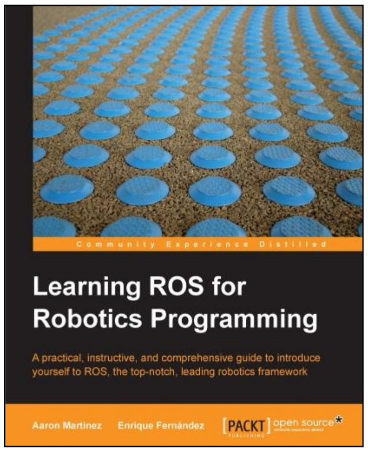
Learning ROS for Robotics Programming

Aaron Martinez

Enrique Fernández



Chapter No. 3

"Debugging and Visualization"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.3 "Debugging and Visualization"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Aaron Martinez is a computer engineer, entrepreneur, and expert in digital fabrication. He did his Master's thesis in 2010 at the IUCTC (Instituto Universitario de Ciencias y Tecnologias Ciberneticas) in the University of Las Palmas de Gran Canaria. He prepared his Master's thesis in the field of telepresence using immersive devices and robotic platforms. After completing his academic career, he attended an internship program at The Institute for Robotics in the Johannes Kepler University in Linz, Austria. During his internship program, he worked as part of a development team of a mobile platform using ROS and the navigation stack. After that, he was involved in some projects related to robotics, one of them is the AVORA project in the University of Las Palmas de Gran Canaria. In this project, he worked on the creation of an AUV (Autonomous Underwater Vehicle) to participate in the Student Autonomous Underwater Challenge-Europe (SAUC-E) in Italy. In 2012, he was responsible for manufacturing this project; in 2013, he helped to adapt the navigation stack and other algorithms from ROS to the robotic platform.

Recently, Aaron created his own company called Biomecan. This company works with projects related to robotics, manufacturing of prototypes, and engineering tissue. The company manufactures devices for other companies and research and development institutes. For the past two years, he has been working on engineering tissue projects, creating a new device to help researchers of cell culture.

Aaron has experience in many fields such as programming, robotics, mechatronics, and digital fabrication, many devices such as Arduino, BeagleBone, Servers, and LIDAR, servomotors, and robotic platforms such as Wifi bot, Nao Aldebaran, and Pioneer P3AT.

I would like to thank my girlfriend who has supported me while writing this book and gave me motivation to continue growing professionally. I also want to thank Donato Monopoli, Head of Biomedical Engineering Department at ITC (Canary-Islands Institute of Technology), and all the staff there.

Thanks for teaching me all I know about digital fabrication, machinery, and engineering tissue. I spent the best years of my life in your workshop. Thanks to my colleagues in the university, especially Alexis Quesada, who gave me the opportunity to create my first robot in my Master's thesis. I have learned a lot about robotics working with them.

Finally, thanks to my family and friends for their help and support.

Enrique Fernández is a computer engineer and roboticist. He did his Master's Thesis in 2009 at the University Institute of Intelligent Systems and Computational Engineering in the University of Las Palmas de Gran Canaria. There he has been working on his Ph.D for the last four years; he is expected to become a Doctor in Computer Science by September 2013. His Ph.D addresses the problem of Path Planning for Autonomous Underwater Gliders, but he has also worked on other robotic projects. He participated in the Student Autonomous Underwater Challenge-Europe (SAUC-E) in 2012, and collaborated for the 2013 edition. In 2012, he was awarded a prize for the development of an underwater pan-tilt vision system.

Now, Enrique is working for Pal-Robotics as a SLAM engineer. He completed his internship in 2012 at the Center of Underwater Robotics Research in the University of Girona, where he developed SLAM and INS modules for the Autonomous Underwater Vehicles of the research group using ROS. He joined Pal-Robotics in June 2013, where he is working with REEM robots using the ROS software intensively and developing new navigation algorithms for wheeled and biped humanoid robots, such as the REEM-H3 and REEM-C.

During his Ph.D, Enrique has published several conference papers and publications. Two of these were sent to the International Conference of Robotics and Automation (ICRA) in 2011. He is the co-author of some chapters of this book, and his Master's Thesis was about the FastSLAM algorithm for indoor robots using a SICK laser scanner and the odometry of a Pioneer differential platform. He also has experience with electronics and embedded systems, such as PC104 and Arduino. His background covers SLAM, Computer Vision, Path Planning, Optimization, and Robotics and Artificial Intelligence in general.

I would like to thank my colleagues in the AVORA team, which participated in the SAUC-E competition, for their strong collaboration and all the things we learned. I also want to thank the members of my research group at the University Institute of Intelligent Systems and Computational Engineering and the people of the Center of Underwater Robotics Research in Girona. During that time, I expended some of the most productive days of my life; I have learned a lot about robotics and had the chance to learn player/stage/Gazebo and start with ROS. Also, thanks to my colleagues in Pal-Robotics, who have received me with open arms, and have given me the opportunity to learn even more about ROS and (humanoid) robots. Finally, thanks to my family and friends for their help and support.

Learning ROS for Robotics Programming

Learning ROS for Robotics Programming gives you a comprehensive review of ROS tools. ROS is the Robot Operating System framework, which is used nowadays by hundreds of research groups and companies in the robotics industry. But it is also the painless entry point to robotics for nonprofessional people. You will see how to install ROS, start playing with its basic tools, and you will end up working with state-of-the-art computer vision and navigation tools.

The content of the book can be followed without any special devices, and each chapter comes with a series of source code examples and tutorials that you can run on your own computer. This is the only thing you need to follow in the book.

However, we also show you how to work with hardware, so that you can connect your algorithms with the real world. Special care has been taken in choosing devices which are affordable for amateur users, but at the same time the most typical sensors or actuators in robotics research are covered.

Finally, the potential of ROS is illustrated with the ability to work with whole robots in a simulated environment. You will learn how to create your own robot and integrate it with the powerful navigation stack. Moreover, you will be able to run everything in simulation, using the Gazebo simulator. We will end the book by providing a list of real robots available for simulation in ROS. At the end of the book, you will see that you can work directly with them and understand what is going on under the hood.

What This Book Covers

Chapter 1, *Getting Started with ROS*, shows the easiest way you must follow in order to have a working installation of ROS. You will see how to install different distributions of ROS, and you will use ROS Fuerte in the rest of the book. How to make an installation from Debian packages or compiling the sources, as well as making installations in virtual machines, have been described in this chapter.

Chapter 2, *The ROS Architecture with Examples*, is concerned with the concepts and tools provided by the ROS framework. We will introduce you to nodes, topics, and services, and you will also learn how to use them. Through a series of examples, we will illustrate how to debug a node or visualize the messages published through a topic.

