Manual para el diseño de módulos básicos

Sergio A. Serrano

Instituto Nacional de Astrofísica, Óptica y Electrónica (INAOE),
Luis Enrique Erro # 1, C.P. 72840,
Santa María Tonantzintla, Puebla, México
sserrano@inaoep.mx

1 Introducción

Uno de los principales intereses en robótica de servicio consiste en el desarrollo de robots capaces de desempeñar la mayor cantidad de tareas que un humano realiza dentro de un entorno doméstico, como puede ser limpiar el hogar, atender en la entrada y poner la mesa para cenar. Para lograr esto el robot debe tener un conjunto de distintas habilidades que le permitan tomar las acciones necesarias para realizar una tarea. Es por esto que equipos que programan software para esta clase de robots trabajan simultáneamente en desarrollo de dichas habilidades.

Sin embargo, cuando varias personas programan componentes (*e.g.*, las habilidades del robot) de un mismo sistema, es necesario establecer una plantilla o estructura que todos los componentes deben cumplir. Hacer esto trae diversas ventajas como son:

- Facilita la integración de diversos componentes en un solo sistema.
- Permite el desarrollo de varios componentes de manera simultánea.
- Al seguir los componentes una misma estructura, su documentación resulta sencilla de entender para quienes no desarrollaron el componente.

Por este motivo, en este documento se presenta una estructura general, que hemos denominado **módulo básico**, para el diseño, implementación y documentación de habilidades para el robot de servicio Markovito[1] del laboratorio de robótica del INAOE. En la sección 2 se presenta el concepto del módulo básico y de cómo este forma parte de un sistema de toma de decisiones de un robot. En la sección 3 se describe la estructura general que define al módulo básico y la notación utilizada para documentarlo. Después, en la sección 4 se detalla una colección de funciones (en **Python**) encargadas de la comunicación de un módulo básico con el resto del sistema, así como una descripción de la estructura general de un sistema de toma de decisiones que integra módulos básicos. Por último, en la sección 5, se presenta un ejemplo de cómo se pueden utilizar los archivos del paquete Basic Module Util como punto de partida para implementar un módulo básico.

[1] http://robotic.inaoep.mx/~markovito/

2 ¿Qué es un módulo básico?

Un módulo básico es una colección de funciones que pueden servir para resolver un mismo tipo de subtarea. Mediante la agrupación de funciones, por el tipo de subtarea que son capaces de resolver, es posible modularizar el diseño sistemas para resolver tareas compuestas por varias subtareas. Además, gracias a que la implementación de un módulo básico no depende de otros módulos, es posible usar distintas combinaciones de ellos según lo requiera una tarea, promoviendo así la reutilización de código.

En la figura 1 se muestra un ejemplo de un sistema que cuenta con tres módulos: i) *reconocimiento de objetos y personas*, ii) *navegación* y iii) *reconocimiento y síntesis de voz*. En la figura 2 se muestran dos ejemplos de tareas que requieren de la solución de varias subtareas: i) *encontrar una manzana* e ii) invitar a una persona al comedor. Se puede apreciar como para resolver ambas tareas el robot emplea funciones de los módulos de *reconocimiento de objetos y personas* y de *navegación*, sin embargo, para notificar a la persona que la cena está lista, se requiere del módulo de *reconocimiento y síntesis de voz*.

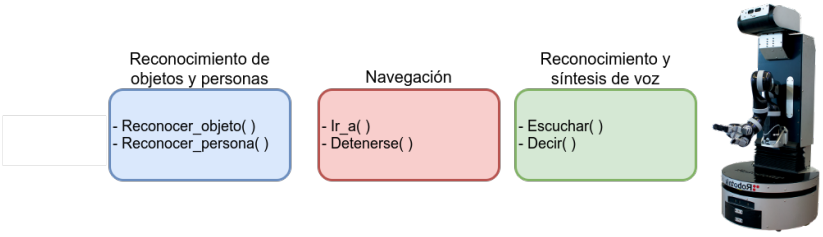

Fig. 1. Ejemplo de un sistema que cuenta con módulos básicos para el reconocimiento de objetos y personas, navegación y reconocimiento de voz. Estos módulos le permiten al robot identificar visualmente cosas, reconocer comando por voz y desplazarse dentro de su entorno.

3 Definición de un módulo básico

Un módulo básico se define por el conjunto de acciones (funciones) que es capaz de realizar. A su vez, cada acción se define por la información que necesita para ejecutarse (parámetros de entrada), la información que es capaz de obtener (parámetros de salida/retorno) y en qué escenarios puede ejecutarse (condiciones de ejecutabilidad).

