UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DESARROLLO DE UNA HERRAMIENTA GRÁFICA DE EXPLORACIÓN DE ROBOTS CON ROS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

DEMIAN ALEY SCHKOLNIK MÜLLER

PROFESOR GUÍA:
JOHAN FABRY

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CÁRDENAS.
AIDAN HOGAN

SANTIAGO DE CHILE
2015

Resumen

ROS, un middleware para robots, ofrece gran variedad de herramientas para su utilización. Sin embargo, carece de una importante herramienta: Algo que permita visualizar de forma dinámica sus componentes, y poder así entender sistemas complejos de mejor forma.

Esta memoria consistió en desarrollar una herramienta, que es capaz de visualizar, mediante un grafo, un sistema completo creado en ROS, pudiendo ver cuáles son sus distintos componentes y cómo interactúan. Además, la herramienta incluye la opción de visualizar tres tipos de mensajes comunes presentes en ROS. El trabajo se realizó en el lenguaje de programación Smalltalk, y usando el ambiente de desarrollo Pharo. Para las visualizaciones se utilizó el motor de visualizaciones Roassal, basado en Pharo. El trabajo se dividió en cuatro grandes etapas.

La primera etapa consistió en un trabajo de investigación, haciendo una revisión de las herramienta a usar en el desarrollo, junto a un análisis de los mensajes comunes de ROS y como se representaban. Esto sería usado en la segunda etapa.

La segunda etapa consistió en el desarrollo de una API, que permitiera comunicación entre Pharo y ROS. Para ello se hizo uso de las herramientas de consola presentes en ROS. Junto a la API, se desarrollaron una serie de tests, a modo de robustecer la API frente a posibles cambios en ROS.

La tercera etapa consistió en la creación del grafo general de ROS. Este grafo nos muestra el sistema completo creado en ROS, indicándonos también de qué forma interactúan los componentes entre ellos. En esta etapa, la comunicación con ROS se hace exclusivamente a través de la API implementada en la segunda etapa.

La cuarta y última etapa consistió en la creación de las visualizaciones para tres mensajes comunes de ROS. Estas visualizaciones son dinámicas, es decir, cambian en tiempo real cuando los mensajes cambian.

El presente trabajo presenta primero el estudio previo, explicando las herramientas, lenguajes y funcionalidades que se usarán en el desarrollo. Esta sección también incluye información de herramientas similares existentes, y explica las falencias de éstas y la justificación de la creación de una nueva herramienta. La segunda parte de este trabajo contiene el desarrollo de la herramienta, comenzando por la API, para luego seguir con el grafo general y las visualizaciones.

A modo de conclusión, la herramienta creada presenta una manera eficaz y usable de explorar robots creados con ROS, y fue creada de manera modular y extensible, generando algunas visualizaciones de ejemplo. Puede servir como base para un trabajo futuro, que genere más visualizaciones así como también incluya un sistema de inserción de mensajes.

Tabla de Contenido

Índice de Ilustraciones

1. Introducción

El sistema operativo para robots, ROS [5] es un set de librerías y herramientas de software. ROS fue creado para programar aplicaciones para robots. Dentro de las estructuras que componen ROS, podemos encontrar nodos, que son procesos que realizan algún tipo de computación. Un ejemplo de nodo en un robot puede ser un motor o bien el código que controla algún comportamiento específico. Además de nodos, existen los tópicos, que básicamente son canales de comunicación, cada uno con un nombre específico, a través de los cuales los nodos intercambian mensajes. El modelo de transmisión de datos usado es el de suscriptor / publicador. Los nodos se pueden suscribir a varios tópicos, es decir, les llegará información de los tópicos a los cuales se suscriban. También pueden publicar a varios tópicos, es decir enviar información a tópicos. Esto se logra a través de mensajes. Un mensaje es una estructura de datos simple.

Pese a las numerosas ventajas que posee ROS, no posee actualmente una herramienta que permita adecuadamente explorar la configuración de sus nodos, tópicos y mensajes. La mayoría de las herramientas existentes hoy en día son netamente basadas en texto, lo que dificulta su uso. En particular, desde el punto de vista de la usabilidad, vemos que numerosos principios de usabilidad [6] [7] son violados por estas herramientas. En primera instancia vemos que el sistema no posee *visibilidad [8]*, dado el uso predominante de herramientas textuales, incluso para datos con estructuras complejas. Luego podemos destacar la baja *retroalimentación [7]* que provee: En un sistema basado en texto únicamente, la retroalimentación viene dada mayoritariamente por los movimientos del robot, que en muchos casos, no nos ayuda a determinar lo que buscamos. Con respecto al *mapping [7] [8]*, nos encontramos nuevamente con un problema: Al estar explorando la configuración, debo descubrir por mi cuenta cómo es la relación de los distintos controles, es decir, de qué forma se ejecutan las funciones, qué botones hacen qué cosa, etc.

Hay una herramienta gráfica presente en ROS, pero esta herramienta sirve únicamente para visualizar de manera estática un grafo de nodos. Posee además un sinnúmero de deficiencias, como ser un sistema que no escala bien con números grandes de nodos, existen aristas del grafo que se repiten en ciertos casos y, como mencionado anteriormente, es un gráfico completamente estático, por lo que no podemos revisar adecuadamente los flujos.

Se pueden encontrar diversos software que permiten cómodamente mostrar grafos, mostrando de qué forma las distintas componentes están conectadas. Ejemplos de esto son Gephi [1] (fig 1), Grafos [2] (fig2) y Graphviz [3] (fig3). Sin embargo, necesitamos mostrar adecuadamente los datos que fluyen por los tópicos, por ejemplo como se puede ver en DataDisplayDebugger [4] (fig. 4). De esta forma es mucho más amigable para el usuario.

Además, para hacer un programa en ROS, si deseamos publicar datos en un tópico, se requiere una gran cantidad de trabajo. Esto es debido a que tenemos que programar un programa específico para eso, compilarlo, desplegar al robot, observar (muchas veces incluso comportamientos extraños), y finalmente repetir. Es por ello que para determinar experimentalmente cuáles son los valores adecuados se deben realizar trabajos que toman demasiado tiempo. El uso de una interfaz apropiada, en tiempo real, reduciría este trabajo enormemente.

Para este trabajo se usará Pharo Smalltalk, un lenguaje de programación orientado a objetos, por recomendación del profesor guía. Para la construcción del grafo se usará Roassal, un motor de visualización gráfico ágil, desarrollado internamente en el DCC (Departamento de Ciencias de la Computación de la Universidad de Chile). Roassal posee la ventaja de ser programable, por lo tanto puede cumplir todos los requerimientos nombrados, además de contar con la ayuda de muchos expertos del departamento.

Los objetivos del trabajo de memoria fueron los siguientes:

- Creación de una herramienta gráfica que permita mostrar gráficamente la topología presente al conectarla a un robot que utilice ROS, mostrando nodos, tópicos y mensajes, y como estas componentes están conectadas entre sí.
- Determinar cuáles son los tipos de mensajes más comunes en ROS. Se debe estudiar cómo se deben visualizar estos mensajes, y como sería una interfaz adecuada para ellos.
- La herramienta debe permitir a un usuario revisar el flujo de mensajes dentro de la topología, mostrando en tiempo real y de la forma más adecuada los mensajes de los tipos más comunes.
- La herramienta deberá estar estructurada de forma modular, permitiendo una adaptación sencilla si la interfaz hacia ROS cambia, o cuando se quieran integrar nuevos tipos de visualizaciones de mensajes.

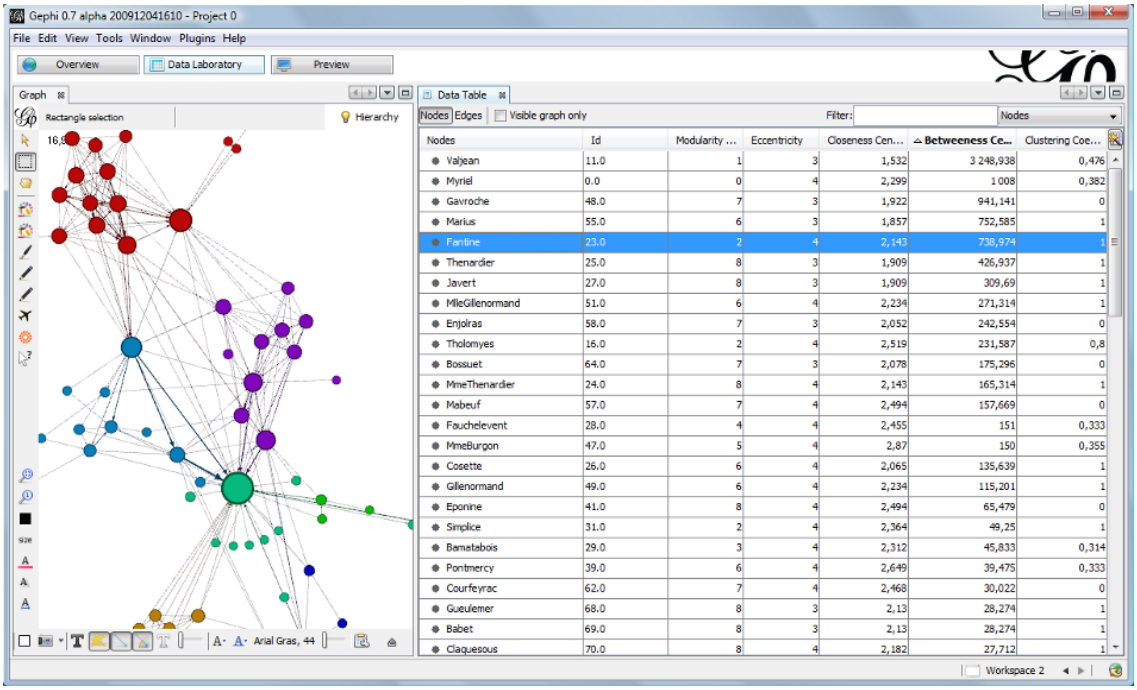


Ilustración 1: Un grafo modelado en Gephi

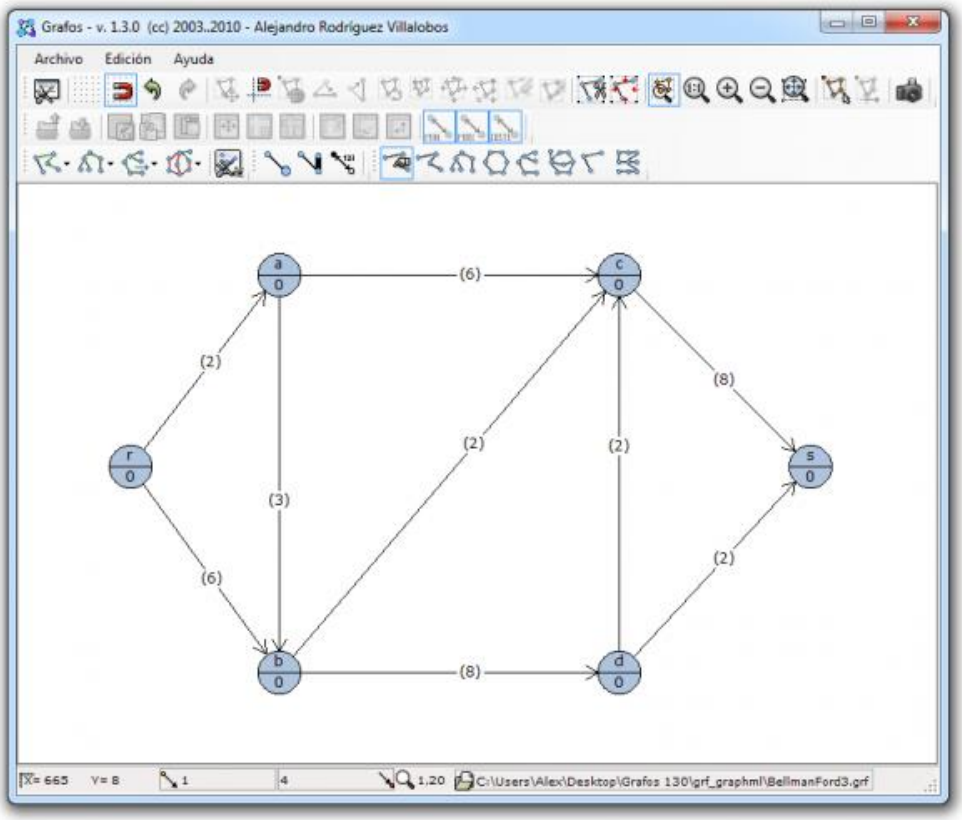


Ilustración 2: Un grafo hecho en Grafos

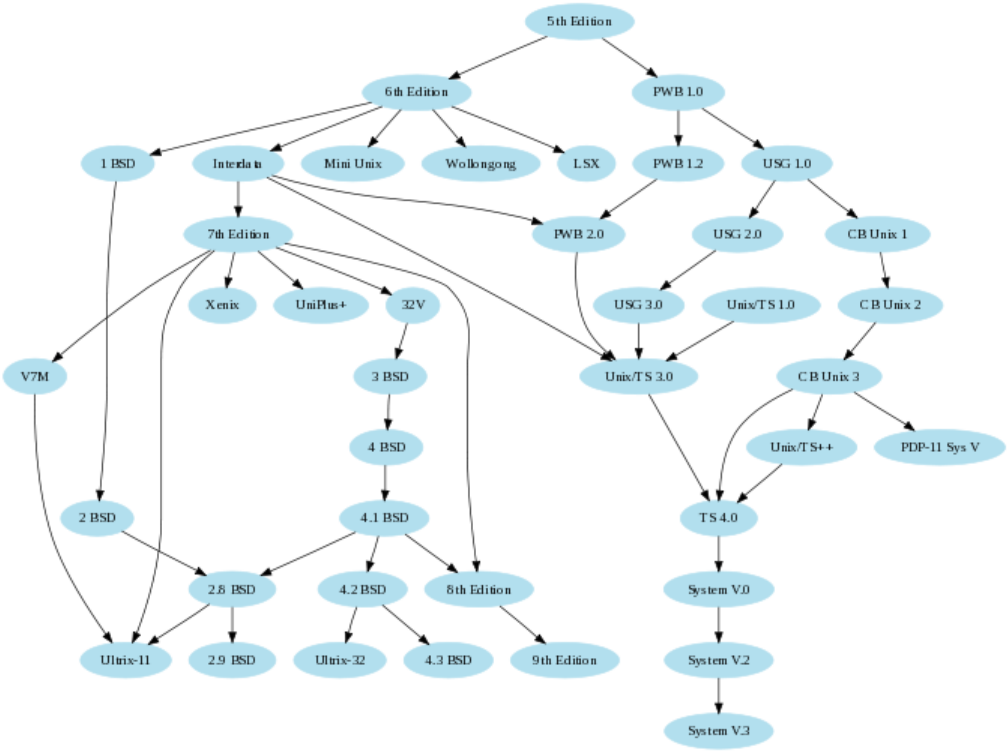


Ilustración 3: Un grafo hecho con Graphviz

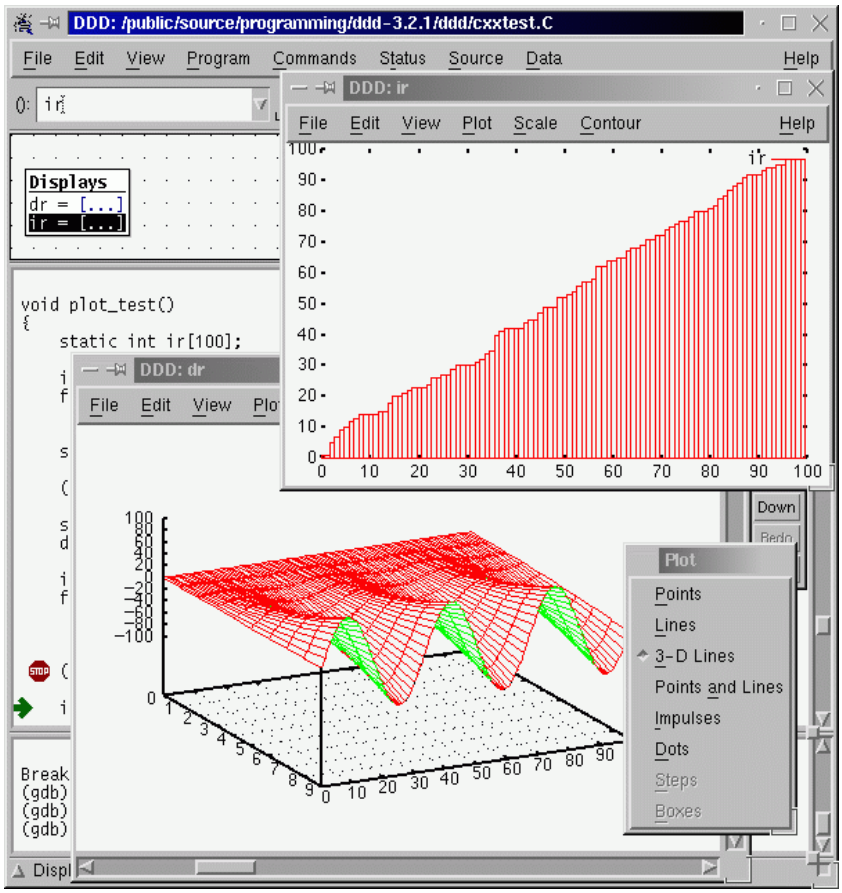


Ilustración 4: Data mostrada gráficamente en Data Display Debugger

2. Antecedentes

2.1 Smalltalk

Como fue mencionado en la introducción, para este trabajo de memoria se utilizará para programar el lenguaje Pharo Smalltalk, para poder así trabajar con Roassal y debido a la gran cantidad de expertos que hay en el departamento.