Chapter 3, *Debugging and Visualization*, goes a step further in order to show you powerful tools for debugging your nodes and visualize the information that goes through the node's graph along with the topics. ROS provides a logging API which allows you to diagnose node problems easily. In fact, we will see some powerful graphical tools such as rxconsole and rxgraph, as well as visualization interfaces such as rxplot and rviz. Finally, this chapter explains how to record and playback messages using rosbag and rxbag.

Chapter 4, *Using Sensors and Actuators with ROS*, literally connects ROS with the real world. This chapter goes through a number of common sensors and actuators that are supported in ROS, such as range lasers, servo motors, cameras, RGB-D sensors, and much more. Moreover, we explain how to use embedded systems with microcontrollers, similar to the widely known Arduino boards.

Chapter 5, *3D Modeling and Simulation*, constitutes one of the first steps in order to implement our own robot in ROS. It shows you how to model a robot from scratch and run it in simulation using the Gazebo simulator. This will later allow you to use the whole navigation stack provided by ROS and other tools.

Chapter 6, *Computer Vision*, shows the support for cameras and computer vision tasks in ROS. This chapter starts with drivers available for FireWire and USB cameras, so that you can connect them to your computer and capture images. You will then be able to calibrate your camera using ROS calibration tools. Later, you will be able to use the image pipeline, which is explained in detail. Then, you will see how to use several APIs for vision and integrate OpenCV. Finally, the installation and usage of a visual odometry software is described.

Chapter 7, *Navigation Stack – Robot Setups*, is the first of two chapters concerned with the ROS navigation stack. This chapter describes how to configure your robot so that it can be used with the navigation stack. In the same way, the stack is explained, along with several examples.

Chapter 8, *Navigation Stack – Beyond Setups*, continues the discussion of the previous chapter by showing how we can effectively make our robot navigate autonomously. It will use the navigation stack intensively for that. This chapter shows the great potential of ROS using the Gazebo simulator and rviz to create a virtual environment in which we can build a map, localize our robot, and do path planning with obstacle avoidance.

Chapter 9, *Combining Everything – Learn by Doing*, builds from the previous chapters and shows a number of robots which are supported in ROS using the Gazebo simulator. In this chapter you will see how to run these robots in simulation and perform several of the tasks learned in the rest of the book, especially those related to the navigation stack.

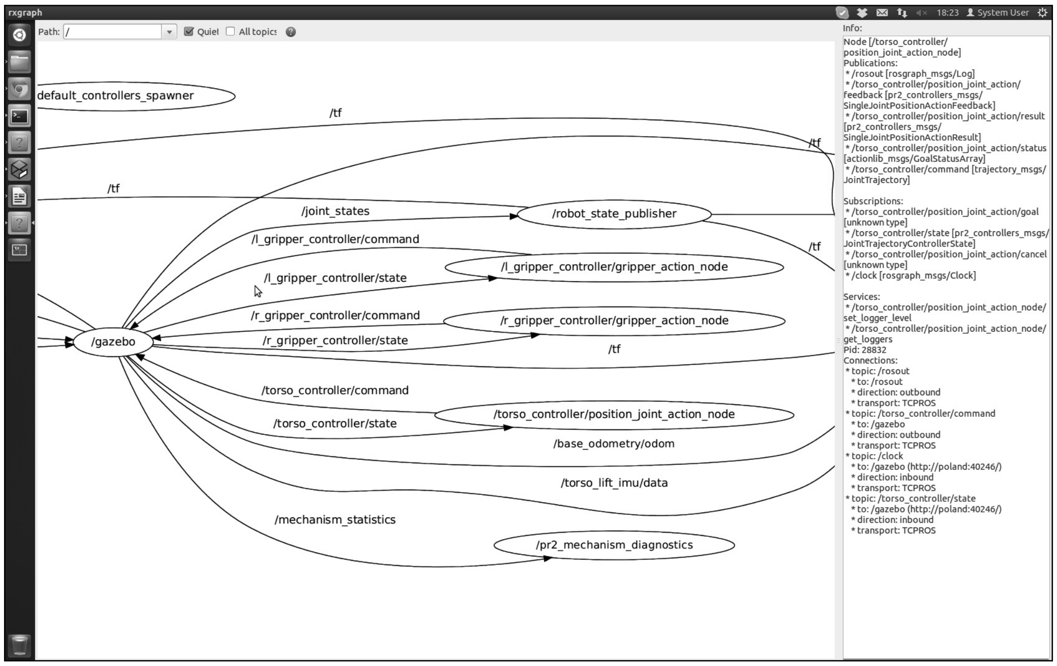
3
Debugging and Visualization

The ROS framework comes with a great number of powerful tools to help the user and developer in the process of debugging the code, and detecting problems with both the hardware and software. This comprises debugging facilities such as log messages as well as visualization and inspection capabilities, which allows the user to see what is going on in the system easily.

Here, we also cover the workflow to debug ROS nodes using GDB debugger as an example. Although this is almost the same as debugging a regular C/C++ program, there are a few aspects that must be taken into account. We will only focus on these particular aspects, since explaining the way to use the debugger is far from the scope of this chapter. You are encouraged to read the GDB reference and user manual for this.

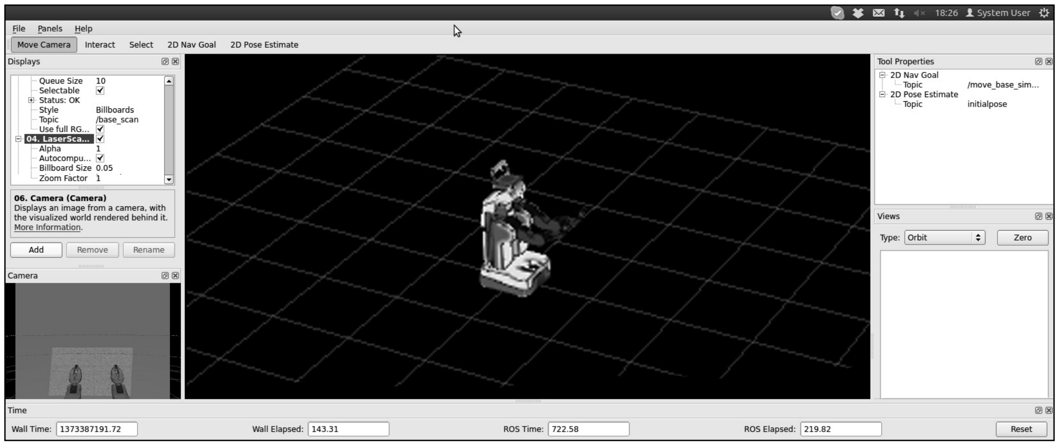
ROS provides an API for logging, which allows setting different logging levels, depending on the semantics of the message to output or print. This is not only with the aim of helping debugging but also to have more informative and robust programs in case of failure. As we will see later, we can inform the user about the stages in the process of an algorithm with high-level informative messages, while also warning the user about missed values or parameters as well as regular or fatal errors, which are unrecoverable.