La definición de un módulo básico se documenta utilizando la notación JSON[2], en un archivo en el que se especifican todos los detalles necesarios para poder

[2] https://www.json.org/json-es.html

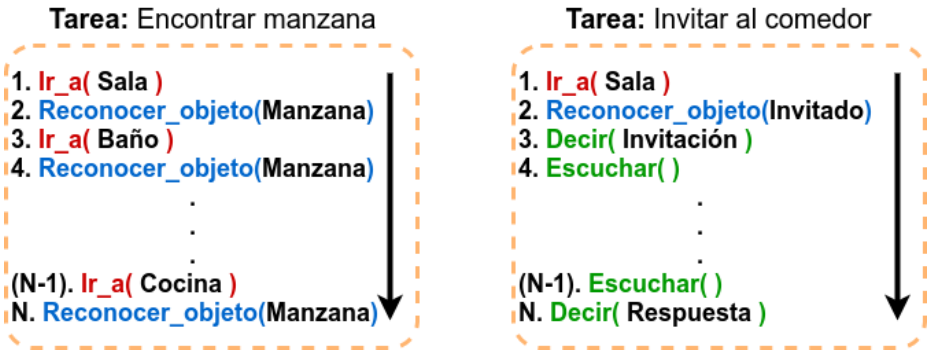

Fig. 2. La mayoría de las tareas que un robot de servicio enfrenta en un hogar requieren de diversas habilidades. Por ejemplo, con los módulos de la figura 1 un robot podría hacer cosas como encontrar una manzana o invitar a alguien a pasar al comedor mediante la ejecución de funciones de diversos módulos.

ejecutar las funciones que contiene el módulo. En la figura 3 se muestra la estructura general que el JSON de un módulo debe seguir.

A continuación se describe cada uno de los atributos de un JSON utilizado para documentar un módulo básico.

- **Basic Module:** Nombre del módulo.
- **Brief Description:** Descripción general de las habilidades que encapsula el módulo, *e.g.*, habilidades para el reconocimiento de objetos o el reconocimiento de comandos por voz.
- **Hardware Required:** Lista de elementos de hardware con los debe contar el robot para poder ejecutar todas las funciones del módulo.
- **Functions:** Lista de funciones que el módulo puede ejecutar. A su vez, cada función está definida por los siguientes campos.
  - **Name:** Nombre de la función.
  - **Exec-condition:** Condiciones en el robot o el entorno que deben cumplirse para que la función pueda ser ejecutada.
  - **Description:** Breve descripción de lo que hace la función.
  - **Input-params:** Lista de parámetros de entrada que la función requiere para poder ejecutarse. Por ejemplo, en la figura 2 la función **Ir_a** requiere de una ubicación meta para saber a donde ir.

```
{
  "Basic Module": "",
  "Brief Description": "",
  "Hardware Required":
  [
    ""
  ],
  "Functions":
  [
    {
      "Name":"",
      "Exec-condition":"",
      "Description":"",
      "Input-params":
      [
        {
          "Name":"",
          "Data-type":"",
          "Description":""
        }
      ],
      "Output-params":
      [
        {
          "Name":"",
          "Data-type":"",
          "Description":""
        }
      ]
    }
  ]
}
```

Fig. 3. Estructura general de un archivo JSON con la que se documenta un módulo básico.

- **Output-params:** Lista de parámetros de salida que la función retorna al terminar su ejecución. Por ejemplo, en la figura 2 la función **Escuchar** retorna una cadena de caracteres equivalente al comando de voz que el robot ha escuchado.

Tanto en **Input-params** y **Output-params**, cada parámetro se define con los siguientes campos:

– **Name:** Nombre del parámetro.

– **Data-type:** Tipo de dato que el parámetro recibe (**Input-params**) o retorna (**Output-params**), y se especifica con el tipo de mensaje ROS, *e.g.*, "std_msgs/String", "geometry_msgs/Point", etc.

– **Description:** Breve descripción de para qué sirve el parámetro.

De esta manera, cuando un desarrollador define el JSON de un módulo básico se compromete a implementar funciones capaces de realizar lo indicado en el JSON, con lo que otros desarrolladores pueden partir de la suposición de que estas funciones ya existen y funcionan, para ocuparse en implementar funciones para otras subtareas.

En la siguiente sección se explica cómo un módulo básico se puede implementar como un nodo ROS[3] y cómo varios módulos se integran en un solo sistema.

4 Implementación de un módulo básico

En esta sección se describe el funcionamiento general de un sistema que integra módulos básicos, así como una colección de métodos (implementados en **Python**) que permite homogeneizar la comunicación con los módulos.

4.1 Esquema general del sistema

La arquitectura de Markovito para la solución de tareas (mostrada en la Fig. 4) se implementa utilizando ROS[4] y está constituido de tres componentes principales: 1) un coordinador de tarea, 2) una memoria de pizarrón y 3) tantos módulos básicos como el robot tenga; donde cada componente se implementa como un nodo (véase la Fig. 5).

1. **Coordinador de tarea:** El nodo del coordinador (**Master Node**) se encarga de la toma de decisiones del robot. Mediante el tópico **/master/output** invoca funciones de los módulos básicos para que el robot realice acciones y en **/blackboard/input** le solicita resultados previos a la memoria de pizarrón. Además, se encuentra subscrito a **/function/output** para recibir el resultado de la última función invocada y a **/blackboard/output** para recibir algún resultado que le haya solicitado a la memoria.

