A Gentle Introduction to ROS

Chapter: Parameters

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

©2014, Jason Matthew O'Kane. All rights reserved.

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 7

Parameters

In which we configure nodes using parameters.

In addition to the messages that we've studied so far, ROS provides another mechanism called **parameters** to get information to nodes. The idea is that a centralized **parameter server** keeps track of a collection of values—things like integers, floating point numbers, strings, or other data—each identified by a short string name.[1][2] Because parameters must be actively queried by the nodes that are interested in their values, they are most suitable for configuration information that will not change (much) over time.

This chapter introduces parameters, showing how to access them from the command line, from within nodes, and in launch files.

7.1 Accessing parameters from the command line

Let's start with a few commands to see how parameters work.

Listing parameters To see a list of all existing parameters, use this command:[3]

```
rosparam list
```

On the author's system, with no nodes running, the output is:

[1] http://wiki.ros.org/roscpp/Overview/Parameter Server
[2] http://wiki.ros.org/Parameter Server
[3] http://wiki.ros.org/rosparam

```
/rosdistro
/roslaunch/uris/host_donatello__38217
/rosversion
/run_id
```

Each of these strings is a name—specifically, a global graph resource name (see Chapter 5)—that the parameter server has associated with some value.

⏭ *In the current version of ROS, the parameter server is actually part of the master, so it is started automatically by* `roscore` *or* `roslaunch`*. In nearly all cases, the parameter server works correctly behind the scenes, and there's no reason to think about it explicitly. Keep in mind however, that all parameters are "owned" by the parameter server rather than by any particular node. This means that parameters—even those created with private names—will continue to exist even after the node they're intended for has terminated.*

Querying parameters To ask the parameter server for the value of a parameter, use the `rosparam get` command:

```
rosparam get parameter_name
```

For example, to read the value of the `/rosdistro` parameter, use this command:

```
rosparam get /rosdistro
```

The output is the string `indigo`, which is not too much of a surprise. It is also possible to retrieve the values of every parameter in a namespace:

```
rosparam get namespace
```

For example, by asking about the global namespace, we can see the values of every parameter all at once:

```
rosparam get /
```

On the author's computer, the output is:

```
rosdistro: indigo
roslaunch:
    uris: host_donatello__38217: 'http://donatello:38217/'
rosversion: 1.11.9
run_id: e574a908-70c5-11e4-899e-60d819d10251
```

Setting parameters To assign a value to a parameter, use a command like this:

```
rosparam set parameter_name parameter_value
```

This command can modify the values of existing parameters or create new ones. For example, these commands create string parameters that store the wardrobe preferences of a certain group of cartoon ducks:

```
rosparam set /duck_colors/huey red
rosparam set /duck_colors/dewey blue
rosparam set /duck_colors/louie green
rosparam set /duck_colors/webby pink
```

⏭ *Alternatively, we can set several parameters in the same namespace at once:*

```
rosparam set namespace values
```

The values should be specified as a YAML dictionary that maps parameter names to values. Here's an example that has the same effect as the four commands above:

```
rosparam set /duck_colors "huey: red
dewey: blue
louie: green
webby: pink"
```

Note that this syntax requires newline characters in the command itself. This is not a problem because the opening quotation mark signals to `bash` *that the command is not yet complete. Pressing* `Enter` *when a quoted argument is open inserts a newline character, as desired, instead of executing the command.*

The spaces after the colons are important, to ensure that `rosparam` *treats this as a collection of parameters in the* `/duck_colors` *namespace, rather than as a single string parameter called* `duck_colors` *in the global namespace.*

Creating and loading parameter files To store all of the parameters from a namespace, in YAML format, to a file, use `rosparam dump`:

```
rosparam dump filename namespace
```

The opposite of `dump` is `load`, which reads parameters from a file and adds them to the parameter server:

```
rosparam load filename namespace
```

For both of these commands, the namespace argument is optional, and defaults to the global namespace (`/`). The combination of `dump` and `load` can be useful for testing, because it provides a quick way to take a "snapshot" of the parameters in effect at a certain time, and to recreate that scenario later.

7.2 Example: Parameters in turtlesim

For a more concrete example of how parameters can be useful, let's see how `turtlesim` uses them. If you start `roscore` and a `turtlesim_node`, and then ask for a `rosparam list`, you'll see output like this:

```
/background_b
/background_g
/background_r
/rosdistro
/roslaunch/uris/host_donatello__59636
/rosversion
/run_id
```

We've already seen those last four parameters, which are created by the master. In addition, it looks like our `turtlesim_node` has created three parameters. Their names (correctly) suggest that they specify the background color that `turtlesim` is using, separated into red, green, and blue channels.

This illustrates that nodes can, and sometimes do, create and modify parameter values. In this case, `turtlesim_node` sets those three parameters as part of its initialization. In this regard, `turtlesim_node` is atypical, because its initialization will clobber any values that might already have been set for those parameters. That is, every `turtlesim_node` starts with the same blue background, at least for a short time, regardless of any steps we might take to specify a different starting color.