Smalltalk es un lenguaje de programación orientado a objetos, dinámicamente tipado y reflexivo. Smalltalk fue creado con la idea de un nexo entre personas y máquinas. Fue creado y diseñado en parte para uso educacional, principalmente para aprendizaje construccionista, por el Learning Research Group (LRG) de Xerox PARC, durante los 70's.

Como en otros lenguajes de programación orientados a objetos, el concepto central de Smalltalk-80 es el de un objeto. Un objeto es siempre una instancia de una clase. Las Clases son "planos" que describe las propiedades y el comportamiento de las instancias. Por ejemplo, una clase de ventanas gráficas puede declarar que las ventanas tienen propiedades tales como etiqueta, posición y si la ventana es visible o no. La clase ademas puede declarar que las instancias soportan operaciones tales como abrir, cerrar, mover u ocultar. Cada objeto particular de ventana tendría sus propios valores de esas propiedades, y cada uno de ellos sería capaz de llevar a cabo las operaciones definidas por su clase.

Un objeto de Smalltalk puede hacer exactamente tres cosas:

1. Mantener un estado (referencias a otros objetos)

2. Recibir un mensaje de sí mismo o de otro objeto

3. En el transcurso del procesamiento de un mensaje, mandar mensajes a si mismo u otro objeto.

El estado que el objeto tiene siempre es privado a ese objeto. Otros objetos pueden realizar consultas o cambiar el estado a través de peticiones (mensajes) al objeto. Cualquier mensaje puede ser enviado a cualquier objeto. Cuando el mensaje es recibido, el receptor determina si el mensaje es apropiado.

Smalltalk es un lenguaje de programación orientado a objetos "puro", lo que significa, que a diferencia de Java o C++, no hay diferencia entre los valores que son objetos y valores que son objetos primitivos. En Smalltalk, objetos primitivos tales como enteros, booleanos y caracteres son también objetos, en

el sentido que son instancias de las clases correspondientes, y sus operaciones se invocan mediante el envío de mensajes. Un programador puede cambiar o extender las clases que implementan los valores primitivos, por lo que el comportamiento nuevo puede ser definido para sus instancias. Dado que todos los valores son objetos, las clases también lo son. Cada clase es una instancia de una metaclase de esa clase. Las metaclases también son objetos, y son todas instancias de una clase llamada Metaclase. Bloques de código, la manera de Smalltalk de funciones anónimas, son también objetos.

2.2 Roassal

Como explicado en la introducción, para la construcción del grafo se utilizará Roassal[1], pues posee la ventaja de que es programable. Así, podemos adecuarlo a las necesidades de este trabajo de memoria. Posee además la gran ventaja de haber sido desarrollado internamente en nuestro departamento.

Roassal es un motor de visualización gráfico ágil. Roassal representa objetos gráficamente, usando Smalltalk. Posee un gran set tipos distintos de interacciones. Dentro de sus funcionalidades podemos encontrar pintar, interconectar, aumentar, disminuir, etc.

Roassal está construido sobre un modelo de visualización de estructuras bastante simple. Está construido sobre las nociones de objetos, elementos, formas, interacción y vista (objects, elements, shapes, interaction and view).

Un elemento es una envoltura (wrapper) a un objeto de dominio proveído externamente (por ejemplo un número, archivo u objeto). A un elemento podemos asignarle una forma (compuesta o simple), y una interacción. Los elementos se agregan a la vista. Esto es necesario para visualizar las cosas en la pantalla.

Un elemento es una representación gráfica de un objeto arbitrario. Un usuario final ve elementos e interactúa con ellos usando el teclado y mouse. Un elemento contiene formas que definen su representación gráfica. Una forma describe una representación gráfica primitiva tales como una caja, circulo, línea u etiqueta textual. Las formas son combinables para crear formas más elaboradas.

Las formas se separan en dos grandes grupos: Elementos y vértices.

Las formas de los elementos pueden ser de los siguientes tipos: Cajas, elipses, etiquetas, polígonos, bitmap, camino o arco. Entienden los siguientes mensajes, entre otros: Tamaño, ancho y alto, color y color de borde.

[1] http://objectprofile.com/Roassal.html

Las formas de los vértices pueden ser líneas, flechas, linea dirigida o multilínea (es decir, líneas que se separan para llegar a más nodos).

Como mencionado anteriormente, a los elementos se les puede asignar un comportamiento. Las interacciones se pueden implementar para vistas, elementos o vértices. Dentro de los comportamientos existen los siguientes: Hacer una vista arrastrable, horizontal, vertical o ambos; Para elementos podemos hacerlos también arrastrables, podemos hacer que salga un popup al pasar el mouse por encima, mostrar etiquetas, etc. Para gráficos, es posible normalizar los datos, además de establecer los mínimos y máximos, configurar el alineamiento de los elementos, las posiciones, etc.

Con todas estas funcionalidades, podemos entonces usar Roassal para la implementación de la herramienta final. Asignamos una forma específica a los nodos de ROS, traspasándolos a nodos del grafo. Los distintos tópicos serán los vértices del grafo, y programamos todo para que al hacer click en algún elemento, nos dé información específica de ese elemento.

2.3 OSProcess en Smalltalk

OSProcess es un paquete de Pharo que nos permite trabajar directamente con procesos de diversos sistemas operativos. Trae paquetes especializados para Mac, windows, Unix, RiscOS, AIO, entre otros. En nuestro caso, dado que ROS se ejecuta primariamente en Linux (Ubuntu, en la mayoría de los casos), usaremos principalmente las librerías de Unix.

Dentro de los métodos más comunes de OSProcess, y los que más usaremos podemos encontrar:

- exitstatus, que nos indicará el estado en el cual una rutina termina.
- forkChild, para crear nuevos clones de procesos hijos.
- initialEnviroment, para configurar el ambiente inicial (en general, esto será un diccionario).
- initialize, para inicializar los objetos necesarios en default.
- pid, para ver el id específico de un proceso.
- command:input, que nos permite correr un comando en un proceso shell.

El método más importante de los anteriores es el método ‘command’, dado que nos permite enviar comandos a la consola. De esta forma podremos enviar comandos ROS. Veamos un ejemplo de código de cómo, desde Pharo, publicamos directamente a un tópico de ROS:

(OSProcess command: 'rostopic pub /turtle1/cmd_vel geometry_msgs/Twist -r 1 -- '' [2.0, 0.0, 0.0]'' ''[0.0,0.0,1.8]'' ').

En el ejemplo anterior, le decimos al objeto OSProcess que envíe el comando ‘rostopic pub …’ a la consola. Con eso estamos publicando en el tópico /turtle1/cmd_vel un mensaje de tipo ‘twist’, que hace que el robot turtle1 se desplace.

Para la comunicación entre Pharo y ROS, debemos usar OSProcess y PipeableOSProcess. Estas clases, nativas de Pharo, nos ayudan a interactuar con la consola, y de esa forma, poder invocar en la consola directamente los comandos de consola de ROS.

Mediante la función ‘OSProcess command:’ podemos enviar un comando directamente a la consola. De esta forma, podemos hacer cosas tales como iniciar ROS (con roscore) o bien iniciar o terminar nodos o tópicos. Sin embargo, OSProcess envía el comando y no posee métodos de retorno de información. Es por ello que se usa exclusivamente cuando no estamos interesados en la respuesta de la consola. Cuando sí nos interesa, usamos PipeableOSProcess.

PipeableOSProcess también posee la función ‘command:’, que permite enviar un comando a la consola. Sin embargo, retorna un output, que podemos redirigir a un stream para posterior procesamiento. Así podemos leer información de la consola, para funciones de la API tales como la obtención de las listas de nodos o tópicos, funciones de echo o de info.

2.4 Unit Tests en Smalltalk

Hacer tests unitarios es fundamental para este trabajo. Aparte de todas las ventajas tradicionales que obtenemos al hacer tests unitarios, el trabajo se construirá sobre herramientas existentes, por lo que hay que probar que la ‘capa base’ este funcionando de manera correcta, para lo cual se usarán los test. Existe la posibilidad de que las versiones nuevas de las herramientas cambien, por lo que con unit test podemos detectar estos cambios automáticamente.

La principal forma de hacer tests unitarios en Pharo es a través de una poderosa herramienta llamada SUnit. Pese a que SUnit está enfocado a tests Unitarios, pueden ser usados como test de integración y como tests funcionales.

SUnit nos permite escribir tests que se autoevalúan. El test mismo define cual debería ser el resultado correcto. Nos ayuda además a organizar los tests en grupos, para describir el contexto en el cual los tests deben correr, y correr un grupo de tests automáticamente.

Lo primero que se debe hacer es crear una nueva subclase de TestCase, llamada (por ejemplo) ExampleSetTest. Así es como se ve un test básico:

```
TestCase subclass: #ExampleSetTest
instanceVariableNames: 'full empty'
classVariableNames: ''
category: 'MySetTest'
```

En el ejemplo, creamos dos variables de instancia llamadas ‘full’ y ‘empty’. Podemos declarar también variables de clase o diccionarios de almacenamiento, sin embargo, no lo haremos en este ejemplo.

Luego, es necesario configurar el ambiente en el cual correrán los tests. Esto se hace con un método inicializador, llamado setUp. setUp es llamado antes de la ejecución de cada test definido en la clase test.

```
ExampleSetTest»setUp
empty := Set new.
full := Set with: 5 with: 6
```

Para ejecutar los tests simplemente podemos hacerlo desde el explorador de Pharo. Los tests ejecutados serán marcados verdes o rojos, dependiendo de si pasan o no.

Sin embargo, que setUp se llame antes de cada test nos pone en problemas. Para cada test, nosotros necesitamos una serie de procesos corriendo, tales como Roscore y unos cuantos nodos. Lanzar estos procesos es lento, de 4 a 20 segundos por proceso, dependiendo del computador, por lo que si debemos hacerlo antes de cada test, nuestro set de tests demorará muchísimo en correr. Es por este motivo que debemos buscar una forma de lanzar estos procesos antes de todos los tests, en vez de cada uno. Afortunadamente SUnit provee un feature avanzado, que nos permite hacer justamente esto. Se logra mediante una clase llamada TestResource, diseñada exactamente para este propósito: TestResource carga recursos pesados antes de todos los tests.

El método setUp de TestResource es el encargado de hacer la carga de procesos pesados. Veamos el código:

```
setUp
        "This is where we load all heavy resources, such as roscore and
some nodes"
        OSProcess command: 'roscore'.
        (Delay forSeconds: 5) wait.
        OSProcess command:'rosrun turtlesim turtlesim_node
__name:=turtle1'.
        (Delay forSeconds: 5) wait.
        OSProcess command: 'rostopic echo /turtle1/cmd_vel
__name:=turtle_echo'.
        (Delay forSeconds: 5) wait.
        OSProcess command:'rosrun turtlesim turtle_teleop_key
__name:=turtle_teleop'.
        (Delay forSeconds: 5) wait.
```

Primero llamamos a Roscore, el proceso principal que nos permitirá correr el resto de los nodos. Dado que no tenemos manera de saber cuándo ha completado de lanzarse, esperamos 5 segundos entre cada instrucción. A continuación corremos un simulador interno de ROS llamado turtleSim, y le asignamos el nombre turtle1. Luego lanzamos un nodo de echo, encargado de retransmitir todos los datos del tópico /turtle1/cmd/vel. Lanzamos también teleop_key, un nodo encargado de transformar señales de teclado en mensajes para el simulador. Todos estos nodos se comunican entre sí, sirviendo de buen set para nuestros tests.

2.5 ROS

El sistema operativo para robots, ROS (por su sigla en inglés: Robot Operating System) es un set de librerías y herramientas de software que ayudan a construir aplicaciones para robots. Posee desde drivers a algoritmos y es open source. ROS se compone principalmente de nodos, tópicos y mensajes.

ROS fue creado debido a la dificultad de crear software para robots que sea robusto y multipropósito. Desde la perspectiva de la robótica, problemas que pueden parecer triviales para humanos varían ampliamente entre instancias de tareas y ambientes. Lidiar con estas variaciones es tan difícil, que ninguna persona, laboratorio o institución puede hacerlo por sí mismo.

Como resultado de esta problemática, ROS fue creado desde cero para alentar desarrollo colaborativo de software de robots. Por ejemplo, un laboratorio puede poseer expertos en mapeo de ambientes interiores, y podría contribuir con un sistema de clase mundial de para producir mapas. Otro grupo podría tener expertos en usar mapas para navegar, mientras que otro grupo podría haber descubierto un enfoque de visión por computador que funciona bien

reconociendo objetos pequeños en desorden. ROS fue diseñado específicamente para grupos como estos para que colaboren y construyan mediante apoyo mutuo.

Las librerías de cliente de ROS permiten trabajar en 2 lenguajes de programación: rospy para python y roscc para c++.

A continuación vamos a hablar más en detalles sobre ROS, su estructura, sus comandos textuales y formato de mensajes, ya que serán usados en este trabajo de memoria.

2.5.1 Nodos

Un **nodo** es un proceso que realiza algún tipo de computación. Los nodos se combinan en un grafo y se comunican entre ellos a través de tópicos, servicios RPC (del inglés, Remote Procedure Call), y el Centro de Parámetros. Estos nodos están hechos para operar en escalas muy finas y pequeñas. El sistema de control de un robot típicamente abarcará muchos nodos. A modo de ejemplo, un nodo controla un dispositivo de distancias láser, un nodo controla los motores de las ruedas del robot, un nodo hace localizaciones, un nodo hace planificaciones de caminos, etc.

