CAPÍTULO 5

MANUAL DE USUARIO

5.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

5.1.1. Identificación del proyecto

Título:	"Integración del dispositivo 'Microsoft Kinect' como parte de una arquitectura robótica genérica"
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

5.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo "Microsoft Kinect" comercializado por la empresa "Microsoft Corporation" La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

5.1.3. Visión general del documento

En el presente documento se ofrece toda la información necesaria para poner en funcionamiento el software desarrollado y comenzar su utilización *por parte del usuario final*. En él se realizará un repaso por las funciones de las dos piezas software que componen el proyecto y la forma de utilizarlas para obtener los resultados deseados, se incluirán también comentarios de las *acciones que son realizadas de forma pasiva* o automatizada para comodidad del usuario.

5.2. Manual de usuario de la interfaz

A continuación se explica cómo manejar las funciones de la interfaz que se ha desarrollado, mediante la utilización de comandos, ficheros de lanzamiento (*launch files*) y canales de comunicación de ROS. Se incluyen la lista de requisitos, la instalación del software necesario y el manejo de las funciones más relevantes para el correcto funcionamiento del proyecto.

Para facilitar la lectura, los comandos a utilizar se encuentran recuadrados siguiendo a la explicación se su funcionamiento en cada caso. Sin embargo parece ser que se modifican caracteres al "seleccionar y copiar" automáticamente el contenido debido a la naturaleza del fichero PDF y será necesario, en cada caso, escribir los comandos manualmente.

5.2.1. Requisitos necesarios

- Requisitos hardware (equipo de las pruebas)
  - Procesador Intel Atom -Silverthorne- (1.3 GHz) o superior
  - 1 GiB de Memoria RAM
  - 10 GiB de espacio en el disco duro
  - Teclado y ratón para configurar y controlar el entorno
  - Dispositivo de destino (por ejemplo Microsoft Kinect)
  - Puerto de conexión USB 2.0, WiFi b/g, Ethernet 10/100, etc. Según dispositivo de destino
- Requisitos software
  - Ubuntu Linux 10.04 (Lucid Lynx)
  - Robot Operating System (ROS) versión *Fuerte*
  - Paquetes de Linux:
    - python2.6
    - python2.6-dev
    - python-yaml
  - Software adicional
    - Paquetes ROS de los controladores de posibles dispositivos a manejar. En el caso del Kinect:
      - ros-fuerte-openni-kinect
      - ros-fuerte-kinect-aux

5.2.2. Instalación del entorno

Antes de poder utilizar la interfaz es necesario instalar el entorno ROS sobre el que trabaja, así como el conjunto de ficheros de la propia interfaz y algunos paquetes adicionales utilizados por el código de la misma. No será necesario repetir los pasos que se hayan realizado en una posible instalación anterior de la parte de odometría.

La instalación recomendada se realiza, acorde a los requisitos mencionados, sobre Ubuntu Linux y con una versión de ROS que puede ser "electric" o una posterior. En este caso se utiliza Ubuntu Linux 10.04 junto con ROS fuerte y la versión 2.6 de python por ser versiones con las que ya se ha probado el software.

1. El primer paso será la instalación del propio entorno ROS siguiendo las instrucciones de la página oficial[1], pero que se enumeran a continuación como parte imprescindible del presente documento.

   a) Si el sistema operativo no está configurado para permitir repositorios "restricted", "universe" y "multiverse", estos pueden activarse desde la ventana de "Software Resources" o "Recursos Software", en el menú de "Sistema / Administración" que se muestra en la Figura 5.1.
   La explicación original y actualizada, en caso de que hubiera cambios en las futuras versiones, puede encontrarse en la siguiente dirección:
   " *https://help.ubuntu.com/community/Repositories/Ubuntu* "

   b) Una vez activados los repositorios, ya se pueden incluir sus direcciones en el fichero *ros-latest.list* dentro de "/etc/apt/sources.list.d". Para las versiones utilizadas en este manual, se puede ejecutar directamente el siguiente comando.

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu lucid main" > /etc/apt/sources.
       list.d/ros-latest.list'
   ```

   c) También es necesario configurar las claves para el acceso al repositorio de la siguiente manera:

   ```
   wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
   ```

   d) Con los repositorios correctamente preparados para la instalación se utiliza la herramienta "apt-get" según los pasos habituales con el entorno ROS como objetivo.

   ```
   sudo apt-get update
   sudo apt-get install ros-fuerte-desktop
   ```

   Sólo para la parte de odometría: la versión "-full" será utilizada debido a que se requieren los paquetes de navegación y de percepción que no se incluyen en la versión "desktop" básica.

   ```
   # Si se actualizo recientemente con "sudo apt-get update"
   sudo apt-get install ros-fuerte-desktop-full
   ```

   e) Recién acabada la instalación, es recomendable incluir el fichero "setup.bash" de ROS en el fichero ".bashrc" para que el primero sea ejecutado al inicio definiendo así las variables de entorno necesarias para el correcto funcionamiento de ROS.

   ```
   echo "source /opt/ros/fuerte/setup.bash" >> \~/.bashrc
   . \~/.bashrc
   ```

[1] Instrucciones de instalación de ROS en su web: " *www.ros.org/wiki/fuerte/Installation/Ubuntu* "

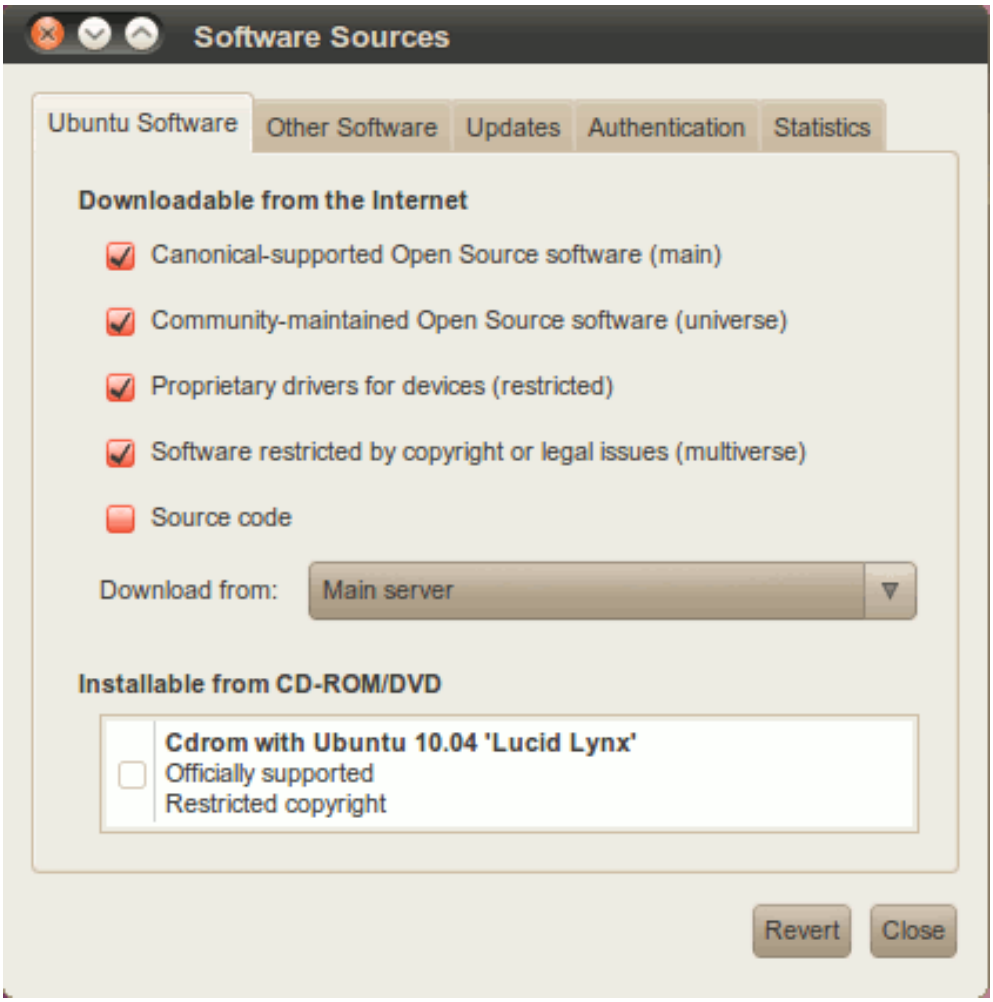

Figura 5.1: Ubuntu - Software Sources Panel

La línea equivalente, si no se desea que los cambios sean permanentes, es « `source /opt/ros/fuerte/setup.bash` ». Las variables de entorno desaparezcan al cargar de nuevo el entorno Linux.

Hecho esto, el entorno ROS, que es la base sobre la que se trabaja, deberá haber quedado completamente instalado y funcional.

2. Será necesario instalar Python 2.6 tanto en su versión de intérprete como de desarrollo y la librería YAML para las configuraciones. Para ello se utiliza la herramienta apt-get:

```
# Si se actualizo recientemente con "sudo apt-get update" basta escribir:
sudo apt-get install python2.6 python2.6-dev python-yaml
```

3. El driver que se quiera controlar desde la interfaz debe ser instalado individualmente según las necesidades del usuario. Para el caso particular del Kinect, la web de ROS recomienda el paquete *openni_kinect* asociado a la versión de ROS utilizada y el paquete *camera_drivers*. Eso se traduce en lo siguiente al utilizar apt-get para la instalación:

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install ros-fuerte-openni-kinect ros-fuerte-camera-drivers
```

Adicionalmente, se puede instalar también *kinect_aux* para mayor control sobre el dispositivo Kinect y aprovechando la capacidad de la interfaz para actuar como acceso común a toda la configuración de ambos drivers. En este caso es necesario utilizar *git* y compilar el paquete en la máquina local al no encontrarse disponible mediante *apt-get*.

a) Es necesario instalar primero el propio cliente git (que sí está disponible mediante apt-get).