However, once our program compiles and runs, it might still fail. At this point, two major things could be happening; first, a problem related to the nodes, topics, services, or any other ROS element, or second, an issue caused by our algorithm itself. ROS provides a set of powerful tools to inspect the state of the system, which include the node's graph, with all the connections (publishers and subscribers) among topics as shown in the following screenshot. Both local and remote nodes are seamlessly addressed, so the user can easily and rapidly detect a problem in a node that is not running or a missed topic connection.


rxgraph
18:23
System User
Path: /
Quiet
All topics
default_controllers_spawner
/tf
/tf
/tf
/joint_states
/robot_state_publisher
/l_gripper_controller/command
/tf
/l_gripper_controller/gripper_action_node
/l_gripper_controller/state
/r_gripper_controller/command
/r_gripper_controller/gripper_action_node
/gazebo
/r_gripper_controller/state
/tf
/torso_controller/command
/torso_controller/position_joint_action_node
/torso_controller/state
/base_odometry/odom
/torso_lift_imu/data
/mechanism_statistics
/pr2_mechanism_diagnostics

```
Info:
Node [/torso_controller/
position_joint_action_node]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /torso_controller/position_joint_action/
feedback [pr2_controllers_msgs/
SingleJointPositionActionFeedback]
 * /torso_controller/position_joint_action/result
[pr2_controllers_msgs/
SingleJointPositionActionResult]
 * /torso_controller/position_joint_action/status
[actionlib_msgs/GoalStatusArray]
 * /torso_controller/command [trajectory_msgs/
JointTrajectory]

Subscriptions:
 * /torso_controller/position_joint_action/goal
[unknown type]
 * /torso_controller/state [pr2_controllers_msgs/
JointTrajectoryControllerState]
 * /torso_controller/position_joint_action/cancel
[unknown type]
 * /clock [rosgraph_msgs/Clock]

Services:
 * /torso_controller/position_joint_action_node/
set_logger_level
 * /torso_controller/position_joint_action_node/
get_loggers
Pid: 28832
Connections:
 * topic: /rosout
   * to: /rosout
   * direction: outbound
   * transport: TCPROS
 * topic: /torso_controller/command
   * to: /gazebo
   * direction: outbound
   * transport: TCPROS
 * topic: /clock
   * to: /gazebo (http://poland:40246/)
   * direction: inbound
   * transport: TCPROS
 * topic: /torso_controller/state
   * to: /gazebo (http://poland:40246/)
   * direction: inbound
   * transport: TCPROS
```

Up to some extent, a bunch of generic plotting tools are provided to analyze the output or results of our own algorithms so that it becomes easier to detect bugs. First of all, we have time series plots for scalar values, which might be fields of the messages transferred between nodes. Then, there are tools to show images, even with support for stereo pairs. Last but not least, we have 3D visualization tools such as rviz, as shown in the following screenshot, for the PR2 robot. They allow rendering point clouds, laser scans, and so on. As an important characteristic, all data belongs to a topic that is placed on a frame so that the data is rendered in that frame. A robot is generally a compound of many frames with transform frames among them. To help the user to see the relationships among them, we also have tools to watch the frame hierarchy at a glance.



```
Queue Size        10
Selectable        ✓
Status: OK
Style             Billboards
Topic             /base_scan
Use full RG...    ✓
04. LaserSca...   ✓
Alpha             1
Autocompu...      ✓
Billboard Size    0.05
Zoom Factor       1
```

06. Camera (Camera)
Displays an image from a camera, with the visualized world rendered behind it. More Information.

```
2D Nav Goal
  Topic           /move_base_sim...
2D Pose Estimate
  Topic           initialpose
```

Type: Orbit

Wall Time: 1373387191.72 Wall Elapsed: 143.31 ROS Time: 722.58 ROS Elapsed: 219.82

In the upcoming sections, we will cover the following aspects:

- Debugging and good practices for developing code when creating ROS nodes.
- Adding logging messages to our code and setting different levels, from debug messages to errors, or even fatal ones.
- Giving names, applying conditions, and throttling the logging messages, which becomes very useful in large projects.
- Presenting a graphical tool to manage all the messages.
- Inspecting the state of the ROS system by listing the nodes running and the topics and services available.
- Visualizing the node's graph representation, which are connected by publishing and subscribing to topics.
- Plotting scalar data of certain messages.
- Visualizing scalar data for complex types. In particular, we will cover images and the case of FireWire cameras, which are seamlessly supported in ROS, as well as the 3D visualization of many topic types.
- Explaining what frames are and their close relationship with the messages published by the topics. Similarly, we will see what a frame transformation in the TF tree is.
- Saving the messages sent by topics and how to replay them for simulation or evaluation purposes.

Debugging ROS nodes

In order to detect problems in the algorithms implemented inside ROS nodes, we can face the problem at different levels to make the debugging of the software easier. First, we must provide some readable information about the progress of the algorithm, driver, or another piece of software. In ROS, we have a set of logging macros for this particular purpose, which are completely integrated with the whole system. Second, we need tools to determine which verbosity level is desired for a given node; this is related to the ability to configure different logging levels. In the case of ROS, we will see how it is possible to set debugging/logging levels even on the fly as well as conditions and names for particular messages. Third, we must be able to use a debugger to step over the source code. We will see that the widely known GDB debugger integrates seamlessly with ROS nodes. Finally, at the abstraction (or semantic) level of ROS nodes and topics, it is useful to inspect the current state of the whole system. Such introspection capabilities in ROS are supported by means of tools that draw the nodes graph with connections among topics. The user or developer can easily detect a broken connection at a glance, as we will explain later in another section.

Using the GDB debugger with ROS nodes

We will start with the standard way of debugging C/C++ executables of any kind. The flexibility of ROS allows using the well-known GDB debugger with a regular C/C++ program. All we have to know is the location of our executable, which in the case of ROS would be a node implemented in C++. Therefore, we only have to move to the path that contains the node binary and run it within GDB. Indeed, we could also run our node directly without the typical `rosrun <package> <node>` syntax.

To make it simple, we will show you how to run a node in GDB for the `example1` node in the `chapter3_tutorials` package. First, move to the package with `roscd` as follows:

```
roscd chapter3_tutorials
```

Then, we only have to recall that C++ binaries are created inside the `bin` folder of the package folder's structure. Hence, we simply run it inside GDB using the following command:

```
gdb bin/example1
```

Remember that you must have a `roscore` command running before you start your node because it will need the master/server running.

