ROS Tutorial 4. ROS Overview: Nodes, Messages and Services

tbocs edited this page on Oct 2, 2011 · 12 revisions

So what is ROS anyway?

As previously said, ROS is a different kind of operating sytsem that serves as a software platorm but is partly based on the "real" operating system (in this case, Ubuntu Linux), so ROS uses the Ubuntu's processes management system, file system, user interface and programming utilities (compiler, threading model, etc.).

But surely ROS does more than what an traditional operating system does. Think about all the software you have made before, especially the ones you write for your CS courses; you seldom run two or more programs simultaneously, or make programs that talk to each other, or even consider other people making new programs and hoping to let their programs take an advantage of yours, right? Concurrency, inter-communication and extensibility, these are the things you don't often do; and you will find it very hard to do by only using the Raw APIs from Operating System directly. However, since robotics requires a lot of these high-level features, it's very important for developers to bring these features to today's programs in a easy, fast as well as unified way.

That's where ROS comes in. Since ROS is based on the current OS as we just said, a ROS program is really nothing essentially different from a traditional program. What I mean is that a program like

```
include <iostream>
using namespace std;
int main()
{
  cout << "hello world.\n";
}
```

will definitely run on ROS. Therefore, instead of completely redefine the "grammar" or "vocabulary" of programming, ROS only adds features to the traditional C++ program. In other words, you can get

- Hierarchical abstraction and management of running programs;
- Communication between programs;
- A collection of powerful programs and code libraries as extension.

without rewriting major code but simply utilizing some classes or function calls. Now let's start by looking at some of the most essential things in ROS artitecture: nodes, messages and services.

Programs as nodes

If programs only run separately, you really don't need to design any abstracted structure for programs, right? But if you want interprocess communication done right and made easy, there has to be a well-designed structure of those program.

As an easy start, let's think about computer programs as all the buildings across the United States. In this analogy, every program has code that does its own thing, just like every building that has people doing their own jobs as well. What do you do when you want people in different buildings communicate with each other? Sending mails, right? You write your own address and the address of the recipient on envelop; that's the very foundation of mailing system. ROS does things just like that, but instead of using street name, room number or zip code, ROS uses URL. So for example, we can have a list of running programs in ROS that looks like

```
/motor/arm/left    # control the left arm based on commands received from oth
/motor/arm/right  # control the right arm based on commands received from oth
/sensor/laser     # send the laser sensor reading to other programs.
/sensor/camera    # send the camera image to other programs.
/control          # use the mathematical model to figure out what to do, and
/control/model    # use filtered sensor data to build mathematical model.
/control/filter   # do optimization and noise filtering on raw sensor data.
```

There are seven programs running; they apparently need each other in order to function. Note that here the URL is quite different because you don't need to have `/parent` running as a program in order to run another program known as `/parent/child` . Note that `/parent` in this context can be two different things: it can be a reference of a running program, or can simply be a description of the location of this program in the abstracted program structure of ROS; and these two sides of the same URL simply do not interfere with each other. In other words, the `/sensor` in `/sensor` and the `/sensor` in `/sensor/laser` mean two different things, as described above.

Now here are some technical terms: a program we run in ROS is no longer called program, but **node**. The URL thing we talked about is called the **name** of the node. If you refer to `/parent` alone, it's a name of the node. If you refer to `/parent` in the context of `/parent/child` , then it's a *namespace* that holds other names. There are three basic relationships of names. A name like `/parent/child` is a **global name**; it means always the same wherever you use it. A name like `child` is called **relative name**, which doesn't have a slash at the beginning; if you refer to `child` in a program called `/parent/another_child` , the `child` name will refer to `/parent/child` , but if you do this in a program called `/another_parent/another_child` , the `child` name will refer to `/another_parent/child` instead. One last type is **private name**, which is always with a prefix "~", basically refer to a name under the current namespace; so saying `~child` in `/parent` means `/parent/child` and saying `~child` in `/parent/child` means `/parent/child/child` .

Still remember the "hello world" code? We have to make it recognized as a part of ROS node system, otherwise it won't have any of the features ROS can provide at all. To do that, we have three lines of code to add:

```
#include <ros/ros.h>

int main(...)
{
  ros::init(...)
  ros::NodeHandle nh;
  ...
}
```

Calling `ros::init()` initializes the whole program in ROS, turning this program into a ROS node. Then a ros::NodeHandle object is created so you can use this object to do

further things with ROS (Just like you cannot work or get a credit card in United States until you get a Social Security). Since some details are omitted here, please do not implement this code yet. We will come back in the near future.