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install git
```

b) Después se accede al directorio de ROS que contiene los paquetes de tipo *stack* (conjuntos o colecciones de paquetes relacionados) dado que *kinect_aux* es de dicho tipo. El directorio actual de stacks se encuentra en */opt/ros/fuerte/stacks* y la instalación se realiza de la siguiente manera:

```
cd /opt/ros/fuerte/stacks
sudo git clone https://github.com/ros-pkg-git/kinect.git
sudo chmod 777 -R kinect_aux # Para que rosmake pueda funcionar sin privilegios
cd kinect_aux
rosmake
```

4. Antes de llegar al último paso es necesario instalar el cliente git que está disponible mediante apt-get y permite descargar y actualizar de forma fácil el código fuente de los paquetes.

```
# Si se actualizo recientemente con "sudo apt-get update"
sudo apt-get install git
```

5. Tal como cabe esperar: el último paso, una vez instaladas todas las dependencias, es instalar los ficheros de la interfaz, en este caso desde el repositorio en el que se encuentran alojados, dentro de *GitHub.com* [2]. A efectos de mostrar un ejemplo más completo, se instalarán los archivos en una carpeta nueva de proyectos en vez de utilizar las carpetas ROS existentes.

```
cd /home/usr_name # Nombre de usuario "usr_name" como ejemplo
sudo git clone https://github.com/michi05/my_adaptor
cd my_adaptor
rosmake
```

5.2.3. Configuración de la interfaz: fichero de lanzamiento

Durante el desarrollo, se tató de aprovechar el sistema de ficheros de lanzamiento de ROS, que favorece el uso de una jerarquía de responsabilidades en los mismos de tal manera que cada fichero invoca a otros hasta que todo el entorno es arrancado. A continuación se detallan los distintos lanzadores disponibles para poder configurar el comportamiento de la interfaz y sus opciones de configuración correspondientes.

robotDrivers.- Define y lanza un único robot al completo pudiendo incluir múltiples dispositivos confinados en un

[2] La dirección del repositorio es " `https://github.com/michi05/my_adaptor` "

espacio de nombres común para el robot. Hay dos argumentos que se pueden asignar a este nivel y sus valores por defecto tras el signo “igual”:

- **robot_ns** = robot1
  Define el espacio de nombres relativo en el cual incluir todos los componentes del robot.
- **absolute_namespace** = /
  Contiene el espacio de nombres absoluto bajo el cual se está ejecutando *de facto* el fichero de lanzamiento actual. Va contra el carácter genérico del proyecto pero es imprescindible para que el driver openni de kinect funcione correctamente.

kinectDrivers.- Es un lanzador a nivel de dispositivo exclusivamente para el Kinect. Se ocupa de lanzar tanto la interfaz (objeto de este manual) como los drivers, para no introducir más complejidad con un nivel adicional innecesario. Este fichero requiere los mismos argumentos que *robotDrivers.launch* y, adicionalmente, los siguientes:

- **camera_name** = kinect1
  El nombre de la cámara para determinar el espacio de nombres.
- **main_file_basename** = interface_node_main
  Contiene el nombre del archivo principal.
- **node_name** = interface_node_main
  El nombre del nodo principal que es por defecto el del archivo principal.
- **pkg_name** = my_adaptor
  Contiene el nombre del nodo principal.
- **pkg_location** = $(find my_adaptor)
  Contiene la localización del paquete especificado bajo *pkg_name*.
- **translation_file** = parameterDictionary.yaml
  Determina el nombre del diccionario de parámetros [siguiente subsección].

Para utilizar dichos ficheros se utiliza el comando « `roslaunch` » y sus parámetros pueden cambiarse con antelación modificando el código de cada fichero o pueden incluirse a continuación usando el operador “:=”.

```
roslaunch paqueteROS ficheroLanzador.launch argumento1:=valor1 argumento2:=valor2
```

Por ejemplo:

```
roslaunch my_adaptor kinectDrivers.launch camera_name:=kinect1
```

El hecho de que el primer lanzador incluya al segundo pero sea éste el que tiene más argumentos se debe a que los argumentos son privados y deben ser declarados en cada fichero “.launch”. Esto significa que si en *robotDrivers* se declaran todos los argumentos de los lanzadores que incluye, el listado podría ser demasiado largo. Por defecto cada cuál tiene sus propios argumentos y será un usuario avanzado el que decida si desea cambiar la estructura de lanzamiento.

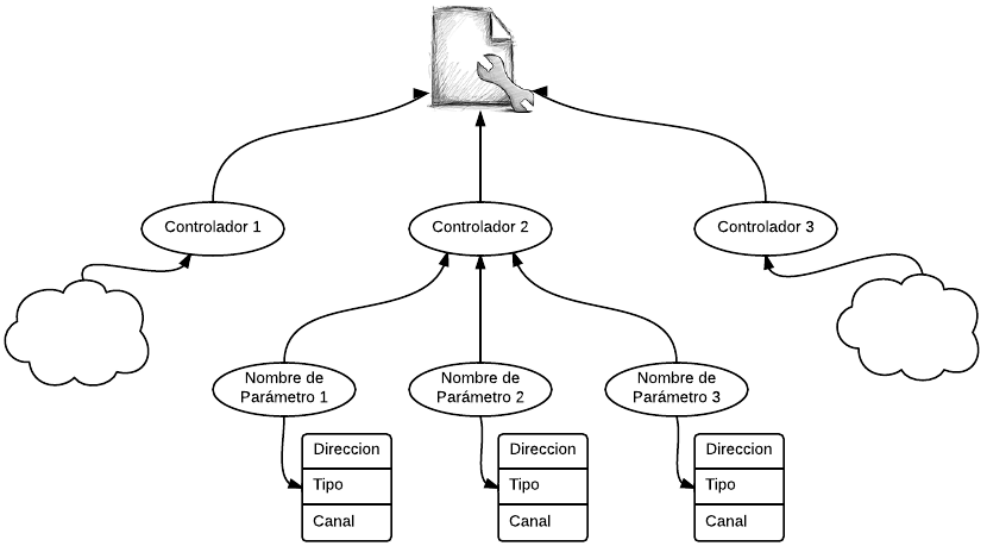

Figura 5.2: Diagrama en árbol de la estructura del diccionario

5.2.4. Configuración de la interfaz: diccionario de parámetros

Quizá la aportación más importante al diseño sea la posibilidad de configurar en un mismo fichero todos los parámetros manejados por el driver (o varios drivers) para que formen parte de la misma lista de *parámetros dinámicos* y, sobre todo, puedan ser configurados de manera análoga y con identificadores (nombres) al gusto del usuario. La estructura del fichero, basada en lenguaje YAML, es la que se describió en el apartado 4.5 del documento *Diseño* , que se correspondía con la figura 5.2.