Once `roscore` is running, you can start your node in GDB by pressing the *R* key and *Enter*. You can also list the associated source code with the *L* key as well as set breakpoints or any of the functionalities that GDB comes with. If everything works correctly, you should see the following output in the GDB terminal after you have run the node inside the debugger:

```
(gdb) r
Starting program: /home/enrique/dev/rosbook/chapter3_tutorials/bin/
example1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_
db.so.1".
[New Thread 0x7ffff2664700 (LWP 3204)]
...
...
[Thread 0x7ffff1e63700 (LWP 3205) exited]
[Inferior 1 (process 3200) exited normally]
```

Attaching a node to GDB while launching ROS

If we have a `launch` file to start our node, we have some attribute in the XML syntax that allows us to attach the node to a GDB session. For the previous node, `example1`, in the package `chapter3_tutorials`, we will add the following node element to the `launch` file:

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1"/>
</launch>
```

Note that the package is passed to the `pkg` attribute and the node to the `type` attribute. We also have to give this instance of the node a name since we can run more than one instance of the same node. In this case, we gave the same name as the node type, that is, the `name` attribute, which has the value `example1`. It is also a good practice to set the attribute `output` to `screen`, so the debugging messages, which we will see in the following code snippet, appear on the same terminal where we launched the node:

```
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"/>
```

To attach it to GDB, we must add `launch-prefix="xterm -e gdb --args"`:

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"
            launch-prefix="xterm -e gdb --args"/>
</launch>
```

What this prefix does is very simple. It starts GDB, loads our node, and waits until the user presses the *C* or *R* key; that is, the node is loaded but waiting to run. This way the user can set breakpoints before the node runs and interact as a regular GDB session. Also, note that a new window opens up. This is because we create the GDB session in xterm, so we have a debugging window separated from the program output window.

Additionally, we can use the same attribute to attach the node to other diagnostic tools; for example, we can run `valgrind` on our program to detect memory leaks and perform some profiling analysis. For further information on `valgrind`, you can check out `http://valgrind.org`. To attach our node to it, we proceed in a similar way as we did with GDB. In this case, we do not need an additional window, so that we do not start xterm, and simply set `valgrind` as the launch prefix:

```
<launch>
<node pkg="chapter3_tutorials" type="example1"
            name="example1" output="screen"
            launch-prefix="valgrind"/>
</launch>
```

Enabling core dumps for ROS nodes

Although ROS nodes are actually regular executables, there are some tricky points to note to enable core dumps that can be used later in a GDB session. First of all, we have to set an unlimited core size. Note that this is required for any executable, not just ROS nodes:

```
ulimit -c unlimited
```

Then, to allow core dumps to be created, we must set the core filename to use the process `pid` by default; otherwise, they will not be created because at `$ROS_HOME`, there is already a core directory to prevent core dumps. Therefore, in order to create core dumps with the name and path `$ROS_HOME/core.PID`, we must do the following:

```
echo 1 > /proc/sys/kernel/core_uses_pid
```

Debugging messages

It is good practice to include messages that indicate what the program is doing. However, we must do it without compromising the efficiency of our software and the clearance of its output. In ROS, we have an API that covers both features and is built on top of log4cxx (a port of the well-known log4j logger library). In brief, we have several levels of messages, which might have a name depending on a condition or even throttle, with a null footprint on the performance and full integration with other tools in the ROS framework. Also, they are integrated seamlessly with the concurrent execution of nodes, that is, the messages do not get split, but they can be interleaved according to their timestamps. In the following sections, we will explain the details and how to use them adequately.

Outputting a debug message

ROS comes with a great number of functions or macros that allow us to output a debugging message as well as errors, warnings, or simply informative messages. It offers a great functionality by means of message (or logging) levels, conditional messages, interfaces for STL streams, and much more. To put things in a simple and straightforward fashion, in order to print an informative message (or information), we can do the following at any point in the code:

```
ROS_INFO( "My INFO message." );
```

Note that we do not have to include any particular library in our source code as long as the main ROS header is included. However, we can add the `ros/console.h` header as shown in the following code snippet:

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

As a result of running a program with the preceding message, we will have the following output:

```
[ INFO] [1356440230.837067170]: My INFO message.
```

All messages are printed with its level and the current timestamp (your output might differ for this reason) before the actual message and both these values are between square brackets. The timestamp is the epoch time, that is, the seconds and nanoseconds since 1970 followed by our message.

This function allows parameters in the same way as the `printf` function in C. This means that we can pass values using all special characters that we can use with `printf`; for example, we can print the value of a floating point number in the variable `val` with this code:

```
const double val = 3.14;
ROS_INFO( "My INFO message with argument: %f", val );
```

Also, C++ STL streams are supported with `*_STREAM` functions. Therefore, the previous instruction is equivalent to the following using streams:

```
ROS_INFO_STREAM(
  "My INFO stream message with argument: " << val
);
```

Please note that we did not specify any stream because it is implicit that we refer to `cout` or `cerr`, depending on the message level, as we will see in the next section.

Setting the debug message level

ROS comes with five classic logging levels, which are in the order of relevance. They are `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`.

These names are part of the function or macro used to output messages that follows this syntax:

`ROS_<LEVEL>[_<OTHER>]`

Both `DEBUG` and `INFO` messages go to `cout` (or `stdout`). Meanwhile, `WARN`, `ERROR`, and `FATAL` go to `cerr` (or `stderr`). Also, each message is printed with a particular color as long as the terminal has this capability. The colors are `DEBUG` in green, `INFO` in white, `WARN` in yellow, `ERROR` in red, and `FATAL` in purple.

The names of these messages clearly say the typology of the message given. The user must use them accordingly. As we will see in the following sections, this allows us to output only messages starting at a user-defined minimum level so that debugging messages can be omitted when our code is stable. Additionally, we have `OTHER` variants that are explained in the sequel.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you. You can also download these code files from `https://github.com/AaronMR/Learning_ROS_for_Robotics_Programming`.

Configuring the debugging level of a particular node

By default, only messages of INFO or a higher level are shown. ROS uses the levels to filter the messages printed by a particular node. There are many ways to do so. Some of them are set at the compile time, while others can be changed before execution using a configuration file. It is also possible to change this level dynamically, as we will see later in the following sections, using rosconsole and rxconsole.