A better strategy—and a better example of how "real" ROS nodes usually work—might have been for `turtlesim` *to first to test whether those parameters exist, and assign the default blue color only if those parameters do not already exist.*

Reading the background color We can inspect the values of the background parameters using `rosparam get`:

```
rosparam get /background_r
rosparam get /background_g
rosparam get /background_b
```

The values returned by these commands are 69, 86, and 255. Since the values are relatively small integers, a good guess (and, it turns out, a correct guess) is that each channel is an 8-bit integer, ranging from 0 to 255. Thus, `turtlesim` defaults to a background color of (69,86,255), corresponding to the deep blue color to which we're accustomed.

Setting the background color Suppose we want to change the background from this blue color to a bright yellow instead. We might try to do this by changing the parameter values after the `turtlesim` node starts up:

```
rosparam set /background_r 255
rosparam set /background_g 255
rosparam set /background_b 0
```

However, even after setting these parameters, the background color remains the same. Why? The explanation is that `turtlesim_node` only reads the values of these parameters when its `/clear` service is called. One way to call this service is using this command:

```
rosservice call /clear
```

After the service call completes, the background color will finally be changed appropriately. Figure 7.1 shows the effect of this change.

The important thing to notice here is that updated parameter values are not automatically "pushed" to nodes. Instead, nodes that care about changes to some or all of their parameters must explicitly ask the parameter server for those values. Likewise, if we expect to change the values of parameters used by an active node, we must be aware of how (or if) that node re-queries its parameters. (Quite often, but not for `turtlesim`, the answer is based on a subsystem called `dynamic_reconfigure`, which we do not cover here.[4])

[4] `http://wiki.ros.org/dynamic_reconfigure`

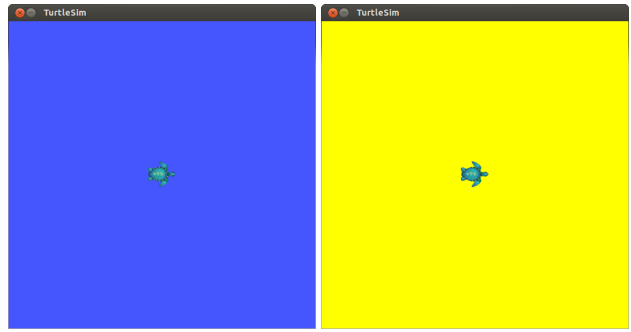

Figure 7.1: Before (left) and after (right) a change to the background color of a turtlesim node.

⏭ *It is possible for a node to ask the parameter server to send updated values when a parameter changes, by using* ros::param::getCached *instead of* ros::param::get. *However, this approach is only intended to improve efficiency, and doesn't eliminate the need for the node to check the parameter's value.*

7.3 Accessing parameters from C++

The C++ interface to ROS parameters is quite straightforward:[5]

```
void ros::param::set(parameter_name, input_value);
bool ros::param::get(parameter_name, output_value);
```

In both cases, the parameter name is a string, which can be a global, relative, or private name. The input value for set can be a std::string, a bool, an int, or a double; the output value for get should be a variable (which is passed by reference) of one of those types. The get function returns true if the value was read successfully and false if there was a problem, usually indicating that the requested parameter has not been assigned a value. To see these functions in action, let's have a look at two examples.

☞ Listing 7.1 illustrates ros::param::set. It assigns integer values to all three turtlesim background color parameters. This program includes code to ensure that the turtlesim node has started up by waiting for the /clear service that it provides—necessary to ensure that turtlesim does not overwrite the values that we set here—and to call that service to force turtlesim to read the new values that we've set. (Our focus here is on the parameters themselves; See Chapter 8 for details about services.)

[5] http://wiki.ros.org/roscpp_tutorials/Tutorials/Parameters

```
1  // This program waits for a turtlesim to start up, and
2  // changes its background color.
3  #include <ros/ros.h>
4  #include <std_srvs/Empty.h>
5
6  int main(int argc, char **argv) {
7    ros::init(argc, argv, "set_bg_color");
8    ros::NodeHandle nh;
9
10   // Wait until the clear service is available, which
11   // indicates that turtlesim has started up, and has
12   // set the background color parameters.
13   ros::service::waitForService("clear");
14
15   // Set the background color for turtlesim,
16   // overriding the default blue color.
17   ros::param::set("background_r", 255);
18   ros::param::set("background_g", 255);
19   ros::param::set("background_b", 0);
20
21   // Get turtlesim to pick up the new parameter values.
22   ros::ServiceClient clearClient
23     = nh.serviceClient<std_srvs::Empty>("/clear");
24   std_srvs::Empty srv;
25   clearClient.call(srv);
26
27 }
```

Listing 7.1: A C++ program called set_bg_color.cpp that sets the background color of a turtlesim window.

☞ Listing 7.2 shows an example of ros::param::get. It extends our original pubvel example (Listing 3.4) by reading a private floating point parameter called max_vel and using that value to scale the randomly-generated linear velocities.

