - Nombre del archivo a incluir.
  - **ns=**"foo" **(opcional)** Importe el archivo relativo al espacio de nombres **'foo'**.
  - **clear_params=**"true|false" **(opcional, predeterminado: falso)** - Elimina todos los parámetros en el espacio de nombres de **<include>** antes del lanzamiento. *Esta característica es muy peligrosa y debe usarse con precaución.* Se debe especificar un **ns**.

```
<include file="$(find pkg-name)/path/filename.launch"/>
```

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 **rosbag**, 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 **rosbag** es llevar un registro de los datos generados durante el proceso para posteriormente visualizarlos y procesarlos fuera de línea.

ROS bag tools

- **rosbag record** [topic_1] [topic_2] **--duration**=[value] **--output-name=** [bag_name]
- **rosbag play** [bag_name]
- **rosbag info** [bag_name]
- **rosbag check** [bag_name]
- Descripción más detallada de los comandos y acciones de los archivos bag pueden ser encontradas en:

  http://wiki.ros.org/rosbag/Commandline

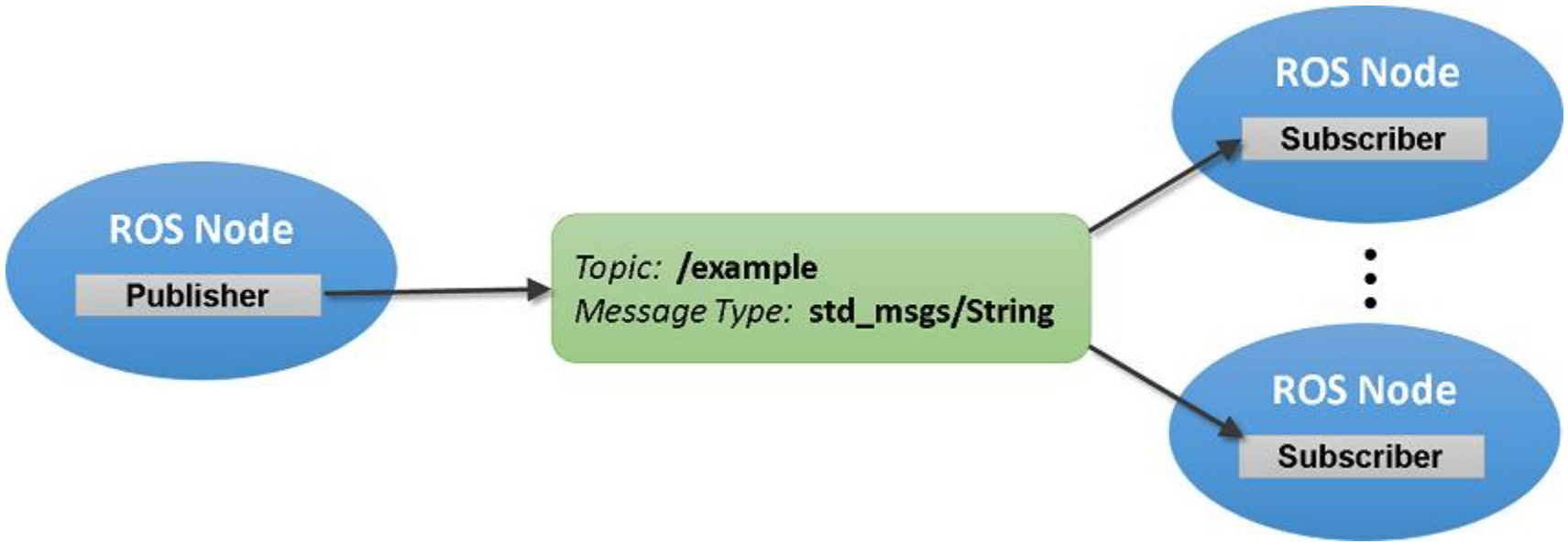
ROS Actions

¿Por qué se necesitan?

¿Qué son ROS Actions?