```
# Los comentarios usan almohadilla segun el convenio del lenguaje YAML
'ruta/driver': # La ruta puede ser vacia.
               #Si un driver usa varias se agregan como si fueran drivers distintos
  nombre_parametro: # El nombre nuevo deseado para este parametro
  - alias_parametro # El nombre o identificador del parametro segun el controlador
  - tipo_dato # El tipo de dato basico: entero (int), texto (string)...
  - clase_parametro # La clase de parametro segun el canal de comunicacion con el driver
                    #Por ahora puede ser: "topic" o "parametro_dinamico" (dynParam)
```

Y este es un ejemplo real utilizado durante el desarrollo:

```
'camera/driver/':
  image_mode:
  - camera/driver/image_mode
  - int
  - dynParam
```

El fichero utilizado por defecto se llama “parameterDictionary.yaml” aunque puede ser sustituido o modificarse fácilmente con los parámetros del apartado anterior.

Fichero de configuración de "dynamic_reconfigure"

Hasta la versión actual de ROS, y desde la reciente introducción del servidor dinámico de parámetros ("dynamic_reconfigure"), es necesario escribir un fichero de extensión ".cfg" por cada nodo que utiliza dicho servidor para conocer los parámetros de los que debe ocuparse. Si bien es probable que este sistema cambie a corto plazo, el usuario se ve obligado a repetir la configuración del fichero de traducción en dicho fichero ".cfg".

La configuración de dicho fichero consiste en la introducción de una línea de la forma « `gen.add("nombre_parametro", tipo_dato, 0, "Descripcion␣si␣se␣desea", valor_por_defecto, valor_min, valor_max)` » con los campos indicados (nombre, tipo de dato, un cero[3], descripción, valor por defecto, valor mínimo y valor máximo) por cada parámetro a controlar. Los valores mínimo y máximo sólo se utilizan para los tipos enteros y reales, y los tipos posibles son: "int_t", "double_t", "str_t" y "bool_t". Si se desea una explicación más amplia y detallada, se puede consultar el tutorial de "Cómo configurar tu primer fichero CFG" en la web de ROS[4].

En el proyecto actual existe ya un fichero "Properties.cfg" (dentro de la carpeta "cfg") que responde al fichero de traducciones por defecto. Para que el entorno reconozca cambios en el fichero habrá que ejecutar lo siguiente:

```
# Colocarse en la carpeta del proyecto
roscd my_adaptor
# Marcar el archivo como ejecutable
#(en caso de que haya borrado el original)
chmod a+x cfg/Properties.cfg
# Compilar el proyecto con la nueva configuracion
rosmake
```

Para futuros desarrollos, si el diseño actual ofrece buenos resultados, se planea programar un generador que produzca ambos ficheros de forma paralela para facilitar la labor del usuario en caso de que el fichero de configuración de su dispositivo no exista ya de antemano.

5.2.5. Arranque inicio y cierre de la interfaz

Una vez completada la instalación del software y las dependencias necesarias, será posible arrancar la interfaz mediante un simple comando *roslaunch paquete launchfile* en la consola de Linux y cerrarlo, en condiciones normales, enviando una señal al mismo mediante la combinación de teclas *Ctrl+C*. Esto se debe al diseño de la interfaz, orientado a facilitar su arranque rápido y configuración automática una vez que los parámetros han sido plasmados en los ficheros de configuración correspondientes.

El comando básico para el arranque es « `roslaunch my_adaptor robotDrivers.launch` », si bien puede sustituirse el fichero de lanzamiento (*robotDrivers.launch*) o realizar el lanzamiento manualmente ejecutando primero los drivers y luego la interfaz tras introducir los parámetros necesarios en el servidor de parámetros:

[3] Aclaración: el cero (0) del tercer campo sirve para distinguir grupos de parámetros. No tiene sentido en el manual de usuario pues los parámetros no recibirán un tratamiento especial sin modificar el código fuente del nodo.

[4] Se puede acceder al tutorial "Cómo configurar tu primer fichero CFG" a través del a siguiente dirección: " *www.ros.org/wiki/dynamic_reconfigure/Tutorials/HowToWriteYourFirstCfgFile* "

```
roscore & # Debe existir una instancia de roscore en todo momento
# Lanzar los drivers deseados y configurados en el espacio de nombres adecuado
roslaunch openni_launch openni.launch # Driver Openni del Kinect
# Guardar los parametros de lanzamiento en el servidor de parametros
#un posible ejemplo seria...
rosparam set property_config_file parameterDictionary.yaml
# Ejecutar el nodo de la interfaz
rosrun my_adaptor interface_node_main.py
```

5.2.6. Consultar el valor de un parámetro

Una vez está en marcha la interfaz, ésta hará de intermediaria entre aplicaciones de la capa superior y los controladores, de forma que no se requiera ninguna acción adicional por parte del usuario. Aún así es posible consultar y modificar la configuración del driver con ayuda de la interfaz. La consulta se puede llevar a cabo de dos maneras distintas: mediante el servicio adecuado en la línea de comando, o con una pequeña aplicación incluida de antemano por ROS.

En el primer caso bastaría con llamar al servicio “get/<tipo>Property” cambiando “tipo” por el tipo de dato del parámetro entre: Int, String, Bool o Float. La llamada completa a un parámetro de tipo entero quedaría como en el siguiente ejemplo:

```
rosservice call /kinect1/get/IntProperty depth_resolution
```

En este ejemplo se consulta “depth_resolution y se obtendría una salida similar a « `paramValue:2` ».

Figura 5.3: Pantalla inicial del configurador de parámetros dinámicos

La ejecución de « `rosrun dynamic_reconfigure reconfigure_gui` » realizaría la ismsma acción, de tal forma que emergerá una ventana de interfaz tan simple como la de la figura 5.3. En ella se debe elegir un nodo del desplegable para llegar a la siguiente pantalla (figura 5.4). En este caso será el que termine con el nombre del nodo: *img_iface* aunque puede personalizarse la ruta desde los ficheros

de lanzamiento. El usuario puede consultar desde ahí la lista completa de parámetros incluyendo el del ejemplo y los valores actuales aparecen en la columna de la derecha.

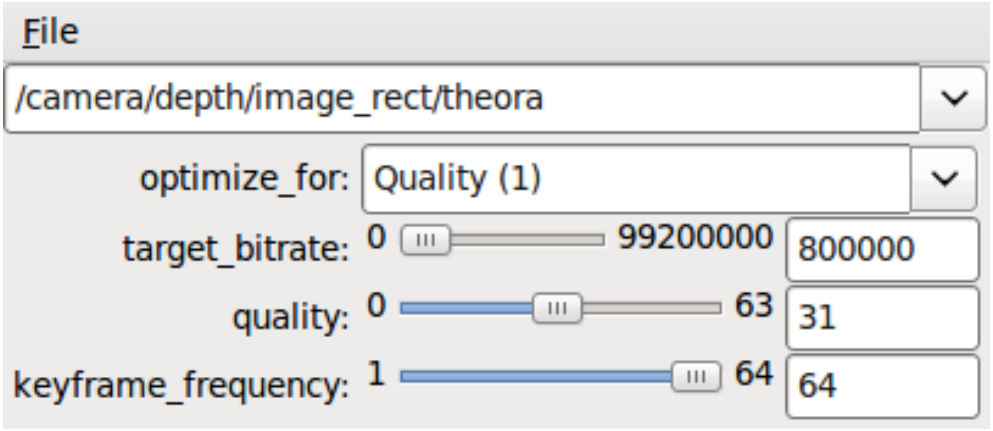

Figura 5.4: Lista de parámetros dynámicos configurables

5.2.7. Modificar parámetros del driver en tiempo de ejecución

También se ofrece la opción de realizar ajustes manuales mediante el servidor dinámico de parámetros (*Dynamic Parameter Server*) ya sea creando un nodo ROS personalizado que actúe como cliente de *dynamic_reconfigure* o mediante la aplicación *reconfigure GUI* del propio entorno ROS. Sólo se explicará aquí la segunda, pues la primera está orientada a desarrolladores.

Para lanzar la herramienta de reconfiguración se ejecuta, con la interfaz ya en marcha: « `rosrun dynamic_reconfigure reconfigure_gui` ». Inmediatamente, aparece la ventana de la figura 5.3 y, como ya se dijo en el apartado anterior, se elige el nodo *img_iface* en el desplegable.

Una vez dentro, en la siguiente pantalla, la propia interfaz indica los valores mínimos y máximos (si los hay) y permite la modificación de los parámetros ya sea mediante barras deslizantes, marcas on/off o cajas de texto. Al ser parámetros dinámicos, los cambios que se realicen serán aplicados de forma instantánea a la interfaz y, desde ahí, a los controladores.

Como alternativa se puede escribir « `rosservice call /kinect1/set/<tipo>Property nombre_parametro` » adaptado al parámetro que se desee sustituyendo “tipo” por el tipo de dato (Bool, Int, Float, String) y siguiendo con el nombre y el nuevo valor del parámetro respectivamente.

