

A Gentle Introduction to ROS

Chapter: Launch files

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 2.0.3 (de33d94) , generated on August 25, 2014.

Typeset by the author using \LaTeX and `memoir.cls`.

ISBN 978-14-92143-23-9

Chapter 6

Launch files

In which we configure and run many nodes at once using launch files.

If you've worked through all of the examples so far, by now you might be getting frustrated by the need to start so many different nodes, not to mention `roscore`, by hand in so many different terminals. Fortunately, ROS provides a mechanism for starting the master and many nodes all at once, using a file called a **launch file**. The use of launch files is widespread through many ROS packages. Any system that uses more than one or two nodes is likely to take advantage of launch files to specify and configure the nodes to be used. This chapter introduces these files and the `roslaunch` tool that uses them.

6.1 Using launch files

Let's start by seeing how `roslaunch` enables us to start many nodes at once. The basic idea is to list, in a specific XML format, a group of nodes that should be started at the same time.¹ Listing 6.1 shows a small example launch file that starts a `turtlesim` simulator, along with the teleoperation node that we saw in Chapter 2 and the subscriber node we wrote in Chapter 3. This file is saved as `example.launch` in the main package directory for the `agitr` package. Before we delve into specifics of the launch file format, let's see how those files can be used.

Executing launch files To execute a launch file, use the `roslaunch` command:²

¹<http://wiki.ros.org/roslaunch/XML>

²[http://wiki.ros.org/roslaunch/CommandLine Tools](http://wiki.ros.org/roslaunch/CommandLine%20Tools)

```
1 <launch>
2   <node
3     pkg="turtlesim "
4     type="turtlesim_node "
5     name="turtlesim "
6     respawn="true "
7   />
8   <node
9     pkg="turtlesim "
10    type="turtle_teleop_key "
11    name="teleop_key "
12    required="true "
13    launch-prefix="xterm -e "
14  />
15  <node
16    pkg="agitr "
17    type="subpose "
18    name="pose_subscriber "
19    output="screen "
20  />
21 </launch>
```

Listing 6.1: A launch file called `example.launch` that starts three nodes at once.

`roslaunch package-name launch-file-name`

You can invoke the example launch file using this command:

`roslaunch agitr example.launch`

If everything works correctly, this command will start three nodes. You should get `turtlesim` window, along with another window that accepts arrow key presses for teleoperating the turtle. The original terminal in which you ran the `roslaunch` command should show the pose information logged by our `subpose` program. Before starting any nodes, `roslaunch` will determine whether `roscore` is already running and, if not, start it automatically.



Be careful not to confuse `roslaunch`, which starts a single node, with `roslaunch`, which can start many nodes at once.

►► It is also possible to use launch files that are not part of any package. To do this, give `roslaunch` only the path to the launch file, without mentioning any package. For example, in the author's account, this command starts our example launch file, without relying on the fact that it's a part of any ROS package:

```
roslaunch ~/ros/src/agitr/example.launch
```

This sort of workaround to circumvent the usual package organization is probably not a good idea for anything but very simple, very short-lived experimentation.

An important fact about `roslaunch`—one that can be easy to forget—is that all of the nodes in a launch file are started at roughly the same time. As a result, you cannot be sure about the order in which the nodes will initialize themselves. Well-written ROS nodes don't care about the order in which they and their siblings start up. (Section 7.3 has an example program in which this becomes important.)

►► This behavior is a reflection of the ROS philosophy that each node should be largely independent of the other nodes. (Recall our discussion of loose coupling of nodes from Section 2.8.) Nodes that function well only when launched in a specific order are a poor fit for this modular design. Such nodes can almost always be redesigned to avoid ordering constraints.

Requesting verbosity Like many command line tools, `roslaunch` has an option to request verbose output:

```
roslaunch -v package-name launch-file-name
```

Listing 6.2 shows an example of the information this option generates beyond the usual status messages. It can occasionally be useful for debugging to see this detailed explanation of how `roslaunch` is interpreting your launch file.