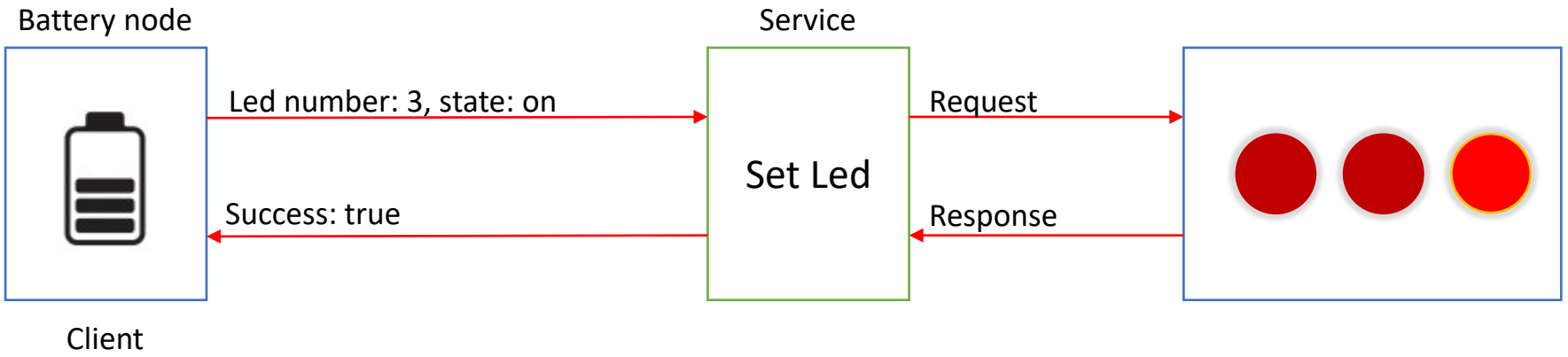
ROS Actions, ¿Por qué se necesitan?

- Herramientas de comunicación de ROS
  - Topics

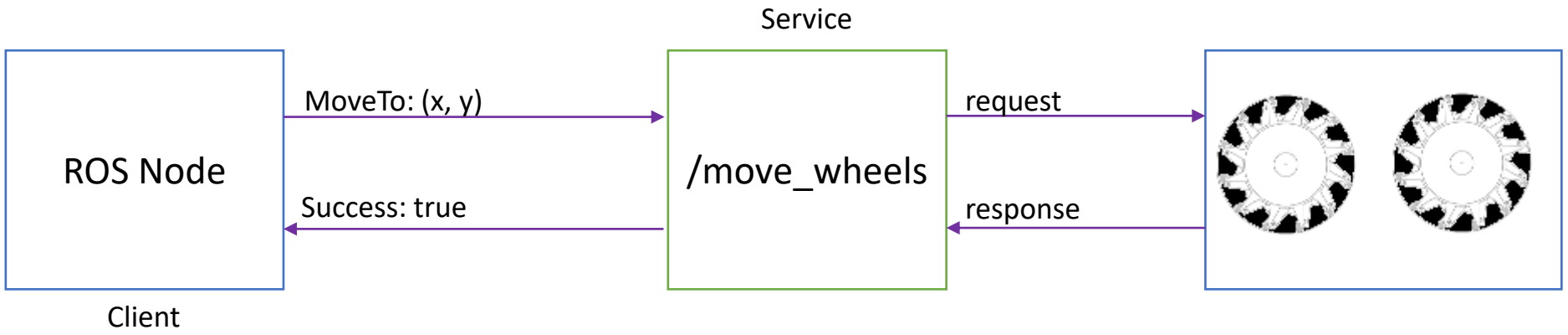


ROS Actions, ¿Por qué se necesitan?

- 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**)?