```
rosservice call /kinect1/set/IntProperty depth_resolution 1
```

5.2.8. Consultar la dirección de un topic

Puede ser deseable, en un momento determinado, averiguar la dirección de un recurso publicado en forma de “topic”, de tal manera que la aplicación que lo solicita pueda conectarse directamente a él sin depender de la interfaz. Existe, en este caso, sólo una manera de realizar esa consulta y es mediante el

servicio apropiado: “get/TopicLocation”. Éste responderá al alias de un parámetro o recurso como único argumento, es decir, al nombre que se le asignó en el diccionario (hay que tener en cuenta que las aplicaciones nunca tienen que conocer el identificador a nivel del controlador). Un ejemplo sería la consulta siguiente:

```
rosservice call /kinect1/get/TopicLocation depth_reg_raw
```

Dado que “depth_reg_raw” es un topic, deberá tener asociada una dirección relativa donde se publica dentro del entorno ROS. En este caso se obtiene la respuesta “paramValue: depth_registered/image_rect_raw”, que significa que el recurso puede ser encontrado en la dirección absoluta “/kinect1/depth_registered/image_rect_raw”.

5.2.9. Renombrar un topic

Es posible renombrar (o reubicar) los topics publicados por el controlador, pero se debe ser consciente de que, para cada topic, se creará un nodo ROS que “repetirá” uno por uno cada mensaje del topic, con las desventajas de rendimiento que ello conlleva. Para conseguir este resultado existen 2 métodos según las necesidades del usuario:

Al inicio, gracias al diccionario de parámetros. Incluyendo un parámetro adicional en el fichero “parameterDictionary.yaml” presentado en 5.2.4, la interfaz renombrará el topic del parámetro correspondiente durante el arranque. El código tiene que ser como el siguiente:

```
'ruta/driver':
  nombre_parametro: # El nombre asignado o alias (no en el driver)
  - ruta_nueva # La direccion de topic para ser asignada
  - tipo_dato # El tipo del topic se mantiene solo por compatibilidad
  - renaming # La clase determina la accion a realizar
```

Al escribir “renaming” como clase del parámetro, la interfaz entiende que se trata de un topic y que debe reubicarlo en la ruta nueva especificada. Tras reubicar el topic se almacena la ruta nueva para el parámetro“nombre_parametro” y se continua con normalidad.

Mediante el servicio apropiado, enviando una petición. Para algunos casos puede ser interesante cambiar la ruta de un topic en tiempo de ejecución. Para ello se provée un servicio “*set/TopicLocation*” que recibe el identificador del topic y la ruta deseada, y modifica su ruta (ya conocida) dentro del entorno ROS.

Un ejemplo de llamada a dicho servicio sería: « `rosservice call /ruta_adaptador/set/TopicLocation nombre_parametro ruta_nueva` », que cambiaría la dirección del topic asociado a “nombre_parametro” por “ruta_nueva”, que a su vez puede ser una ruta absoluta o relativa, en cuyo caso se ubicará usando el driver como raíz.

CAPÍTULO 6

MANUAL TÉCNICO

6.1. Introducción

Se dedica este primer apartado de cada capítulo a presentar la materia sobre la que se extenderá el mismo. Se expone primeramente una visión general del proyecto con sus datos identificativos y un breve resumen de su naturaleza con el objetivo de que el lector pueda situarse con total facilidad sin necesidad de haber leído la *Memoria* en la que se amplía esa misma información. Se ubican al final de la documentación las herramientas de consulta como el glosario, anexos y bibliografía, que si bien pueden ser de utilidad para comprender cualquiera de los capítulos no responden a ninguno en particular.

6.1.1. Identificación del proyecto

Título:	"Integración del dispositivo 'Microsoft Kinect' como parte de una arquitectura robótica genérica"
Directores:	Luciano Sánchez Ramos Enrique H. Ruspini
Autor:	Miguel A. García Glez
Fecha:	noviembre de 2012

6.1.2. Visión general del proyecto

El proyecto que se presenta está compuesto de dos partes complementarias pero diferenciadas. Por una parte el diseño e implementación de un software intermedio para comunicar de forma genérica distintas aplicaciones robóticas y dispositivos de entrada a través de un entorno genérico llamado ROS (Robot Operating System). Si bien dichos periféricos deben ser intercambiables, el desarrollo actual está orientado al uso de la cámara de profundidad incluida en el dispositivo "Microsoft Kinect" comercializado por la empresa "Microsoft Corporation" La otra etapa del proyecto consiste en la creación de un software para dicho sistema ROS que, haciendo uso de la interfaz ya diseñada, obtenga información 3D del entorno y haga los cálculos correspondientes a la estimación de movimiento (odometría) del robot que lleva instalado el sistema.

6.1.3. Visión general del documento

El documento de manuales técnicos ha sido concebido con el objetivo principal de aportar toda la información necesaria para guiar a futuros desarrolladores que deseen reutilizar, corregir o adaptar el código de la manera que le sea necesario. Se trata de un aspecto importante dado que se solicitó expresamente al inicio del proyecto que el código estuviera orientado un mantenimiento sencillo mediante el uso de clases con responsabilidades claras y separadas, además de las indispensables aclaraciones mediante comentarios en el código fuente.

Este documento puede verse, de alguna manera, complementado con el documento 4 sobre el *Diseño e Implementación* donde se introduce la estructura de la solución tanto a nivel de clases como de ficheros y de comunicaciones con el entorno. Por ese motivo se repetirán aquí algunos de los conceptos y diagramas más importantes con el objetivo de que el presente texto sea autosuficiente para su objetivo, pero sin perjuicio de que una lectura del primero podría aportar una mejor comprensión del conjunto.

Se comenzará por enumerar la relación de recursos necesarios para la modificación del código fuente y su puesta en marcha en forma de producto ejecutable, siguiendo con su estructura de clases y una explicación más en profundidad de cada clase individual. Finalmente, se hablará de los ficheros en que se divide el código para terminar detallando la capacidad de configuración implementada para los nodos.

6.2. Recursos necesarios

Para el mantenimiento del código no se necesitan grandes medios pero sí un PC con el software adecuado y unos determinados conocimientos de programación. A continuación se expone una relación de los recursos que deberán ser suficientes para dichas modificaciones.

Recursos hardware

Un PC de características básicas es suficiente para realizar cualquier modificación al código de la interfaz y generar la solución correspondiente. Se obviarán en este apartado los dispositivos físicos requeridos para las pruebas y para el uso del software, pues no son requisitos y dependen plenamente de necesidades puntuales. El equipo que se toma como referencia es un portátil "*Asus eeePC*", el más básico de los disponibles en el laboratorio.

- Procesador Intel Atom -Silverthorne- (1.3 GHz) o superior
- 1 GiB de Memoria RAM
- 10 GiB de espacio en el disco duro
- Teclado y ratón

Recursos software

Toma mayor importancia en este caso el entorno software que el hardware en el que se instala. Se necesitará, por supuesto, el entorno de destino (un sistema ROS dentro de Ubuntu Linux), así como el compilador del lenguaje (Python, en este caso) y una aplicación que permita modificar ficheros de texto plano. Una vez más se excluyen del listado los paquetes relacionados con el dispositivo final al no ser requisitos para la modificación del código en sí.

- Ubuntu Linux 10.04 (Lucid Lynx) o superior
- Robot Operating System (ROS) versión *Fuerte*
- Paquetes de Linux:
  - python2.6
  - python2.6-dev
  - **Sólo para la interfaz** Librería de YAML para Python: "python-yaml"
  - **Sólo para la odometría** Paquete ROS de la librería de nubes de puntos (*PCL*): "ros-fuerte-perception-pcl"[1]

[1] Referencia de perception_pcl: " `http://www.ros.org/wiki/pcl` "

  - **Sólo para pruebas con MS Kinect** Paquete ROS de los controladores *OpenNI* : "ros-fuerte-openni-kinect"[2]

- Y un editor de texto para el código como pueden ser por ejemplo:

  - Editor de texto básico como *gedit text editor*.
    Instalado por defecto con Ubuntu.
  - Entorno de desarrollo integrado como *Eclipse IDE*.
    Descargable desde su web oficial " `www.eclipse.org` "

Todos los paquetes mencionados tanto de Ubuntu como de ROS están disponibles en los repositorios de *Ubuntu* y el propio sistema operativo puede ser tanto descargado, como solicitado por correo postal desde la página web " `http://www.ubuntu.com/` ".

Recursos humanos

También son importantes los conocimientos necesarios para el mantenimiento y modificación del código. Siempre es conveniente tener conocimientos amplios y abundantes de todas las herramientas que intervienen, pero se enumeran a continuación los campos importantes que podrían ser estudiados para adquirir una buena comprensión del código y del entorno previamente a realizar modificaciones en el primero.

- **Robot Operating System** Es necesario poseer conocimientos sobre el funcionamiento de ROS y algunas de sus herramientas para comprender y adaptar el comportamiento de la aplicación, del propio entorno y, sobre todo, de la comunicación entre ellos. Se trata de un sistema modular con una colección de herramientas de las cuales se destacan las más indispensables para el caso actual:

  - Ficheros de lanzamiento ROS - " `http://ros.org/wiki/roslaunch` "
  - Servidor de parámetros - " `www.ros.org/wiki/ParameterServer` "
  - Canales de comunicación: topics y servicios - " `www.ros.org/wiki/Topics` " y " `www.ros.org/wiki/Services` "
  - Y adicionalmente, herramientas de prueba y diagnóstico: roswtf, rosbag, rostest, etc.

- **Ficheros de lanzamiento ROS** Si bien este punto forma parte del anterior, al tratarse de una herramienta de ROS, se destaca específicamente por ser una herramienta opcional en el entorno e imprescindible en el presente proyecto. Especialmente en el caso de la interfaz, los ficheros de lanzamiento permiten secuenciar el lanzamiento de nodos para garantizar que los controladores están en marcha.

- **Git revision control** Es necesario manejar el sistema de control de versiones (o revisiones) utilizado (*Git*) para obtener el código inicial y, muy especialmente, si se pretende que el repositorio actual refleje los cambios futuros.

[2]Referencia de openni_kinect: " `http://www.ros.org/wiki/openni_kinect` "

Sólo para el paquete interfaz

- **Lenguaje Python** El código está escrito en lenguaje Python. No se utilizan operaciones ni estructuras de gran complejidad en el código de la interfaz, por lo que tener unas nociones básicas será suficiente para poder realizar ciertas modificaciones.
- **Lenguaje YAML** YAML es un lenguaje de marcado que es el utilizado tanto en el diccionario de parámetros (5.2.4), como en los ficheros generados por el servidor de parámetros de ROS.

Sólo para el paquete odometría

- **Lenguaje C++** Dado que en el paquete sobre odometría la eficiencia era uno de los aspectos más importantes, ésta fue implementada en lenguaje C++ y es, por tanto, necesario comprender dicho lenguaje para poder realizar futuras modificaciones del nodo. Es, además, conveniente que el desarrollador esté familiarizado con la programación orientada a objetos dado que el código trata de seguir dicho paradigma.
- **Librería de nubes de puntos (*PCL*)** Las estimaciones de odometría visual corren a cargo de la librería de nubes de puntos (conocida como *PCL*) que implementa el algoritmo un algoritmo *ICP* (por sus siglas en inglés *"Iterative Closest Point"* y que es explicado en el anexo) para la correlación de nubes de puntos y algunas herramientas auxiliares para su tratamiento. Toda la información sobre esta librería se encuentra en su página web " `http://pointclouds.org` " e incluye otras herramientas y algoritmos que podrían ser útiles para futuras ampliaciones del proyecto actual.

6.3. Instalación del entorno de programación

Una vez se han especificado los recursos software necesarios, el siguiente paso es instalarlos en un ordenador que se utilizará para las modificaciones. Dado que Python es un lenguaje semi interpretado y que, en todo caso, es necesario compilar el paquete una vez descargado, la instalación del *Manual de usuario de la interfaz* (5.2.2) es exactamente la misma que la de esta sección y el lector puede remitirse a dicho apartado para encontrarla.

6.4. Ficheros fuente

Es necesario especificar la forma en que se organiza el código fuente dentro del sistema de ficheros del ordenador, pues de ello puede depender dónde se encontrarán distintos elementos y aspectos del software o dónde deben introducirse otros nuevos. A continuación se analiza la estructura que forman los ficheros en su conjunto, lo que contienen y cómo interactúan entre ellos para formar el producto final.