2. **Memoria de pizarrón:** El nodo de la memoria (**Blackboard Node**) se subscribe a **/function/output** para recibir los resultados de las funciones

[3] https://www.ros.org/
[4] En caso de no estar familiarizado con ROS, se le sugiere visitar el sitio de tutoriales http://wiki.ros.org/ROS/Tutorials.

y almacenarnos, a **/blackboard/input** para recibir peticiones del coordinador de algún resultado previo, mientras que en **/blackboard/output** publica el resultado solicitado.

3. **Módulo básico:** El sistema incluye un nodo por cada módulo básico. Todos los nodos de módulo básico se subscriben a **/master/output** para recibir comandos del coordinador de invocar una función. Además, todos los nodos de módulo publican en **/function/output** el resultado de ejecutar sus funciones.

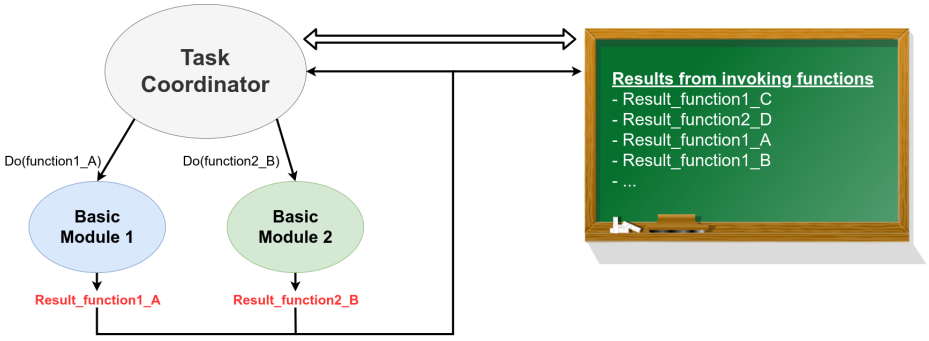

Fig. 4. Arquitectura general de Markovito para solución de tareas. El coordinador de tareas invoca funciones contenidas en los módulos básico. Los resultado de ejecutar una función se envían al coordinador y se almacenan en una memoria (pizzarrón). Además el coordinador puede solicitar resultados de invocaciones anteriores al pizarrón.

4.2 Funciones de comunicación para módulos básicos

Para integrar un módulo básico que esté implementado en Python a la arquitectura, se deben seguir los siguientes pasos:

1. Instalar el paquete ROS **rospy_message_converter**[5].

2. Importar en el script python el archivo **commBM.py**.

3. Subscribirse a **/master/output** y publicar en **/function/output** (tipo **std_msgs/String**).

En este punto su script **Python** está listo para comunicarse con la arquitectura mediante las siguientes funciones (implementadas en **commBM.py**):

[5] http://wiki.ros.org/rospy_message_converter

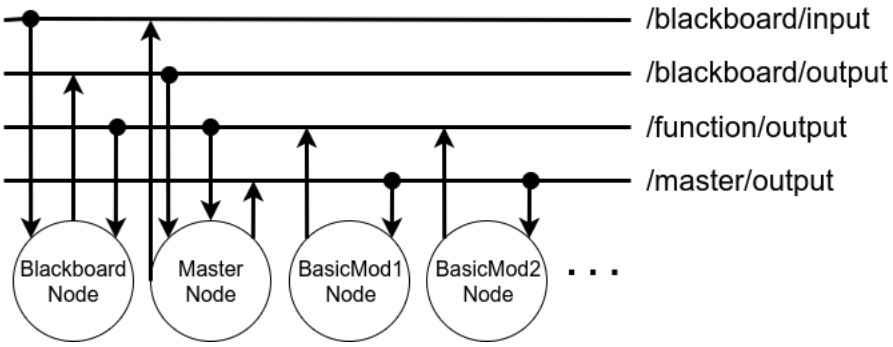

Fig. 5. Esquema general de sistema para la solución de tareas que integra módulos básicos implementado en ROS. El nodo *Blackboard* se encarga de estar guardando los parámetros de salida resultantes de invocar funciones. El nodo *Master* invoca funciones, se queda con algunos de los parámetros de salida y, de ser necesario, puede solicitar al nodo *Blackboard* valores de invocaciones pasadas.

- **readFunCall:** Esta función transforma mensajes publicados por el **Master Node** en **/master/output** en una lista que contiene el nombre del módulo solicitado, el nombre dela función solicitada y la lista de parámetros de entrada para la función.

- **writeMsgFromRos:** Esta función transforma una lista de mensajes ROS en una cadena de texto, lista para publicarse en **/function/output**, como se muestra en la Fig. 6.

