

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 Markovito¹ del laboratorio de robótica del INAOE. En la sección 3 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 2 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.

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 subtarear. 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 subtarear: 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*.

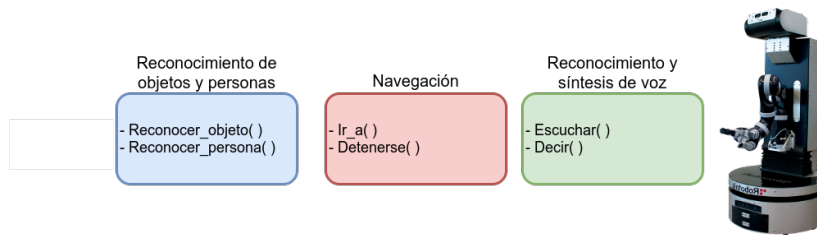


Fig. 1. Ejemplo de un sistema que cuenta con módulos básicos para el reconocimiento de objetos y personas, navegación y reconocimiento de voz. Estos módulos le permiten al robot identificar visualmente cosas, reconocer comando por voz y desplazarse dentro de su entorno.

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 JSON², en un archivo en el que se especifican todos los detalles necesarios para poder

² <https://www.json.org/json-es.html>

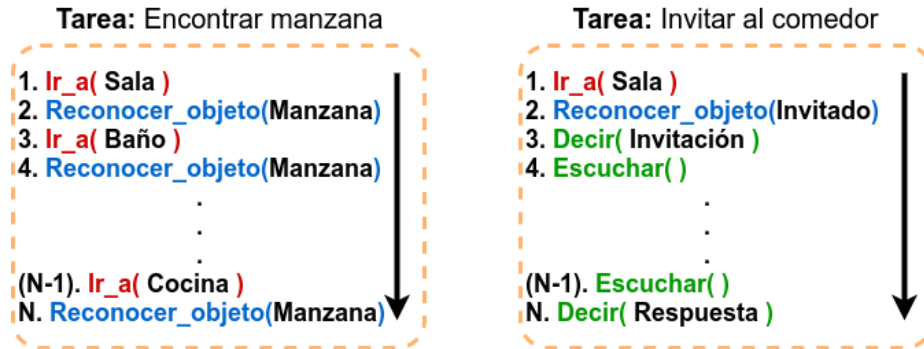


Fig. 2. La mayoría de las tareas que un robot de servicio enfrenta en un hogar requieren de diversas habilidades. Por ejemplo, con los módulos de la figura 1 un robot podría hacer cosas como encontrar una manzana o invitar a alguien a pasar al comedor mediante la ejecución de funciones de diversos módulos.

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.

```

1 {
2   "Basic Module": "",
3   "Brief Description": "",
4   "Hardware Required":
5   [
6     ""
7   ],
8   "Functions":
9   [
10    {
11      "Name": "",
12      "Exec-condition": "",
13      "Description": "",
14      "Input-params":
15      [
16        {
17          "Name": "",
18          "Data-type": "",
19          "Description": ""
20        }
21      ],
22      "Output-params":
23      [
24        {
25          "Name": "",
26          "Data-type": "",
27          "Description": ""
28        }
29      ]
30    }
31  ]
32 }

```

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 ROS³ 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 ROS⁴ 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 suscrito 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 suscribe 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.

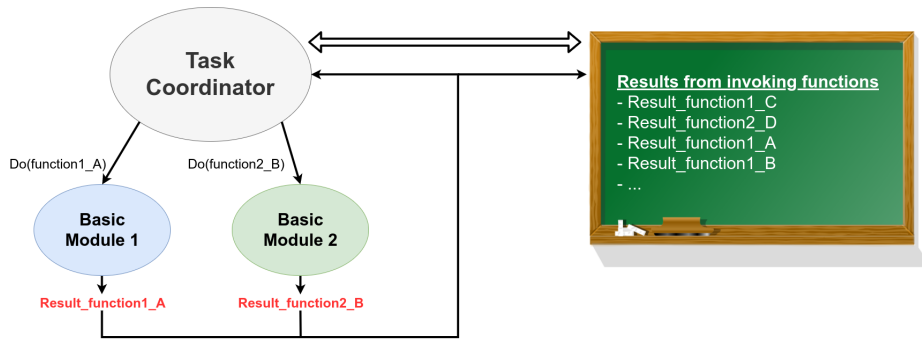


Fig. 4. Arquitectura general de Markovito para solución de tareas. El coordinador de tareas invoca funciones contenidas en los módulos básico. Los resultado de ejecutar una función se envían al coordinador y se almacenan en una memoria (pizarrón). Además el coordinador puede solicitar resultados de invocaciones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

