A Gentle Introduction to ROS

Chapter: Graph resource names

Jason M. O'Kane

Jason M. O'Kane
University of South Carolina
Department of Computer Science and Engineering
315 Main Street
Columbia, SC 29208

http://www.cse.sc.edu/~jokane

This is version 2.1.5 (5378703), generated on April 12, 2018.

Typeset by the author using LaTeX and memoir.cls.

ISBN 978-14-92143-23-9

Chapter 5

Graph resource names

In which we learn how ROS resolves the names of nodes, topics, parameters, and services.

In Chapter 3, we used strings like "hello_ros" and "publish_velocity" to give names to nodes, and strings like "turtle1/cmd_vel" and "turtle1/pose" as the names of topics. All of these are examples of **graph resource names**. ROS has a flexible naming system that accepts several different kinds of names. (These four, for example, are all **relative names**.) In this chapter, we'll take a short detour to understand the various kinds of graph resource names, and how ROS resolves them. We present these ideas, which are actually quite simple, as a separate chapter because they're relevant to most of the concepts in the second half of this book.

5.1 Global names

Nodes, topics, services, and parameters are collectively referred to as **graph resources**. Every graph resource is identified by a short string called a **graph resource name**.[1] Graph resource names are ubiquitous, both in ROS command lines and in code. Both rosnode info and ros::init expect node names; both rostopic echo and the constructor for ros::Publisher expect topic names. All of these are instances of graph resource names. Here are some specific graph resource names that we've encountered already:

```
/teleop_turtle
/turtlesim
```

[1] http://wiki.ros.org/Names

```
/turtle1/cmd_vel
/turtle1/pose
/run_id
/count_and_log/set_logger_level
```

These names are all examples of a specific class of names called **global names**. They're called global names because they make sense anywhere they're used. These names have clear, unambiguous meanings, whether they're used as arguments to one of the many command line tools or inside a node. No additional context information is needed to decide which resource the name refers to.

There are several parts to a global name:

- A leading slash `/`, which identifies the name as a global name.
- A sequence of zero or more **namespaces**, separated by slashes. Namespaces are used to group related graph resources together. The example names above include two explicit namespaces, called `turtle1` and `count_and_log`. Multiple levels of namespaces are allowed, so this is also a valid (but rather unlikely) global name, consisting of 11 nested namespaces:

  ```
  /a/b/c/d/e/f/g/h/i/j/k/l
  ```

  Global names that don't explicitly mention any namespace—including three of the examples above—are said to be in the **global namespace**.
- A **base name** that describes the resource itself. The base names in the example above are `teleop_turtle`, `turtlesim`, `cmd_vel`, `pose`, `run_id`, and `set_logger_level`.

Notice that, if global names were required everywhere, then there would be little to gain from the complexity of using namespaces, other than perhaps making it easier for humans to keep track of things. The real advantage of this naming system comes from the use of relative names and private names.

5.2 Relative names

The main alternative to providing a global name, which—as we've just seen—includes a complete specification of the namespace in which the name lives, is to allow ROS to supply a default namespace. A name that uses this feature is called a **relative graph resource name**, or simply a **relative name**. The characteristic feature of a relative name is that it lacks a leading slash (`/`). Here are some example relative names:

```
teleop_turtle
turtlesim
cmd_vel
turtle1/pose
run_id
count_and_log/set_logger_level
```

The key to understanding relative names is to remember that relative names cannot be matched to specific graph resources unless we know the default namespace that ROS is using to resolve them.

Resolving relative names The process of mapping relative names to global names is actually quite simple. To resolve a relative name to a global name, ROS attaches the name of the current default namespace to the front of the relative name. For example, if we use the relative name `cmd_vel` in a place where the default namespace is `/turtle1`, then ROS resolves the name by combining the two:

$$\underbrace{\texttt{/turtle1}}_{\substack{\text{default}\\\text{namespace}}} + \underbrace{\texttt{cmd\_vel}}_{\text{relative name}} \Rightarrow \underbrace{\texttt{/turtle1/cmd\_vel}}_{\text{global name}}$$

Relative names can also begin with a sequence of namespaces, which are treated as nested namespaces inside the default namespace. As an extreme example, if we use the relative name `g/h/i/j/k/l` in a place where the default namespace is `/a/b/c/d/e/f`, ROS performs this combination:

$$\underbrace{\texttt{/a/b/c/d/e/f}}_{\substack{\text{default}\\\text{namespace}}} + \underbrace{\texttt{g/h/i/j/k/l}}_{\text{relative name}} \Rightarrow \underbrace{\texttt{/a/b/c/d/e/f/g/h/i/j/k/l}}_{\text{global name}}$$

The resulting global name is then used to identify a specific graph resource, just as though a global name had been specified originally.

