A Gentle Introduction to ROS

Chapter: Log messages

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 2.1.6 (ab984b3), generated on April 24, 2018.

Typeset by the author using LaTeX and memoir.cls.

ISBN 978-14-92143-23-9

Chapter 4

Log messages

In which we generate and view log messages.

We have already seen, in the example programs from Chapter 3, a macro called ROS-_INFO_STREAM that displays informative messages to the user. These messages are examples of **log messages**. ROS provides a rich logging system that includes ROS_INFO-_STREAM along with a number of other features. In this chapter, we'll see how to use that logging system.

4.1 Severity levels

The idea of ROS's logging system—and, for the most part, software logging in general—is to allow programs to generate a stream of short text strings called log messages. In ROS, log messages are classified into five groups called **severity levels**, which are sometimes called just **severities** and sometimes called just **levels**. The levels are, in order of increasing importance:[1]

DEBUG
INFO
WARN
ERROR
FATAL

The idea is that DEBUG messages may be generated very frequently, but are not generally interesting when the program is working correctly. At the other end of the spectrum,

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

Severity	Example message
DEBUG	reading header from buffer
INFO	Waiting for all connections to establish
WARN	Less than 5GB of space free on disk
ERROR	Publisher header did not have required element: type
FATAL	You must call ros::init() before creating the first NodeHandle

Figure 4.1: Examples log messages for each severity level.

FATAL messages are likely to be very rare but very important, indicating a problem that prevents the program from continuing. The other three levels, INFO, WARN, and ERROR, represent intermediate degrees of importance between these two extremes. Figure 4.1 shows examples, from the ROS source, of each of these severity levels.

This variety of severity levels is intended to provide a consistent way to classify and manage log messages. We'll see shortly, for example, how to filter or highlight messages based on their severity levels. However, the levels themselves don't carry any inherent meaning: Generating a FATAL message will not, in itself, end your program. Likewise, generating a DEBUG message will not (alas) debug your program for you.

4.2 An example program

The remainder of this chapter deals with how to generate and view log messages. As usual, it will be helpful to have a concrete example program to illustrate what's going on. It would be possible to use turtlesim for this purpose—under the right conditions, turtlesim_node will produce log messages at every level except FATAL—but for learning purposes it will be more convenient to work with a program that produces *lots* of log messages at predictable intervals.

Listing 4.1 shows a program that fits this description. It generates a steady stream of messages at all five severity levels. An example of its console output appears in Listing 4.2. We'll use this as a running example throughout the rest of the chapter.

4.3 Generating log messages

Let's have a more complete look at how to generate log messages from C++ code.

Generating simple log messages There are five basic C++ macros for generating log messages, one for each severity level:

```
1  // This program periodically generates log messages at
2  // various severity levels.
3  #include <ros/ros.h>
4
5  int main(int argc, char **argv) {
6    // Initialize the ROS system and become a node.
7    ros::init(argc, argv, "count_and_log");
8    ros::NodeHandle nh;
9
10   // Generate log messages of varying severity regularly.
11   ros::Rate rate(10);
12   for(int i = 1;ros::ok();i++) {
13     ROS_DEBUG_STREAM("Counted to " << i);
14     if((i % 3) == 0) {
15       ROS_INFO_STREAM(i << " is divisible by 3.");
16     }
17     if((i % 5) == 0 ) {
18       ROS_WARN_STREAM(i << " is divisible by 5.");
19     }
20     if((i % 10) == 0) {
21       ROS_ERROR_STREAM(i << " is divisible by 10.");
22     }
23     if((i % 20) == 0) {
24       ROS_FATAL_STREAM(i << " is divisible by 20.");
25     }
26     rate.sleep();
27   }
28 }
```

Listing 4.1: A program called count.cpp that generates log messages at all five severity levels.

