Universidad Carlos III de Madrid

Escuela Politécnica Superior

Máster Universitario en Robótica y Automatización

Sistemas Operativos de Robots

Ejercicio práctico con el Framework de ROS

Autor y NIA:

David Yagüe Cuevas 100330413

Índice

Introducción 2

Arquitectura 2

Desarrollo de los nodos 3

Conclusiones 5

Introducción

Este documento explicará de forma breve el desarrollo del ejercicio práctico propuesto en el curso de Sistemas Operativos de Robots del Máster en Robótica y Automatización. Para ello se utilizará el Framework de ROS intentando implementar una arquitectura de software lo más parecida posible a la propuesta. Así, se realizará un análisis descriptivo de las actividades realizadas y las decisiones tomadas durante el proceso.

Para tal cometido, el documento se divide en varias secciones: La descripción de la arquitectura, que es básicamente un análisis rápido del ejercicio propuesto, el desarrollo de los nodos, donde se explica el proceso de implementación y las decisiones tomadas durante el mismo y, finalmente, una parte de conclusiones, para culminar el ejercicio práctico con una valoración sobre la utilidad del ejercicio y las posibilidades dentro del desarrollo propio en conjunción con ROS.

Arquitectura

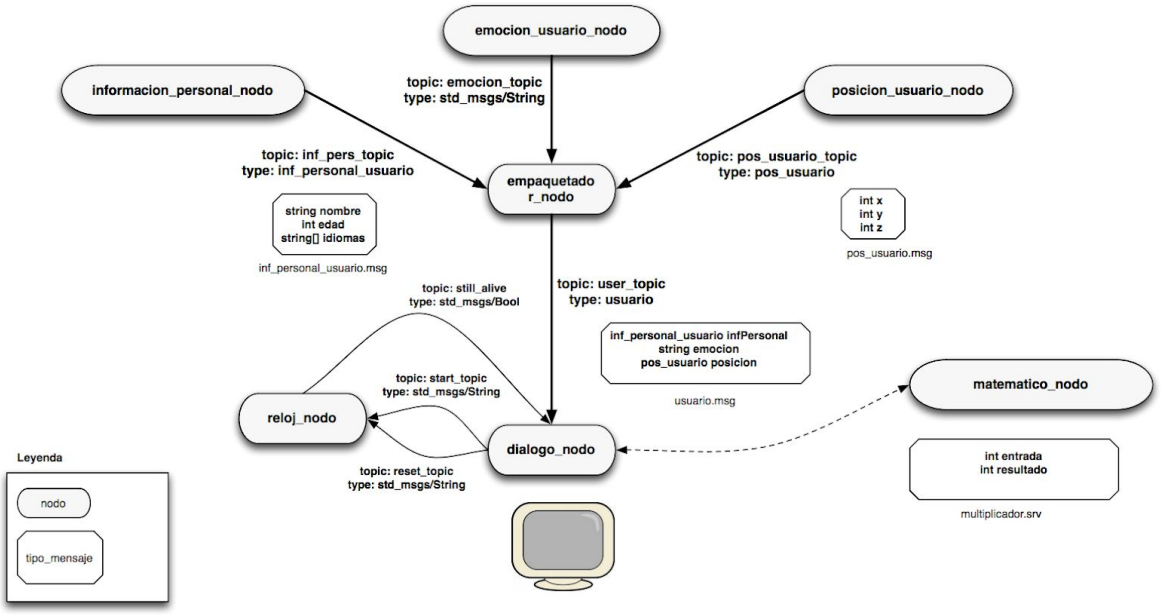

Ilustración 1 - Arquitectura propuesta

La arquitectura que se va a desarrollar es la propuesta en el ejercicio práctico introducido en el curso. Como se puede ver en la Ilustración 1, esta arquitectura cuenta con varios nodos interconectados entre sí que publican mensajes en relación a la información ofrecida por un usuario. Se deberán definir los tipos de los mensajes, ya sean primitivos y/o generados, y los canales de comunicación para cada nodo publicador (topics). Además todas estas directrices, junto con las de compilación se deberán añadir al fichero CMake debido a que se utilizará C++ para el proceso de implementación.

Desarrollo de los nodos

La arquitectura se compone de un total de 7 nodos. Con el objetivo de ser más claro y preciso vamos a dividir su análisis en diferentes secciones:

1. **Entrada del usuario**

   En esta sección se encuentran 3 nodos: *información_personal_nodo*, *posicion_usuario_nodo* y *emocion_usuario_nodo*. Vamos a explicar cada uno brevemente.

   - **emocion_usuario_nodo** → Este nodo se encarga de procesar la entrada del usuario que será la emoción. Se implementa un stringstream que nos proporciona el flujo de caracteres introducidos por el usuario. Una vez obtenida la entrada se publica en el topic */interaccion/emocion_usuario_nodo*. Ej. Feliz.

   - **posicion_usuario_nodo** → Este nodo es el encargado de procesar la posición del usuario. Como en el caso anterior se utiliza un stringstream para almacenar la entrada del usuario, que por motivos de simplificación se ha decidido que sea con el siguiente formato: Entero Entero Entero. Ej. 4 6 8. Una vez obtenida la posición se publica en un topic nuevo: */interaccion/posicion_usuario_topic*.

   - **informacion_personal_nodo** → Este nodo es el más complejo de los tres. Se encarga de procesar la información personal del usuario.Se recoge el nombre, la edad y los idiomas (vector de strings) mediante la lectura del Standard Input. Nótese que para el correcto almacenamiento de los idiomas se ha decidido hacer uso de una variable de control. Los idiomas se van pidiendo en bucle hasta que el usuario introduce “done”. Una vez suministrada toda la información se publica un topic nuevo: */interaccion/informacion_personal_topic*.