Ending a launched session To terminate an active `roslaunch`, use `Ctrl-C`. This signal will attempt to gracefully shut down each active node from the launch, and will forcefully kill any nodes that do not exit within a short time after that.

```
1 ... loading XML file [/opt/ros/hydro/etc/ros/roscore.xml]
2 ... executing command param [rosversion roslaunch]
3 Added parameter [/rosversion]
4 ... executing command param [rosversion -d]
5 Added parameter [/rostdistro]
6 Added core node of type [rosout/rosout] in namespace [/]
7 ... loading XML file [/home/jokane/ros/agitr/example.launch]
8 Added node of type [turtlesim/turtlesim_node] in namespace [/]
9 Added node of type [agitr/pubvel] in namespace [/]
10 Added node of type [agitr/subpose] in namespace [/]
```

Listing 6.2: Extra output generated by the verbose mode of roslaunch.

6.2 Creating launch files

Having seen how launch files can be used, we're ready now to think about how to create them for ourselves.

6.2.1 Where to place launch files

As with all other ROS files, each launch file should be associated with a particular package. The usual naming scheme is to give launch files names ending with `.launch`. The simplest place to store launch files is directly in the package directory. When looking for launch files, roslaunch will also search subdirectories of each package directory. Some packages, including many of the core ROS packages, utilize this feature by organizing launch files into a subdirectory of their own, usually called `launch`.

6.2.2 Basic ingredients

The simplest launch files consist of a root element containing several node elements.

Inserting the root element Launch files are XML documents, and every XML document must have exactly one **root element**. For ROS launch files, the root element is defined by a pair of launch tags:

```
<launch>
...
</launch>
```

All of the other elements of each launch file should be enclosed between these tags.

Launching nodes The heart of any launch file is a collection of node elements, each of which names a single node to launch.³ A node element looks like this:

```
<node
  pkg="package-name"
  type="executable-name"
  name="node-name"
/>
```



The trailing slash near the end of the node tag is both important and easy to forget. It indicates that no closing tag (“</node>”) is coming, and that the node element is complete. XML parsers are required to be very strict about this sort of thing. If you omit this slash, be prepared for errors like this:



Invalid roslaunch XML syntax: mismatched tag

You can also write the closing tag explicitly:

```
<node pkg="..." type="..." name="..."></node>
```

In fact, this explicit closing tag is needed if the node has children, such as remap or param elements. These elements are introduced in Section 6.4 and 7.4, respectively.

A node element has three required attributes:

-  The `pkg` and `type` attributes identify which program ROS should run to start this node. These are the same as the two command line arguments to `roslaunch`, specifying the package name and the executable name, respectively.
-  The `name` attribute assigns a name to the node. This overrides any name that the node would normally assign to itself in its call to `ros::init`.

►► This override fully clobbers the naming information provided to `ros::init`, including any request that the node might have made for an anonymous name.

³<http://wiki.ros.org/roslaunch/XML/node>

(See Section 5.4.) To use an anonymous name from within a launch file, use an anon substitution⁴ for the name attribute, like this:

```
name="$(anon base_name)"
```

Note, however, that multiple uses of the same base name will generate the same anonymous name. This means that (a) we can refer to that name in other parts of the launch file, but (b) we must be careful to use different base names for each node we want to anonymize.

Finding node log files An important difference between `roslaunch` and running each node individually using `roslaunch` is that, by default, standard output from launched nodes is redirected to a log file, and does not appear on the console.¹ The name of this log file is:

```
~/ros/log/run_id/node_name-number-stdout.log
```

The `run_id` is a unique identifier generated when the master is started. (See page 70 for details about how to look up the current `run_id`.) The numbers in these file names are small integers that number the nodes. For example, running the launch file in Listing 6.1 sends the standard output of two of its nodes to log files with these names:

```
turtlesim-1-stdout.log  
teleop_key-3-stdout.log
```

These log files can be viewed with the text editor of your choice.