ROS_DEBUG_STREAM(*message*);
ROS_INFO_STREAM(*message*);
ROS_WARN_STREAM(*message*);
ROS_ERROR_STREAM(*message*);
ROS_FATAL_STREAM(*message*);

The *message* argument of each of these macros can handle exactly the kinds of expressions that work with a C++ ostream, such as std::cout. This includes using the insertion operator (<<) on primitive data types like int or double, on composite types for which

```
1 [ INFO] [1375889196.165921375]: 3 is divisible by 3.
2 [ WARN] [1375889196.365852904]: 5 is divisible by 5.
3 [ INFO] [1375889196.465844839]: 6 is divisible by 3.
4 [ INFO] [1375889196.765849224]: 9 is divisible by 3.
5 [ WARN] [1375889196.865985094]: 10 is divisible by 5.
6 [ERROR] [1375889196.866608041]: 10 is divisible by 10.
7 [ INFO] [1375889197.065870949]: 12 is divisible by 3.
8 [ INFO] [1375889197.365847834]: 15 is divisible by 3.
```

Listing 4.2: Sample output from running count for a few seconds. This output does not contain any DEBUG-level messages, because the default minimum level is INFO.

that operator is properly overloaded, and on standard stream manipulators like std::fixed, std::setprecision, or std::boolalpha.

Stream manipulators are effective only for the log message in which they appear. Any manipulators you would like to use must be re-inserted every time.

> ▶▶| *Here's why this limitation on stream manipulators exists: As their all-capital names suggest, the ROS_..._STREAM constructions are macros. Each expands to a short block of code that creates a std::stringstream and inserts the arguments you provide into that stream. The expanded code then ships the fully-formatted contents of that std::stringstream to an internal logging system, namely log4cxx.*[2] *Because the std::stringstream is destroyed when this process completes, its internal state, including any formatting configuration established by stream manipulators, is lost.*

▶▶| *If you prefer a printf-style interface instead of C++-style streams, there are also macros whose names omit the _STREAM suffix. For example, the macro*

ROS_INFO(*format*, ...);

[2] http://wiki.apache.org/logging-log4cxx/

generates INFO*-level log messages. These macros work exactly as you might expect, at least if you're familiar with* printf*. As a concrete example, the output line in Listing 3.4 is roughly equivalent to:*

```
ROS_INFO("position=(%0.2f,%0.2f) direction=%0.2f",
    msg.x, msg.y, msg.theta);
```

There are also printf*-style versions of the one time (*. . . _ONCE*) and throttled (*. . . _THROTTLE*) families of macros introduced below, again with names that omit the* _STREAM *part.*

Notice that there's no need to use std::endl nor any other line terminator, because the logging system is already line-oriented. Each call to any of these macros will generate a single, complete log message which will be displayed as a single line.

Generating one-time log messages Sometimes, log messages that are generated inside loops or in frequently-called functions are important to the user, but also irritatingly repetitive. One natural way to deal with these situations would be to use a static variable to ensure that the message is generated only once, the first time it is reached. Listing 4.3 shows a C++ fragment that would accomplish this. To avoid repeating this cumbersome block of code—Wrapping it in a function would not work, because this technique needs a distinct static variable for each statement—ROS provides shorthand macros that generate precisely these sorts of one-time only log messages.

```
ROS_DEBUG_STREAM_ONCE(message);
ROS_INFO_STREAM_ONCE(message);
ROS_WARN_STREAM_ONCE(message);
ROS_ERROR_STREAM_ONCE(message);
ROS_FATAL_STREAM_ONCE(message);
```

The first time these macros are encountered during a program's execution, they generate the same log messages as the corresponding non-ONCE versions. After that first execution, these statements have no effect. Listing 4.4 shows a minimal example, in which the logging macros each generate one message, on the first iteration of the loop, and are ignored on all future iterations.