El código fuente completo se encuentra almacenado y publicado en la web de "*GitHub*" que se encarga del sistema de control de versiones (siguiendo el sistema "Git" en particular) así como de permitir a otros usuarios obtener el código o tratar de contribuir si fuera necesario. Los espacios registrados a tal efecto se encuentran en " `https://github.com/Michi05/my_adaptor` " para el paquete de interfaz y " `https://github.com/Michi05/my_odometry` " para la odometría.

6.4.1. Estructura general

Dado que el proyecto está dividido en dos partes que representan dos paquetes software individuales y autónomos, será necesario estudiar cada uno por separado. Se utiliza, sin embargo, la estructura común que viene dada por el entorno ROS y que es explicada en detalle en su página de consulta " `www.ros.org/wiki/Packages` ". A continuación el listado de ficheros y directorios comunes:

- **bin/:** archivos ejecutables ya compilados.
- **cfg/:** directorio para la configuración del servidor dinámico de parámetros (sólo en la interfaz).
- **include/:** archivos de cabecera de C++ (.h, .hpp).
- **launch/:** *launch files* encargados de lanzar los nodos con sus correspondientes configuraciones y dependencias.
- **msg/:** Especificación de los tipos de datos asociados a los topics y servicios.
- **nodes/:** Archivos de código fuente específico para nodos. Se utilizará para los ficheros python de la interfaz.
- **scripts/:** Archivos de secuencias de comandos (no se utilizó).
- **src/:** Archivos de código fuente principal (.c, .cpp). Se eligió usar esto sólo en el segundo paquete (*my_odometry*).
- **srv/:** Tipos de datos específicos para los servicios (conjunto de tipos de mensajes agrupados en "solicitud" y "respuesta".
- **CMakeLists.txt:** Configuración para "CMake" con los directorios y ficheros a incluir en la compilación.
- **manifest.xml:** Manifiesto del paquete (información sobre su estructura y dependencias).

Los tipos de mensajes y servicios imprescindibles para determinar qué información se podrá transmitir y en qué manera u orden. Los tipos de mensajes determinan estructuras de datos complejas formadas a partir de tipos básicos y/o de otros mensajes; mientras que los servicios agrupan estructuras de datos construidas del mismo modo pero asociadas a dos partes separadas: solicitud y respuesta.

Durante la compilación también se generan algunos directorios y ficheros adicionales que varían con la configuración y las herramientas del entorno utilizadas, pero que no es necesario conocer ni mucho menos es aconsejable modificar porque se trata de ficheros para uso propio del sistema.

6.4.2. Ficheros de la interfaz

El código fuente, a nivel de ficheros, se basa en dos elementos principales: "interface_node_main.py" y "ctrl_interface.py" que se encuentran en el directorio *nodes* y representan, respectivamente: la función de partida principal del nodo (función *"main"*) y la definición de las distintas clases que intervienen durante la ejecución. Intervienen además 4 carpetas más con ficheros de importancia:

- **cfg/** contiene el fichero "properties.cfg" que determina los parámetros conocidos por el servidor dinámico de parámetros. El modo de configurarlo se explica en la sección de configuraciones (6.7) y se profundiza en la documentación oficial de ROS[3].
- **launch/** permite escribir ficheros de lanzamiento que realicen automáticamente una serie de tareas principalmente de configuración, ejecución del nodo principal y de las dependencias. Es una parte indispensable de la interfaz dado que tiene que ejecutar los controladores de los dispositivos antes de lanzarla, pero también se encarga de asignar un espacio de nombres común a los nodos dentro del entorno ROS.
  - **kinectDrivers.launch** es el fichero de lanzamiento básico para controlar el Kinect.
  - **adaptor_openni.launch** lanza sólo los drivers del Kinect. Es utilizado por los otros lanzadores.
  - **robotDrivers.launch** permite lanzar "kinectDrivers" dentro de un espacio de nombres y conlleva más configuración.
  - **runMultiplexers.launch** ayuda a lanzar los multiplexadores de topics (utilidad interna para el adaptador).
- **srv/** guarda en su interior las especificaciones de los servicios que vienen definidos por dos mensajes o conjuntos de mensajes: de solicitud y de respuesta.
  - **Solicitudes de lectura para tipos de datos básicos:** booleanValue.srv, intValue.srv, floatValue.srv, stringValue.srv
  - **Solicitudes para modificar tipos de datos básicos:** setBoolean.srv, setInteger.srv, setFloat.srv, setString.srv
  - **Obtención de la dirección (topic) asociada a un parámetro:** requestTopic.srv

[3]Enlace: "`http://www.ros.org/wiki/dynamic_reconfigure`"

6.4.3. Ficheros de la odometría

El paquete de odometría visual, por su parte, está dividido en tres partes: una función *main* principal y dos clases que se implementan según el convenio habitual de código y cabecera por separado (con extensiones .cpp y .hpp respectivamente). Sus correspondientes ficheros se denominan "odometry_main.cpp", "pcl_odometry.cpp" y "pcl_myTransf.cpp" junto con sus respectivas cabeceras con extensiones *.hpp* . También en este caso cabe mencionar algunas carpetas adicionales indispensables para las modificaciones:

- **launch/** permite escribir ficheros de lanzamiento que realicen automáticamente una serie de tareas principalmente de configuración, ejecución del nodo principal y de las dependencias. En el caso de la odometría se encarga de cargar la configuración guardada y se incluye un segundo lanzador que, sin ejecutar el nodo principal, publica una TF estándar prefijada para las simulaciones.
  - **my_odometry.launch** es una propuesta de lanzador básico que carga un fichero de configuración que puede ser distinta de la configuración por defecto del nodo y pone en marcha el nodo principal.
  - **tf_publisher.launch** lanza una herramienta ROS de publicación de TFs de referencia útiles para las simulaciones y visualizaciones.
- **msg/** es el directorio donde se almacenan las especificaciones de "mensajes" que, como ya se adelantó, son los tipos o estructuras de datos que se pueden transmitir a través de los canales ROS: topics y servicios.
  - odom_answer.msg
- **srv/** guarda en su interior las especificaciones de los servicios que vienen definidos por dos mensajes o conjuntos de mensajes: de solicitud y de respuesta.
  - **emptyRequest.srv** es necesario para los servicios que no requieren parámetros, con la particularidad de que incorpora en la respuesta los datos de estado del nodo, es decir, un mensaje de tipo *odom_answer* .
  - **odom_update_srv.srv** si bien se trata de un servicio idéntico a "emptyRequest" y podría prescindirse de él, se mantiene este servicio por separado

6.5. Estructura de clases de la interfaz

La interfaz se diseñó, en aras de favorecer la mantenibilidad, con un número de clases bajo como para favorecer la simplicidad pero suficiente para separar las responsabilidades, en este caso, organizadas en forma de capas de abstracción. Esto se traduce, concretamente, en tres clases complementarias encargadas de distintas partes de la comunicación: la más baja (*driver_manager*) es la que se comunica directamente con el driver del dispositivo, la capa de arriba, *upward_interface* es accesible por parte del usuario o por las aplicaciones que hacen uso de la interfaz, y ambas se comunican entre ellas si bien es la tercera clase *iface_translator* la que permite que se entiendan al traducir los "nombres" de los parámetros entre los conocidos por el driver del dispositivo y los especificados por el usuario en el fichero de configuración.

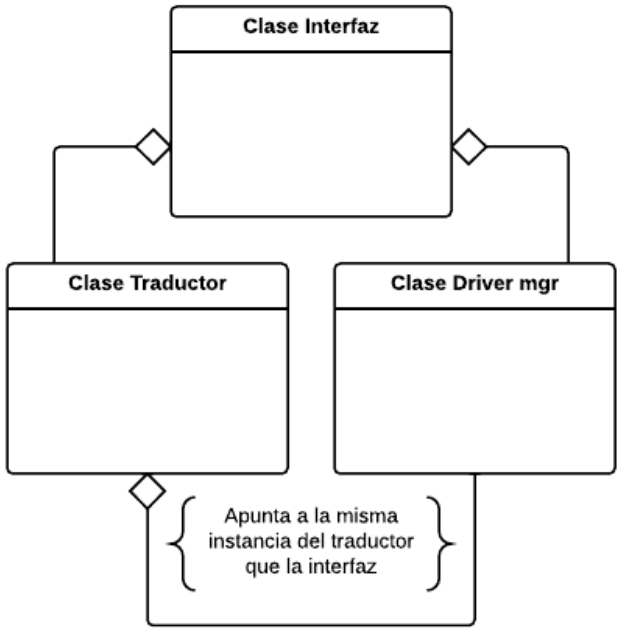

Figura 6.1: Diagrama de clases que incluye las de la interfaz

En la figura 6.1 se representa gráficamente la estructura que forman entre sí las clases del nodo y sus relaciones. En los apartados siguientes se profundiza sobre los datos miembro que incorpora cada una de ellas y se describe la utilidad de cada método que implementan usando el convenio de documentación de "Python Dosctring" de escribir "@param" para especificar los parámetros de entrada y "@return" antes de exponer el contenido de la variable de retorno.

La instanciación de estas clases y el modo de inicializarlas durante el arranque forman parte de la explicación sobre el *Código fuente del producto* dentro del documento de *Diseño* (4.6).

6.5.1. Clases del paquete interfaz

En los siguientes apartados, que corresponden a las tres clases que componen la interfaz, se dedicará en cada caso: un punto a especificar los datos miembro que almacena cada una de ellas y otro a las funciones miembro que implementan en cada caso. Las explicaciones giran en torno un elemento principal, que son los parámetros configurables de los controladores; y cuatro conceptos básicos. Los dos primeros conceptos son los niveles de abstracción alto y bajo, que designan respectivamente lo que debe "conocer" el agente que usa la interfaz (usuario o aplicación) y lo que "entiende" el driver. Les siguen los "nombres locales" o "alias" de los parámetros, que son los identificadores que se utilizan para designar los parámetros desde el punto de vista del usuario (alto nivel). Finalmente, los nombres "remotos" o "a nivel de driver" se utilizarán sólo en operaciones de bajo nivel, para comunicarse con el controlador. Salvar el obstáculo entre los alias y los nombres a nivel de driver de forma transparente es el objetivo más importante del presente nodo.

Clase *upward_interface*

La denominada *upward_interface* es la clase que representa la capa más alta de abstracción dentro del código, es decir, orientada a la comunicación con el agente (aplicación o usuario) que quiere comunicarse con el controlador. Es importante advertir que esta clase contiene referencias a las otras dos para evitar las comunicaciones globales entre clases, lo cual no responde intuitivamente al diseño conceptual pero sí lo hace su comportamiento o funcionamiento resultante.

Su trabajo consiste en habilitar canales de comunicación ROS en forma de servicios y servidor de parámetros para recibir y transmitir modificaciones de parámetros tanto en dirección al driver por parte de un nodo externo, como en el sentido contrario en caso de que se producan cambios en la capa inferior que la interfaz interpretará como un evento para propagar dicho cambio.

Datos Miembro

- **listOf_services** es un dato estático de tipo lista en el que se almacenan las instancias de los servicios publicados para recibir peticiones externas. En caso de futuras implementaciones que permitan varias instancias de *upward_interface*, los servicios seguirían siendo únicos al almacenarse en la misma lista.
- **translator** almacena una referencia a la instancia de la clase *translator* que debe ser utilizada para las traducciones.
- **driverMgr** contiene una referencia a la instancia de la clase *driver_manager* que se usará para la comunicación con el controlador.