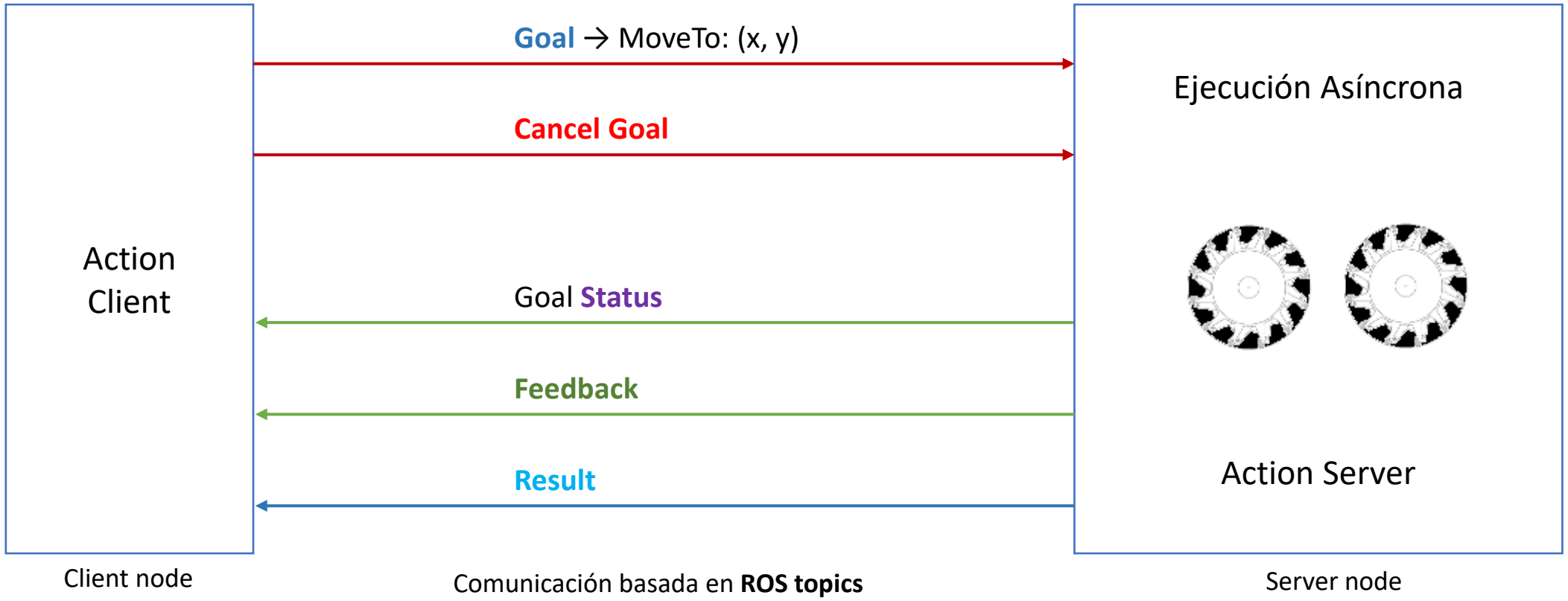
ROS Actions - Problema

Problemas con Servicios

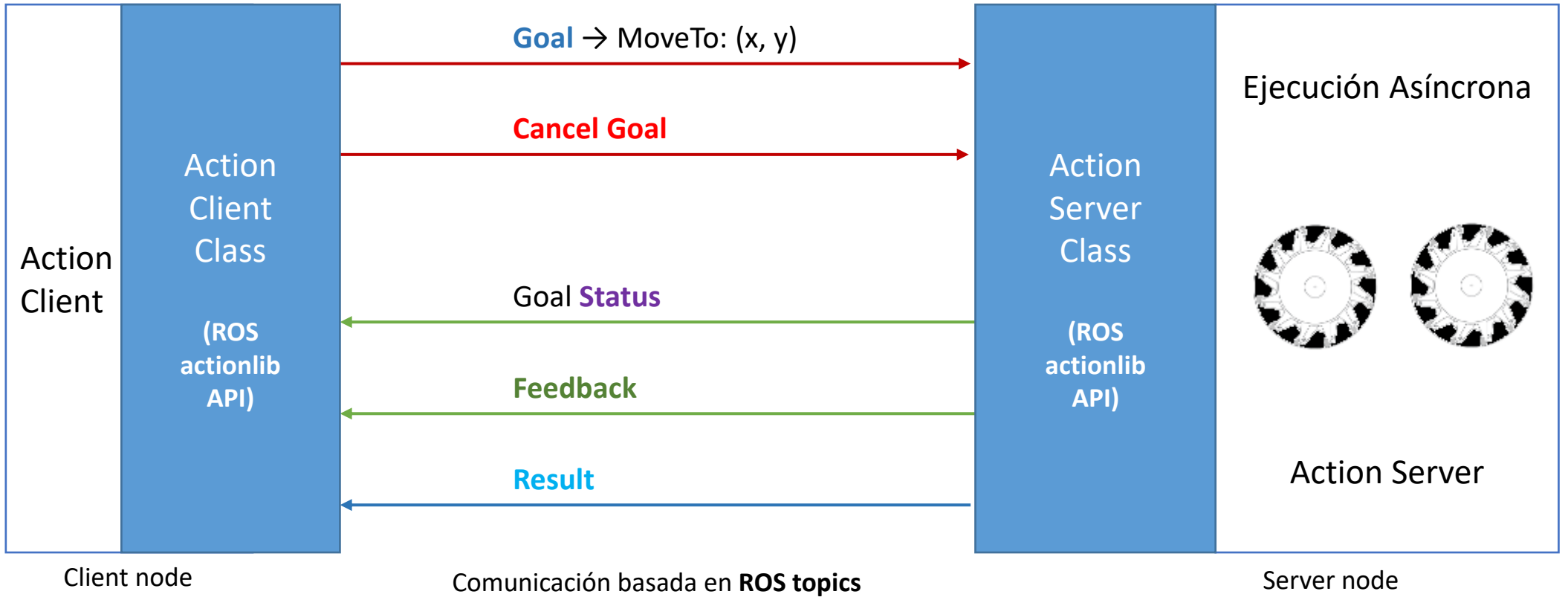
→ **Síncronos**

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

ROS Actions, ¿Qué son?