Setting the default namespace This default namespace is tracked individually for each node, rather than being a system-wide setting. If you don't take any specific steps to set the default namespace, then ROS will, as you might expect, use the global namespace (`/`). The best and most common method for choosing a different default namespace for a node or group of nodes is to use `ns` attributes in a launch file. (See Section 6.3.) However, there are also a couple of mechanisms for doing this manually.

☞ Most ROS programs, including all C++ programs that call `ros::init`, accept a command line parameter called `__ns`, which specifies a default namespace for that program.

`__ns:=`*default-namespace*

☞ You can also set the default namespace for every ROS program executed within a shell, using an environment variable.

`export ROS_NAMESPACE=`*default-namespace*

This environment variable is used only when no other default namespace is specified by the `__ns` parameter.

Understanding the purpose of relative names Aside from the question how to determine the default namespace used for relative names, one other likely question is "Who cares?" At first glance, the concept of relative names appears to be just a shortcut to avoid typing the full global names every time. Although relative names do provide this kind of convenience, their real value is that they make it easier to build complicated systems by composing smaller parts.

When a node uses relative names, it is essentially giving its users the ability to easily push that node and the topics it uses down into a namespace that the node's original designers did not necessarily anticipate. This kind of flexibility can make the organization of a system more clear and, more importantly, can prevent name collisions when groups of nodes from different sources are combined. In contrast, every explicit global name makes it harder to achieve this kind of composition. Therefore, when writing nodes, it's recommended to avoid using global names, except in the unusual situations where there is a very good reason to use them.

5.3 Private names

Private names, which begin with a tilde (~) character, are the third and final class of graph resource names. Like relative names, private names do not fully specify the namespace in which they live, and instead rely on the ROS client library to resolve the name to a complete global name. The difference is that, instead of using the current default namespace, private names *use the name of their node as a namespace*.

For instance, in a node whose global name is `/sim1/pubvel`, the private name `~max_vel` would be converted to a global name like this:

$$\underbrace{\texttt{/sim1/pubvel}}_{\text{node name}} + \underbrace{\texttt{\~{}max\_vel}}_{\text{private name}} \Rightarrow \underbrace{\texttt{/sim1/pubvel/max\_vel}}_{\text{global name}}$$

The intuition is that each node has its own namespace for things that are related only to that node, and are not interesting to anyone else. Private names are often used for parameters—`roslaunch` has a specific feature for setting parameters that are accessible by private names; see page 113—and services that govern the operation of a node. It is usually a mistake to use a private name to refer to a topic because, if we're keeping our nodes loosely coupled, no topic is "owned" by any particular node.

Private names are private only in the sense that they are resolved into a namespace that is unlikely to be used by any other nodes. Graph resources referred to by private names remain accessible, via their global names, to any node that knows their name. This is a contrast, for example, to the `private` *keyword in C++ and similar languages, which prevents other parts of a system from accessing certain class members.*

5.4 Anonymous names

In addition to these three primary types of names, ROS provides one more naming mechanism called **anonymous names**, which are specifically used to name nodes. The purpose of an anonymous name is to make it easier to obey the rule that each node must have a unique name. The idea is that a node can, during its call to `ros::init`, request that a unique name be assigned automatically.

To request an anonymous name, a node should pass `ros::init_options::AnonymousName` as a fourth parameter to `ros::init`:

```
ros::init(argc, argv, base_name, ros::init_options::AnonymousName);
```

The effect of this extra option is to append some extra text to the given base name, ensuring that the node's name is unique.

▶▶| *Although the details of what specific extra text is added are not particularly important, it is interesting to note that* `ros::init` *uses the current wall clock time to form anonymous names.*

```
1  // This program starts with an anonymous name, which
2  // allows multiple copies to execute at the same time,
3  // without needing to manually create distinct names
4  // for each of them.
5  #include <ros/ros.h>
6
7  int main(int argc, char **argv) {
8    ros::init(argc, argv, "anon",
9      ros::init_options::AnonymousName);
10   ros::NodeHandle nh;
11   ros::Rate rate(1);
12   while(ros::ok()) {
13     ROS_INFO_STREAM("This␣message␣is␣from␣"
14       << ros::this_node::getName());
15     rate.sleep();
16   }
17 }
```

Listing 5.1: A program called anon.cpp whose nodes have anonymous names. We can start as many simultaneous copies of this program as we like, without any node name conflicts.

Listing 5.1 shows a sample program that uses this feature. Instead of simply being named anon, nodes started from this program get names that look like this:

```
/anon_1376942789079547655
/anon_1376942789079550387
/anon_1376942789080356882
```

The program's behavior is quite unremarkable, but because it requests an anonymous name, we are free to run as many simultaneous copies of that program as we like, knowing that each will be assigned a unique name when it starts.

5.5 Looking forward

In this chapter, we learned about how ROS interprets the names of graph resources. In particular, non-trivial ROS systems with many interacting nodes can benefit from the flexibility arising from using relative or private names. The next chapter introduces a tool called roslaunch that simplifies the process of starting and configuring these kinds of multi-node ROS sessions.