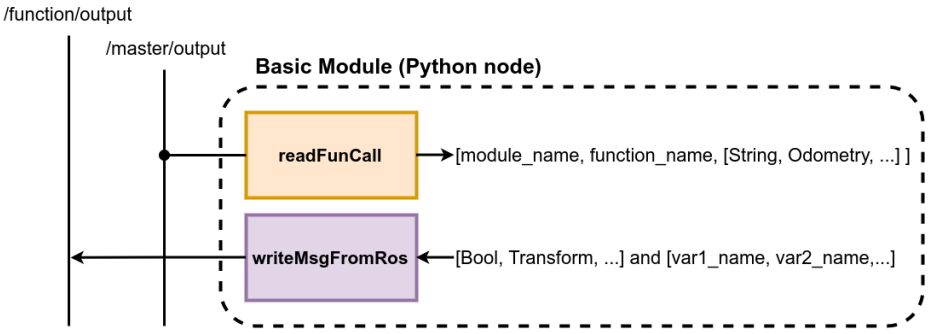

Fig. 6. Para que un nodo de módulo básico se comunique con el sistema, basta con que use las funciones **readFunCall** (para recibir peticiones de invocación de funciones) y **writeMsgFromRos** (para publicar el resultado de una función del módulo como una cadena de caracteres).

4.3 Nodo puente para módulos básicos en C++

Debido a que el paquete ROS de **rospy_message_converter** no provee soporte para C++ y es necesaria para la conversión de mensajes ROS a cadenas de caracteres (y viceversa), se requiere de un nodo **python** que sirva como puente entre un módulo básico que ha sido implementado como un nodo en C++ y el resto del sistema.

En la Fig. 7 se muestra la configuración de conexión con la que un módulo básico implementado en C++ puede comunicarse con el sistema. Los tópicos ROS que permiten establecer dicha conexión se han agrupado en tres categorías:

- **comm-topics:** Los tópicos **/master/output** y **/function/output**, mediante los cuales el nodo puente se comunica con el coordinador y la memoria pizarrón.

- **data-topics:** Tópicos que a través de los cuales el nodo puente y el nodo del módulo se transmiten parámetros de entrada y de salida de las funciones del módulo. Tanto el nodo puente y el nodo del módulo básico deben suscribirse y publicar en cada uno de estos tópicos. Debe crearse un tópico por cada tipo de dato que las funciones en el módulo básico recibe o retorna.

- **sync-topics:** Par de tópicos con los que el nodo puente y el del módulo básico de coordinan para recibir y enviar datos a su contra parte.

El nodo puente maneja dos tipos de eventos: 1) cuando recibe una petición para invocar una función desde el coordinador y 2) cuando recibe el resultado de haber ejecutado una función desde el módulo básico. Estos eventos con procesados de la siguiente manera:

1. **Recepción de petición**
   (a) Con la función **readFunCall** transforma la cadena de texto recibida.

   (b) Verifica que la función solicitada es del módulo básico al que está conectado.

   (c) Publica los parámetros de entrada recibidos en su correspondiente **data-topic** y realiza un pequeño *delay*.

   (d) Publica en su **sync-topic** el nombre de la función solicitada.

2. **Recepción de resultados**
   (a) Por medio del **sync-topic** al que está suscrito, recibe la lista de los **data-topics** en los que se publicaron los valores y nombres de los parámetros de salida que conforman el resultado de haber terminado de ejecutar la función.

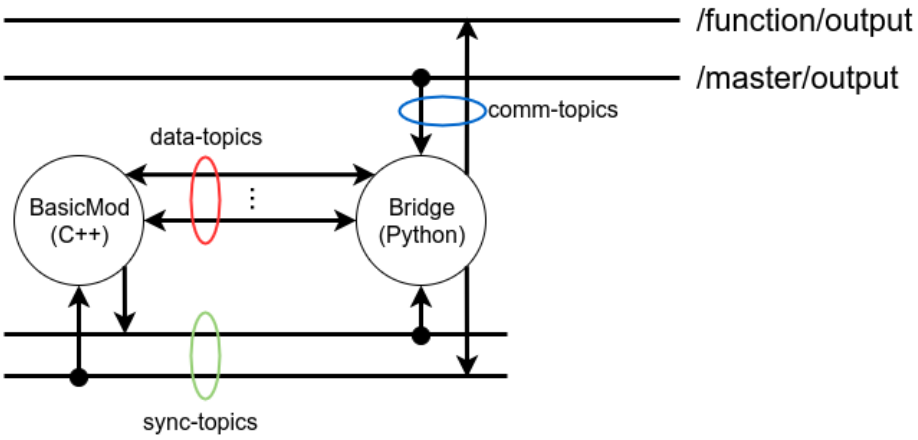

Fig. 7. Un nodos de módulo básico que ha sido implementado en **C++** requiere de un nodo puente (implementado en **Python**) para poder recibir parámetros de entrada por **/master/output** y publicar sus parámetros de salida en **/function/output**. Mediante los tópicos **data-topics** el nodo puente transmite y recibe parámetros del nodo del módulo, mientras que por los **sync-topics** se notifican cuando ya se han enviado nuevos datos.