Para comenzar a trabajar usando ROS es necesario correr el comando *$ roscore*. Roscore inicializará los principales procesos requeridos, entre ellos master, *rosout* y el servidor de parámetros. Master es el que provee los nombres de servicio para ros, mientras que *rosout* permite trabajar con la salida estándar del sistema (stdout/stderr). Por otro lado, tenemos el comando *$ rosnode*, que nos permite obtener información de nodos. *$ rosnode list* nos mostrará una lista con los nodos activos, mientras que *$ rosnode info* nos mostrará información específica de un nodo en particular. *Rosrun* nos permite ejecutar un nodo directamente desde su paquete, de la forma *$ rosrun [package_name] [node_name].*

2.5.2 Tópicos

Los tópicos son canales de comunicación, etiquetados, a través de los cuales los nodos intercambian mensajes. Los tópicos poseen semánticas anónimas del tipo publicar/suscribirse, la cual desacopla la producción de información del consumo de ésta. En general, los nodos no están conscientes de con quien se están comunicando. En vez de esto, los nodos que están interesados en información se suscriben a un tópico relevante. Nodos que

generan información publican al tópico relevante. Puede haber múltiples publicadores y suscriptores a un tópico. La idea de los tópicos es proveer comunicación de transmisión continua y unidireccional.

Usando el comando *$ rostopic -h* nos dará la lista de posibles sub comandos de rostopic:

- rostopic bw - muestra el ancho de banda usado por un tópico
  rostopic echo - imprime mensajes en la pantalla
  rostopic hz - muestra la tasa de refresco para un tópico
  rostopic list - muestra información acerca de tópicos activos
  rostopic pub - publica data a un tópico
  rostopic type - muestra el tipo de un tópico

2.5.3 Mensajes

Los nodos se comunican entre sí a través de la publicación de mensajes a tópicos. Un mensaje es una estructura de datos simple, y posee campos tipados. Tipos estándar primitivos (enteros, punto flotante, booleanos, etc.) están soportados, así como también arreglos de tipos primitivos. Los mensajes pueden contener estructuras y arreglos anidados (como un struct de C).

2.5.4 Ejemplo de código (Python)

A continuación se mostrará un simple código para indicar el modo de funcionamiento de un nodo que publica mensajes y otro que se suscribe. Se explican los códigos, para entender mejor el funcionamiento normal de un programa ROS. Además, serán usados al crear la herramienta. En los ejemplos, se mostrará primero el código completo, para luego separarlo y explicarlo línea por línea. De este modo es más comprensible.

Ejemplo 1, Nodo publicador:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    r = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        str = "hello world %s"%rospy.get_time()
        rospy.loginfo(str)
        pub.publish(str)
        r.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException: pass
```

Podemos ahora hacer un pequeño desglose del código del publicador:

```
#!/usr/bin/env python
```

Cada nodo Python de ROS debe tener esta declaración. Esto nos asegura que el código es ejecutado como código Python.

```
import rospy
from std_msgs.msg import String
```

Si queremos escribir un nodo ROS debemos importar rospy. La importación de std_msgs.msg es para poder reusar el tipo de mensajes std_msgs/String.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

Esta sección del código define la interfaz del publicador hacia el resto de ROS. pub = rospy.Publisher("chatter", String, queue_size=10) declara que el nodo está publicando al tópico 'chatter' usando el tipo de mensaje String. En este caso, String es realmente la clase std_msgs.msg.String. El argumento queue_size

limita el número de mensajes encolados si algún suscriptor no los está recibiendo suficientemente rápido.

La línea rospy.init_node(NAME) es muy importante, pues le dice a rospy el nombre del nodo. Hasta que rospy posea esta información, no puede comunicarse con ROS Master.

```
r = rospy.Rate(10) # 10hz
```

En esta línea creamos el objeto de tipo *rate* r. Con la ayuda de su método sleep() podemos iterar a la frecuencia deseada.

```
while not rospy.is_shutdown():
        str = "hello world %s"%rospy.get_time()
        rospy.loginfo(str)
        pub.publish(str)
        r.sleep()
```

Hay que verificar is_shutdown() para ver si el programa debería salirse. La llamada pub.publish(String(str)) publicará al tópico 'chatter' usando un mensaje String recién creado. El bucle llama a r.sleep(), lo cual duerme el tiempo suficiente para mantener la frecuencia del bucle.

```
try:
        talker()
    except rospy.ROSInterruptException: pass
```

Este último código captura excepciones.

Ejemplo 2, Nodo suscriptor:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id()+"I heard %s",data.data)

def listener():

    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    rospy.spin()

if __name__ == '__main__':
    listener()
```

Analicemos ahora el nodo suscriptor.

```
rospy.init_node('listener', anonymous=True)

rospy.Subscriber("chatter", String, callback)

# spin() simply keeps python from exiting until this node is stopped
rospy.spin()
```

Esto declara que nuestro nodo se está suscribiendo al tópico ‘chatter’, el cual es del tipo std_msgs.msgs.String. Cuando se reciben nuevos mensajes, se invoca callback, con el mensaje como primer argumento.

ROS requiere que cada nodo tenga un nombre único. Si un nodo aparece con un nombre ya usado, botará el anterior. Esto es para poder botar fácilmente nodo que no estén funcionando. El argumento anonymous=True el dice a rospy que genere un nombre único para poder tener varios nodos de tipo suscriptor.

rospy.spin() simplemente previene que el nodo termine hasta que se cierre manualmente.

2.5.5 Herramienta de gráficos de RQT (rqt_graph)

Rqt_graph es la única herramienta gráfica de visualización de nodos y tópicos de ROS. Provee una interfaz de usuario para visualizar el grafo. Sus componentes están hechos de forma genérica de forma que otros paquetes que deseen poseer representación gráfica puedan depender de este paquete.

Sin embargo, esta herramienta sirve únicamente para visualizar de manera estática, y hay muchas deficiencias, como mostrará a continuación. El trabajo realizado (la herramienta) mejorará estos aspectos junto con introducir aspectos dinámicos tales como inserción de mensajes o interpretación de mensajes.

Análisis de Usabilidad de la herramienta de gráficos de RQT (rqt_graph)

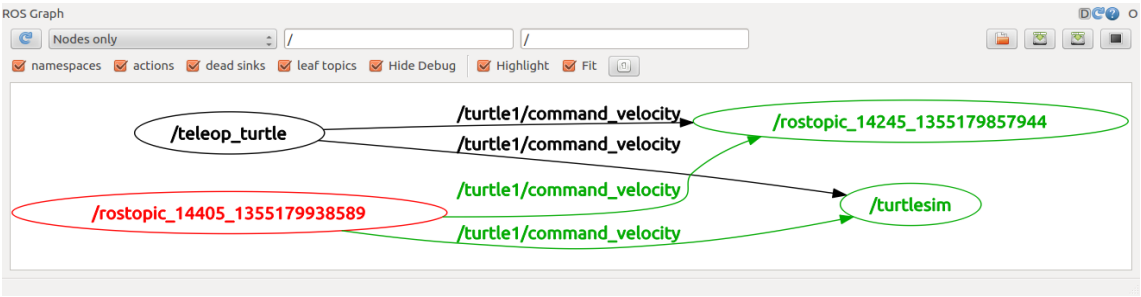


Ilustración 5: RQT_Graph

En la ilustración, vemos cuatro nodos, representados por elipses, y cuatro tópicos, representados por las flechas. Recibir una flecha representa estar suscrito a un tópico particular. /teleop_turtle es un nodo que captura entrada del teclado.

/rostopic de arriba a la derecha es un echo que imprime en pantalla.

/rostopic de abajo a la izquierda es un nodo que publica un stream de entrada. /turtlesim es un pequeño simulador con una tortuga.

Visibilidad

Del punto de vista de la visibilidad, los colores ayudan a separar componentes. Sin embargo, podemos ver que las primeras dos flechas apuntan con color negro hacia dos nodos verdes, y del nodo rojo de abajo salen flechas verdes hacia los nodos verdes. Esto confunde, por lo que el uso de colores debe estandarizarse. Por otro lado, los nombres deben ser autoexplicativos, como el caso de /teleop_turtle y /turtlesim, en contraste con los otros dos nodos (/rostopic_xxxxx). Por otro lado debemos asegurarnos de que al agregar muchos nodos y tópicos a nuestro grafo, las cosas escalen adecuadamente para no perder visibilidad.

Mapping

Si analizamos de qué forma la interfaz se relaciona con las distintas funciones, podemos notar que puede resultar difícil en ingreso de nuevos tópicos. Lo ideal aquí sería tener una lista desplegable. Por otro lado, no es posible actualmente ingresar mensajes, dado que rqt_graph es únicamente una herramienta de visualización.

Modelo Mental

Considerando el modelo mental de los usuarios, la forma de desplegar el gráfico es la correcta dado que indica de buena manera las relaciones entre componentes. Sin embargo, si deseamos visualizar una mayor cantidad de nodos, el usuario debería tener la posibilidad de reordenar los nodos para mayor claridad y orden.

Feedback

Algo positivo es que al pasar el puntero sobre algún componente, se iluminan las zonas relevantes, entregando buen feedback. De igual manera, el sistema funciona a tiempo real, lo que es esencial para su correcto uso. Sin embargo, hay información clave que no se está mostrando al usuario, tal como contenido de mensajes o estados de los nodos y tópicos. Para navegar el grafo, la herramienta toma tiempos demasiado largos, aproximadamente medio segundo cada click.

Aspectos generales

Otro problema a mencionar son los tiempos de ejecución de la herramienta. Para hacer un grafo típico de mediano tamaño, el programa tarda aproximadamente 10 minutos. Como mencionado en feedback, hacer zoom o scroll en la herramienta puede tomar hasta medio segundo cada click.

2.5.6 Mensajes más comunes en ROS

A continuación se mostrarán los mensajes más comunes utilizados en ROS, divididos por categorías [9]. Es de vital importancia presentar los mensajes más comunes, pues en la herramienta será posible visualizar estos mensajes, y posteriormente debemos definir, mediante estos mensajes, como será la representación gráfica que tendrán.

actionlib_msgs: Mensajes principalmente para la propagación de metas (goals).

Mensaje	Utilización
GoalID	Envía el ID único de una meta.

GoalStatus	Define el estado de una meta.
GoalStatusArray	Guarda los estados para metas que están siendo seguidos por un servidor de acción.

diagnostic_msgs: Mensajes destinados principalmente al diagnóstico del robot.

Mensaje	Utilización
DiagnosticArray	Usado para enviar información de diagnostic acerca del estado del robot.
DiagnosticStatus	Contiene el estado de un componente individual del robot.
KeyValue	Contiene la información de que nombre ponerle al valor, junto a un valor que se desea seguir.
SelfTest	Un servicio que ordena hacerse un test.

geometry_msgs: Mensajes destinados a trabajar con geometría.

Mensaje	Utilización
Point	Contiene un punto en el espacio
Point32	Contiene un punto en el espacio con 32 bits de precisión.
PointStamped	Representa un punto con tiempo específico
Polygon	Especificaciones de un polígono
PolygonStamped	Un polígono con un punto de tiempo específico
Pose	Una posición en el espacio, compuesta de posición y orientación
Pose2D	Posición y orientación en 2 dimensiones
PoseArray	Un arreglo de poses
PoseStamped	Una pose con un tiempo específico
PoseWithCovariance	Una pose en el espacio con incertidumbre

PoseWithCovarianceStamped	Una pose estimada junto a un tiempo específico
Quaternion	Una orientación en el espacio en forma de cuaternión
QuaternionStamped	Un cuaternión con un tiempo específico
Transform	Una transformada entre dos coordenadas en el espacio
TransformStamped	Una transformada con un tiempo determinado
Twist	Expresa velocidad en el espacio dividido entre velocidad lineal y velocidad angular
TwistStamped	Referencia a Twist con un tiempo específico
TwistWithCovariance	Velocidad en el espacio con incertidumbre
TwistWithCovarianceStamped	Un twist estimado junto a un tiempo específico
Vector3	Un vector en el espacio
Vector3Stamped	Un vector junto a un tiempo específico
Wrench	Representa fuerza en el espacio, separado en sus componentes lineales y angulares
WrenchStamped	Wrench junto a un tiempo específico

nav_msgs: Mensajes destinados a la navegación, mapeo y similares.

Mensaje	Utilización
GridCells	Un arreglo de celdas en una cuadrícula 2D
GetMap	Servicio que obtiene un mapa como OccupancyGrid
GetMap	Acción que obtiene un mapa como OccupancyGrid
MapMetaData	Contiene información acerca de las características de OccupancyGrid
GetPlan	Obtiene un plano desde la posición actual hasta la pose meta

OccupancyGrid	Representa un mapa de grilla 2D, en donde cada celda representa la probabilidad de ocupación
Odometry	Representa un estimado de la posición y velocidad en el espacio
Path	Un arreglo de poses que representan un camino para que el robot siga

sensor_msgs: Mensajes destinados al uso de sensores.

Mensaje	Utilización
CameraInfo	Contiene información para una cámara
ChannelFloat32	Mensaje usado por PointCloud para contener datos opcionales asociados a cada punto en la nube.
CompressedImage	Contiene una imagen comprimida
FluidPressure	Lectura única de presión, para ser usado dentro de un fluido (aire, agua, etc.).
Illuminance	Medimiento único de illuminancia fotométrica.
Image	Contiene una imagen no comprimida
Imu	Contiene datos de IMU (Innertial Measurement Unit - Unidad de medida inercial)
JointState	Contiene datos para describir el estado de un conjunto de articulaciones de torque.
Joy	Reporta el estado de joysticks y botones
JoyFeedback	Para entregar feedback a un joystick
JoyFeedbackArray	Un arreglo para publicar múltiples feedbacks de una sola vez

LaserEcho	Es un submensaje de MultiEchoLaserScan (no se usa por separado)
LaserScan	Scaneo único de un encontrador de rangos laser planar
MagneticField	Medición de un campo magnético en un lugar específico
MultiDOFJointState	Representación de un conjunto de articulación con múltiples grados de libertad
MultiEchoLaserScan	Exploración individual de un telémetro láser plano multi-eco
NatSatFix	Solución de navegación por satélite para cualquier Sistema Global de Navegación por Satélite
NavSatStatus	El estado para NavSat
PointCloud	Colección de puntos 3D, además de información opcional adicional de cada punto.
PointCloud2	Colleción de puntos N-dimensionales, que puede contener información tal como normales, intensidad, etc.
PointField	Descripción de un punto en pointCloud2
Range	Lectura de la distancia individual de un ranger activo que emite energía e informa de una lectura de la distancia que es válido lo largo de un arco en la distancia medida.
RegionOfInterest	Especifica una región de interés dentro de una imagen
RelativeHumidity	Lectura de un medidor de humedad relativo
Temperature	Lectura individual de temperatura
TimeReference	Medición de fuente temporal externa no sincronizada con el reloj del sistema

SetCameraInfo	Servicio que le solicita a una camara guardar información como información de calibración

2.5.7 Representación Textual

Esta representación es lo que producen y consumen los comandos textuales ROS. Es entonces importante describirlos pues al implementar la herramienta final, debemos procesar esta información para mostrarla al usuario de forma gráfica, además de tener que enviar mensajes de esta forma en caso de input de usuario.

La idea es descomponer cada mensaje en tipos básicos, tales como vectores, time stamps, números, etc. De esta forma podremos hacer posteriormente el diseño de la herramienta reutilizando al máximo cada uno de los tipos básicos. De esta forma aprovechamos la correspondencia entre la estructura de cada tipo de mensaje y su forma textual.