Directing output to the console To override this behavior for a single node, use the `output` attribute in its node element:

```
output="screen"
```

Nodes launched with this attribute will display their standard output on screen instead of in the log files discussed above. The example uses this attribute for the `subpose` node, which explains why the `INFO` messages from this node appear on the console. It also explains why this node is missing from the list of log files above.

¹In the current version of `roslaunch`, output on standard error—notably including console outputs of `ERROR`- and `FATAL`-level log messages—appears on the console, rather than in log files. However, a comment in the `roslaunch` source code notes that this behavior may be changed in the future.

⁴<http://wiki.ros.org/roslaunch/XML>

In addition to the output attribute, which affects only a single node, we can also force roslaunch to display output from all of its nodes, using the `--screen` command-line option:

```
roslaunch --screen package-name launch-file-name
```



If a program, when started from roslaunch, does not appear to be producing the output you expect, you should verify that that node has the `output="screen"` attribute set.

Requesting respawning After starting all of the requested nodes, roslaunch monitors each node, keeping track of which ones remain active. For each node, we can ask roslaunch to restart it when it terminates, by using a `respawn` attribute:

```
respawn="true"
```

This can be useful, for example, for nodes that might terminate prematurely, due to software crashes, hardware problems, or other reasons.

The `respawn` attribute is not really necessary in our example—All three programs are quite reliable—but we include it for the `turtlesim_node` to illustrate how respawning works. If you close the `turtlesim` window, the corresponding node will terminate. ROS quickly notices this and, since that node is marked as a `respawn` node, a new `turtlesim` node, with its accompanying window, appears to replace the previous one.

Requiring nodes An alternative to `respawn` is to declare that a node is required:

```
required="true"
```

When a required node terminates, roslaunch responds by terminating all of the other active nodes and exiting itself. That sort of behavior might be useful, for example, for nodes that (a) are so important that, if they fail, the entire session should be abandoned, and (b) cannot be gracefully restarted by the `respawn` attribute.

The example uses the `required` attribute for the `turtle_teleop_key` node. If you close the window in which the teleoperation node runs, roslaunch will kill the other two nodes and exit.



Because their meanings conflict with one another, roslaunch will complain if you set both the `respawn` and `required` attributes for a single node.

Launching nodes in their own windows One potential drawback to using roslaunch, compared to our original technique of using rosrund in a separate terminal for each node, is that all of the nodes share the same terminal. This is manageable (and often helpful) for nodes that simply generate log messages, and do not accept console input. For nodes that do rely on console input, as `turtle_teleop_key` does, it may be preferable to retain the separate terminals.

Fortunately, roslaunch provides a clean way to achieve this effect, using the `launch-prefix` attribute of a node element:

```
launch-prefix="command-prefix"
```

The idea is that roslaunch will insert the given prefix at the start of the command line it constructs internally to execute the given node. In `example.launch`, we used this attribute for the teleoperation node:

```
launch-prefix="xterm -e"
```

Because of this attribute, this node element is roughly equivalent to this command:

```
xterm -e rosrund turtlesim turtle_teleop_key
```

As you may know, the `xterm` command starts a simple terminal window. The `-e` argument tells `xterm` to execute the remainder of its command line (in this case, `rosrund turtlesim turtle_teleop_key`) inside itself, in lieu of a new interactive shell. The result is that `turtle_teleop_key`, a strictly text-based program, appears inside a graphical window.