(b) Con la función **writeMsgFromRos**, los nombres y los valores de los parámetros, construye el mensaje de retorno y lo publica en **/function/output**.

En cuanto a la recepción de peticiones por parte del nodo del módulo básico, recibe el nombre de la función solicitada y la lista de **data-topics** en los que se publicaron los parámetros de entrada, mediante el **sync-topic** al que está suscrito.

4.4 Código de ejemplo

Para ver detalles de implementación, ir a Basic Module Util para descargar un paquete ROS que tiene implementado:

- Nodo maestro
- Nodo puente
- Nodo módulo básico (en **C++** y **Python**)

Es importante recordar que para que este código funcione, es necesario haber instalado el paquete ROS rospy_message_converter. Además, en el directorio **utils** del paquete Basic Module Util se encuentra el script **writeBM.py**, el cual sirve

para capturar la descripción de un módulo básico en formato JSON (como se describió en la sección 3).

5 Utilización de archivos plantilla

En esta sección se describe cómo utilizar los archivos plantilla de módulo básico (en **C++** y **Python**) para implementar los propios. Comenzamos por presentar un caso de ejemplo de un módulo básico, mostrado en la Fig. 8.

```
{
    "Basic Module": "Modulo-Base",
    "Brief Description": "Para controlar la base movil del robot.",
    "Functions": [
        {
            "Description": "Realiza un desplazamiento con componente lineal y radial.",
            "Exec-condition": "",
            "Input-params": [
                {
                    "Data-type": "geometry_msgs/Twist",
                    "Description": "Velocidad lineal y angular de la base.",
                    "Name": "velocidad"
                },
                {
                    "Data-type": "std_msgs/Float64",
                    "Description": "Periodo por el cual la base debe moverse a la velocidad especificada por 'velocidad'.",
                    "Name": "duracion"
                }
            ],
            "Name": "desplazar",
            "Output-params": []
        },
        {
            "Description": "Desplaza la base movil a una posicion meta aplicando un control.",
            "Exec-condition": "",
            "Input-params": [
                {
                    "Data-type": "geometry_msgs/Point",
                    "Description": "Posicion meta del robot.",
                    "Name": "posicion_meta"
                }
            ],
            "Name": "posicion",
            "Output-params": [
                {
                    "Data-type": "std_msgs/Bool",
                    "Description": "Indica si la base logro llegar a la posicion meta.",
                    "Name": "bandera_exito"
                },
                {
                    "Data-type": "geometry_msgs/Point",
                    "Description": "Posicion final del robot al momento en que esta funcion termina su ejecucion.",
                    "Name": "posicion_final"
                }
            ]
        }
    ],
    "Hardware Required": [
        "Base movil diferencial"
    ]
}
```

Fig. 8. Módulo de ejemplo, el cual tiene por objetivo implementar funciones para mover la base móvil de un robot mediante dos modalidades: i) mediante desplazamientos en lazo abierto y ii) aplicando un control para llegar a una posición meta.

5.1 Plantilla de módulo básico en Python

El archivo plantilla de módulo básico para Python es **src/basicmodTemplate.py**. Para implementar el módulo de la Fig. 8, la plantilla se debe modificar en dos secciones:

1. Por cada función del módulo básico, implementar una función **Python** que reciba como entrada la lista de mensajes ROS que requiere para desempeñar su tarea; tal como se muestra en la Fig. 9.

2. Dentro de cada función **Python**, se deberán almacenar en dos listas los parámetros de salida de la función y los nombres de los parámetros, tal como se muestra en la Fig. 9. Esto con el objetivo de pasarlos al método **commBM.writeMsgFromRos** y los convierta en una cadena de caracteres que se publique al nodo maestro.

3. Incluir dentro del bloque **if** que comienza en la línea **48**, se debe incluir un caso por cada función del módulo básico. En la Fig. 10 se muestra la plantilla ya modificada para nuestro módulo de ejemplo.

4. El nombre del módulo básico que se esté implementando deberá definirse en la variable **basic_module_name** de la línea **134**.

```python
def __desplazar(self,msgs):
    #Extract the input-parameters
    velocidad = msgs[0]
    duracion = msgs[1]

    #Do the task with the input-parameters

    #Return the output-params
    out_params = []
    names = []
    msg_str = commBM.writeMsgFromRos(out_params, names)
    out_msg = String(msg_str)
    self.__comm_pub.publish(out_msg)
    return
```

```python
def __posicion(self,msgs):
    #Extract the input-parameters
    posicion_meta = msgs[0]


    #Do the task with the input-parameters

    #Return the output-params
    out_params = [Bool(some_variable), Point(another_variable)]
    names = ['banderas_exito', 'posicion_final']
    msg_str = commBM.writeMsgFromRos(out_params, names)
    out_msg = String(msg_str)
    self.__comm_pub.publish(out_msg)
    return
```

Fig. 9.

```python
#Check if this basic module is being requested & invoke the
#requested function
if(bm == self.__basic_mod_name):
    if(func == 'desplazar'):
        self.__desplazar(msgs)
    elif(func == 'posicion'):
        self.__posicion(msgs)
return
```

Fig. 10.