- **avoidRemoteReconf** contabiliza el número de cambios recibidos desde el driver que aún quedan por tratar, reduciéndose de nuevo hasta '0' tras tratarlos. Si el cambio es local, el contador no aumenta y los cambios son propagados hacia abajo. Su objetivo es evitar que los cambios se propaguen de nuevo hacia el controlador formando un bucle descontrolado de actualizaciones.
- **dynServer** es la instancia del *servidor dinámico de parámetros* del nodo interfaz. Una vez iniciado podrá ser leído desde nodos externos o recibir actualizaciones de estos.

Funciones Miembro

- **dynServerCallback(self, dynConfiguration, levelCode)**
  Manejador para los eventos del *Servidor de Reconfiguración Dinámica* . Retorna una configuración de parámetros en el mismo formato que la recibe (un diccionario).)

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param dynConfiguration - un diccionario {parámetro:valor} con los cambios a realizar. Precondición: el diccionario se refiere a los parámetros según sus nombres LOCALES.

  @param levelCode - el código que se genera durante el *callback* como marca del grupo de parámetros que han cambiado.

  @return El diccionario con la configuración final resultante (obligado para todos los callback).

- **getAnyProperty(self, propertyName)**
  Método genérico para obtener el valor de cualquier parámetro tras llamar al método específico correspondiente. Necesita el nombre LOCAL del parámetro y responde con el valor remoto correspondiente.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro a leer.

  @return el valor leído o *None* si hubo algún error.

- **getDispImage(self, srvMsg)**
  Método para obtener una secuencia de imágenes de disparidad (“Disparity Images”) cuando es llamado desde un servicio.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param srvMsg - una estructura de servicio incluyendo:

  –int64 nImages - número de imágenes a retransmitir.

  –string sourceTopic - topic original del cual obtener las imágenes.

  –string responseTopic - dirección del topic al que enviar las imágenes.

  @return las imágenes leídas de la fuente.

- **getFixedTypeProperty(self, localPropName, valueType)**
  Método principal para obtener valores de parámetros conociendo ya el nombre y tipo de dato para devolver el valor actual en el driver.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param localPropName - el nombre LOCAL del parámetro a leer.

  @param valueType - el tipo de dato esperado del parámetro a leer.

  @return el valor leído o *None* si hubo algún error.

- **getImage(self, srvMsg)**
  Método para obtener una secuencia de imágenes cuando es llamado desde un servicio.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param srvMsg - una estructura de servicio incluyendo:

  –int64 nImages - número de imágenes a retransmitir.

  –string sourceTopic - topic original del cual obtener las imágenes.

  –string responseTopic - dirección del topic al que enviar las imágenes.

  @return las imágenes leídas de la fuente.

- **getStringProperty(self, srvMsg)**
  Método genérico para obtener valores de tipo cadena que llama al método "getFixedTypeProperty" con los valores adecuados en función del tipo y canal al que corresponde el parámetro.

  @param srvMsg - una estructura de servicio residual al hacer la llamada pero que es ignorada.

  @return la respuesta inmediata del método "getFixedTypeProperty" al que llama.

- **getIntProperty(self, srvMsg)**
  Método análogo a getStringProperty para datos de tipo entero.

- **getFloatProperty(self, srvMsg)**
  Método análogo a getStringProperty para datos de tipo real.

- **getBoolProperty(self, srvMsg)**
  Método análogo a getStringProperty para datos de tipo booleano.

- **getTopicLocation(self, getStrMsg)**
  Obtiene el topic de un parámetro dentro del entorno ROS. Precondición: el parámetro tiene que estar publicado como topic.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param getStrMsg - una estructura incluyendo:

  – string topicName - the name of the topic which location is to be returned.

  – string topicValue - espacio para el valor de retorno en la respuesta.

  @return un mensaje de texto describiendo si hubo éxito o describiendo un posible error.

- **get_property_list(self)**
  Método auxiliar para obtener la lista de parámetros conocidos por el traductor.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @return una lista completa de parámetros del traductor.

- **listenToRequests(self)**
  Método para poner en marcha los servicios principales que estarán a la espera de solicitudes. No recibe nada y sólo devuelve éxito por si pudiera necesitarse más adelante.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @return Verdadero si se completa. En caso contrario propaga una excepción en función del problema.

- **publishDispImages(self, srvMsg)**
  Método específico para retransmitir una secuencia de imágenes de disparidad (“Disparity Images”) cuando es llamado desde un servicio. Retransmite la fuente especificada hacia el topic que se solicite en el parámetro correspondiente.

  Specific purpose method meant to retransmit, when asked via-service, a DISPARITY image topic through a topic path receiving the original path, the new path and the amount of messages.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param srvMsg - una estructura de servicio incluyendo:

  –int64 nImages - número de imágenes a retransmitir.

  –string sourceTopic - topic original del cual obtener las imágenes.

  –string responseTopic - dirección del topic al que enviar las imágenes.

  @return un mensaje de texto describiendo si hubo éxito o describiendo un posible error.

- **publishImages(self, srvMsg)**
  Método específico para retransmitir una secuencia de imágenes cuando es llamado desde un servicio. Retransmite la fuente especificada hacia el topic que se solicite en el parámetro correspondiente.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param srvMsg - una estructura de servicio incluyendo:

  –int64 nImages - número de imágenes a retransmitir.

  –string sourceTopic - topic original del cual obtener las imágenes.

  –string responseTopic - dirección del topic al que enviar las imágenes.

  @return un mensaje de texto describiendo si hubo éxito o no.

- **setAnyProperty(self, propertyName, newValue)**
  Método genérico para modificar el valor de cualquier parámetro tras llamar al método específico correspondiente. Necesita el nombre LOCAL del parámetro y actualiza el valor remoto correspondiente.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro a modificar.

  @param newValue - el nuevo valor para asignar al parámetro.

  @return Verdadero si hubo éxito; Falso si no.

- **setFixedTypeProperty(self, localPropName, newValue, valueType)**
  Método principal para modificar valores de parámetros conociendo ya el nombre y tipo de dato e informar de si éste fue cambiado correctamente en el driver. Precondition: localPropName debe ser un nombre LOCAL conocido y el cambio se realiza en un servidor ajeno (otro nodo). No está diseñado para realizar cambios en el servidor propio mediante "loopback".

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param localPropName - el nombre LOCAL del parámetro a modificar.

  @param newValue - el nuevo valor para asignar al parámetro.

  @param valueType - el tipo de dato esperado del parámetro a modificar.

  @return Verdadero si hubo éxito; Falso si no.

- **setStrProperty(self, setterMessage)**
  Método genérico para modificar valores de tipo cadena que llama al método "setFixedTypeProperty" con los valores adecuados en función del tipo y canal al que corresponde el parámetro.

  @param srvMsg - una estructura de servicio residual al hacer la llamada pero que es ignorada.

  @return la respuesta inmediata del método "setFixedTypeProperty" al que llama.

- **setIntProperty(self, setterMessage)**
  Método análogo a setStrProperty para datos de tipo entero.

- **setFloatProperty(self, setterMessage)**
  Método análogo a setStrProperty para datos de tipo real.

- **setBoolProperty(self, setterMessage)**
  Método análogo a setStrProperty para datos de tipo booleano.

- **setTopicLocation(self, setStrMsg)**
  Establece una nueba ubicación para un topic existente dentro del entorno ROS.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param setStrMsg - una estructura de servicio incluyendo:

  –string topicName - el nombre de parámetro cuyo topic se reubicará.

  –string newValue - la nueva dirección donde reubicar el topic.

  –string setAnswer - espacio para almacenar la respuesta correspondiente.

  @return un mensaje de texto describiendo si hubo éxito o describiendo un posible error.

- **updateSelfFromRemote(self, dynConfiguration)**
  Este método se diseñó para "traducir" nombres locales en sus asociados remotos (los del driver) en caso de ser necesario previamente a utilizar el método "updateSelfParameters" (orientado sólo a los nombres LOCALES).

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param dynConfiguration - un diccionario parámetro:valor con los cambios a realizar. Precondición: el diccionario se refiere a los parámetros según sus nombres REMOTOS.

  @return El diccionario con la configuración final resultante tras los cambios (obligado para todos los callback).

- **updateSelfParameters(self, newConfig, avoidPropagation=False)**
  This method is responsible for changing the node's own dynamic reconfigure parameters from its own code ***but avoiding a chain of uncontrolled callbacks!!.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param newConfig - un diccionario {parámetro:valor} con los cambios a realizar. Precondición: el diccionario se refiere a los parámetros según sus nombres LOCALES.

  @param avoidPropagation - valor booleano que indica si es necesario o no evitar la propagación de los cambios hacia abajo (en caso de que estos ya provengan del controlador).

  @return El diccionario con la configuración final resultante tras los cambios (obligado para todos los callback).

6.5.2. Clase *driver_manager*

La clase *driver_manager* contiene las funciones necesarias para comunicarse con la clase superior (*upward_interface*) y para enviar los posibles mensajes correspondientes al controlador de dispositivo, pero también los procedimientos necesarios para leer la configuración del driver y un método de callback para ser informado de sus cambios.

Desde dicha clase se pueden realizar conexiones a servidores de parámetros, llamadas a servicios y escuchar y publicar en topics ROS. En este nivel se utilizan los identificadores de parámetros tal y como son conocidos por el driver, en vez de sus alias. Para comunicarse con la clase superior se solicitan las traducciones de parámetros a la clase *iface_translator* .

Datos Miembro

- **dynSrvTimeout** es un valor que representa el tiempo, en milésimas de segundo, que se espera una conexión a un servicio antes de descartarla como fallida.
- **dynServers** es una lista con las direcciones de los servidores de parámetros remotos de los drivers en el entorno ROS.
- **createdMuxes** almacena, en una estructura de tipo diccionario, las reubicaciones de topics que se han realizado, con el objetivo de saber dónde se encuentran actualmente dichos topics.