►► The `launch-prefix` attribute is, of course, not limited to `xterm`. It can also be useful for debugging (via `gdb` or `valgrind`), or for lowering the scheduling priority of a process (via `nice`).⁵

⁵http://wiki.ros.org/rqt_console

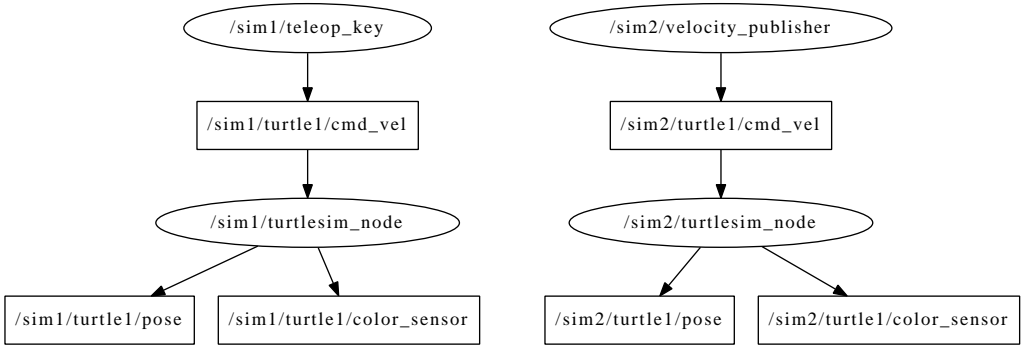



Figure 6.1: Nodes and topics (in ellipses and rectangles, respectively) created by `doublesim.launch`.


6.3 Launching nodes inside a namespace

We saw in Section 5.2 that ROS supports relative names, which utilize the concept of a default namespace. The usual way to set the default namespace for a node—a process often called **pushing down** into a namespace—is to use a launch file, and assign the `ns` attribute in its node element:

```
ns="namespace"
```

Listing 6.3 shows an example launch file that uses this attribute to create two independent `turtlesim` simulators. Figure 6.1 shows the nodes and topics that result from this launch file.

 The usual `turtlesim` topic names (`turtle1/cmd_vel`, `turtle1/color_sensor`, and `turtle1/pose`) are moved from the global namespace into separate namespaces called `/sim1` and `/sim2`. This change occurs because the code for `turtlesim_node` uses relative names like `turtle1/pose` (instead of global names like `/turtle1/pose`), when it creates its `ros::Publisher` and `ros::Subscriber` objects.

 Likewise, the node names in the launch file are relative names. In this case, both nodes have the same relative name, `turtlesim_node`. Such identical relative names are not a problem, however, because the global names to which they are resolved, namely `/sim1/turtlesim_node` and `/sim2/turtlesim_node`, are different.

```
1 <launch>
2   <node
3     name="turtlesim_node "
4     pkg="turtlesim "
5     type="turtlesim_node "
6     ns="sim1 "
7   />
8   <node
9     pkg="turtlesim "
10    type="turtle_teleop_key "
11    name="teleop_key "
12    required="true "
13    launch-prefix="xterm -e "
14    ns="sim1 "
15  />
16  <node
17    name="turtlesim_node "
18    pkg="turtlesim "
19    type="turtlesim_node "
20    ns="sim2 "
21  />
22  <node
23    pkg="agitr "
24    type="pubvel"
25    name="velocity_publisher "
26    ns="sim2 "
27  />
28 </launch>
```

Listing 6.3: A launch file called `doublesim.launch` that starts two independent `turtlesim` simulations. One simulation has a turtle moved by randomly-generated velocity commands; the other is teleoperated.

►► In fact, `roslaunch` requires the node names in the launch files to be base names—relative names without mention of any namespaces—and will complain if a global name appears in the name attribute of a node element.

This example has some similarities to the system discussed in Section 2.8.² In both cases, we start multiple `turtlesim` nodes. The results, however, are quite different. In Section 2.8, we changed only the node names, and left all of the nodes in the same namespace. As a result, both `turtlesim` nodes subscribe to and publish on the same topics. There is no straightforward way to interact with either of the two simulations individually. In the new example from Listing 6.3, we pushed each simulator node into its own namespace. The resulting changes to the topic names make the two simulators truly independent, enabling us to publish different velocity commands to each one.

►► In this example, the namespaces specified by the `ns` attributes are themselves relative names. That is, we used the names `sim1` and `sim2` in a context within the launch file in which the default namespace is the global namespace `/`. As a result, the default namespaces for our two nodes are resolved to `/sim1` and `/sim2`.