Después de aplicar estas modificaciones se tiene implementada la interfaz del módulo descrito en el JSON de la Fig. 8.

5.2 Plantilla de módulo básico en C++

Para implementar el módulo descrito en la Fig. 8 es necesario modificar tres archivos plantilla: i) los dos del módulo básico (**src/utilBM.cpp** y **include / basicmodutil_pkg/utilBM.hpp**) y ii) el del nodo puente (**src / bridgeTemplate.py**).

Para empezar, se debe determinar la cantidad de **data-topics** necesarios para comunicar el nodo del módulo con su nodo puente. En el caso de nuestro ejemplo, dado el conjunto de tipos de dato que reciben y retornan las funciones (**geometry_msgs/Twist**, **std_msgs/Float64**, **geometry_msgs/Point** y **std_msgs/Bool**) se requerieren cuatro.

El primer archivo a modificar es **src/bridgeTemplate.py**, mediante los siguientes pasos.

1. Además de **std_msgs/String**, importar cada tipo de mensaje que se requiere transmitir por los **data-topics**, como se muestra en la Fig. 11.

2. Empezando en la línea **27**, definir un tópico por cada **data-topic**, como en la Fig. 12.

3. Por cada **data-topic**, crear un método callback, el cual almacene el dato recibido en el diccionario **__buffer**, usando como llave el tópico al que se encuentra suscrito el callback, como se muestra en la Fig. 13.

4. Por cada **data-topic**, empezando en la línea **45**, instanciar un publicador y subscriptor que publiquen y se subscriban a dicho tópico, respectivamente. Además, se deben almacenar en los diccionarios **__topics** y **__pubs** (usando como llave el tipo de mensaje) el **data-topic** y su publicador, como se muestra en la Fig. 14.

5. Por cada **data-topic**, empezando en la línea **65**, crear un elemento en el diccionario **__buffer** (usando como llave el **data-topic**), como se muestra en la Fig. 15.

Una vez modificada la plantilla para el nodo puente, sigue modificar la plantilla para el módulo básico. El archivo de cabecera **include/basicmodutil_pkg/utilBM.hpp** se debe modificar de la siguiente manera.

1. Además de **std_msgs/String**, incluir el archivo de cabecera de cada tipo de mensaje a transmitir por los **data-topics**, como se muestra en la Fig. 16.

```
8  # Import each data-type required by the basic module (see the input and output
9  # parameters of its functions)
10 from std_msgs.msg import String
11 from geometry_msgs.msg import Twist
12 from std_msgs.msg import Float64
13 from geometry_msgs.msg import Point
14 from std_msgs.msg import Bool
```

Fig. 11.

```
26          #data-topics
27          self.__top_data_string = "/"+bm_name+"/twist"
28          self.__top_data_float64 = "/"+bm_name+"/float64"
29          self.__top_data_point = "/"+bm_name+"/point"
30          self.__top_data_bool = "/"+bm_name+"/bool"
```

Fig. 12.

```
145     #Data callbacks only have to store the received data in the buffer in its
146     #respective attribute
147     def __twist_cb(self,data):
148         self.__buffer[self.__top_data_twist] = data
149         return
150     def __float64_cb(self,data):
151         self.__buffer[self.__top_data_float64] = data
152         return
153     def __point_cb(self,data):
154         self.__buffer[self.__top_data_point] = data
155         return
156     def __bool_cb(self,data):
157         self.__buffer[self.__top_data_bool] = data
158         return
```

Fig. 13.

2. Por cada **data-topic**, declarar una variable **string** y un subscriptor como miembros privados de la clase **BMTemplate**, como se muestra en la Fig. 17.

3. Por cada **data-topic**, declarar un método callback como métodos privados de la clase **BMTemplate**, como se muestra en la Fig. 18.