A continuación se ahondará un poco más en los siguientes tipos:

a) Geometría: Pose
El mensaje Pose es una representación de una pose en el espacio, y está compuesto por posición y orientación. Su definición compacta es:

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation
```

Su representación textual en rostopic pub y echo es:

```
 x: 0.0
  y: 0.0
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 0.0
```

geometry_msgs/Point es una posición en el espacio, definido por 3 valores:

```
float64 x
float64 y
float64 z
```

Por otro lado, geometry_msgs/Quaternion, está definido como:

```
float64 x
float64 y
float64 z
float64 w
```

b) Geometría: TwistStamped
TwistStamped define un giro, pero con coordenadas de referencia y un timestamp:

```
std_msgs/Header header
geometry_msgs/Twist twist
```

Su representación textual en rostopic pub y echo es:

```
geometry_msgs/TwistStamped "header:
  seq: 0
  stamp:
        secs: 0
        nsecs: 0
  frame_id: ''
twist:
  linear:
        x: 0.0
        y: 0.0
        z: 0.0
  angular:
        x: 0.0
        y: 0.0
        z: 0.0"
```

std_msgs/Header define una cabecera de mensajes estándar, y es generalmente utilizado para comunicar datos con timestamp:

```
uint32 seq
time stamp
string frame_id
```

Por otro lado, geometry_msgs/Twist representa un giro. Se divide entre sus componentes lineales y angulares:

```
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

Donde Vector3 es simplemente un vector en el espacio:

```
float64 x
float64 y
float64 z
```

c) Sensores: JointState

Este mensaje contiene datos para describir el estado de un conjunto de articulaciones de torque. El estado de cada articulación (de revolución o prismática) se define por:

- La posición de la articulación
- La velocidad de la articulación
- El esfuerzo aplicado en la articulación

Cada articulación se identifica de forma única por su nombre.

La cabecera especifica el tiempo en el cual cada estado de una articulación fue tomado. Todos los estados de las articulaciones en un mensaje deben ser tomados simultáneamente.

El mensaje consiste de múltiples arregles, uno por cada parte del estado de la articulación. El objetivo es hacer cada uno de los campos opcionales. El mensaje se define como se ve a continuación:

```
std_msgs/Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

(std_msgs/Header fue definido anteriormente).

Su representación textual en rostopic pub y echo es:

```
sensor_msgs/JointState "header:
  seq: 0
  stamp: {secs: 0, nsecs: 0}
  frame_id: ''
name: ['']
position: [0]
velocity: [0]
effort: [0]"
```

d) PointCloud

Este mensaje contiene una colección de puntos tridimensionales, además de información opcional adicional de cada punto. Consiste principalmente de una cabecera con información del tiempo de toma de la muestra, un arreglo de puntos tridimensionales, y una lista de canales:

```
std_msgs/Header header
geometry_msgs/Point32[] points
sensor_msgs/ChannelFloat32[] channels
```

Su representación textual en rostopic pub y echo es:

```
sensor_msgs/PointCloud "header:
  seq: 0
  stamp:
        secs: 0
        nsecs: 0
  frame_id: ''
points:
- x: 0.0
  y: 0.0
  z: 0.0
channels:
- name: ''
  values:
  - 0"
```

Point32 es simplemente un punto con 32 bits de precisión.

sensor_msgs/ChannelFloat32 es usado para contener datos opcionales asociados a cada punto en la nube:

```
string name
float32[] values
```

e) LaserScan

Este mensaje contiene un escaneo único de un telémetro láser plano. Contiene una cabecera, indicando el tiempo de toma de la muestra. Contiene también ángulos para especificar el ángulo inicial, el ángulo final y los incrementos angulares.

La parte de los incrementos temporales nos indica los deltas de las mediciones. Por otro lado tenemos también los rangos mínimos y máximos. También podemos almacenar las intensidades y los rangos:

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Su representación textual en rostopic pub y echo es:

```
sensor_msgs/LaserEcho "echoes:
- 0"
```

3. Desarrollo de la solución

3.1 Introducción

La herramienta creada en esta memoria se divide en tres partes. Por un lado, está la herramienta de exploración, que nos permite revisar la configuración de un robot. Esto se logra haciendo un grafo de todos los nodos y tópicos del sistema, mostrando la relación entre estos elementos. La segunda parte es la diagramación (dibujo) de ciertos tópicos frecuentes, y mostrar en tiempo real la información que poseen.

Todo lo anterior, será implementado en el lenguaje *Smalltalk*, en el ambiente de desarrollo *Pharo*. Es imperioso, entonces, construir una API de ROS para Pharo. Esta API estará encargada de manejar toda la comunicación entre ROS y Pharo, siendo lo más transparente, extensible y robusto posible. La API corresponde a la tercera parte desarrollada.

La API busca crear una capa de comunicación entre Pharo y ROS, como vemos en el diagrama.

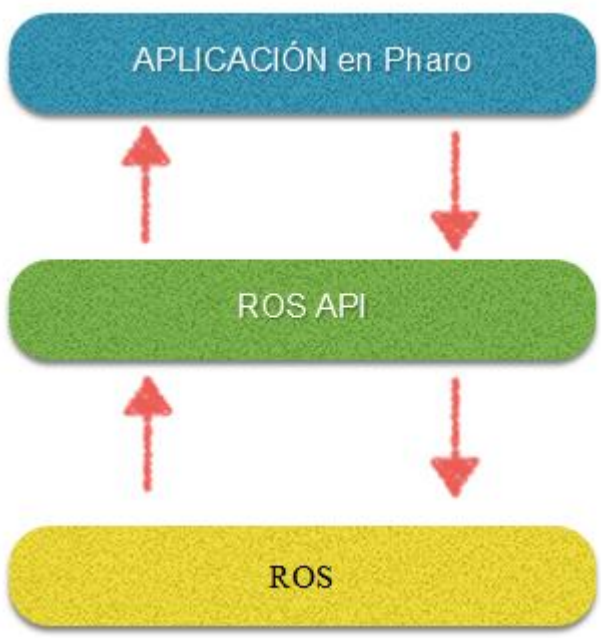


Ilustración 6: Diagrama de comunicación Pharo - ROS

De esta forma, creamos una manera transparente de acceder a toda la información que nos provee ROS. La idea es que se usen exclusivamente las herramientas de consola *(command line tools* [10]) que ROS posee integradas. Debemos, entonces, hacer una serie de llamadas a la consola para obtener la información necesaria.

3.2 API

La API de ROS, está contenida en el paquete ROSAPI. Dentro de este paquete, encontramos la clase ROSAPI. A continuación se listará el conjunto de funciones implementadas, dentro de esta clase, junto a una breve explicación del uso.
La API consiste de las siguientes llamadas:

- Funciones de inicio y término de ROS:
  - **startROSTurtle** Función que inicia ROS, junto a *turtlesim* [11], un *echo* y *teleop_key* (programas de ejemplo, en simulador, de ROS). Es usado para hacer pruebas, y también es usado al comenzar los tests. Esta función no tiene un valor de retorno.
  - **endAllROS** Esta función termina todos cada uno de los nodos y tópicos de ROS, y finaliza ROS mismo. Es usado principalmente para hacer pruebas, y es usado al final de todos los tests. Esta función no tiene un valor de retorno.
- Funciones de echo: Estas funciones trabajan con los *echo* de ROS, cuando le instruimos a ROS que escriba, en la consola, toda la información que pasa por un tópico:
  - **echoProcessOfTopic:{topicName}** Crea un proceso de Smalltalk que se encarga de leer el echo de la consola, para el tópico llamado topicName. El retorno de esta función es el proceso de Smalltalk, al cual podemos pedirle que nos muestre, como un stream, los valores que están siendo pasados a la consola por ROS.
  - **topicEcho:{topicName}** Esta función, similar a la anterior, se encarga de leer lo que echo de ROS escribe en la consola. Sin embargo, en vez de quedarse leyendo, lee todo lo que ya hay, y lo retorna, como un string.
- Funciones de nodos: Las siguientes funciones son usadas para interactuar con los nodos de ROS:
  - **getNodesList** Esta función nos entregará la lista completa de nodos ROS. Con esta lista podemos construir parte del grafo general. Nos retorna una lista de strings con los nombres de los nodos.
  - **newNodeFromROS:{nodeName}** Esta función creará y retornará un objeto de la clase Node. Para rellenar los campos del nodo, tales como suscripciones o publicaciones, realiza una consulta a ROS.
  - **nodeInfo:{nodeName}** Esta función retorna la información específica de un nodo, en un string, como suscripciones o publicaciones, entre otros. Es usada por métodos como newNodeFromROS.

- Funciones de Tópicos: Estas funciones se encargan de interactuar directamente con los tópicos de ROS:
    - **getTopicType:{topicName}** Esta función nos dirá el tipo específico de un tópico. Con esto, podemos separar los tópicos por tipo, para de esa forma saber cómo dibujarlo, según cada tipo. Retorna un string.
    - **getTopicsList** Esta función nos entregará la lista completa de tópicos. Con esta lista, podemos construir una parte del grafo general. Retorna una lista de strings.
    - **sendMessage:{message,type,topicName**} Esta función nos permite enviar un mensaje específico, de un tipo específico, a un tópico. Podemos por ejemplo publicar información a un tópico. No tiene un valor de retorno.
    - **topicInfo:{topicName}** Esta función nos devuelve toda la información de un tópico. Retorna un string.

3.2.1 Tests

Junto a la API, se desarrollaron una serie de tests unitarios. Estos tests son muy importantes, para garantizar la persistencia de la API; ROS, al estar constantemente cambiando, nos presenta un problema para la API: es posible que partes o funciones de la API dejen de funcionar correctamente al ir cambiando ROS. Por ejemplo es muy posible que se hagan cambios en las herramientas de línea de comando de ROS, con lo cual muchos de los métodos que hacen uso de dichas herramientas presentarían problemas.

Si existen cambios en las firmas de las herramientas de consola de ROS, ciertas llamadas de la API fallarán inmediatamente, dado que enviarán a la consola comandos no reconocidos por ROS. Si, por otro lado, los retornos de las funciones de las herramientas de consola de ROS cambian, es decir, hay modificaciones en el formato de las respuestas, entonces los métodos que tienen que parsear información (como newNodeFromROS) fallarán, dado que esperan cierto formato establecido.

Es por ello entonces que se desarrolló una suite de tests unitarios, destinados a cubrir todas las funciones de la API. De esta forma, si los tests pasan exitosamente, nos aseguramos que la API seguirá con su correcto funcionamiento, y por extensión, las herramientas desarrolladas. De lo contrario, es decir, si existen tests que fallan, tendremos información detallada de que parte específica de la API no funcionará bien, y podremos así realizar una corrección a la API. Este paso es fundamental, ya que nos da consistencia en toda la herramienta, y una sólida base de trabajo.

Un gran problema al momento de desarrollar los tests, fue el hecho de que para correr los tests adecuadamente, eran necesarios una serie de procedimientos previos, tales como levantar el proceso principal de ROS (Roscore), creación de tópicos y nodos de prueba, término de todos estos procesos al finalizar el test, etc. Estos procesos son lentos en el sistema. Es por ello que si debemos hacerlos cada vez que hacemos cada uno de los tests, nos encontramos frente al problema de que los tests demoran cantidades enormes de tiempo en correr. La solución a este problema fue la configuración de los tests para que realicen este proceso únicamente antes del primer test, y que pongan término a todos los procesos una vez terminado el último test, en vez de antes y después de cada uno de los tests individuales (ver sección 2.5).

Los tests se dividieron en dos grandes grupos. Por un lado, están los tests fundamentales, destinados a probar los comandos básicos de ROS. Por otro lado, están los tests de la API, destinados a probar las funcionalidades específicas relacionadas a la API, y cómo Pharo interpreta y lee la información.

Tests Fundamentales, pertenecientes a la clase ROSFundamentalsTest:

- **testEcho**: Este test prueba si podemos hacer un echo a la consola.
- **testNodeInfo**: Test que prueba si nos llega información correcta de un nodo.
- **testNodeList**: Test que prueba si nos llega correctamente la lista de nodos.
- **testPublish**: Test que prueba que podamos publicar información a un tópico.
- **testTopicInfo**: Test que prueba que podamos obtener información de un tópico correctamente.
- **testTopicList**: Test que prueba que podamos obtener la lista correcta de tópicos.

Tests de API, pertenecientes a la clase ROSAPITest:

- **testGetNodesList**: Este test prueba que la API esté obteniendo e interpretando la lista de nodos correctamente.
- **testGetTopicType**: Test para probar que la API está obteniendo el tipo correcto de tópico.
- **testGetTopicsList**: Test que prueba que la API está obteniendo e interpretando correctamente la lista de tópicos.
- **testGetNodeInfo**: Test que nos ayuda a verificar que la API esté obteniendo correctamente la información de un nodo.
- **testSendMessage**: Test que nos ayuda a ver que la API está enviando un mensaje correctamente.

3.2.2 Implementación de la API

ROS command line tools

Para la creación de la API, se usaron principalmente las herramientas de consola incluidas en ROS. Las herramientas usadas fueron las siguientes:

- **roscore**: Este comando inicia una colección de nodos y programas, pre requisitos de ROS. Debe haber un roscore corriendo para que los nodos de ROS se puedan comunicar. Para el trabajo, esto se usó principalmente para los tests.
- **rosrun**: Este comando lanza el ejecutable de un paquete ROS. En el caso de la API se utiliza para lanzar la el simulador de tortuga (turtlesim), y también lo usamos en los tests.
- **rosnode info**: Este comando es usado para obtener información de un nodo particular, y en el caso de la herramienta, lo usamos para generar el grafo que luego presentaremos en pantalla.
- **rosnode list**: Este comando nos da la lista completa de nodos ROS. Esto nos permite crear el grafo.
- **rosnode kill**: Este comando nos permite terminar el proceso de un nodo. Es usado al final de los tests cuando ya no usamos un nodo.
- **rostopic echo**: Este comando nos crea un proceso que escucha toda la información que pasa por ese tópico. Es usado por los tests.
- **rostopic info**: Mediante este comando obtenemos toda la información de un tópico. Es usado para la generación del grafo y para las visualizaciones.
- **rostopic list**: Con este comando obtenemos la lista completa de tópicos. Esto lo usamos para la creación del grafo.
- **rostopic pub**: Con este comando podemos publicar información en un tópico. Esto es usado en los tests.
- **rostopic type**: Este comando nos permite obtener el tipo específico del tópico. Esto nos sirve para poder diferenciarlos más adelante y así poder ocupar el tipo de gráfico adecuado para representarlo.

En general, la API está implementada usando llamadas *command* de OSProcess y PipeableOSProcess (ver sección 2.3), y luego, en algunos casos, parseando o trabajando los resultados. Veamos esto con dos ejemplo típicos en su implementación:

A modo de ejemplo, veamos el método getTopicType:topicName, de la clase ROSAPI, que dado un nombre de un tópico, nos devuelve el tipo:

```
getTopicType:topicName
```

```
	^(PipeableOSProcess  command:('rostopic type ',topicName)) output
readStream upToAll: {Character linefeed }.
```

Lo que vemos, es que le pasamos el string ‘rostopic type (topicName)’ a la consola, y leemos hasta que termine la línea del stream sacado del output de la función, y eso es lo que retornaremos.

Otro ejemplo puede ser el método getNodesList, de la clase ROSAPI, que nos retorna la lista de nodos de ROS:

```
getNodesList
	"returns an array containing all nodes. This will be used to build the
graph"
	|stringArray nodeArray|

	stringArray := Array  new.
	stringArray := Utils makeListArray: ((PipeableOSProcess command:
'rosnode list') output ).

	nodeArray := OrderedCollection new.