It is technically possible to provide a global name for this attribute instead. However, that's almost always a bad idea, for essentially the same reason that using global names inside nodes is a bad idea. Doing so would prevent the launch file from itself being pushed into a namespace of its own, for example as a result of being included by another launch file.

6.4 Remapping names


In addition to resolving relative names and private names, ROS nodes also support **remappings**, which provide a finer level of control for modifying the names used by our nodes.⁶ Remappings are based on the idea of substitution: Each remapping provides an original name and a new name. Each time a node uses any of its remappings' original names, the ROS client library silently replaces it with the new name from that remapping.

6.4.1 Creating remappings

There are two ways create remappings when starting a node.

²Now that we've seen launch files, you should be able to replace the ugly series of four `roslaunch` commands that section with a single small launch file.

⁶[http://wiki.ros.org/Remapping Arguments](http://wiki.ros.org/Remapping%20Arguments)

-  To remap a name when starting a node from the command line, give the original name and the new name, separated by a `:`, somewhere on the command line.

original-name:=new-name

For example, to run a `turtlesim` instance that publishes its pose data on a topic called `/tim` instead of `/turtle1/pose`, use a command like this:

```
roslaunch turtlesim turtlesim_node turtle1/pose:=tim
```

-  To remap names within a launch file, use a remap element:⁷

```
<remap from="original-name" to="new-name" />
```

If it appears at the top level, as a child of the launch element, this remapping will apply to all subsequent nodes. These remap elements can also appear as children of a node element, like this:

```
<node node-attributes >
  <remap from="original-name" to="new-name" />
  ...
</node>
```

In this case, the given remappings are applied only to the single node that owns them. For example, the command line above is essentially equivalent to this launch file construction:

```
<node pkg="turtlesim" type="turtlesim_node"
      name="turtlesim" >
  <remap from="turtle1/pose" to="tim" />
</node>
```

There is one important thing to remember about the way remappings are applied: All names, including the original and new names in the remapping itself, are resolved to global names, before ROS applies any remappings. As a result, names that appear in remappings are often relative names. After name resolution is complete, remapping is done by a direct string comparison, looking for names used by a node that exactly match the resolved original name in any remapping.

⁷<http://wiki.ros.org/roslaunch/XML/remap>

6.4.2 Reversing a turtle

For a concrete example of how these kinds of remappings might be helpful, consider a scenario in which we want to use `turtle_teleop_key` to drive a `turtlesim` turtle, *but with the meanings of arrow keys reversed*. That is, suppose we want the left and right arrow keys to rotate the turtle clockwise and counterclockwise, respectively, and the up and down arrows to move the turtle backward and forward, respectively. The example may seem contrived, but it does represent a general class of real problems in which the messages published by one node must be “translated” into a format expected by another node.

One option, of course, would be to make a copy of the `turtle_teleop_key` source code and modify it to reflect the change we want. This option is very unsatisfying, because it would require us to understand and, even worse, to duplicate the `turtle_teleop_key` code. Instead, let’s see how to do this compositionally, creating a new program that inverts the velocity commands published by the existing teleoperation node.

Listing 6.4 shows a short program that performs the change that we need: It subscribes to `turtle1/cmd_vel` and, for each message it receives, it inverts the both linear and angular velocities, publishes the resulting velocity command on `turtle1/cmd_vel_reversed`.

The only complication—and the reason this example belongs in a section about remappings—is that the `turtlesim` simulator does not actually subscribe to those reversed velocity messages. Indeed, starting the three relevant nodes with simple `roslaunch` commands leads to the graph structure shown in Figure 6.2. From the graph, it’s clear that this system would not have the desired behavior. Because velocity commands still travel directly from `teleop_turtle` to `turtlesim`, the turtle will still respond in its usual, unreversed way.