This program requires a value for a parameter called max_vel in its private namespace, which must be set before the program starts:

```
rosparam set /publish_velocity/max_vel 0.1
```

If that parameter is not available, the program generates a fatal error and terminates.

```
1  // This program publishes random velocity commands, using
2  // a maximum linear velocity read from a parameter.
3  #include <ros/ros.h>
4  #include <geometry_msgs/Twist.h>
5  #include <stdlib.h>
6
7  int main(int argc, char **argv) {
8    ros::init(argc, argv, "publish_velocity");
9    ros::NodeHandle nh;
10   ros::Publisher pub = nh.advertise<geometry_msgs::Twist>(
11       "turtle1/cmd_vel", 1000);
12   srand(time(0));
13
14   // Get the maximum velocity parameter.
15   const std::string PARAM_NAME = "~max_vel";
16   double maxVel;
17   bool ok = ros::param::get(PARAM_NAME, maxVel);
18   if(!ok) {
19     ROS_FATAL_STREAM("Could not get parameter "
20       << PARAM_NAME);
21     exit(1);
22   }
23
24   ros::Rate rate(2);
25   while(ros::ok()) {
26     // Create and send a random velocity command.
27     geometry_msgs::Twist msg;
28     msg.linear.x = maxVel*double(rand())/double(RAND_MAX);
29     msg.angular.z = 2*double(rand())/double(RAND_MAX)-1;
30     pub.publish(msg);
31
32     // Wait until it's time for another iteration.
33     rate.sleep();
34   }
35 }
```

Listing 7.2: A C++ program called pubvel_with_max.cpp that extends the original pubvel.cpp by reading its maximum linear velocity from a parameter.

⏭ *It is technically possible (but somewhat messy) to assign a private parameter to a node on its command line using a remap-like syntax, by prepending the name with an underscore (_):*

_param-name:=param-value

These kinds of arguments are converted to ros::param::set *calls, replacing the _ with a ~ to form a proper private name, by* ros::init*. For example, we could successfully launch* pubvel_with_max *using this command:*

```
rosrun agitr pubvel_with_max    _max_vel:=1
```

7.4 Setting parameters in launch files

Another very common method for setting parameters is to do so within a launch file.

Setting parameters To ask roslaunch to set a parameter value, use a param element:[16]

```
<param name="param-name" value="param-value" />
```

This element, as one would expect, assigns the given value to parameter with the given name. The parameter name should, as usual, be a relative name. For example, this launch file fragment is equivalent to the rosparam set commands on page 107:

```
<group ns="duck_colors">
   <param name="huey" value="red" />
   <param name="dewey" value="blue" />
   <param name="louie" value="green" />
   <param name="webby" value="pink" />
</group>
```

Setting private parameters Another option is to include param elements as children of a node element.

[16] http://wiki.ros.org/roslaunch/XML/param

```
<node ...>
    <param name="param-name" value="param-value" />
        ...
</node>
```

With this construction, the parameter names are treated as private names for that node.

> ⏭ *This is an exception to the usual rules for resolving names. Parameter names given in* param *elements that are children of* node *elements are* always *resolved as private names, regardless of whether they begin with ~ or even /.*

For example, we might use code like this to launch our pubvel_with_max node with its private max_vel parameter set correctly:

```
<node
    pkg="agitr"
    type="pubvel_with_max"
    name="publish_velocity"
/>
    <param name="max_vel" value="3" />
</node>
```

Listing 7.3 shows a complete launch file that launches a turtlesim and our two example programs. Its result should be to show a turtlesim turtle moving quickly across a yellow background.

Reading parameters from a file Finally, launch files also support an equivalent to rosparam load, to set many parameters at once.[7]

```
<rosparam command="load" file="path-to-param-file" />
```

The parameter file listed here is usually one created by rosparam dump. As with other references to specific files (such as the include element from Section 6.5.1) it is typical to use a find substitution to specify the file name relative to a package directory:

```
<rosparam
    command="load"
    file="$(find package-name)/param-file"
/>
```

[7] http://wiki.ros.org/roslaunch/XML/rosparam

```
<launch>
  <node
    pkg="turtlesim"
    type="turtlesim_node"
    name="turtlesim"
  />
  <node
    pkg="agitr"
    type="pubvel_with_max"
    name="publish_velocity"
  >
    <param name="max_vel" value="3" />
  </node>
  <node
    pkg="agitr"
    type="set_bg_color"
    name="set_bg_color"
  />
</launch>
```

Listing 7.3: A launch file called `fast_yellow.launch`. It starts the example programs from Listings 7.1 and 7.2 and sets the `max_vel` parameter.

Along with `rosparam load`, this facility can be helpful for testing, because it allows us to recreate the parameters that were in effect at a certain time in the past.

7.5 Looking forward

Parameters are a relatively simple idea that can lead to substantial flexibility and configurability in ROS nodes. The next chapter deals with one final communication mechanism, called services, which implements one-to-one, bidirectional flows of information.