	stringArray  do: [:item|  nodeArray addLast: ( self
newNodeFromROSNamed: item )].
	^nodeArray.
```

Luego de crear un arreglo vacío, invocamos la función makeListArray de la clase Utils, que nos retorna una lista dado un arreglo. Esta función la llamamos sobre el retorno que nos da PipeableOSProcess command. A command le pasaremos el string ‘*rosnode list’*. Finalmente, a cada nombre de nodo, lo llamaremos con *newNodeFromROSNamed*, de la clase ROSAPI, que va a la consola a buscar la información de un nodo específico.

3.3 Ros Explorer

3.3.1 Introducción

Cada robot que funciona con un sistema ROS, está compuesto, a grandes rasgos, de nodos y tópicos, dentro de otras cosas. Los nodos se suscriben a tópicos para obtener la información que éstos van teniendo, mientras que publican información en otros tópicos. Comprender de qué forma se relacionan los nodos y tópicos de un sistema ROS es fundamental para su entendimiento completo.

ROS Explorer es una herramienta que permite visualizar de forma sencilla y ordenada, de qué forma se ordenan los nodos y tópicos de ROS, que nodo se

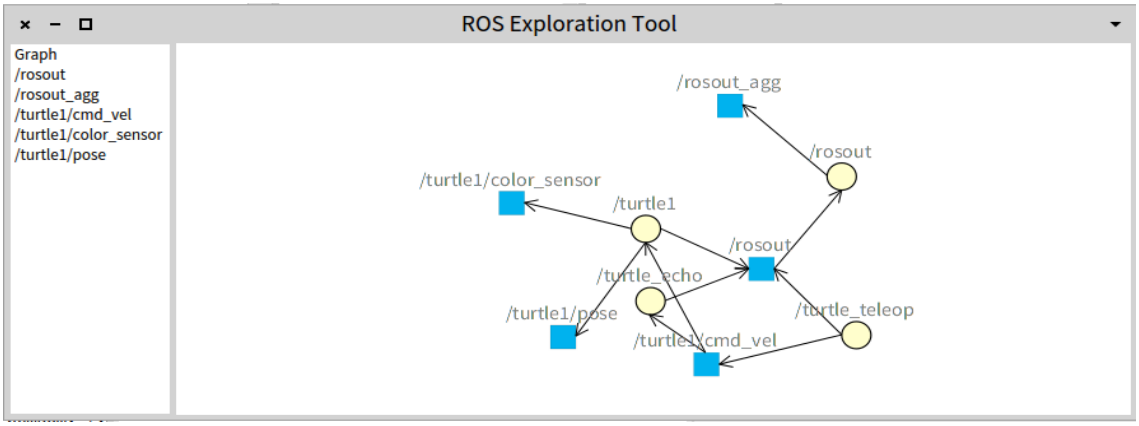


Ilustración 7: Ros Exploration Tool, para un diagrama simple

suscribe a que tópico y que nodos publican en qué tópicos.

En la ilustración vemos la interfaz de ROS Explorer. En el lado izquierdo de la interfaz, vemos la lista de tópicos (/rosout, /rosout_agg, etc.), mientras que en la ventana principal, a la derecha, vemos el grafo de un sistema de ejemplo. Los círculos amarillos son los nodos, mientras que los cuadrados azules son los tópicos. Una flecha desde un nodo a un tópico representa que el nodo está publicando información en ese tópico, mientras que una flecha de un tópico a un nodo representa que el nodo está suscrito a la información de ese tópico. En el ejemplo de más arriba, vemos que por ejemplo el nodo /turtle1 escribe información en turtle1/color_sensor, turtle1/pose y /rosout, mientras que lee información del tópico turtle1/cmd_vel.

3.3.2 Implementación de ROS Explorer

Para la implementación del grafo, se siguieron varios pasos, los cuales serán descritos en detalle a continuación.

DrawGraphOn (de la clase Graph)

Esta es una de las funciones que realiza el trabajo pesado de construir el grafo. Lo primero que hace es hacer llamadas a getROSTopicElements y getROSNodeElements, ambas en la clase Graph. Estas funciones tienen como propósito generar una lista de elementos de Roassal, y retornarlas a la función principal para su posterior dibujo. Ambas funciones, hacen llamados a loadROSTopics y loadROSNodes respectivamente (ambas de la clase Graph). Una vez obtenidos los nodos y los tópicos, se crean los elementos, designando formas, tamaños y colores (círculos amarillos para los nodos y cuadrados azules para los tópicos).

Las funciones auxiliares de loadROSTopics y loadROSNodes, se encargan de llamar a la API, y almacenar las listas respectivas en variables de instancia para su posterior acceso y uso.

Una vez listos en el proceso de creación de nodos y tópicos, creamos un grupo de Roassal, agregamos nodos y tópicos, y agregamos todo a la vista. Es ahora cuando debemos agregar los vértices de nuestro grafo, es decir, las suscripciones y publicaciones. Para éste motivo, llamamos a la funciones de addPubEdges y addSubEdges (ambas de la clase Graph).

En la función addPubEdges, lo primero que hacemos es definir la forma del vértice, en nuestro caso, una flecha negra. Posteriormente, debemos recorrer toda la lista de nodos (guardada en un paso anterior, en la función loadROSNodes). Para cada nodo, vamos a obtener su lista de publicaciones. Una vez obtenida esta información, configuramos el vértice (flecha) para apuntar desde el nodo correspondiente al tópico al cual el nodo está publicando. Para el caso de addSubEdges es el mismo procedimiento, con la diferencia que la flecha irá desde el tópico hacia el nodo suscrito a ese tópico.

A esta altura, ya tenemos en nuestro grafo los nodos, tópicos, suscripciones y publicaciones. Solo queda definir los últimos detalles. Definimos que los elementos de nuestro grafo sean arrastrables, y que bajo cada tópico y cada nodo estén el nombre correspondiente. Finalmente usaremos ForceLayout para ordenar todos los elementos.

Problemas de rendimiento para grafos grandes

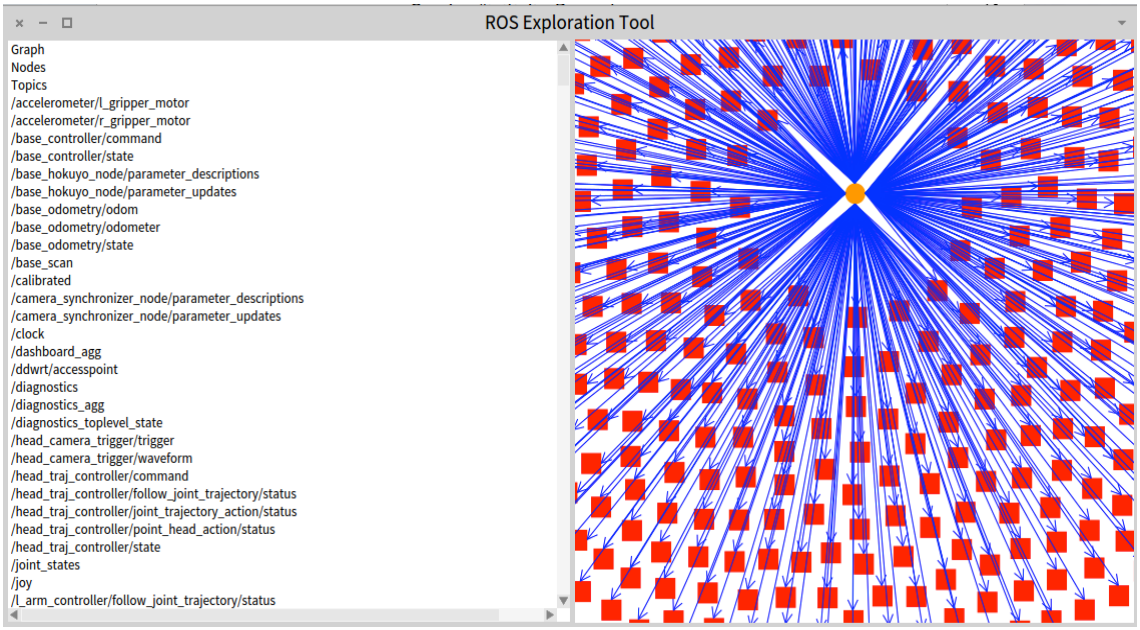


Ilustración 8: ROS Exploration Tool para grandes cantidades de elementos

Al probar la herramienta en un sistema extenso, como es el del robot pr2, que posee cerca de 500 tópicos, podemos notar una grave baja en el rendimiento. Este problema se debe principalmente a dos factores.

El primer factor es la cantidad de llamadas realizadas a la consola. Para cada uno de los tópicos y nodos, debemos realizar una llamada a la consola para verificar y obtener información de suscripciones y publicaciones. Como es de esperar, para 500+ llamadas el sistema se vuelve muy lento. La solución ideal a este problema es realizar una única llamada a ROS que nos entregue el estado completo del sistema. Sin embargo, no existe tal función entre las command-line-tools de ROS.

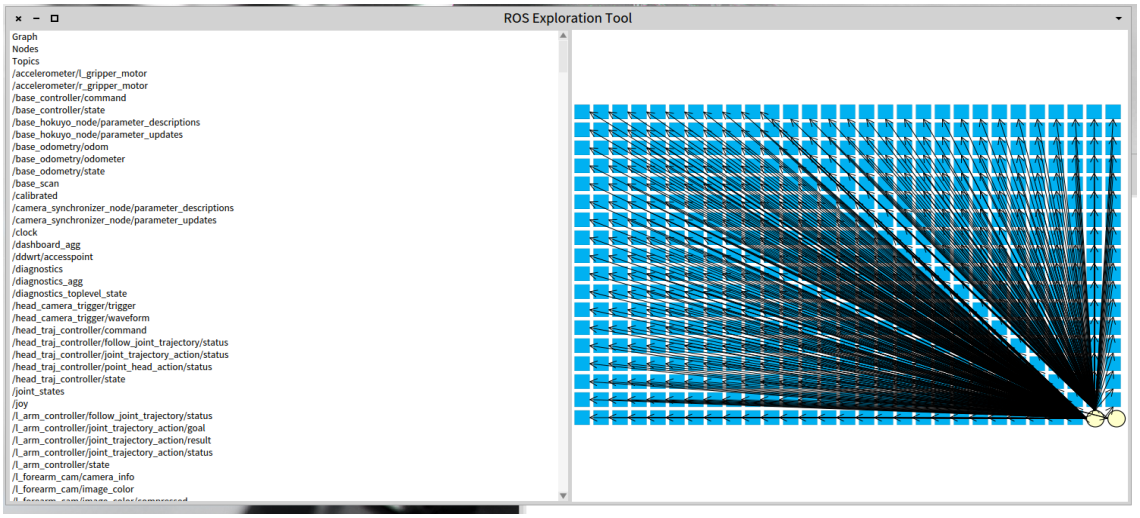


Ilustración 9: ROS Exploration Tool usando Grid Layout

Afortunadamente, ROS posee una master-api, que posee una función exactamente como la que buscamos, llamada getSystemState, que entrega la lista completa de nodos y tópicos, junto a las suscripciones. Lamentablemente, para hacer uso de la master-api es necesario hacerlo a través de un protocolo HTTP llamado XMLRPC. Se implementó una solución alternativa, usando este protocolo, que mejora enormemente el rendimiento, pero requiere la instalación de ciertas librerías para poder usar el protocolo XMLRPC. Veremos detalles de esta implementación alternativa en la sección 3.3.

El segundo factor que produce bajas más considerables aún, es el layout aplicado. ForceLayout contiene dentro de su implementación un algoritmo de orden cuadrático. En la implementación alternativa del grafo, para mejorar el rendimiento, usaremos gridLayout, que si bien visualmente no tiene el atractivo de ForceLayout, si es sumamente rápido.

3.4 Implementación alternativa de Graph usando master-api de ROS y XMLRPC

Como mencionado anteriormente, si usamos la herramienta en una configuración que posee demasiados nodos, estaremos sobrecargando el sistema con llamadas a la consola. Es por ello que se diseñó una implementación alternativa para el grafo. Esta implementación alternativa hace uso de la master-api de ROS, la cual está implementada para funcionar con el protocolo XMLRPC. XMLRPC es un protocolo ligero de comunicación HTTP.

La implementación alternativa se encuentra en el método de instancia *drawGraphOn2*, de la clase Graph. Primero que nada, guarda en la variable

systemState el estado del sistema ROS. Esto lo hace a través del método *getSystemState*, de la clase ROSAPI2. ROSAPI2 fue una clase creada para separarla de la implementación de ROSAPI original (que no requiere el uso de la master-api de ROS y por lo tanto no requiere el uso de XMLRPC).

La clase ROSAPI2 posee un único método de clase, llamado *getSystemState*, que vemos a continuación:

```
getSystemState

        | proxy systemState|
        proxy := XMLRPCProxy withUrl:'http://localhost:11311'.
        systemState := (proxy invokeMethod:'getSystemState'
withArgs:{'/script'}) third.
        ^ systemState.
```

Primero que nada, creamos un proxy, mediante el método XMLRPCProxy withUrl, de la clase XMLRPCProxy, del paquete XMLRPC-Client-Core. La dirección de localhost presente corresponde a la dirección por defecto de la master-api de ROS. Luego guardamos y retornamos lo que nos devolverá la llamada ‘getSystemState’ a la master-api, invocando este método a través del proxy. Notar que retornamos el tercer elemento de esa colección. El primer elemento es el código de retorno, parte del protocolo XMLRPC. El segundo elemento es un mensaje de status. El tercer elemento es el estado del sistema, que es lo que usaremos. El retorno del estado del sistema posee la siguiente forma:

- El retorno es una lista de strings con la siguiente forma:
    - [publicadores, suscriptores, servicios]
- Los publicadores son una lista de strings con la siguiente forma:
    - [[topic1, [topic1Publisher1...topic1PublisherN]] ...]
- Los suscriptores son una lista de strings con la siguiente forma:
    - [[topic1, [topic1Subscriber1...topic1SubscriberN]] ...]
- Los servicios son una lista de strings con la siguiente forma:
    - [[service1, [service1Provider1...service1ProviderN]] ...].

Luego de eso, volviendo al método *drawGraphOn2*, ya poseemos el sistema completo, y ordenado. Basta ahora pasar los arreglos de systemState a los métodos previamente utilizados y explicados en la sección 3.2.2.

3.5 Gráficos de Mensajes en tópicos

3.5.1 Introducción

La tercera parte de la memoria consiste en la visualización de ciertos mensajes dentro de los tópicos. Todo el sistema está armado de forma extensible, por lo que en el futuro será fácil para alguien que desee continuar con el trabajo, extender la cantidad de visualizaciones. Para crear una nueva visualización hay que seguir el proceso explicado en 3.3.4.

3.5.2 Visualizaciones

Twist

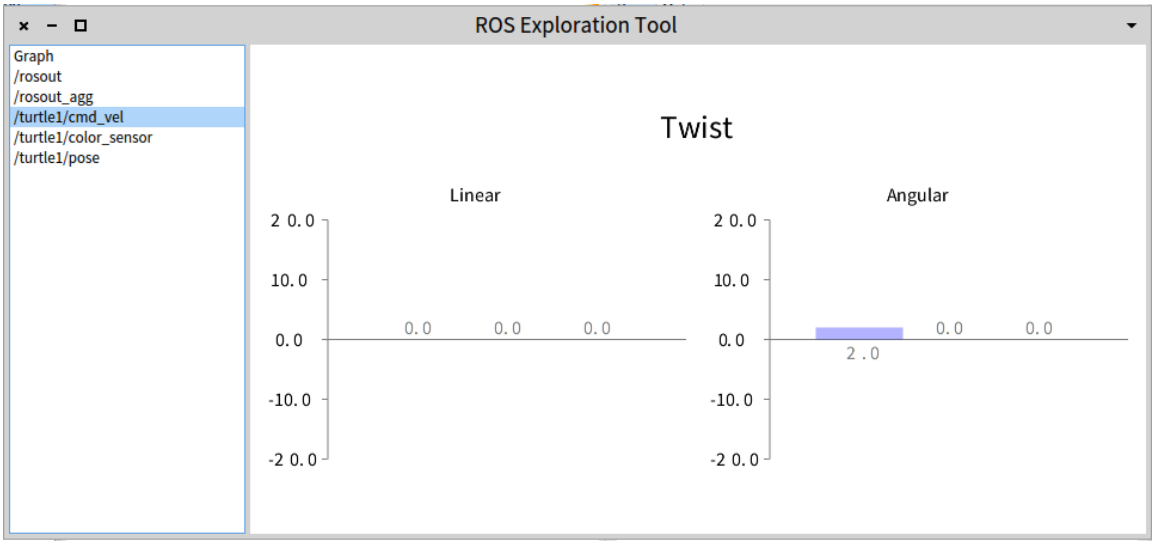


Ilustración 10: Gráfico de un Twist

Un twist representa una velocidad, dividida en sus componentes lineal y angular.

Al momento de explorar un robot que funciona con ROS, nos gustaría poder saber rápidamente cuales son las distintas componentes de velocidad que presenta. La velocidad linear y angular debemos presentarlas de forma separada, dado que comparar componentes angulares y lineales no hace sentido y no es de utilidad. Es por ello, que se decidió hacer este gráfico como dos sub gráficos separados. Sin embargo, la comparación de dos elementos dentro de una misma componente, es decir, por ejemplo, comparar las velocidades en X y en Z de un robot, si hace sentido; es por ello que un gráfico de barras nos presenta con la mejor manera de realizar estas comparaciones con sólo una mirada. Por ejemplo, en la ilustración, podemos de inmediato reconocer que el robot no tiene velocidad linear, pero si una pequeña velocidad angular en la primera componente (2.0, en este caso).

La clase twist posee cuatro variables de instancia, llamadas linear, angular, graphLinear y graphAngular. Linear y angular representan arreglos que contienen los datos para los grafos linear y angular respectivamente, mientras que graphLinear y graphAngular son los objetos de los grafos. Los necesitamos (las referencias) para poder modificarlos dinámicamente.

Los métodos de clase de Twist son creadores de instancias, a partir de ciertos objetos. El principal que se usa es newFromString:aString, que crea un twist de acuerdo a un string, como visto en la sección 2.5.7 Representación Textual. Para esto, separa el string en líneas, y luego llama al método newFromArray:anArray, que toma un arreglo de líneas y lo transforma en un twist.

Dentro de los métodos de instancia, posee el método draw, que retorna una vista con el twist dibujado. Además posee el método drawOn:aView, que dibuja el twist sobre la vista entregada. Ambos métodos son muy similares en su implementación. Primero, debemos crear arreglos de datos para entregarle a los gráficos. Estos arreglos tienen la siguiente forma:
{ {‘x’ . valorX} . {‘y’ . valorY } . {‘z’ . valorZ} } .
‘x’, ‘y’ y ‘z’ representan las tres coordenadas que usaremos, mientras que valorX, valorY y valorZ son los valores para cada una de las coordenadas.

Luego llamamos a la función *drawComposedDynamicGraph* (y *drawComposedDynamicGraphOn*, respectivamente), de la clase Utils, que hará lo necesario para dibujar nuestro twist.

Otro método interesante de instancia de la clase twist es *equals:aTwist*, que compara si dos twists son iguales (comparando cada una de sus componentes por separado).

Laser Scan

Un laser scan representa un barrido de un láser en una zona.
Podemos ver en el gráfico el arreglo de valores tomado, ordenado según el orden en el que los valores fueron capturados.

Al explorar una configuración de un robot con un laserscan, nos gustaría poder ver las distancias de los puntos tomados. Es por ello que se decidió usar un gráfico de puntos. Con este tipo de visualización podemos, de una sola mirada, saber cómo los puntos están ordenados en el espacio, y cuáles fueron las distancias que fueron capturadas.

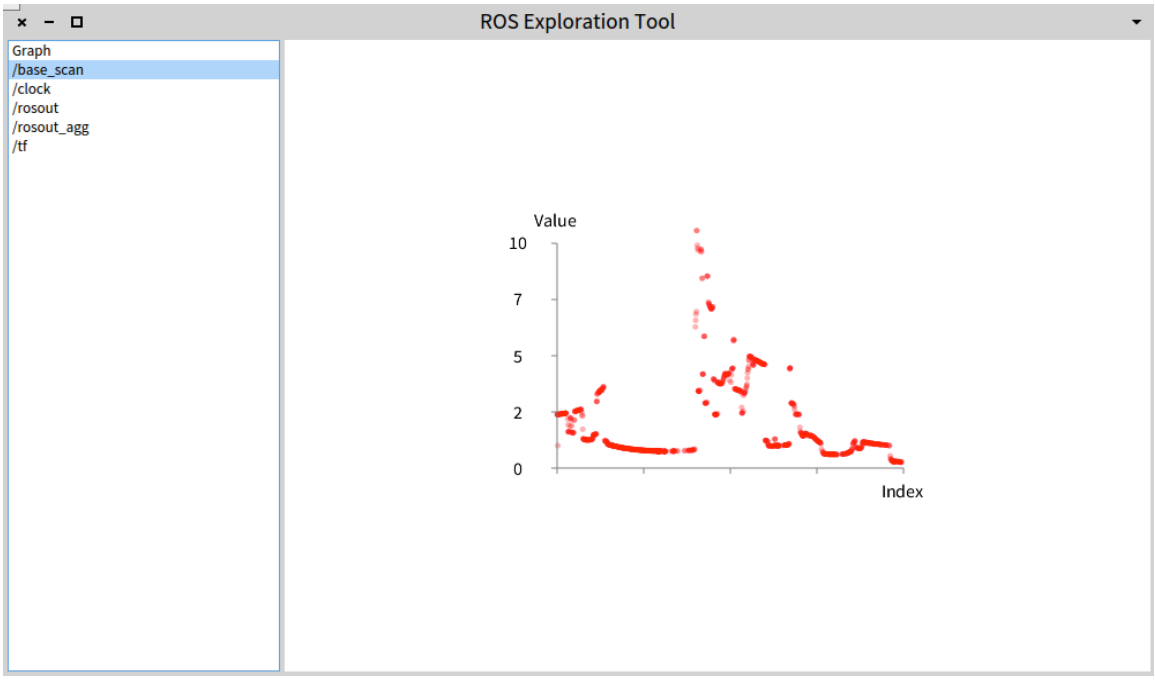


Ilustración 11: Gráfico de un Laser Scan

La clase LaserScan, posee numerosas variables de instancia. La mayoría de ellas almacena arreglos de valores para graficar, además de tener almacenado el grafo mismo.

Dentro de sus métodos de clase, podemos encontrar numerosos métodos inicializadores, que nos permiten crear un laser scan a partir de otros objetos. El que usamos en nuestro caso es *newFromString:aString*, que dado un string textual de un BaseScan, como visto en la sección 2.5.7, crea un objeto laser scan parseando el texto, similar a como se hace en la clase twist.

De los métodos de instancia, podemos destacar los métodos *draw* y *drawOn:aView*, métodos que grafican el laser scan, y retornan una vista, o lo dibujan sobre una vista que entreguemos, respectivamente.

Para ellos, lo primero es crear un nuevo *RTDynamicGrapher*, de la clase RTDynamicGrapher. Configuramos los rangos mínimos y máximos del grafo, y configuramos los colores y formas que tendrá. A continuación agregamos los puntos al grafo, ponemos las etiquetas a los ejes, y retornamos. La vista (o bien ponemos el grafo en la vista entregada, respectivamente).

Joint States

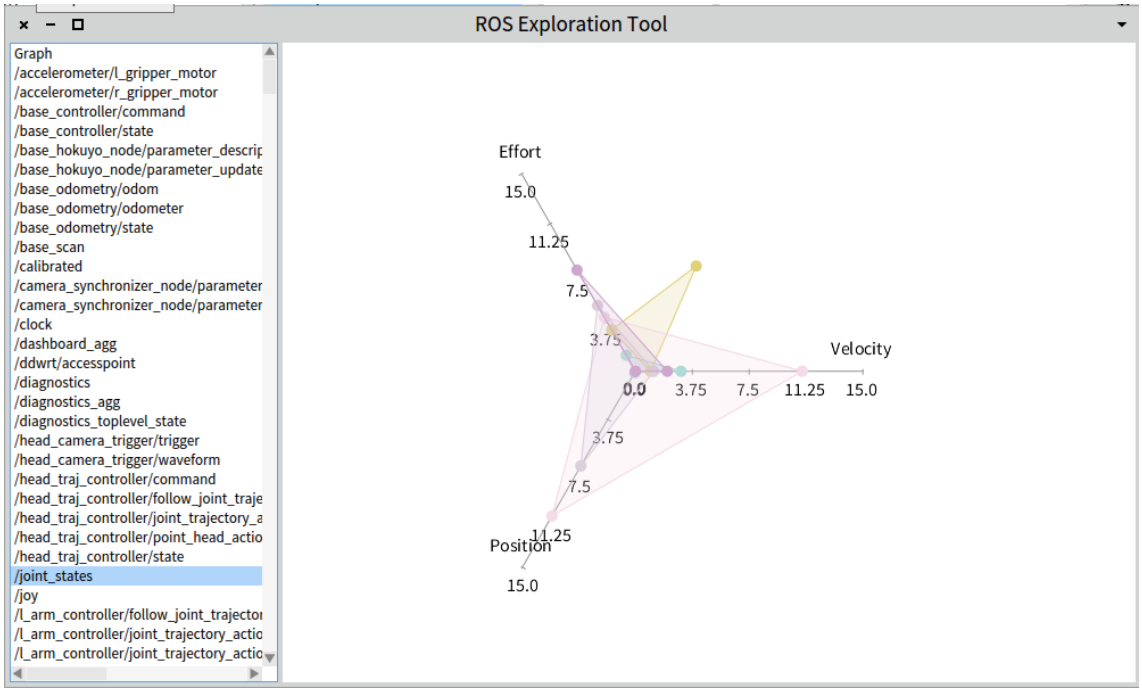


Ilustración 12: Gráfico de Joint States

Joint States representa el estado de las 'articulaciones' de un robot. Nos muestra, para cada una de las articulaciones, las tres componentes de la articulación, es decir, posición, velocidad y esfuerzo.

Cada una de las tres componentes (esfuerzo, velocidad y posición) son medidas físicas independientes, por lo que compararlas entre sí no tiene sentido alguno. Sin embargo, sí es posible comparar el esfuerzo, la velocidad o la posición de dos articulaciones distintas, por lo que la visualización presentada nos ayuda enormemente a ello. Podemos distinguir, rápidamente, cuáles son las articulaciones que presentan más esfuerzo, velocidad o posición.

Al igual que las dos clases anteriores, JointState almacena como variables de instancia los arreglos con datos que contiene. En este caso es un arreglo para cada componente, es decir, un arreglo para velocidad, otro para posición, y otro para esfuerzo. Por otro lado también posee almacenado el objeto del grafo, para poder modificarlo dinámicamente de ser necesario.

Dentro de sus métodos de clase, al igual que en los casos anteriores, podemos encontrar los inicializadores, para poder convertir un string en un objeto JointState.

Sus métodos de instancia también son casi idénticos a los objetos anteriores, salvo el método *createKiviatArrays*. Este método nos sirve de utilidad para transponer los arreglos entregados. La manera de entregar los valores al gráfico, es a través de una serie de 3-tuplas, conteniendo posición, velocidad y esfuerzo en cada una. Entonces, el método toma los tres arreglos con valores uqe tenemos almacenados, y lo transforma en la serie de 3-tuplas requeridas para graficar.

3.5.3 Implementación de visualizaciones

ROSDrawManager

Esta clase fue creada para manejar todas las visualizaciones que se crearon para los distintos tipos de mensajes. Se usa mediante el método de clase draw:{aTopic} On:{aView}, que recibe un nombre de tópico, y una vista, para dibujar el contenido de ese tópico en la vista entregada.

Lo primero que hace es hacer una llamada a la API, para obtener el tipo de tópico que queremos graficar. Posteriormente, y dependiendo del tipo de tópico, hace una llamada específica para dibujarlo, como veremos a continuación. De no tener un tipo especificado, pide a la API la información específica del tópico, y la muestra como un texto.

En las visualizaciones creadas, usamos drawTwistOn, drawLaserScanOn, y drawJointStateOn. Todas estas llamadas se hacen a un objeto de la clase ROSEchoManager.

ROSEchoManager

Todas las visualizaciones mostradas son de carácter dinámico. Eso significa que van cambiando a medida que los mensajes del tópico graficado van cambiando, a tiempo real. Es por ello que debemos estar permanentemente verificando si es que ha habido cambios en el tópico. Para este propósito se creó la clase ROSEchoManager, que maneja la conexión con el tópico.

Los métodos de ROSEchoManager: drawTwistOn, drawLaserScanOn & drawJointStateOn, se encargarán de dibujar un twist, laser scan o joint states dinámicos sobre la vista que le entreguemos. Para ello, proceden como sigue.

En primer lugar, crea un twist, laser scan o joint states vacíos, es decir, con todas sus componentes en cero. Luego, crea una nueva instancia de ROSEchoManager, para manejar las conexiones que hará con la API y con la

consola. También definimos un string auxiliar. Este string auxiliar almacenará el mensaje obtenido de la consola. En iteraciones futuras, lo compararemos con nuevos mensajes obtenidos, y si vemos que cambia, sabemos que poseemos nueva información, y actualizamos los gráficos. Actualizamos también el string auxiliar, para usarlo de punto de comparación para las siguientes iteraciones.

Dado que tenemos que estar permanentemente leyendo la consola, a través de la API, para ver si hay cambios en un tópico, tenemos que trabajar con hilos y paralelismo a fin de no bloquear todo el sistema mientras hacemos las lecturas. Creamos por tanto dicho hilo, y entramos en un ciclo infinito dentro de ese hilo. El hilo será retornado por el método, de forma que cuando cerremos la ventana, o cambiemos de tópico, podemos cerrar el hilo.

Dentro del ciclo principal, comparamos el mensaje auxiliar (inicializado vacío) con el mensaje nuevo leído por nuestra API. Si es que son distintos, significa que tenemos un nuevo mensaje, o el mensaje ha cambiado, por lo que debemos actualizar nuestro grafo. Creamos entonces un nuevo Twist, laser scan o joint states, con el mensaje nuevo, y lo agregamos al grafo, reemplazando el anterior.