This situation—one in which some node subscribes to the “wrong” topic—is precisely the kind of situation for which remappings are intended. In this case, we can correct the problem by sending a remapping to `turtlesim` that replaces `turtle1/cmd_vel` with `turtle1/cmd_vel_reversed`. Listing 6.5 shows a launch file that starts all three nodes, including the appropriate remap for the `turtlesim_node`; Figure 6.3 shows the correct graph that results.

```
1  // This program subscribes to turtle1/cmd_vel and
2  // republishes on turtle1/cmd_vel_reversed,
3  // with the signs inverted.
4  #include <ros/ros.h>
5  #include <geometry_msgs/Twist.h>
6
7  ros::Publisher *pubPtr;
8
9  void commandVelocityReceived(
10     const geometry_msgs::Twist& msgIn
11 ) {
12     geometry_msgs::Twist msgOut;
13     msgOut.linear.x = -msgIn.linear.x;
14     msgOut.angular.z = -msgIn.angular.z;
15     pubPtr->publish(msgOut);
16 }
17
18 int main(int argc, char **argv) {
19     ros::init(argc, argv, "reverse_velocity");
20     ros::NodeHandle nh;
21
22     pubPtr = new ros::Publisher(
23         nh.advertise<geometry_msgs::Twist>(
24             "turtle1/cmd_vel_reversed",
25             1000));
26
27     ros::Subscriber sub = nh.subscribe(
28         "turtle1/cmd_vel", 1000,
29         &commandVelocityReceived);
30
31     ros::spin();
32
33     delete pubPtr;
34 }
```

Listing 6.4: A C++ program called `reverse_cmd_vel` that reverses turtlesim velocity commands.

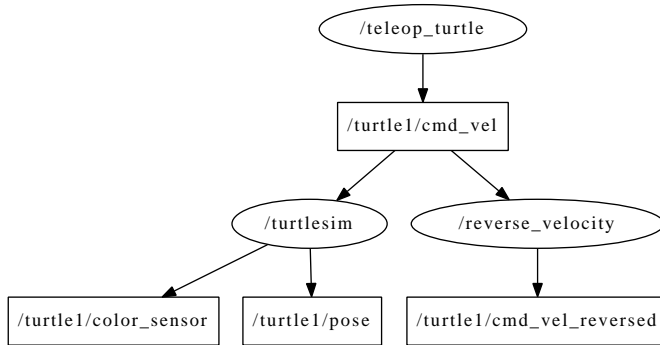


Figure 6.2: The ROS graph resulting from an incorrect attempt to use `reverse_cmd_vel` to reverse a `turtlesim` turtle.

6.5 Other launch file elements

This section introduces a few additional `roslaunch` constructions.⁸ To illustrate these features, we'll refer to the launch file in Listing 6.6. This launch file starts either two or three independent `turtlesim` simulators, depending on how it is launched.

6.5.1 Including other files

To include the contents of another launch file, including all of its nodes and parameters, use an `include` element:⁹

```
<include file="path-to-launch-file" />
```

The `file` attribute expects the full path to the file we want to include. Because it can be both cumbersome and brittle to enter this information directly, most include elements use a `find` substitution to search for a package, instead of explicitly naming a directory:

```
<include file="$(find package-name)/launch-file-name" />
```

The `find` argument is expanded, via a string substitution, to the path to the given package. The specific launch file is usually much easier to name from there. The example uses this technique to include our previous example, `doublesim.launch`.

⁸[http://wiki.ros.org/ROS/Tutorials/Roslaunch tips for larger projects](http://wiki.ros.org/ROS/Tutorials/Roslaunch%20tips%20for%20larger%20projects)

⁹<http://wiki.ros.org/roslaunch/XML/include>

```
1 <launch>
2   <node
3     pkg="turtlesim "
4     type="turtlesim_node "
5     name="turtlesim "
6   >
7     <remap
8       from="turtle1/cmd_vel"
9       to="turtle1/cmd_vel_reversed "
10    />
11  </node>
12  <node
13    pkg="turtlesim "
14    type="turtle_teleop_key "
15    name="teleop_key "
16    launch-prefix="xterm -e "
17  />
18  <node
19    pkg="agitr "
20    type="reverse_cmd_vel "
21    name="reverse_velocity "
22  />
23 </launch>
```

Listing 6.5: A launch file called `reverse.launch` that starts a `turtlesim` that can be teleoperated with directions reversed.