4. Por cada función del módulo básico, declarar un método de la clase **BMTemplate** que reciba como argumentos de entrada los mismos parámetros de entrada que se especifican en el archivo de descripción JSON del módulo, como se muestra en la Fig. 18.

```
44          #data pubs & subs
45          tmp = commBM.getRosType(Twist())
46          self.__topics[tmp] = self.__top_data_twist
47          self.__pubs[tmp] = rospy.Publisher(self.__top_data_twist,Twist,queue_size=1)
48          self.__twist_sub = rospy.Subscriber(self.__top_data_twist,Twist,self.__twist_cb)
49
50          tmp = commBM.getRosType(Float64())
51          self.__topics[tmp] = self.__top_data_float64
52          self.__pubs[tmp] = rospy.Publisher(self.__top_data_float64,Float64,queue_size=1)
53          self.__float64_sub = rospy.Subscriber(self.__top_data_float64,Float64,self.__float64_cb)
54
55          tmp = commBM.getRosType(Point())
56          self.__topics[tmp] = self.__top_data_point
57          self.__pubs[tmp] = rospy.Publisher(self.__top_data_point,Point,queue_size=1)
58          self.__point_sub = rospy.Subscriber(self.__top_data_point,Point,self.__point_cb)
59
60          tmp = commBM.getRosType(Bool())
61          self.__topics[tmp] = self.__top_data_bool
62          self.__pubs[tmp] = rospy.Publisher(self.__top_data_bool,Bool,queue_size=1)
63          self.__bool_sub = rospy.Subscriber(self.__top_data_bool,Bool,self.__bool_cb)
```

Fig. 14.

```
61          #A dictionary will serve as a receiving data buffer, that must have an
62          #attribute for each data-topic. The data-topics are employed as keys
63          #in the dictionary.
64          self.__buffer = {}
65          self.__buffer[self.__top_data_twist] = Twist()
66          self.__buffer[self.__top_data_float64] = Float64()
67          self.__buffer[self.__top_data_point] = Point()
68          self.__buffer[self.__top_data_bool] = Bool()
```

Fig. 15.

```
15 //...include as many messages your basic-module requires
16 #include <std_msgs/String.h>
17 #include <geometry_msgs/Twist.h>
18 #include <std_msgs/Float64.h>
19 #include <geometry_msgs/Point.h>
20 #include <std_msgs/Bool.h>
```

Fig. 16.

Por último, se debe modificar el archivo **src/utilBM.cpp** de la siguiente manera.

1. Dentro del método constructor de la clase **BMTemplate**, definir el cada **data-topic** y subscriptor declarado en el archivo de cabecera, así como instanciar un publicador para cada **data-topic**, tal como se muestra en la Fig. 19.

2. Dentro del método constructor de la clase **BMTemplate**, almacenar el publicador de cada **data-topic** en el diccionario **data_pubs**, usando como llave

```
52                  //data-topics
53                  std::string top_data_twist;
54                  std::string top_data_float64;
55                  std::string top_data_point;
56                  std::string top_data_bool;
57
58                  //data subs
59                  ros::Subscriber twist_sub;
60                  ros::Subscriber float64_sub;
61                  ros::Subscriber point_sub;
62                  ros::Subscriber bool_sub;
```

Fig. 17.

```
96              void twist_cb(const geometry_msgs::Twist::ConstPtr& msg);
97
98              void float64_cb(const std_msgs::Float64::ConstPtr& msg);
99
100             void point_cb(const geometry_msgs::Point::ConstPtr& msg);
101
102             void bool_cb(const std_msgs::Bool::ConstPtr& msg);
103
104             void desplazar(const geometry_msgs::Twist &param1,const std_msgs::Float64 &param2);
105
106             void posicion(const geometry_msgs::Point &param1);
```

Fig. 18.

el tipo de dato, como se muestra en la Fig. 20.

3. Dentro del método constructor de la clase **BMTemplate**, por cada **data-topic**, crear un elemento en el diccionario **buffer**, usando como llave el **data-topic**, como se muestra en la Fig. 20.

4. En el método **returnOutParams**, dentro del ciclo **for**, por cada **data-topic** agregar un caso **if** para saber a qué tipo se debe convertir cada parámetro de salida (almacenado en el vector **params**), como se muestra en la Fig. 21.

5. El método callback de cada **data-topic** debe almacenar en el diccionario **buffer** (usando como llave el **data-topic** al que está suscrito el callback) el mensaje ROS recibido, como se muestra en la Fig. 22 para el mensaje de tipo **geometry_msgs/Twist**.

6. En el método **sync_cb**, al final del bloque **try**, agregar un caso **if** por cada función del módulo básico, como se muestra en la Fig. 23. Dentro de cada caso se convierten los parámetros de entrada a su tipo de dato correspondiente, usando la función **boost::any_cast**.

7. Dentro de cada método de la clase **BMTemplate** que implementa una función del módulo básico, sus valores de salida y los nombres de sus parámetros de salida se deben almacenar dentro de vectores y pasar al método **returnOutParams** para que los envíe al nodo puente, como se muestra en la Fig. 24 para el método que implementa la función **posicion** de nuestro ejemplo.

```
77        //data-topics
78        top_data_twist = "/"+bm_name+"/twist";
79        top_data_float64 = "/"+bm_name+"/float64";
80        top_data_point = "/"+bm_name+"/point";
81        top_data_bool = "/"+bm_name+"/bool";
82        // ... and as many as the basic module requires
83
84        //data pubs & subs
85        ros::Publisher twist_pub = nh->advertise<geometry_msgs::Twist>(top_data_twist);
86        twist_sub = nh->subscribe<geometry_msgs::Twist>(top_data_twist,1,&BMTemplate::twist_cb,this);
87
88        ros::Publisher float64_pub = nh->advertise<std_msgs::Float64>(top_data_float64);
89        float64_sub = nh->subscribe<std_msgs::Float64>(top_data_float64,1,&BMTemplate::float64_cb,this);
90
91        ros::Publisher point_pub = nh->advertise<geometry_msgs::Point>(top_data_point);
92        point_sub = nh->subscribe<geometry_msgs::Point>(top_data_point,1,&BMTemplate::point_cb,this);
93
94        ros::Publisher bool_pub = nh->advertise<std_msgs::Bool>(top_data_bool,1);
95        bool_sub = nh->subscribe<std_msgs::Bool>(top_data_bool,1,&BMTemplate::bool_cb,this);
96        // ... and as many as the basic module requires
```