Generating throttled log messages Similarly, there are macros for throttling the rate at which a given log message appears.

```
1 // Don't do this directly.  Use ROS_..._STREAM_ONCE instead.
2 {
3   static bool first_time = true;
4   if(first_time) {
5     ROS_INFO_STREAM("Here's some important information"
6       << " that will only appear once.");
7     first_time = false;
8   }
9 }
```

Listing 4.3: A fragment of C++ that disables a log message after its first execution. The ROS_..._STREAM_ONCE macros expand to very similar code blocks.

```
1 // This program generates a single log message at each
2 // severity level.
3 #include <ros/ros.h>
4
5 int main(int argc, char **argv) {
6   ros::init(argc, argv, "log_once");
7   ros::NodeHandle nh;
8
9   while(ros::ok()) {
10    ROS_DEBUG_STREAM_ONCE("This appears only once.");
11    ROS_INFO_STREAM_ONCE("This appears only once.");
12    ROS_WARN_STREAM_ONCE("This appears only once.");
13    ROS_ERROR_STREAM_ONCE("This appears only once.");
14    ROS_FATAL_STREAM_ONCE("This appears only once.");
15  }
16 }
```

Listing 4.4: A C++ program called once.cpp that generates only five log messages.

ROS_DEBUG_STREAM_THROTTLE(*interval*, *message*);
ROS_INFO_STREAM_THROTTLE(*interval*, *message*);
ROS_WARN_STREAM_THROTTLE(*interval*, *message*);
ROS_ERROR_STREAM_THROTTLE(*interval*, *message*);
ROS_FATAL_STREAM_THROTTLE(*interval*, *message*);

The *interval* parameter is a double that specifies the minimum amount of time, measured in seconds, that must pass between successive instances of the given log message.

```
1  // This program generates log messages at varying severity
2  // levels, throttled to various maximum speeds.
3  #include <ros/ros.h>
4
5  int main(int argc, char **argv) {
6    ros::init(argc, argv, "log_throttled");
7    ros::NodeHandle nh;
8
9    while(ros::ok()) {
10     ROS_DEBUG_STREAM_THROTTLE(0.1,
11       "This␣appears␣every␣0.1␣seconds.");
12     ROS_INFO_STREAM_THROTTLE(0.3,
13       "This␣appears␣every␣0.3␣seconds.");
14     ROS_WARN_STREAM_THROTTLE(0.5,
15       "This␣appears␣every␣0.5␣seconds.");
16     ROS_ERROR_STREAM_THROTTLE(1.0,
17       "This␣appears␣every␣1.0␣seconds.");
18     ROS_FATAL_STREAM_THROTTLE(2.0,
19       "This␣appears␣every␣2.0␣seconds.");
20   }
21 }
```

Listing 4.5: A C++ program called `throttle.cpp` that shows throttled log messages.

Each instance of any `ROS_..._STREAM_THROTTLE` macro will generate its log message the first time it is executed. Subsequent executions will be ignored, until the specified amount of time has passed. The timeouts are tracked separately (using a local `static` variable that stores the "last hit" time) for each instance of any of these macros.

Listing 4.5 shows a program that uses these macros to get behavior very similar to the `count` program from Listing 4.1. The key difference, apart from the the content of the messages, is that the program in Listing 4.5 will consume more computation time, because it uses polling, rather than timed sleeping, to decide when it's time to generate new messages. This sort of polling is, in real programs, generally a bad idea.

4.4 Viewing log messages

So far, we've said quite a bit about how to create log messages, but very little about where those messages actually go. There are actually three different destinations for log mes-

sages: Each log message can appear as output on the console, as a message on the rosout topic, and as an entry in a log file. Let's see how to use each of these.

4.4.1 Console

First, and most visibly, log messages are sent to the console. Specifically, DEBUG and INFO messages are printed on standard output, whereas WARN, ERROR, and FATAL messages are sent to standard error.[3]

⏭ *The distinction here between standard output and standard error is basically irrelevant, unless you want to redirect one or both of these streams to a file or a pipe, in which case it causes some complications. The usual file redirection technique*

```
command > file
```

redirects standard output, but not standard error. To capture all of the log messages to the same file, use something like this instead:

```
command &> file
```

Be careful, however, because differences in the way these two streams are buffered can cause the messages to appear out of order—with DEBUG and INFO messages appearing later than one might expect—in the result. You can force the messages into their natural order by using the stdbuf command to convince standard output to use line buffering:

```
stdbuf -oL command &> file
```

Finally, note that ROS inserts ANSI color codes—which look, to humans and to software that does not understand them, something like this: ^[[0m—into its output, even if the output is not being directed to a terminal. To view a file containing these sorts of codes, try a command like this:

```
less -r file
```