Don't forget that `roslaunch` will search through a package's subdirectories when searching for a launch file given on its command line. On the other hand, include elements must name the specific path to the file they want, and cannot rely on this search of subdirectories. This difference explains how the include element above might generate errors, even though a call to `roslaunch`, with the same package name and launch file name, succeeds.

The include element also supports the `ns` attribute for pushing its contents into a namespace:

```
<include file=". . ." ns="namespace" />
```

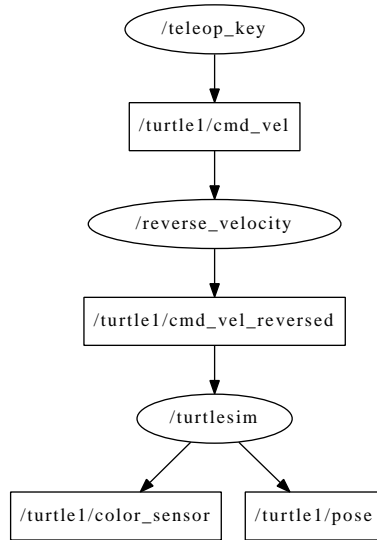


Figure 6.3: The correct ROS graph resulting from `reversed.launch`. The `remap` element enables the nodes to connect properly.

This setup occurs fairly commonly, especially when the included launch file is part of a another package, and should operate mostly independently of the rest of the nodes.

6.5.2 Launch arguments

To help make launch files configurable, `roslaunch` supports **launch arguments**, also called **arguments** or even **args**, which function somewhat like local variables in an executable program.¹⁰ The advantage is that you can avoid code duplication by writing launch files that use arguments for the small number of details that might change from run to run. To illustrate this idea, the example launch file uses one argument, called `use_sim3`, to determine whether to start three copies of `turtlesim` or only two.



Although the terms `argument` and `parameter` are used somewhat interchangeably in many computing contexts, their meanings are quite different in ROS. Parameters are values used by a running ROS system, stored on the parameter server and accessible to active nodes via the `roscpp::param::get` functions and to users via `rosparam`.

¹⁰<http://wiki.ros.org/roslaunch/XML/arg>

```
1 <launch>
2   <include
3     file="$(find agittr)/doublesim.launch"
4   />
5   <arg
6     name="use_sim3"
7     default="0"
8   />
9
10  <group ns="sim3" if="$(arg use_sim3)" >
11    <node
12      name="turtlesim_node"
13      pkg="turtlesim"
14      type="turtlesim_node"
15    />
16    <node
17      pkg="turtlesim"
18      type="turtle_teleop_key"
19      name="teleop_key"
20      required="true"
21      launch-prefix="xterm -e"
22    />
23  </group>
24 </launch>
```

Listing 6.6: A launch file called `triplesim.launch` that illustrates `group`, `include`, and `arg` arguments.

(See Chapter 7.) In contrast, arguments make sense only within launch files; their values are not directly available to nodes.

Declaring arguments To **declare** the existence of an argument, use an `arg` element:

```
<arg name="arg-name" />
```

Declarations like this are not strictly required (unless you want to assign a default or a value—see below), but are a good idea because they can make it clear to a human reader what arguments the launch file is expecting.

Assigning argument values Every argument used in a launch file must have an assigned value. There are a few ways to accomplish this. You can provide a value on the roslaunch command line:

```
roslaunch package-name launch-file-name arg-name:=arg-value
```

Alternatively, you can provide a value as part of the arg declaration, using one of these two syntaxes:

```
<arg name="arg-name" default="arg-value" />
<arg name="arg-name" value="arg-value" />
```

The only difference between them is that a command line argument can override a default, but not a value. In the example, use_sim3 has a default value of 0, but this can be changed from the command line, like this:

```
roslaunch agitr triplesim.launch use_sim3:=1
```

If we were to modify the launch file, replacing default with value, then this command would generate an error, because argument values set by value cannot be changed.