Fig. 19.

```
 98        //Store data publishers in the std::map 'data_pubs'
 99        // key = data-type, value = ROS-publisher
100        data_pubs[getRosType(geometry_msgs::Twist())] = twist_pub;
101        data_pubs[getRosType(std_msgs::Float64())] = float64_pub;
102        data_pubs[getRosType(geometry_msgs::Point())] = point_pub;
103        data_pubs[getRosType(std_msgs::Bool())] = bool_pub;
104        // ... and as many as the basic module requires
105
106        //Create an attribute in the buffer for each data-topic
107        // key = data-topic, value = ROS-message object
108        buffer[top_data_twist] = geometry_msgs::Twist();
109        buffer[top_data_float64] = std_msgs::Float64();
110        buffer[top_data_point] = geometry_msgs::Point();
111        buffer[top_data_bool] = std_msgs::Bool();
112        // ... and as many as the basic module requires
```

Fig. 20.

Después de aplicar estas modificaciones se tiene implementada en **C++** la interfaz para del módulo descrito en la Fig. 8.

```
192         //Publish the output-params in their respective publisher
193         vector<string> outparam_topics;
194         for(unsigned int i = 0; i < params.size(); i++)
195         {
196                 string ros_type = getRosType(params[i]);
197                 //Cast the i-th param to its data-type
198                 if(ros_type  == "geometry_msgs/Twist")
199                         data_pubs[ros_type].publish(boost::any_cast<geometry_msgs::Twist>(params[i]));
200                 else if(ros_type  == "std_msgs/Float64")
201                         data_pubs[ros_type].publish(boost::any_cast<std_msgs::Float64>(params[i]));
202                 else if(ros_type  == "geometry_msgs/Point")
203                         data_pubs[ros_type].publish(boost::any_cast<geometry_msgs::Point>(params[i]));
204                 else if(ros_type  == "std_msgs/Bool")
205                         data_pubs[ros_type].publish(boost::any_cast<std_msgs::Bool>(params[i]));
206                 //... and as many cases for the data-types the basic-module has
207                 //Get the topic in which 'params[i]' is published
208                 outparam_topics.push_back(data_pubs[ros_type].getTopic());
209         }
```

Fig. 21.

```
222 void BMTemplate::twist_cb(const geometry_msgs::Twist::ConstPtr& msg)
223 {
224         //Store the received data in the buffer
225         try
226         {
227                 buffer[top_data_twist] = *msg;
228         }
229         catch(std::exception& e)
230         {
231                 return;
232         }
233 }
```

Fig. 22.

```
159                 //Invoke the requested function
160                 if(fun == "desplazar")
161                 {
162                         // invoke 'desplazar'
163                         // cast the input parameters to their corresponding data-type
164                         auto param1 = boost::any_cast<geometry_msgs::Twist>(buffer[inparam_topics[0]]);
165                         auto param2 = boost::any_cast<std_msgs::Float64>(buffer[inparam_topics[1]]);
166                         desplazar(param1,param2);
167                 }
168                 else if(fun == "posicion")
169                 {
170                         //invoke 'posicion'
171                         // cast the input parameters to their corresponding data-type
172                         auto param1 = boost::any_cast<geometry_msgs::Point>(buffer[inparam_topics[0]]);
173                         posicion(param1);
174                 }
175                 //Add a case for each function in the basic module
176         }
177         catch(std::exception& e)
```

Fig. 23.

```
248 void BMTemplate::posicion(const geometry_msgs::Point &param1)
249 {
250         // Do something with the input parameters...
251 
252         //Generate the output value names
253         vector<string> tmp_n;
254         tmp_n.push_back(string("bandera_exito"));
255         tmp_n.push_back(string("posicion_final"));
256 
257         //Store the output values in a vector
258         vector<boost::any> tmp_p;
259         std_msgs::Bool out1(true);
260         geometry_msgs::Point out2(0.1, 1.5, 4.3);
261         tmp_p.push_back(out1);
262         tmp_p.push_back(out2);
263 
264         //Return the function's output-parameters
265         returnOutParams(tmp_n,tmp_p);
266 }
```

Fig. 24.