   Es importante destacar que todos estos nodos implementan un handler para SIGINT, esto permite que la entrada en terminal no quede bloqueado por el proceso.

2. **Aglomeración de datos**

   En esta sección tenemos al nodo *empaquetador_nodo*. Este nodo es el encargado de encapsular en un solo mensaje de ROS todos los mensajes recibidos de los nodos anteriormente explicados. Para ello se definen unas variables globales y se implementan los respectivos callbacks para cada mensaje publicado por los nodos. Una vez se ha agregado toda la información en el mensaje de tipo *interaccion::usuario* se publica en un nuevo topic */interaccion/user_topic*. Nótese que el nodo tiene un control booleano para impedir publicar un mensaje no completo, de esta manera, solo se empaqueta el mensaje si el usuario ha completado la transacción; esto nos permite también actualizar los mensajes dinámicamente.

3. **Diálogo**

   Esto nodo es el encargado de utilizar el mensaje empaquetado publicado por *empaquetador_nodo* para presentar la información en pantalla. Para ello se subscribe al topic */interaccion/user_topic*, va leyendo cada dato y lo va mostrando. El nodo también implementa

la llamada al servidor para multiplicar la edad del usuario por dos e interacciona con *reloj_nodo* para comprobar que este último sigue vivo. Por otro lado, para implementar esta relación se hace uso de un contador que discrimina el topic en el que se publica, que será un topic dinámico: */interaccion/start_topic* que inicia el reloj o */interaccion/reset_topic*.

Finalmente el nodo hace una llamada al sistema con el comando espeak para realizar la síntesis de voz de un texto, que en este caso notifica el nombre, la emoción y la posición del usuario. Nótese que se ha tenido que utilizar un template para poder realizar la conversión de entero a string; es algo raro porque el compilador que utiliza ROS va acorde al estándar de c++11 pero no es capaz de compilar la directriz *std::to_string(number)*. Además para que el sistema no esté continuamente procesando el texto, se ha añadido una variable booleana que cambia a *True* cuando cada vez que se produce un *Callback* del empaquetador.

4. Servidor

El nodo *matematico_nodo* implementa un servicio, basado en multiplicar por 2 un parámetro. Para ello se implementa la directriz que levanta el servidor y se añade la función *run* que define la acción de multiplicar por 2 un parámetro de entrada (que en este caso será la edad). Cuando el servidor se lanze, el nodo *dialogo_nodo* podrá hacer peticiones; así, cada vez que este nodo saque en pantalla la edad del usuario, esta será multiplicada por 2 llamando al método *run* del servidor.

5. Temporizador

Este nodo se encarga de implementar un temporizador y dar información temporal de distintos sucesos: hora local, UTC, tiempo entre mensajes… Para ello se van a utilizar distintas funcionalidad de ROS, a saber: las *Durations* y los *TimerEvents*. Como el objetivo es informar al nodo *dialogo_nodo* de que el *reloj_nodo* sigue activo, hay que implementar cierta relación entre ellos. El nodo *dialogo_nodo* contiene un contador, mientras que este no llegue a 0, *dialogo_nodo* publicará en el topic */interaccion/start_topic*. Esto le dirá al reloj cuando empezar. Cuando el valor pase de 0 comenzará a publicar en */interaccion/reset_topic*.

Una vez el temporizador se ha iniciado, lo primero que hace es imprimir en pantalla la hora local (y UTC) con el uso de la librería *Boost* de C++. Dado que la idea es que el nodo informe a *dialogo_nodo* si sigue vivo o no, se implementa una nueva clase que da acceso a un *Timer*. Esta nueva clase, *ClockTimer*, no solo nos va a permitir instanciar un *Timer* propio sino que nos da la posibilidad de publicar mensajes desde un *Callback*. Este *Callback* es el encargado de capturar el *Event* del *Timer* (definido cada minuto) y publicar un mensaje en el topic */interaccion/still_alive* que será capturado por el nodo *dialogo_nodo*, el cual informará por pantalla que el reloj sigue vivo.

Finalmente, tenemos la última funcionalidad, que se encarga de imprimir en pantalla el tiempo entre mensajes entre el *reloj_nodo* y el *dialogo_nodo*. Para ello se utiliza una variable de tipo *Time* de ROS en el *Callback* que leerá del topic dinámico publicado por *dialogo_nodo*. Se evalúa el tiempo actual en el nodo *reloj_nodo* y se computa el tiempo entre mensajes con el capturado en el *Callback*. Finalmente se imprime en pantalla.

6. Launcher

Para que el lanzamiento de la arquitectura sea más dinámico y sencillo, se añade un launcher que lanza los nodos necesarios. Este launcher lanza los nodos que se han considerado apropiados, a saber: *matematico_nodo*, *empaquetador_nodo*, *dialogo_nodo* y *reloj_nodo*. Debido a que los nodos restantes hacen peticiones al usuario, se asumirá que estarán ya lanzados manual e individualmente.

Conclusiones

Tras el desarrollo de la arquitectura propuesta podemos comentar ciertos aspectos sobre el ejercicio práctico. Lo cierto es que, en lo que respecta al desarrollo de comportamientos robóticos, ROS acaba siendo un framework bastante útil que promueve el desarrollo modular y ágil. Si bien es cierto que puede ser algo complicado al principio, la curva de aprendizaje se adecúa a las expectativas puestas sobre el framework. En mi caso particular, yo tuve la suerte de ya saber un poco de ROS al haber trabajado con él y el ejercicio propuesto me ha ayudado a refrescar aspectos de los que no me acordaba o a introducir nuevos conocimientos que no poseía. En definitiva, un ejercicio tan práctico como adecuado, muy buena introducción a ROS.

Enlace al repositorio en Github: https://github.com/Weasfas/ROSCourseUC3M