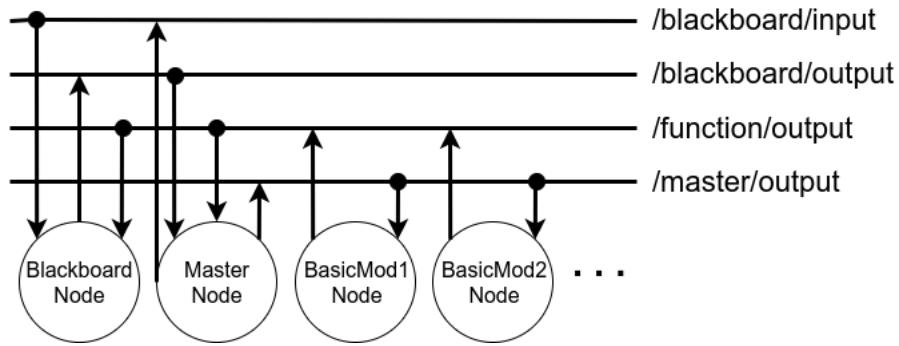


Fig. 5. Esquema general de sistema para la solución de tareas que integra módulos básicos implementado en ROS. El nodo *Blackboard* se encarga de estar guardando los parámetros de salida resultantes de invocar funciones. El nodo *Master* invoca funciones, se queda con algunos de los parámetros de salida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

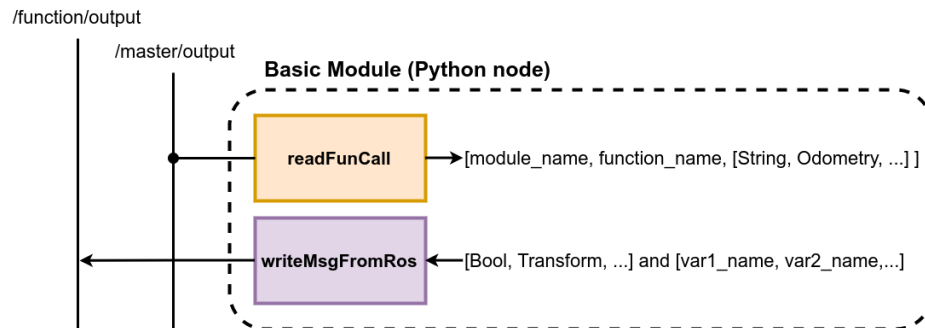


Fig. 6. Para que un nodo de módulo básico se comunique con el sistema, basta con que use las funciones **readFunCall** (para recibir peticiones de invocación de funciones) y **writeMsgFromRos** (para publicar el resultado de una función del módulo como una cadena de caracteres).

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 son 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.

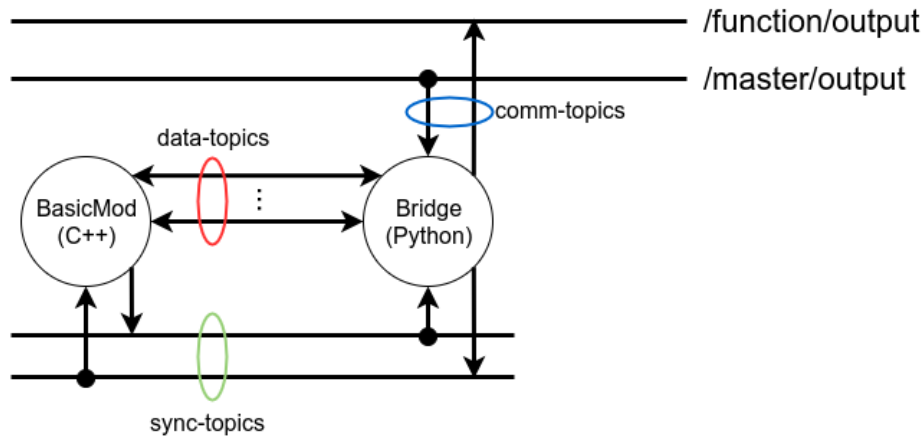


Fig. 7. Un nodo de módulo básico que ha sido implementado en **C++** requiere de un nodo puente (implementado en **Python**) para poder recibir parámetros de entrada por `/master/output` y publicar sus parámetros de salida en `/function/output`. Mediante los tópicos **data-topics** el nodo puente transmite 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.

5 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

Es importante recordar que para que este código funcione, es necesario haber instalado el paquete ROS [rospy_message_converter](#).