First, we will see how to set the debugging level at compile time in our source code. Just go to the main function, and after the ros::init call, insert the following code:

```
// Set the logging level manually to DEBUG
ROSCONSOLE_AUTOINIT;
log4cxx::LoggerPtr my_logger =
  log4cxx::Logger::getLogger( ROSCONSOLE_DEFAULT_NAME );
my_logger->setLevel(
  ros::console::g_level_lookup[ros::console::levels::Debug]
);
```

You do not need to include any particular header, but in the CMakeLists.txt file, we must link the header to the log4cxx library. To do so, we must put:

```
find_library(LOG4CXX_LIBRARY log4cxx)
```

And our node must link to it:

```
rosbuild_add_executable(example1 src/example1.cpp)
target_link_libraries(example1 ${LOG4CXX_LIBRARY})
```

Now, DEBUG (and higher) messages are shown when our node runs since we set ros::console::levels::Debug in the preceding example. You can run the example1 node to check it and even change the level.

An alternative to the preceding method consists of using the compile-time-logger-removal macros. Note that this will remove all the messages below a given level at compilation time, so later we will not have them; this is typically useful for the release build of our programs. To do so, we must set ROSCONSOLE_MIN_SEVERITY to the minimum severity level or even none, in order to avoid any message (even FATAL); this macro can be set in the source code or even in the CMakeLists.txt file. The macros are ROSCONSOLE_SEVERITY_DEBUG, ROSCONSOLE_SEVERITY_INFO, ROSCONSOLE_SEVERITY_WARN, ROSCONSOLE_SEVERITY_ERROR, ROSCONSOLE_SEVERITY_FATAL, and ROSCONSOLE_SEVERITY_NONE.

The ROSCONSOLE_MIN_SEVERITY macro is defined in ros/console.h as DEBUG if not given. Therefore, we can pass it as a built argument (with -D) or put it before all the headers; for example, to show only ERROR (or higher) messages we will execute the following code as we did in the example2 node:

```
#define ROSCONSOLE_MIN_SEVERITY ROSCONSOLE_SEVERITY_DEBUG
```

On the other hand, we have a more flexible solution of setting the minimum debugging level in a configuration file. We will create a folder, just for convenience, named config with the file chapter3_tutorials.config and this content:

```
log4j.logger.ros.chapter3_tutorials=DEBUG
```

We can put any of the levels supported in ROS. Then, we must set the ROSCONSOLE_CONFIG_FILE environment variable to point our file. However, there is a better option. It consists of using a launch file that does this and also runs our node directly. Therefore, we can extend the launch files shown before to do so with an env element as shown in the following code snippet:

```
<launch>
  <!-- Logger config -->
  <env name="ROSCONSOLE_CONFIG_FILE"
       value="$(find chapter3_tutorials)/config/chapter3_tutorials.
config"/>

  <!-- Example 1 -->
  <node pkg="chapter3_tutorials" type="example1" name="example1"
        output="screen"/>
</launch>
```

The environment variable takes the config file, described previously, that contains the logging level specification for each named logger. Then, in the launch file, our node is simply run.

Giving names to messages

Since we can put messages in many places inside the same node, ROS allows us to give a name to each node in our program. This way, later on, it will be easier to detect from which part of the code is such a message coming. To do so, we use the ROS_<LEVEL>[_STREAM]_NAMED function as shown in the following code snippet (taken from the example2 node):

```
ROS_INFO_STREAM_NAMED(
  "named_msg",
  "My named INFO stream message; val = " << val
);
```

With named messages, we can go back to the `config` file and set different debugging levels for each named message. This allows for fine tuning using the name of the messages as the children of the node in the specification; for example, we can set the `named_msg` messages that are shown only for the `ERROR` (or higher) level with (note that although ROS uses log4cxx, the configuration files use the log4j root name) the following command line:

```
log4j.logger.ros.chapter3_tutorials.named_msg=ERROR
```

Conditional and filtered messages

Conditional messages are printed only when a given condition is satisfied. In some way, they are like conditional breakpoints using debugging messages. To use them, we have the `ROS_<LEVEL>[_STREAM]_COND[_NAMED]` functions; note that they can be named messages as well. The following are the examples of the `example2` node:

```
// Conditional messages:
ROS_INFO_STREAM_COND(
  val < 0.,
  "My conditional INFO stream message; val (" << val << ") < 0"
);
ROS_INFO_STREAM_COND(
  val >= 0.,
  "My conditional INFO stream message; val (" << val << ") >= 0"
);

// Conditional Named messages:
ROS_INFO_STREAM_COND_NAMED(
  "cond_named_msg", val < 0.,
  "My conditional INFO stream message; val (" << val << ") < 0"
);
ROS_INFO_STREAM_COND(
  "cond_named_msg", val >= 0.,
  "My conditional INFO stream message; val (" << val << ") >= 0"
);
```

Filtered messages are similar to conditional messages in essence, but they allow us to specify a user-defined filter that extends `ros::console::FilterBase`. We must pass a pointer to such a filter in the first argument of a macro with the format `ROS_<LEVEL>[_STREAM]_FILTER[_NAMED]`. The following example is taken from the `example2` node:

```
struct MyLowerFilter : public ros::console::FilterBase {
  MyLowerFilter( const double& val ) : value( val ) {}
  inline virtual bool isEnabled()
  {
    return value < 0.;
  }

  double value;
};

struct MyGreaterEqualFilter : public ros::console::FilterBase {
  MyGreaterEqualFilter( const double& val ) : value( val ) {}

  inline virtual bool isEnabled()
  {
    return value >= 0.;
  }

  double value;
};

MyLowerFilter filter_lower( val );
MyGreaterEqualFilter filter_greater_equal( val );

ROS_INFO_STREAM_FILTER(
  &filter_lower,
  "My filter INFO stream message; val (" << val << ") < 0"
);
ROS_INFO_STREAM_FILTER(
  &filter_greater_equal,
  "My filter INFO stream message; val (" << val << ") >= 0"
);
```

More messages – once, throttle, and combinations

It is also possible to control how many times a given message is shown. We can print it only once with `ROS_<LEVEL>[_STREAM]_ONCE[_NAMED]`. This kind of message is useful in loops where we do not want to output so many messages for efficiency reasons, and it is enough to know that we entered.

```
for( int i = 0; i < 10; ++i ) {
  ROS_INFO_STREAM_ONCE(
    "My once INFO stream message; i = " << i
  );
}
```

This code from the `example2` node will show the message only once for `i == 0`.

However, it is usually better to show the message at every iteration. This is where we can use throttle messages. They have the same format as that of the `ONCE` message, but if you replace `ONCE` with `THROTTLE` they will have `Duration` as the first argument; that is, it is printed only at the specified time interval:

```
for( int i = 0; i < 10; ++i ) {
  ROS_INFO_STREAM_ONCE(
    2,
    "My once INFO stream message; i = " << i
  );
  ros::Duration( 1 ).sleep();
}
```

Finally, note that named, conditional, and once/throttle messages can be combined together for all the available levels.