Formatting console messages You can tweak the format used to print log messages on the console by setting the ROSCONSOLE_FORMAT environment variable. This vari-

[3] http://wiki.ros.org/roscpp/Overview/Logging

able will generally contain one or more field names, each denoted by a dollar sign and curly braces, showing where the log message data should be inserted. The default format is equivalent to:

```
[${severity}] [${time}]: ${message}
```

This format is probably suitable for most uses, but there are a few other fields that might be useful:[4]

- ☞ To insert details about the source code location from which the message was generated, use some combination of the `${file}`, `${line}`, and `${function}` fields.
- ☞ To insert the name of the node that generated the log message, use the `${node}` field.

The `roslaunch` *tool (which we'll introduce in Chapter 6) does not, by default, funnel standard output and standard error from the nodes it launches to its own output streams. To see output from a* `roslaunch`*ed node, you must explicitly use the* `output="screen"` *attribute, or force all nodes to have this attribute with the* `--screen` *command-line parameter to* `roslaunch`. *See page 88.*

4.4.2 Messages on `rosout`

In addition to appearing on the console, every log message is also published on the topic `/rosout`. The message type of this topic is `rosgraph_msgs/Log`. Listing 4.6 shows the fields in this data type, which includes the severity level, the message itself, and some other associated metadata.

You might notice that the information in each of these messages is quite similar to the details in the console output discussed above. The primary usefulness of the `/rosout` topic, compared to the console output, is that it includes, in a single stream, log messages from every node in the system. All of those log messages show up on `/rosout`, regardless of where, when, or how their nodes were started, or even which computer they're running on.

Since `/rosout` is just an ordinary topic, you could, of course, use

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

```
1  byte DEBUG=1
2  byte INFO=2
3  byte WARN=4
4  byte ERROR=8
5  byte FATAL=16
6  std_msgs/Header header
7    uint32 seq
8    time stamp
9    string frame_id
10 byte level
11 string name
12 string msg
13 string file
14 string function
15 uint32 line
16 string[] topics
```

Listing 4.6: Fields in the rosgraph_msgs/Log message type.

```
rostopic echo /rosout
```

to see the messages directly. If you insist, could even write a program of your own to subscribe to /rosout and display or process the messages however you like. However, the simplest way to see /rosout messages is to use this command:[5][6]

```
rqt_console
```

Figure 4.2 depicts the resulting GUI. It shows log messages from all nodes, one per line, along with options to hide or highlight messages based on various kinds of filters. The GUI itself should not need any extra explanation.

⏭ *The description of* rqt_console *above is not quite true. In fact,* rqt_console *subscribes to* /rosout_agg *instead of* /rosout*. Here's the true graph, when both our* count *example and an instance of* rqt_console *are running:*

[5] http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch
[6] http://wiki.ros.org/rqt_console

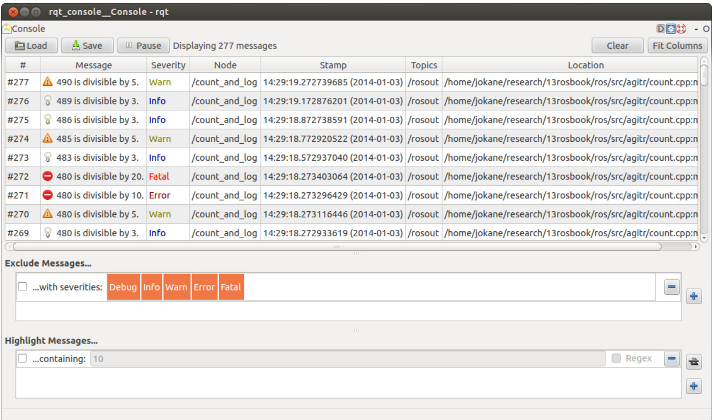
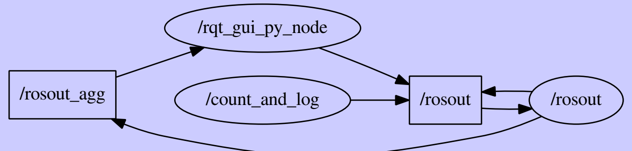


Figure 4.2: The GUI for rqt_console.