Cabe mencionar, en este punto, que para el caso de twist, tenemos ambos grafos (el lineal y el angular) almacenados convenientemente en variables de instancia de la clase Twist, a modo de poder acceder fácilmente a cada uno de ellos por separado.

En el caso del joint states, el mensaje que recibimos son tres arreglos para cada componente de una articulación, y cada componente con su lista de articulaciones. Es por ello que debemos hacer aquí una pequeña transformación, para cambiarlo por una serie de trios para cada una de las articulaciones. Esto se debe a que el grafo de joint states recibe la información de esta forma.

3.5.4 Creación de nuevas visualizaciones

Para crear una nueva visualización debemos seguir el siguiente proceso:

1. Crear la clase del objeto que queremos visualizar. Dentro de sus variables de instancia, debemos tener una para los datos del objeto, y una para el grafo.
2. Crear un método de clase, para iniciar el objeto dada su representación textual en string.
3. Crear un método de instancia llamado *draw*, que crea la visualización.

4. Crear un método de instancia llamado *drawOn:aView*, que dada una vista, crea la visualización sobre esa vista.

5. Crear el método de instancia *equals:anObject*, que compara dos objetos para determinar si poseen los mimos valores. Esto es usado para actualizar las visualizaciones y permiten que sean dinámicas.

6. El método *type*, que debe retornar un string indicando el tipo de objeto (por ejemplo, para el caso del laser scan, corresponde a 'sensor_msgs/LaserScan'.

7. Agregar, al método *draw:object on:aView* de la clase ROSDrawManager, el tipo de objeto y hacer una llamada al método *drawOn* del objeto que queremos visualizar.

3.6 Errores

3.6.1 Problema de rendimiento para más de 50 nodos y tópicos

Como mencionado anteriormente, la implementación original del grafo de nodos y tópicos, presenta una considerable baja en el rendimiento cuando tratamos con grandes cantidades de nodos y tópicos. A modo de referencia, en una máquina virtual, graficar cerca de 500 tópicos tarda alrededor de 15 minutos. A continuación se entrará en mayor detalle respecto a este problema, junto con plantear una solución alternativa encontrada.

Los problemas eran ocasionados por dos factores importantes.

El primer factor era la cantidad de accesos a consola. Para construir el grafo, es necesario saber precisamente a que tópicos se está suscribiendo un nodo, así como también a que tópicos está publicando. Es por ello que debemos, para cada nodo, realizar una llamada a la consola que nos entrega esta información. Para altas cantidades de elementos, estas llamadas comienzan a alentar el sistema completo, bajando el rendimiento general. Para solucionar este problema se propuso una segunda implementación, que hace uso de la master-api de ROS. Esta API funciona a través del protocolo XMLRPC, un protocolo ligero de conexión http. Llamamos entonces a getSystemState, la función que nos entrega instantáneamente toda la información necesaria para armar el grafo. La desventaja es que ya no estamos haciendo uso exclusivamente de las command-line-tools de ROS, y debemos instalar librerías adicionales en pharo.

El segundo factor, y más importante, es el algoritmo usado por ForceLayout, que posee componentes cuadráticas dentro de su implementación. Para solucionar este problema, en la nueva implementación, se usó GridLayout,

que aunque no posee el atractivo visual que posee ForceLayout, si es muy veloz en su ordenamiento.

3.6.2 Problema de hilos sin terminar

Para la implementación de los grafos dinámicos de visualización de tópicos, debemos crear un hilo que esté constantemente accediendo a los tópicos, a través de la API, para monitorear posibles cambios y así actualizar la visualización.

En la implementación de las visualizaciones, retornamos el hilo encargado de las lecturas, y lo guardamos como variable de instancia del grafo. De esta forma, si cerramos la ventana de la herramienta, o bien cambiamos de tópico o nos vamos al grafo principal nuevamente, revisamos si el hilo existe, y de ser así, lo terminamos. Este debería, en teoría, también acabar con los procesos hijos de este hilo. Sin embargo, en la práctica, nos encontramos que al terminar con el hilo de las consultas quedan procesos residuales, algunos con accesos a consola, otros con semáforos, dando vueltas en nuestro sistema. Esto genera un problema, dado que se van acumulando los procesos sueltos y hacen que todo el sistema funcione más lento. Los procesos se pueden cerrar manualmente desde el explorador de procesos de pharo.

3.6.3 Turtle/teleop_key deja la consola bloqueada

Dentro de los ejemplos de ROS, es común usar el simulador de tortuga (TurtleSim) incluido en ROS. En los ejemplos también podemos usar teleop_key, un nodo ros que nos permite manejar la tortuga, desde la consola, con las flechas del teclado. Sin embargo, esto genera que la consola quede bloqueada en escritura, capturando únicamente las flechas. Pese a cerrar este nodo posteriormente, la consola misma queda bloqueada.

La API de ROS, para su correcto funcionamiento, debe hacer una serie de escrituras en la consola para todos sus métodos. Esto significa, que si corremos el ejemplo de la tortuga y de teleop_key desde pharo, la consola desde la cual se lanzó pharo quedará bloqueada. Esto lleva como consecuencia que la API dejará de responder adecuadamente. Cerrar Pharo, y la consola, solucionan este problema.

3.6.4 Aio Plugin

Cuando corremos la herramienta, la primera vez que la API hace uno de sus accesos, Pharo nos lanza un error de plugin.
Una vez cerrada esta ventana, es posible usar la herramienta con normalidad.

Este error impide la comunicación entre Pharo y ROS, mediante la API y lecturas a la consola. Este es un problema sumamente grave, debido a que toda la herramienta se basa en las lecturas a la consola con la información de ROS.

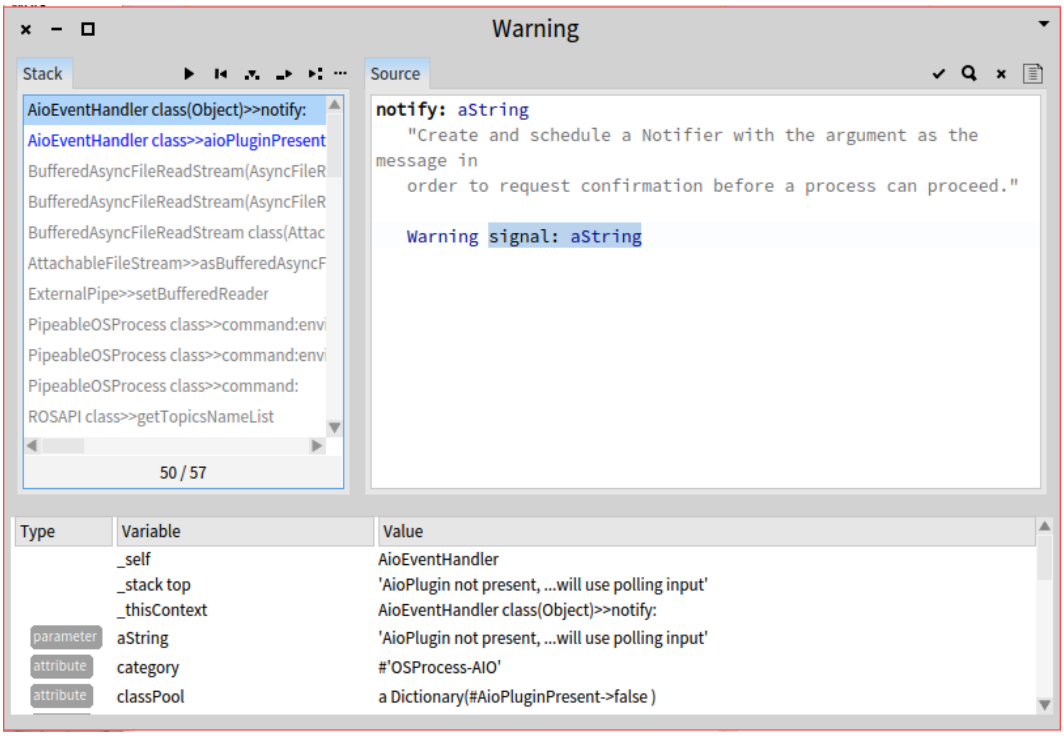
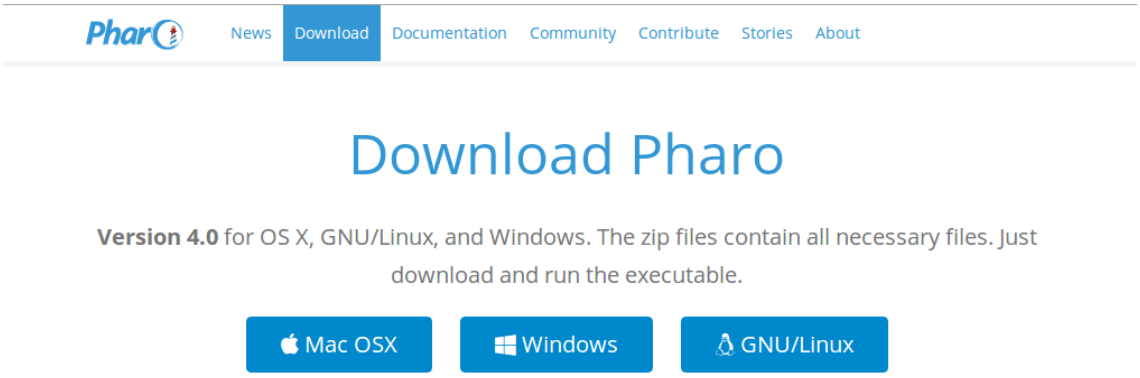


Ilustración 13: AIO Plugin ERROR

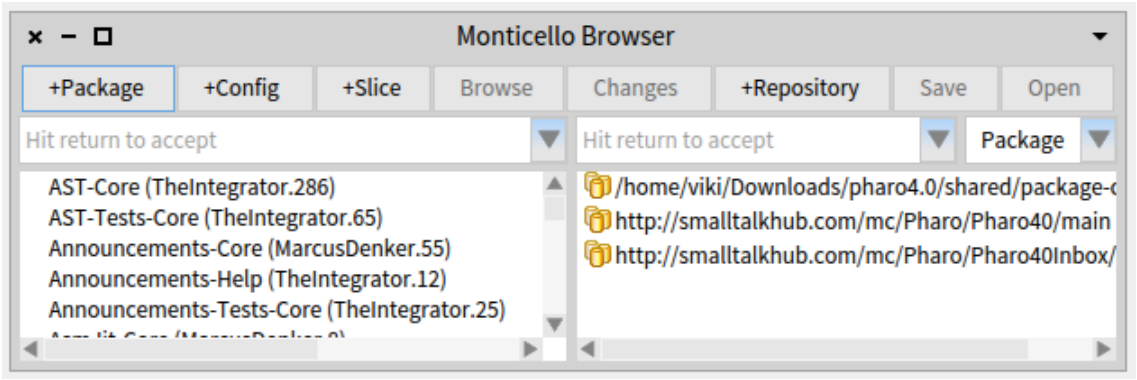
3.7 Instrucciones de Instalación

La herramienta desarrollada es completamente open-source, y puede usarse de la siguiente manera:

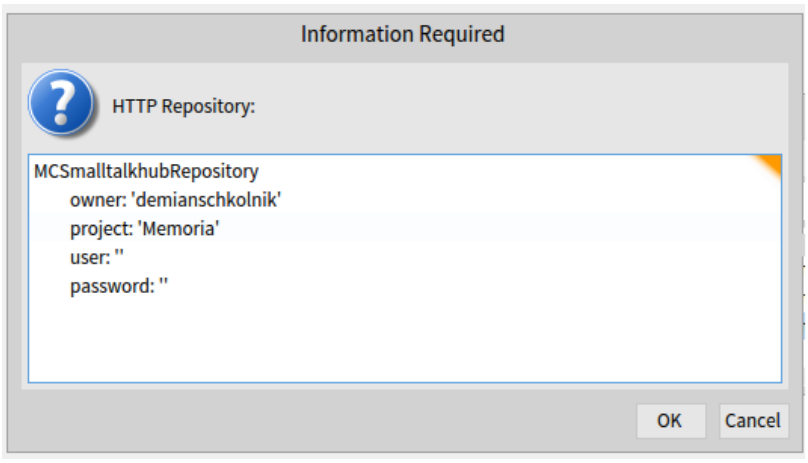
1. Descargar la última versión de Pharo, desde www.pharo.org/download.



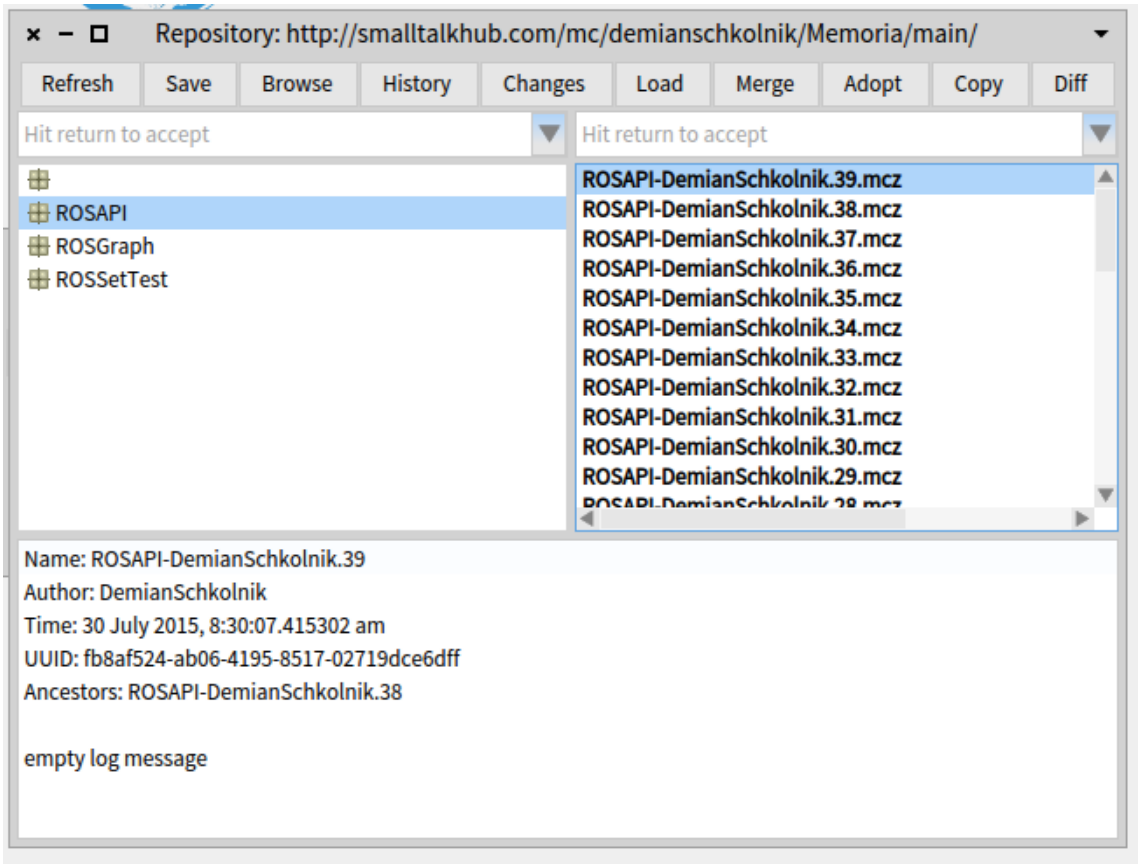
2. En Pharo, abrir Monticello Browser.



3. Presionar el botón +Repository. Luego Escoger el tipo de repositorio 'SmallTalkHub'.
4. Configurar el repositorio, poniendo en owner: 'demianschkolnik', y en proyect: 'Memoria'.



5. Seleccionar ROSAPI y ROSGraph, seleccionar la última versión (en la ilustración, la 39 por ejemplo), y poner Load.



4. Conclusiones

La presente memoria consistió en la creación de una herramienta, que permite la visualización y exploración de un sistema hecho en ROS. Para ello, había que hacer un primer análisis, que estudiara de qué forma se podía acceder a ROS, y cuáles eran los mensajes más comunes. Luego, debía hacerse la herramienta, usando el lenguaje Smalltalk y el ambiente de desarrollo Pharo. La razón de esto es porque se iba a usar un motor de visualizaciones llamado Roassal, el cual está siendo desarrollado por varios miembros del mismo departamento de computación.

Para resolver el problema, se creó una API que permitiera interactuar desde Pharo directamente con ROS. Esto era un paso indispensable, dado que Roassal está hecho en Pharo, y debíamos usarlo desde ahí. Junto a la API, hubo que implementar una serie de tests, que nos permitieran darle robustez a la API. De esta forma, si algo en ROS cambia, nos será posible ver que tenemos que cambiar en la API para que la herramienta siga funcionando.

Por otro lado, se implementó el grafo general que ilustra de qué forma nodos y tópicos se relacionan. Para ello, se realizan llamadas a ROS, a través de la API que creamos. Las respuestas de ROS, están en texto plano, y debimos parsearlas a objetos de Pharo. Específicamente, los nodos y tópicos de ROS, y almacenar, para cada nodo, cuáles eran los tópicos a los cuales se suscribe y cuáles eran los tópicos en los cuales publica. Finalmente, usando Roassal, se crea el grafico.

La tercera y última parte de la solución era crear gráficos para distintos tipos de mensajes. Se determinaron tres tipos básicos, y se implementaron los gráficos para estos tipos. Cabe mencionar que los gráficos implementados son dinámicos: esto significa que si los valores del objeto graficado cambian, estos cambios se verán reflejados automáticamente en el gráfico.

Debemos destacar que la mayoría de los elementos de la solución se encuentran ordenados en una serie de paquetes y clases, de forma que la solución es extensible o modificable. La herramienta en sí está desacoplada de la API, por lo que es posible modificar cualquiera de las dos por separado, lo que es una gran ventaja si se desea seguir trabajando en este proyecto a futuro.

Por otro lado, se detectaron numerosos problemas, sobre todo en sistemas ROS con grandes cantidades de elementos. Para solucionar los problemas de rendimiento en estos casos, se diseñó e implementó una solución alternativa, que puede usarse para agilizar el proceso de graficado

y reducir la cantidad de recursos compartidos. Se mantiene como solución alternativa debido a que usa paquetes externos, y no es autocontenida, como la solución principal.

Con respecto a los objetivos del trabajo de memoria:

- Creación de una herramienta gráfica que permita mostrar gráficamente la topología presente al conectarla a un robot que utilice ROS, mostrando nodos, tópicos y mensajes, y como estas componentes están conectadas entre sí.
    - Mediante la obtención de listas de nodos y tópicos, en conjunto con la obtención de la información específica de éstos, todo a través de la API, se creó la herramienta para visualizar de manera ordenada la configuración presente en un sistema ROS.
- Determinar cuáles son los tipos de mensajes más comunes en ROS. Se debe estudiar cómo se deben visualizar estos mensajes, y como sería una interfaz adecuada para ellos.
    - A través de investigación y un análisis realizado (sección 2.5.6), se llegó a la conclusión de los mensajes más comunes de ROS, además del diseño de interfaces adecuadas para la correcta visualización e interpretación de dichos mensajes.
- La herramienta debe permitir a un usuario revisar el flujo de mensajes dentro de la topología, mostrando en tiempo real y de la forma más adecuada los mensajes de los tipos más comunes.
    - Usando la información recopilada, y usando el motor de visualizaciones ágiles ROASSAL, se generaron diversas visualizaciones para los tipos de mensaje más comunes.
- La herramienta deberá estar estructurada de forma modular, permitiendo una adaptación sencilla si la interfaz hacia ROS cambia, o cuando se quieran integrar nuevos tipos de visualizaciones de mensajes.
    - A través de una buena separación de API con la implementación de la herramienta, logramos exitosamente separar las componentes de forma muy modular. De esta forma podemos actualizar o mejorar las partes sin tener que modificar todo el conjunto. De la misma forma, el hecho de haber usado Smalltalk y la forma como se estructuran las clases (orientado a objetos), nos permite extender este sistema fácilmente.

5. Bibliografía