Nodelets also have some support in terms of logging and debugging messages. Since they have their own namespace, they have a specific name to differentiate the messages of one nodelet from another. Simply, all the macros shown so far are valid, but instead of `ROS_*` we have `NODELET_*`. These macros will only compile inside nodelets. Also, they operate by setting up a named logger, with the name of the nodelet running, so that you can differentiate between the output of two nodelets of the same type running in the same nodelet manager. Another advantage of nodelets is that they will help you turn one specific nodelet to the `DEBUG` level, instead of all the nodelets of a specific type.

Using rosconsole and rxconsole to modify the debugging level on the fly

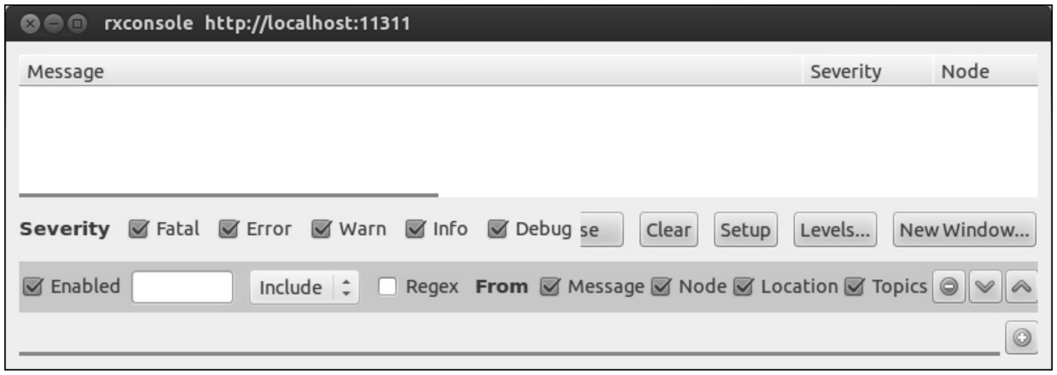
A logging message integrates with a series of tools to visualize and configure them. The ROS framework comes with an API known as **rosconsole** that was used to some extent in the previous sections. We advised you to consult the API for more advanced features, but we believe that this book covers everything a regular robotics user or developer might need.

Apart from the API to configure the logging message in your nodes, ROS provides a graphical tool, which is part of the `rxtools` package. This tool is `rxconsole`, and you only have to type it in the command line to see the graphical interface that allows seeing, inspecting, and configuring the logging subsystem of all running nodes.

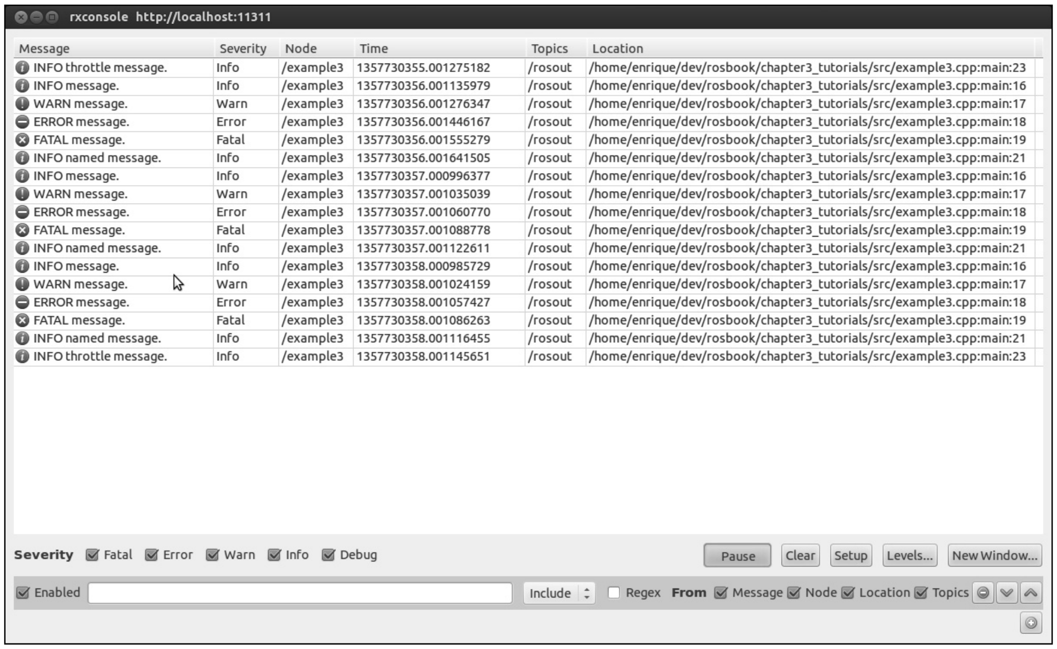
In order to test this, we are going to use `example3`, so we will run `roscore` in one terminal and the node in another using the following command line:

```
rosrun chapter3_tutorials example3
```

Now, with the node running, we will open another terminal and run rxconsole. The following window will open; note that you can also run rxconsole first. So the logging message will now appear immediately as shown in the following screenshot:



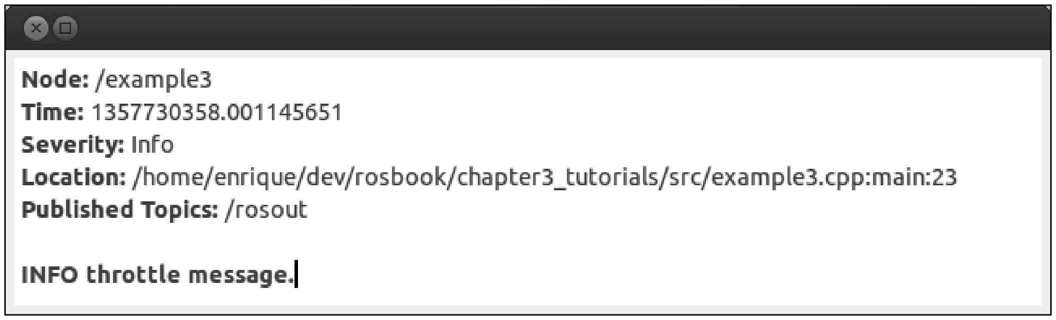
Once we have our `example3` node running, we will start to see messages as shown in the following screenshot:



Message	Severity	Node	Time	Topics	Location
INFO throttle message.	Info	/example3	1357730355.001275182	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:23
INFO message.	Info	/example3	1357730356.001135979	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:16
WARN message.	Warn	/example3	1357730356.001276347	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:17
ERROR message.	Error	/example3	1357730356.001446167	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:18
FATAL message.	Fatal	/example3	1357730356.001555279	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357730356.001641505	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
INFO message.	Info	/example3	1357730357.000996377	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:16
WARN message.	Warn	/example3	1357730357.001035039	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:17
ERROR message.	Error	/example3	1357730357.001060770	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:18
FATAL message.	Fatal	/example3	1357730357.001088778	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357730357.001122611	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
INFO message.	Info	/example3	1357730358.000985729	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:16
WARN message.	Warn	/example3	1357730358.001024159	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:17
ERROR message.	Error	/example3	1357730358.001057427	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:18
FATAL message.	Fatal	/example3	1357730358.001086263	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357730358.001116455	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
INFO throttle message.	Info	/example3	1357730358.001145651	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:23

In the table, we have several columns that give information (aligned in order, as shown in the preceding screenshot) about the message itself, its severity, the node that generated the message and the timestamp, the topic (the `/rosout` aggregator from the ROS server is usually with it), and the location of the logging macro in the source code of the node.

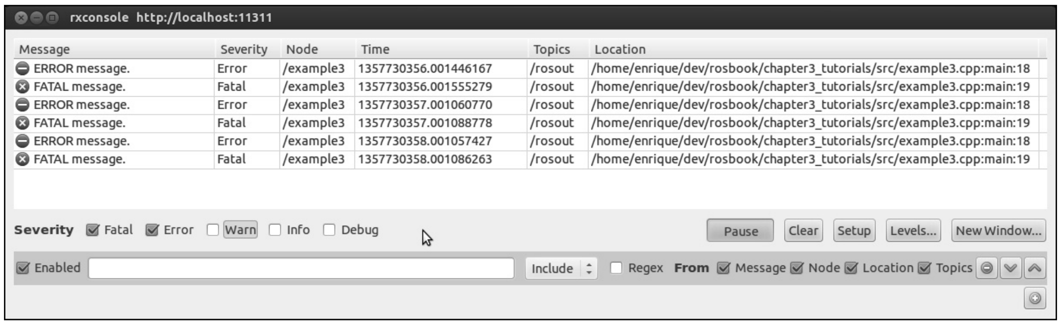
We can click on the **Pause** button to stop receiving new messages, and click on it again to resume monitoring. For each message in the table, we can click on **Pause** to see all its details as shown in the following screenshot for the last message of the previous screenshot:



Node: /example3
Time: 1357730358.001145651
Severity: Info
Location: /home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:23
Published Topics: /rosout

INFO throttle message.

One of the great features of rxconsole is the ability to filter messages. The most basic way to filter is by severity level. If we want to see only the `FATAL` and `ERROR` messages in our example, we have to unselect the other severity levels as shown in the following screenshot:



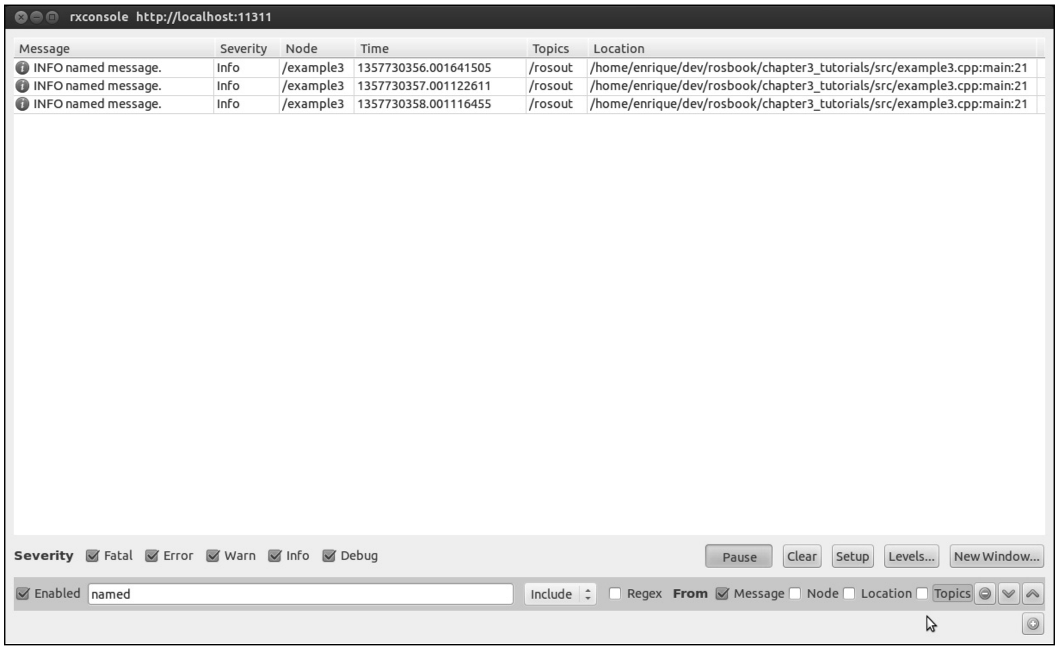
rxconsole http://localhost:11311

Message	Severity	Node	Time	Topics	Location
ERROR message.	Error	/example3	1357730356.001446167	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:18
FATAL message.	Fatal	/example3	1357730356.001555279	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
ERROR message.	Error	/example3	1357730357.001060770	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:18
FATAL message.	Fatal	/example3	1357730357.001088778	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
ERROR message.	Error	/example3	1357730358.001057427	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:18
FATAL message.	Fatal	/example3	1357730358.001086263	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19

Severity [✓ Fatal] [✓ Error] ☐ Warn ☐ Info ☐ Debug

[✓ Enabled] Include ☐ Regex **From** [✓ Message] [✓ Node] [✓ Location] [✓ Topics]

Immediately, we will see only the messages with `FATAL` and `ERROR` severity levels. Additionally, we can filter (include or exclude) by message content, node name, location, or topic name, and also using regular expression in our queries. The following screenshot shows an example in which we filtered the messages to show only those with the word `named` in the message for all the severity levels:



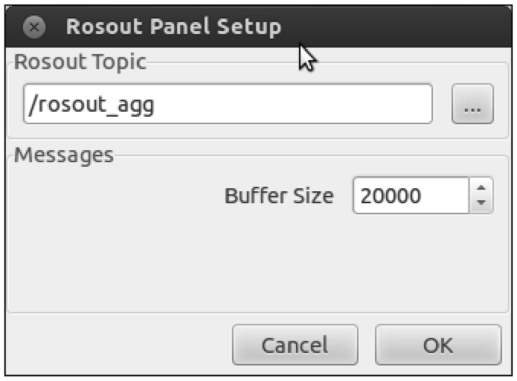
Message	Severity	Node	Time	Topics	Location
INFO named message.	Info	/example3	1357730356.001641505	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
INFO named message.	Info	/example3	1357730357.001122611	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
INFO named message.	Info	/example3	1357730358.001116455	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21

Severity [✓ Fatal] [✓ Error] [✓ Warn] [✓ Info] [✓ Debug]

[✓ Enabled] named Include ☐ Regex From [✓ Message] ☐ Node ☐ Location ☐ Topics

Note that we can add more filter entries (as much as we want) and also remove or disable the ones defined.