Accessing argument values Once an argument is declared and a value assigned to it, you can use its value using an arg substitution, like this:

```
$(arg arg-name)
```

Anywhere this substitution appears, roslaunch will replace it with the value of the given argument. In the example, we use the use_sim3 argument once, inside the if attribute of a group element. (We'll introduce both if and group shortly.)

Sending argument values to included launch files One limitation of the argument setting technique presented so far is that it does not provide any means for passing arguments to subordinate launch files that we import using include elements. This is important because, much like (lexical) local variables, arguments are only defined for the launch file that declares them. Arguments are not “inherited” by included launch files.

The solution is to insert arg elements as children of the include element, like this:

```
<include file="path-to-launch-file">
  <arg name="arg-name" value="arg-value" />
  ...
</include>
```

Note that this usage of the `arg` element differs from the `arg` declarations we've seen so far. The arguments mentioned between the `include` tags are arguments for the included launch file, not for the launch file in which they appear. Because the purpose is to establish values for the arguments needed by the included file, the `value` attribute is required in this context.


One common scenario is that both launch files—the included one and the including one—have some arguments in common. In such cases, we might want to pass those values along unchanged. An element like this, using the same argument name in both places, does this:

```
<arg name="arg-name" value="$(arg arg-name)" />
```

In this instance, the first appearance of the argument's name refers, as usual, to that argument in the included launch file. The second appearance of the name refers to the argument in the including launch file. The result is that the given argument will have the same value in both launch files.

6.5.3 Creating groups

One final launch file feature is the `group` element, which provides a convenient way to organize nodes in a large launch file.¹¹ The `group` element can serve two purposes:

 Groups can push several nodes into the same namespace.

```
<group ns="namespace" />
...
</group>
```

Every node within the `group` starts with the given default namespace.

►► If a grouped node has its own `ns` attribute, and that name is (as it probably should be) a relative name, then the resulting node will be started with a default namespace that nests the latter namespace within the former. These rules, which match what one would expect for resolving relative names, also apply to nested groups.

 Groups can conditionally enable or disable nodes.

¹¹<http://wiki.ros.org/roslaunch/XML/group>

```
<group if="0-or-1" />
...
</group>
```

If the value of the `if` attribute is 1, then the enclosed elements are included normally. If this attribute has value 0, then the enclosed elements are ignored. The `unless` attribute works similarly, but with the meanings reversed:

```
<group unless="1-or-0" />
...
</group>
```

Of course, it is unusual to directly type a 0 or 1 for these attributes. Combined with the `arg` substitution technique, however, they form a powerful way of making your launch files configurable.



Note that 0 and 1 are the only legitimate values for these attributes. In particular, the usual boolean AND and OR operations that you might expect are not directly available.

The example has a single group that combines these two purposes. The group has both the `ns` attribute (to push the group's two nodes into the `sim3` namespace) and the `if` attribute (to implement the enabling or disabling of that third simulation based on the `use_sim3` argument).



Notice that group is never strictly necessary. It's always possible to write the `ns`, `if`, and `unless` attributes manually for each element that we might otherwise include in a group. However, groups can often reduce duplication—the namespaces and conditions appear only once—and make the launch file's organization more readily apparent.

►► Unfortunately, only these three attributes can be passed down via a group. For example, as much as we might want to, `output="screen"` cannot be set for a group element, and must be given directly to each node to which we want to apply it.

6.6 Looking forward

In the chapters so far, we've seen how to create nodes that communicate by passing messages and how to start many nodes at once, with potentially complex configurations. The next chapter introduces the parameter server, which provides a centralized way to provide parts of that configuration information to nodes.