- **paramServers** es un diccionario que tiene como clave las direcciones a los distintos servidores de parámetros externos y el valor almacenado son las conexiones ya realizadas para poder comunicarse con ellos durante la ejecución de la interfaz.
- **avoidSelfReconf** contabiliza el número de cambios realizados localmente que aún quedan por tratar, reduciéndose de nuevo hasta '0' tras tratarlos. Si el cambio viene del driver, el contador no aumenta y los cambios son aplicados localmente. Su objetivo es evitar que los cambios generen nuevos eventos formando un bucle descontrolado de actualizaciones.

Funciones Miembro

- **callService(service, arguments, valueType)**
  Realiza llamadas a servicios de forma genérica para poder llamar independientemente del tipo de dato. No está en uso actualmente pero podría ser útil en el futuro.

  @param service - el nombre del servicio que será llamado.

  @param arguments - los argumentos que se pasarán durante la llamada al servicio.

  @param valueType - el tipo de dato de los mensajes solicitud/respuesta según los ficheros .srv correspondientes.

  @return Verdadero en caso de éxito.

- **dynClientCallback(self, dynConfiguration)**
  Método de callback que responde cuando hay cambios en el servidor de parámetros remoto, es decir: que recibe nombres REMOTOS y sus valores en la variable "dynConfiguration".

  Si el cambio fue iniciado por este mismo nodo, el valor de "avoidSelfReconf" será distinto de "0" y no se realiza ninguna acción. En otro caso los cambios son enviados al método "updateSelfFromRemote" para actualizar los valores en el servidor local.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param dynConfiguration - un diccionario con la configuración a aplicar.

@return la configuración resultante tras la actualización. (Que puede no ser la esperada)

- **getTopic(self, topicName, data_type)**
  Obtener el valor actual de un topic.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param topicName - nombre del topic en el que leer el valor.

  @param data_type - el tipo de datos esperado en el topic.

  @return el valor obtenido (sin previa comprobación o validación).

- **getValue(self, propName, dynServerPath)**
  Obtiene el valor actual de un parámetro en el servidor de destino a partir de su nombre REMOTO.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propName - nombre REMOTO del parámetro a leer.

  @param dynServerPath - dirección del servidor que almacena el parámetro.

  @return El valor obtenido si fue posible y "None" si no.

- **relocateTopic(self, oldAddress, newAddress)**
  Este método está diseñado para cambiar en tiempo de ejecución la ubicación de un topic a una nueva dirección. Dado que el entorno no permite esta acción propiamente dicha, se replica el topic con la herramienta "mux".

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param oldAddress - la dirección del topic (ya existente) a reubicar.

  @param newAddress - la nueva dirección en la que publicar el topic.

  @return Falso si hubo algún error (si el primer carácter de la nueva dirección NO es una letra), verdadero si se realizó con éxito. Si hubo un error desconocido se lanza una excepción.

- **sendByTopic(self, topic, value, data_type)**
  Publica el valor dado una sola vez en el topic especificado y retorna. Si no hay conexiones a la escucha, el método espera hasta un segundo antes de realizar la publicación y sale en cualquier caso.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param topic - el nombre del topic en el que publicar el valor.

  @param value - valor a publicar.

  @param data_type - el tipo de datos que se espera publicar en el topic.

- **setRemoteValue(self, rPropName, newValue, dynServerPath, ifcNodeInstance)**
  Trata de asignar un valor dado a un parámetro especificado según su nombre REMOTO en el servidor de destino.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param rPropName - nombre REMOTO del parámetro a modificar.

  @param newValue - valor para asignar al parámetro.

  @param dynServerPath - dirección del servidor remoto en el que modificar el parámetro.

  @param ifcNodeInstance - el objeto "upward_interface" que permite cambiar los parámetros en el servidor local.

  @return El valor obtenido si es posible o "none" si hubo algún problema

- **topic_is_published(topicName)**
  Método estático que comprueba si un topic está publicado actualmente en el entorno.

  @param topicName - el nombre del topic cuya existencia será comprobada.

  @return Verdadero en caso de éxito.

6.5.3. Clase *iface_translator*

Es la parte con menor complejidad pero no menos importante pues aunque con numerosas funciones auxiliares, destacan tres funciones principales, a saber: "readYAMLConfig", para leer las traducciones desde el diccionario de parámetros (fichero de configuración); "interpret", que traduce un alias de parámetro a su nombre de bajo nivel para que sea reconocido por el driver; y "reverseInterpret", que a partir de un identificador de bajo nivel, retorna su alias. Ésta clase será utilizada por las otras dos clases para que puedan comunicarse y entenderse a pesar de utilizar "nombres" distintos.

Datos Miembro

- **property_config_file** es el nombre del fichero de configuración, que es útil también como identificador único en caso de utilizar múltiples instancias de la clase de traducción.
- **translations** almacena una lista de cadenas de texto con las direcciones ROS de los drivers y otra indexada de forma análoga (el mismo índice corresponde al mismo driver) de diccionarios con los pares "nombre local" - "nombre remoto".
- **ReversePropDict** es una estructura de tipo diccionario que tiene como claves los nombres remotos y como valor los nombres locales para poder realizar la traducción en el sentido contrario sin recorrer por completo la variable "translations".

Funciones Miembro

- **canGet(self, propertyName)**
  Comprueba si el parámetro cuyo nombre recibe puede ser leído o no en función de si es conocido, accesible, etc.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro a comprobar.

  @return Verdadero si el parámetro fue encontrado y puede ser leído. Falso si no.

- **canSet(self, propertyName)**
  Comprueba si el parámetro cuyo nombre recibe puede ser modificado o no en función de si es conocido, sólo-lectura, etc.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro a comprobar

  @return Verdadero si el parámetro fue encontrado y puede ser modificado. Falso si no.

- **cleanRenamings(self, driverManager)**
  Lanza un nodo "mux" de ROS que reubica los topics a nuevas direcciones según los parámetros de clase "renaming" en el fichero de traducciones. Precondición: el *driver_manager* tiene que estar previamente inicializado y los topics originales ya publicados.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param driverManager - referencia a una instancia de *driver_manager* que se usará en las reubicaciones

- **dynamicServers(self, checkDynamic=True)**
  Genera una lista con las direcciones relativas o absultas de los diferentes servidores de reconfiguración dinámica según los datos del fichero de traducciones YAML cargado.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param checkDynamic - decide si comprobar que realmente hay parámetros dinámicos en cada servidor de la lista.

  @return La lista de direcciones de los servidores disponibles.

- **generateReverseDictionary(self)**
  Genera un diccionario para el traductor usando como clave el nombre REMOTO (en el driver) y el nombre LOCAL como valor para hacer búsquedas inversas.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @return El diccionario inverso de parámetros indexados por nombre REMOTO (dos entradas por cada uno para incluirlos con y sin la dirección completa).

- **getServerPath(self, propName)**
  Dado un parámetro, obtiene la dirección ROS del servidor que lo contiene para poder modificar la configuración del driver correspondiente.

  @param propName - el nombre local del parámetro a buscar.

- **get_property_list(self)**
  Método que devuelve la lista completa de parámetros.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @return lista completa de los parámetros conocidos por el traductor.

- **get_topic_path(self, topicName)**
  Método para obtener la dirección de un topic determinado. Si se recibe un topic existente se comprueba su existencia y se devuelve; si no se trata de traducir como un parámetro.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param topicName - el nombre el parámetro o topic a buscar.

  @return la dirección completa del topic asociado al nombre recibidio.

- **interpret(self, propertyName)**
  Devuelve los datos asociados a un nombre LOCAL de los conocidos gracias al fichero de configuración (o traducción) según: [nombre REMOTO, tipo de dato, canal].

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro a utilizar.

  @return una tupla "[nombre REMOTO, tipo de dato, canal]" si encuentra el parámetro. Si no: "None".

- **prop_exists(self, propertyName)**
  Comprueba la existencia en el diccionario de un parámetro dado según su nombre LOCAL.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro a comprobar.

  @return Verdadero si se encontró el parámetro en el diccionario. Falso si no.

- **readYAMLConfig(self, file_name)**
  Carga todo el fichero "file_name" (que contiene los diccionarios de parámetros) en el traductor. Retorna los diccionarios para que estos sean recordados durante la ejecución.

  a modo de in which to read the properties. Returned Value MUST be a tuple of two lists with strings and dictionaries respectively.

@param self - la instancia cuyo método es llamado (común en los métodos Python)

@param file_name - el nombre y dirección de un fichero del que leer la configuración (diccionarios).

@return Una tupla de dos listas. La primera contiene las direcciones a cada driver y la segunda el diccionario asociado a cada elemento de la primera.

- **reverseInterpret(self, reverseProperty)**
  Localiza el parámetro "reverseProperty" (según su nombre REMOTO) en el diccionario inverso para obtener el nombre LOCAL.

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param reverseProperty - el nombre REMOTO del parámetro.

  @return El nombre LOCAL del parámetro si fue encontrado. Si no: "None".

- **updatePptyRef(self, propertyName, newPath)**
  Cambia el nombre REMOTO o dirección de un parámetro en el traductor. (Sólo durante la ejecución, no se almacenan los cambios).

  @param self - la instancia cuyo método es llamado (común en los métodos Python)

  @param propertyName - el nombre LOCAL del parámetro.

  @param newPath - el nombre REMOTO para asignarle al parámetro.

  @return la nueva dirección o nombre asignado al parámetro si éste fue encontrado. Si no: "None".

- **get_basic_name(propName)**
  Método estático para obtener el nombre de base en una dirección, es decir la última sección: /rootDir/secondary/propName/ = /rootDir/secondary/propName = propName

  @param propName - el nombre completo de un parámetro, posiblemente incluyendo una dirección relativa.

  @return la última parte del nombre (la base) sin la dirección o una cadena vacía si no fue posible la acción.