[1] Gephi: The Open Graph Viz Platform: https://gephi.github.io/
[2] Grafos - software para la construcción, edición y análisis de grafos: http://arodrigu.webs.upv.es/grafos/doku.php
[3] Graphviz - Graph Visualization Software: http://www.graphviz.org/
[4] Data Display Debugger: http://www.gnu.org/software/ddd/
[5] The Robot Operating System (ROS): http://www.ros.org/
[6] Nielsen, Jakob. *Designing web usability: The practice of simplicity*. New Riders Publishing, 1999.
[7] Nielsen, Jakob. "Ten usability heuristics." (2005): 2008.
[8] Shneiderman, Shneiderman Ben, and Catherine Plaisant. "Designing the user interface 4 th edition." ed: Pearson Addison Wesley, USA (2005).
[9] http://wiki.ros.org/common_msgs
[10] http://wiki.ros.org/ROS/CommandLineTools
[11] http://wiki.ros.org/turtlesim

Apéndice 1 : Tablas de mensajes más comunes en ROS:

A continuación se mostrarán los mensajes más comunes utilizados en ROS, divididos por categorías (http://wiki.ros.org/common_msgs).

actionlib_msgs: Mensajes principalmente para la propagación de metas (goals).

Mensaje	Utilización	Definición del mensaje
GoalID	Envía el ID único de una meta.	time stamp string id
GoalStatus	Define el estado de una meta.	uint8 PENDING=0 uint8 ACTIVE=1 uint8 PREEMPTED=2 uint8 SUCCEEDED=3 uint8 ABORTED=4 uint8 REJECTED=5 uint8 PREEMPTING=6 uint8 RECALLING=7 uint8 RECALLED=8 uint8 LOST=9 actionlib_msgs/GoalID goal_id uint8 status string text
GoalStatusArray	Guarda los estados para metas que están siendo seguidos por un servidor de acción.	std_msgs/Header header actionlib_msgs/GoalStatus[] status_list

diagnostic_msgs: Mensajes destinados principalmente al diagnóstico del robot.

Mensaje	Utilización	Definición del mensaje
DiagnosticArray	Usado para enviar información de diagnostic acerca del estado del robot.	std_msgs/Header header diagnostic_msgs/DiagnosticStatus[] status
DiagnosticStatus	Contiene el estado de un componente individual del robot.	byte OK=0 byte WARN=1 byte ERROR=2 byte STALE=3 byte level string name string message string hardware_id diagnostic_msgs/KeyValue[] values

KeyValue	Contiene la información de que nombre ponerle al valor, junto a un valor que se desea seguir.	string key string value
SelfTest	Un servicio que ordena hacerse un test.	string id byte passed diagnostic_msgs/DiagnosticStatus[] status

geometry_msgs: Mensajes destinados a trabajar con geometría.

Mensaje	Utilización	Definición del mensaje
Point	Contiene un punto en el espacio	float64 x float64 y float64 z
Point32	Contiene un punto en el espacio con 32 bits de precisión.	float32 x float32 y float32 z
PointStamped	Representa un punto con tiempo específico	std_msgs/Header header geometry_msgs/Point point
Polygon	Especificaciones de un polígono	geometry_msgs/Point32[] points
PolygonStamped	Un polígono con un punto de tiempo específico	std_msgs/Header header geometry_msgs/Polygon polygon
Pose	Una posición en el espacio, compuesta de posición y orientación	geometry_msgs/Point position geometry_msgs/Quaternion orientation
Pose2D	Posición y orientación en 2 dimensiones	float64 x float64 y float64 theta
PoseArray	Un arreglo de poses	std_msgs/Header header geometry_msgs/Pose[] poses
PoseStamped	Una pose con un tiempo específico	std_msgs/Header header geometry_msgs/Pose pose
PoseWithCovariance	Una pose en el espacio con incertidumbre	geometry_msgs/Pose pose float64[36] covariance
PoseWithCovarianceStamped	Una pose estimada junto a un tiempo específico	std_msgs/Header header geometry_msgs/PoseWithCovariance pose

Quaternion	Una orientación en el espacio en forma de cuaternión	float64 x float64 y float64 z float64 w
QuaternionStamped	Un cuaternión con un tiempo específico	std_msgs/Header header geometry_msgs/Quaternion quaternion
Transform	Una transformada entre dos coordenadas en el espacio	geometry_msgs/Vector3 translation geometry_msgs/Quaternion rotation
TransformStamped	Una transformada con un tiempo determinado	std_msgs/Header header string child_frame_id geometry_msgs/Transform transform
Twist	Expresa velocidad en el espacio dividido entre velocidad lineal y velocidad angular	geometry_msgs/Vector3 linear geometry_msgs/Vector3 angular
TwistStamped	Referencia a Twist con un tiempo específico	std_msgs/Header header geometry_msgs/Twist twist
TwistWithCovariance	Velocidad en el espacio con incertidumbre	geometry_msgs/Twist twist float64[36] covariance
TwistWithCovarianceStamped	Un twist estimado junto a un tiempo específico	std_msgs/Header header geometry_msgs/TwistWithCovariance twist
Vector3	Un vector en el espacio	loat64 x float64 y float64 z
Vector3Stamped	Un vector junto a un tiempo específico	std_msgs/Header header geometry_msgs/Vector3 vector
Wrench	Representa fuerza en el espacio, separado en sus componentes lineales y angulares	geometry_msgs/Vector3 force geometry_msgs/Vector3 torque
WrenchStamped	Wrench junto a un tiempo específico	std_msgs/Header header geometry_msgs/Wrench wrench

nav_msgs: Mensajes destinados a la navegación, mapeo y similares.

Mensaje	Utilización	Definición del mensaje

GridCells	Un arreglo de celdas en una cuadrícula 2D	std_msgs/Header header float32 cell_width float32 cell_height geometry_msgs/Point[] cells
GetMap	Servicio que obtiene un mapa como OccupancyGrid	nav_msgs/OccupancyGrid map
GetMap	Acción que obtiene un mapa como OccupancyGrid	nav_msgs/OccupancyGrid map
MapMetaData	Contiene información acerca de las características de OccupancyGrid	time map_load_time float32 resolution uint32 width uint32 height geometry_msgs/Pose origin
GetPlan	Obtiene un plano desde la posición actual hasta la pose meta	geometry_msgs/PoseStamped start geometry_msgs/PoseStamped goal float32 tolerance nav_msgs/Path plan
OccupancyGrid	Representa un mapa de grilla 2D, en donde cada celda representa la probabilidad de ocupación	std_msgs/Header header nav_msgs/MapMetaData info int8[] data
Odometry	Representa un estimado de la posición y velocidad en el espacio	std_msgs/Header header string child_frame_id geometry_msgs/PoseWithCovariance pose geometry_msgs/TwistWithCovariance twist
Path	Un arreglo de poses que representan un camino para que el robot siga	std_msgs/Header header geometry_msgs/PoseStamped[] poses

sensor_msgs: Mensajes destinados al uso de sensores.

Mensaje	Utilización	Definición del mensaje

CameraInfo	Contiene información para una cámara	std_msgs/Header header uint32 height uint32 width string distortion_model float64[] D float64[9] K float64[9] R float64[12] P uint32 binning_x uint32 binning_y sensor_msgs/RegionOfInterest roi
ChannelFloat32	Mensaje usado por PointCloud para contener datos opcionales asociados a cada punto en la nube.	string name float32[] values
CompressedImage	Contiene una imagen comprimida	std_msgs/Header header string format uint8[] data
FluidPressure	Lectura única de presión, para ser usado dentro de un fluido (aire, agua, etc.).	std_msgs/Header header float64 fluid_pressure float64 variance
Illuminance	Medimiento único de illuminancia fotométrica.	std_msgs/Header header float64 illuminance float64 variance
Image	Contiene una imagen no comprimida	std_msgs/Header header uint32 height uint32 width string encoding uint8 is_bigendian uint32 step uint8[] data
Imu	Contiene datos de IMU (Innertial Measurement Unit - Unidad de medida inercial)	std_msgs/Header header geometry_msgs/Quaternion orientation float64[9] orientation_covariance geometry_msgs/Vector3 angular_velocity float64[9] angular_velocity_covariance geometry_msgs/Vector3 linear_acceleration float64[9] linear_acceleration_covariance

JointState	Contiene datos para describir el estado de un conjunto de articulaciones de torque.	std_msgs/Header header string[] name float64[] position float64[] velocity float64[] effort
Joy	Reporta el estado de joysticks y botones	std_msgs/Header header float32[] axes int32[] buttons
JoyFeedback	Para entregar feedback a un joystick	uint8 TYPE_LED=0 uint8 TYPE_RUMBLE=1 uint8 TYPE_BUZZER=2 uint8 type uint8 id float32 intensity
JoyFeedbackArray	Un arreglo para publicar múltiples feedbacks de una sola vez	sensor_msgs/JoyFeedback[] array
LaserEcho	Es un submensaje de MultiEchoLaserScan (no se usa por separado)	float32[] echoes
LaserScan	Scaneo único de un encontrador de rangos laser planar	std_msgs/Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities
MagneticField	Medición de un campo magnético en un lugar específico	std_msgs/Header header geometry_msgs/Vector3 magnetic_field float64[9] magnetic_field_covariance
MultiDOFJointState	Representación de un conjunto de articulación con múltiples grados de libertad	std_msgs/Header header string[] joint_names geometry_msgs/Transform[] transforms geometry_msgs/Twist[] twist geometry_msgs/Wrench[] wrench

MultiEchoLaserScan	Exploración individual de un telémetro láser plano multi-eco	std_msgs/Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max sensor_msgs/LaserEcho[] ranges sensor_msgs/LaserEcho[] intensities
NatSatFix	Solución de navegación por satélite para cualquier Sistema Global de Navegación por Satélite	uint8 COVARIANCE_TYPE_UNKNOWN=0 uint8 COVARIANCE_TYPE_APPROXIMATED=1 uint8 COVARIANCE_TYPE_DIAGONAL_KNOWN=2 uint8 COVARIANCE_TYPE_KNOWN=3 std_msgs/Header header sensor_msgs/NavSatStatus status float64 latitude float64 longitude float64 altitude float64[9] position_covariance uint8 position_covariance_type
NavSatStatus	El estado para NavSat	int8 STATUS_NO_FIX=-1 int8 STATUS_FIX=0 int8 STATUS_SBAS_FIX=1 int8 STATUS_GBAS_FIX=2 uint16 SERVICE_GPS=1 uint16 SERVICE_GLONASS=2 uint16 SERVICE_COMPASS=4 uint16 SERVICE_GALILEO=8 int8 status uint16 service
PointCloud	Colección de puntos 3D, además de información opcional adicional de cada punto.	std_msgs/Header header geometry_msgs/Point32[] points sensor_msgs/ChannelFloat32[] channels

PointCloud2	Colleción de puntos N-dimensionales, que puede contener información tal como normales, intensidad, etc.	std_msgs/Header header uint32 height uint32 width sensor_msgs/PointField[] fields bool is_bigendian uint32 point_step uint32 row_step uint8[] data bool is_dense
PointField	Descripción de un punto en pointCloud2	uint8 INT8=1 uint8 UINT8=2 uint8 INT16=3 uint8 UINT16=4 uint8 INT32=5 uint8 UINT32=6 uint8 FLOAT32=7 uint8 FLOAT64=8 string name uint32 offset uint8 datatype uint32 count
Range	Lectura de la distancia individual de un ranger activo que emite energía e informa de una lectura de la distancia que es válido lo largo de un arco en la distancia medida.	uint8 ULTRASOUND=0 uint8 INFRARED=1 std_msgs/Header header uint8 radiation_type float32 field_of_view float32 min_range float32 max_range float32 range
RegionOfInterest	Especifica una región de interés dentro de una imagen	uint32 x_offset uint32 y_offset uint32 height uint32 width bool do_rectify
RelativeHumidity	Lectura de un medidor de humedad relativo	std_msgs/Header header float64 relative_humidity float64 variance
Temperature	Lectura individual de temperatura	std_msgs/Header header float64 temperature float64 variance
TimeReference	Medición de fuente temporal externa no sincronizada con el reloj del sistema	std_msgs/Header header time time_ref string source

SetCameraInfo	Servicio que le solicita a una camara guardar información como información de calibración	sensor_msgs/CameraInfo camera_info bool success string status_message