Now, it's time for you to go through this tutorial to do some real experiments. You should understand pretty well how ROS node system looks like as well as knowing the basic commands for viewing/managing nodes. Since right now we only know nodes, please hold your patients about other unfamiliar things on that page and just focus on understanding how nodes work. As you will see, ROS has all the name resolving functionality built-in, so you will have a lot of fun, rather than stress, with that.

Communication as messaging

Ok, now we know how nodes (don't say "programs" anymore) can refer to each other. What about sending messages? Recall the analogy, we give the mail to the post office, and the mailing network will automatically deliver our mail, from one end to the other. However, things are a little bit different in ROS. As a node, you don't really care who gets your message; you only care about having sent them to the "post office", and hope that whoever needs it can get it. This is more like a "P.O. Box" mailing system. There are often a number of boxes, each is meant to contain mails about some particular general themes or keywords (like news categories). A node puts mail into the right box, and others checkout the right box to get mail copies.

So you have already got the idea. Here are some technical terms, again. A "mail" in the world of ROS is called **message**. A "P.O. Box" is called **message topic**. You can name the topics as you like, but it is recommended that you name them in the same way as you name the nodes. Yes, ROS supports URL resolving for message names as well. Just like news categories, messages within the same topic should be about same sort of things. A topic of message must have a definite **message type**, so the ROS can convert them from the data structures to byte streams at sender's end, transport them to the recipient, and convert them back to data structures on that end. You don't want your message of type string to be interpreted as floats, do you?

To receive a certain topic of messages, the node must have an instance of object that is called **ROS Subscriber**; to send a certain topic of messages, the node must have an instance of object that is called **ROS Publisher**. In C++ source code, these are something like

```
// Define a publisher object instance.
ros::Publisher pub = nh.advertise<std_msgs::String>("/topic1", ...); // Don't

// Create and send a message.
std_msgs::String msg("test message");
pub.publish(msg);

// Define a subscriber object instance.
ros::Subscriber sub = nh.subscribe("/topic1", ... , sub_callback);

// Called when the message is received.
void sub_callback (const std_msgs::String::ConstPtr msg)
{
    // Do something with the message.
    cout << "I receive: " << msg->data << " !\n";
}
```

As you can see, it is the creation of the publisher defines the topic name ("/topic1") and

the topic type (std_msgs::String). It is totally okay, however, if you start listening to a topic before any node actually publish to it, because in this case the subscriber will not receive anything, so not knowing the message type of the message actually doesn't matter. So it is totally up to your choice to first start either the node with subscriber or the node with publisher. We will come back to more details later.

You can now play with ROS messaging system by following this tutorial. One interesting I would like to point out that message name and node name are actually related in several aspects. For example, if you create a topic called `data` in node `/parent/child`, the full topic name is `/parent/child/data`. Are these two names really linked tightly "under the hood"? I don't know; I think the node name is just used for generation of the topic name, and that's all. In fact, since I haven't found it possible to change a name of the node or a message topic after they are created, this problem should not bother us. For more information about names, please look at this article as well.

Services and parameters

So far, the messaging system looks really good. But ROS does even more than that. Sometimes you want to let another node do something for you and get result from there, just like calling a local function. If you want to do this with message system, you have to set up two topics, one for input parameters and one for output returning result. Since messages are passed and handled asynchronously, you have to set up a wait loop to check if you have heard back from the other node. These thing are simply troublesome.

Luckily, ROS supports the concept of "remote procedure call" (in contrast to local function call), in the form of `ROS services`. Wit ROS services, calling a function in another node is as easy as calling local functions. Like message, a ROS service also has a **name**, which is used to refer to the service and is preferred to be written in URL format, and a **type**, which defines the data structure of function inputs and outputs. An example would be:

```
// Create a service client, which can handle multiple instances of same servi
ros::ServiceClient client = nh.serviceClient<my_package::MyService>("/MyServi

// Create service instances.
my_package::MyService srv1, srv2;
srv1.request.parameter = ...;
srv2.request.parameter = ...;

// Call the service through client.
client.call(srv1);
client.call(srv2);
cout << srv1.response.result << " " << srv2.response.result << endl;
```

One of the best example about ROS services is the ROS parameter server. Since robotics algorithms always have a lot of parameters to be optimized for specific optimization, it's very important to declare and place them in a good place (rather than randomly scattering variables everywhere in the source file) so that you can always do parameter tuning easily. What's really great about ROS parameter server is that it not only helps you centralize all the parameters used by various nodes, but also enables you to tune the parameters without recompiling or even live (without restarting the node). To take a real look at ROS services and parameter server, please go through this tutorial. Make sure you are not just moving your eyeballs, but really get your hands dirty!