The _agg *suffix refers to the fact that messages are **aggregated** by the* rosout *node. Every message published on the* /rosout *topic is echoed on the* /rosout_agg *topic by the* rosout *node.*

The reason for this apparent redundancy is to reduce the overhead of debugging. Because each publisher-subscriber relationship leads to a direct network connection between the two nodes, subscribing to /rosout *(for which every node is a publisher) can be costly on systems with many nodes, especially when those nodes generate many log messages. The idea is that the* rosout *node will be the only subscriber to* /rosout *and the only publisher on* /rosout_agg*. Then debugging tools can access the complete stream of log messages, without creating extra work for every node in the system, by subscribing to* /rosout_agg*.*

As an aside, ROS packages for some robots, including the PR2 and the TurtleBot, use the same pattern for diagnostic messages, which are originally published on a topic called /diagnostics *and echoed by an aggregator node on another topic called* /diagnostics_agg*.*

4.4.3 Log files

The third and final destination for log messages is a log file generated by the rosout node. As part of its callback function for the /rosout topic, this node writes a line to a file with a name like this:

~/.ros/log/*run_id*/rosout.log

This rosout.log log file is a plain text file. It can be viewed with command line tools like less, head, or tail, or with your favorite text editor. The run_id is a universally-unique identifier (UUID) which is generated—based on your computer's hardware MAC address and the current time—when the master is started. Here's an example run_id:

```
57aa1860-d765-11e2-a830-f0def1e189cc
```

The use of this sort of unique identifier makes it possible to distinguish logs from separate ROS sessions.

Finding the run id There at least two easy ways to learn the run_id of the current session.

☞ You can examine the output generated by roscore. Near the end of this output, you'll see a line that looks something like this.

setting /run_id to *run_id*

☞ You can ask the master for the current run_id, using a command like this:

```
rosparam get /run_id
```

This works because the run_id is stored on the parameter server. More details about parameters are in Chapter 7.

Checking and purging log files These log files accumulate over time, which can be problematic if you use ROS for a while on a system that has meaningful limitations (due either to an account quota or to hardware limits) on disk space. Both roscore and roslaunch perform checks to monitor the size of existing logs, and warn you when they exceed 1GB, but neither will take any steps to reduce the size. You can use this command to see the amount of disk space in the current user account consumed by ROS logs:[7]

```
rosclean check
```

[7] http://wiki.ros.org/rosclean

If the logs are consuming too much disk space, you can remove all of the existing logs using this command:

```
rosclean purge
```

You can also, if you prefer, delete the log files by hand.

4.5 Enabling and disabling log messages

If you executed the programs in Listings 4.1, 4.4, and 4.5 for yourself (or read the sample output in Listing 4.2 carefully), you might have noticed that no `DEBUG`-level messages are generated, even though those programs call the `ROS_DEBUG_STREAM` macro. What happened to those `DEBUG`-level messages? The answer is that, by default, ROS C++ programs only generate log messages at the `INFO` level and higher; attempts to generate `DEBUG`-level messages are discarded.

This is a specific example of the concept of **logger levels**, which specify, for each node, a minimum severity level. The default logger level is `INFO`, which explains the absence of `DEBUG`-level messages from our example program. The general idea behind logger levels is to provide, at run time, the ability to regulate the level of detail for each node's logs.

Setting the logger level is somewhat similar to the severity filtering options in `rqt_console`*. The difference is that changing the logger level prevents log messages from ever being generated at their source, whereas the filters in* `rqt_console` *accept any incoming log messages, and selectively choose not to display some of them. Except for some overhead, the effect is similar.*

⏭ *For log messages that are disabled by the logger level, the message expression is not even evaluated. This is possible because* `ROS_INFO_STREAM` *and similar constructions are macros rather than function calls. The expansions of these macros check whether the message is enabled, and only evaluate the message expression itself if the answer is yes. This means (a) that you should not rely on any side effects that might occur from building the message string, and (b) that disabled log messages will not slow your program, even if the parameter to the logging macro would be time-consuming to evaluate.*

There are several ways to set a node's logger level.