ROS Actions, ¿Qué son?



ROS Actions

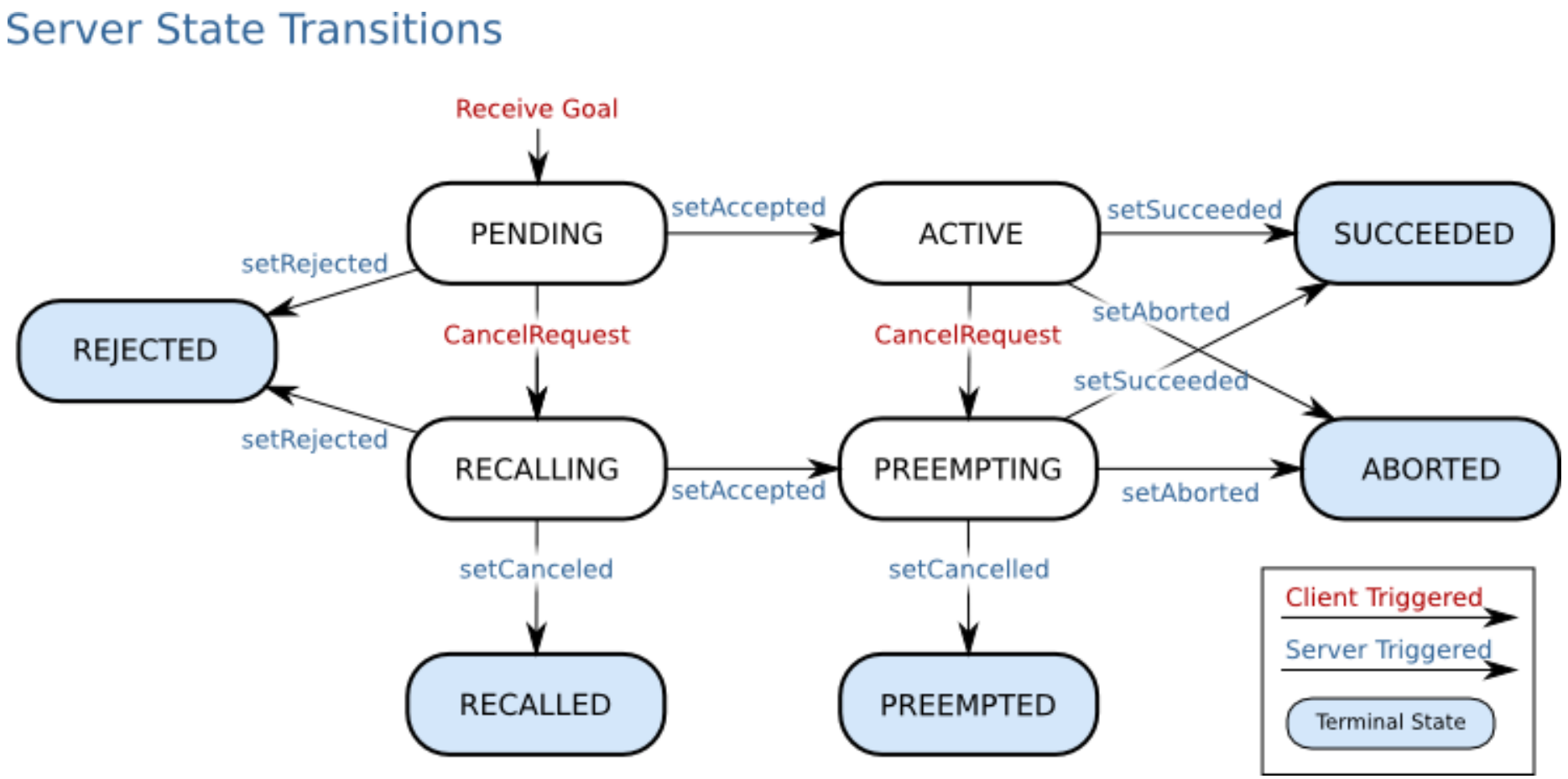
Con **ROS Actions** es posible:

→ 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



Actionlib - Client State Transitions