Setting the logger level from the command line To set a node's logger level from the command line, use a command like this:

```
rosservice call /node-name/set_logger_level ros.package-name level
```

This command calls a service called set_logger_level, which is provided automatically by each node. (We'll study services more carefully in Chapter 8.)

☞ The *node-name* is the name of the node whose logger level you would like to set.

☞ The *package-name* is, as you might expect, the name of the package that owns the node.

☞ The *level* parameter is a string, chosen from DEBUG, INFO, WARN, ERROR, and FATAL, naming the logger level to use for that node.

For example, to enable DEBUG-level messages in our example program, we could use this command:

```
rosservice call /count_and_log/set_logger_level ros.agitr DEBUG
```

Note that, because this command communicates directly with the node in question, we cannot use it until after the node is started. If everything works correctly, this call to rosservice will output nothing but a blank line.

The set_logger_level *service will report an error if you misspell the desired logger level, but not if you misspell the* ros.*package-name* *part.*

⏭ *The* ros.*package-name* *argument to* rosservice *is needed to specify the name of the* ***logger*** *we would like to configure. Internally, ROS uses a library called* log4cxx *to implement its logging features. Everything we've discussed in this chapter uses, behind the scenes, the default logger, whose name is* ros.*package-name.*

However, the ROS C++ client library also uses several other loggers internally, to track things that are not usually interesting to users, down to the level of things like

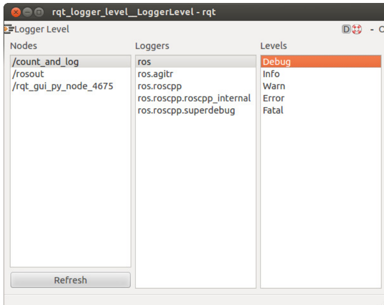


Figure 4.3: The GUI for `rqt_logger_level`.

bytes being written and read, connections being established and dropped, and callbacks being invoked. Because the `set_logger_level` *service provides an interface to all of these loggers, we must explicitly specify which logger we want to configure.*

This extra level of complexity is the reason that the `rosservice` *command above won't complain if you misspell the logger name. Instead of generating an error,* `log4cxx` *silently (and, one might add, uselessly) creates a new logger with the specified name.*

Setting the logger level from a GUI If you prefer a GUI instead of this command line interface, try this command:

```
rqt_logger_level
```

The resulting window, shown in Figure 4.3, allows you to select from a list of nodes, a list of loggers—You almost certainly want `ros.`*`package-name`*—and finally a list of logger levels. Changing the logger level using this tool has the same effect as the `rosservice` command mentioned above, because it uses the same service call interface to each node.

Setting the logger level from C++ code It is also possible for a node to modify its own logger levels. The most direct way to do this is to access the `log4cxx` infrastructure that ROS uses to implement its logging features, using code like this:

```
#include <log4cxx/logger.h>
    ...
log4cxx::Logger::getLogger(ROSCONSOLE_DEFAULT_NAME)->setLevel(
    ros::console::g_level_lookup[ros::console::levels::Debug]
);
ros::console::notifyLoggerLevelsChanged();
```

Aside from the necessary syntactic camouflage, this code should be readily identifiable as setting the logger level to `DEBUG`. The `Debug` token can, of course, be replaced by `Info`, `Warn`, `Error`, or `Fatal`.

⏭ *The call to* `ros::console::notifyLoggerLevelsChanged()` *is necessary because the enabled/disabled status of each logging statement is cached. It can be omitted if you set the logger level before any logging statements are executed.*

4.6 Looking forward

In this chapter, we saw how to generate log messages from within ROS programs, and how to view those messages in several different ways. These messages can be useful for tracking and debugging the behavior of complex ROS systems, especially when those systems span many different nodes. The next chapter discusses ROS names, which, when used wisely, can also help us to compose complicated systems of nodes from smaller parts.