We can also remove all the messages captured by rxconsole by clicking on the **Clear** button. The **Setup** button allows configuring of the diagnostic aggregator topic, which is typically `rosout_agg`, and the number of messages that the GUI keeps in its history before rolling over (as shown in the following screenshot). Note that the diagnostic aggregator is just a sink in the ROS server that receives all the logging messages. This way, we can inspect which devices are failing or how are they working. An advanced developer might find it useful to learn about the diagnostic API to use a higher layer to build hierarchical layout that is already supported and used for complex systems or robots.



Rosout Panel Setup

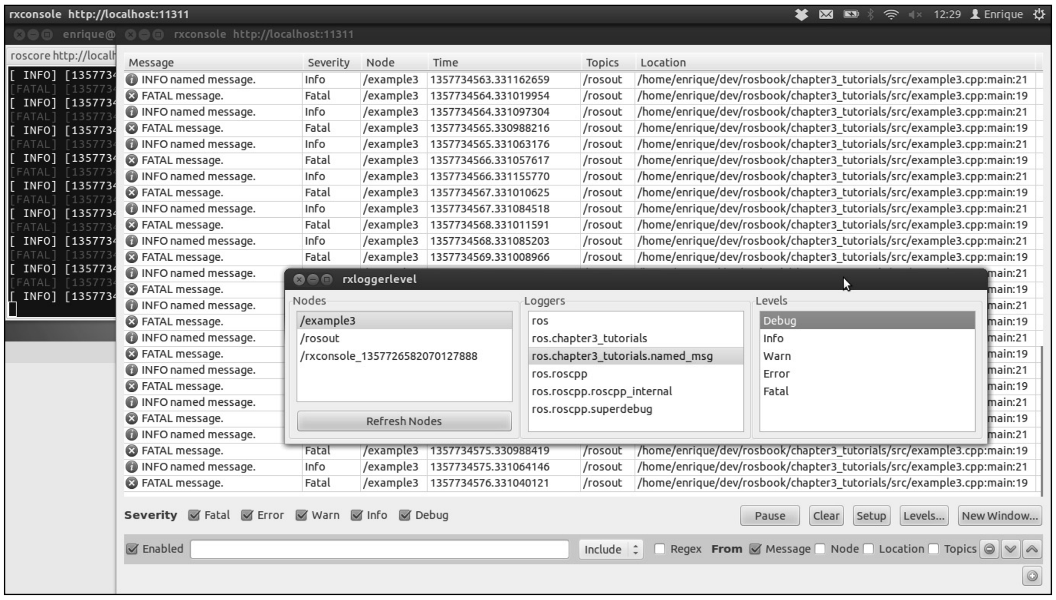
Rosout Topic: /rosout_agg

Messages

Buffer Size: 20000

Cancel OK

Finally, we can set the logger severity level for each named logger. By default, each node has a logger with its package name, but we also can define named loggers using the `NAMED` macros described in the previous sections. For the `example3` node, we have the `ros.chapter3_tutorials` and `ros.chapter3_tutorials.named_msg` loggers. If we click on the **Levels...** button, we will see the following screenshot:



rxconsole http://localhost:11311

roscore http://localh

```
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
[FATAL] [135773
[ INFO] [135773
```

Message	Severity	Node	Time	Topics	Location
INFO named message.	Info	/example3	1357734563.331162659	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734564.331019954	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357734564.331097304	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734565.330988216	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357734565.331063176	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734566.331057617	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357734566.331155770	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734567.331010625	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357734567.331084518	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734568.331011591	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357734568.331085203	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734569.331008966	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.					main:21
FATAL message.					main:19
INFO named message.					main:21
FATAL message.					main:19
INFO named message.					main:21
FATAL message.					main:19
INFO named message.					main:21
FATAL message.					main:19
INFO named message.					main:21
FATAL message.					main:19
INFO named message.					main:21
FATAL message.	Fatal	/example3	1357734575.330988419	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19
INFO named message.	Info	/example3	1357734575.331064146	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:21
FATAL message.	Fatal	/example3	1357734576.331040121	/rosout	/home/enrique/dev/rosbook/chapter3_tutorials/src/example3.cpp:main:19

rxloggerlevel

Nodes	Loggers	Levels
/example3	ros	Debug
/rosout	ros.chapter3_tutorials	Info
/rxconsole_1357726582070127888	ros.chapter3_tutorials.named_msg	Warn
	ros.roscpp	Error
	ros.roscpp.roscpp_internal	Fatal
	ros.roscpp.superdebug	

Refresh Nodes

Severity ☑ Fatal ☑ Error ☑ Warn ☑ Info ☑ Debug

Pause Clear Setup Levels... New Window...

☑ Enabled Include ☐ Regex From ☑ Message ☐ Node ☐ Location ☐ Topics

Here, we can select the `example3` node and then the logger in order to set its level. Note that there are some internal loggers that you can use apart from the ones mentioned before. In the preceding screenshot, we set the `DEBUG` severity level for the `ros.chapter3_tutorials.named_msg` logger so that all the messages with this level or higher are shown; that is, all messages in this case.

Inspecting what is going on

When our system is running, we might have several nodes and even more topics publishing messages and connected by subscription among nodes. Also, we might have some nodes providing services as well. For large systems, it is important to have some tools that let us see what is running at a given time. ROS provides us with some basic but powerful tools to do so, and also to detect a failure in any part of the nodes graph; that is, the architecture that emerges from the connection of ROS nodes using topics.

Listing nodes, topics, and services

In our honest opinion, we should start with the most basic level of introspection. We are going to see how to obtain the list of nodes running, and topics and services available at a given time. Although extremely simple, this is very useful and robust.

- To obtain the list of nodes running use:

  ```
  rosnode list
  ```

- The topics of all nodes are listed using:

  ```
  rostopic list
  ```

- And, similarly, all services are shown by:

  ```
  rosservice list
  ```

We recommend you to go back to *Chapter 2*, *The ROS Architecture with Examples* to see how these commands also allow you to obtain the message type sent by a particular topic, as well as its fields, using `rosmsg show`.

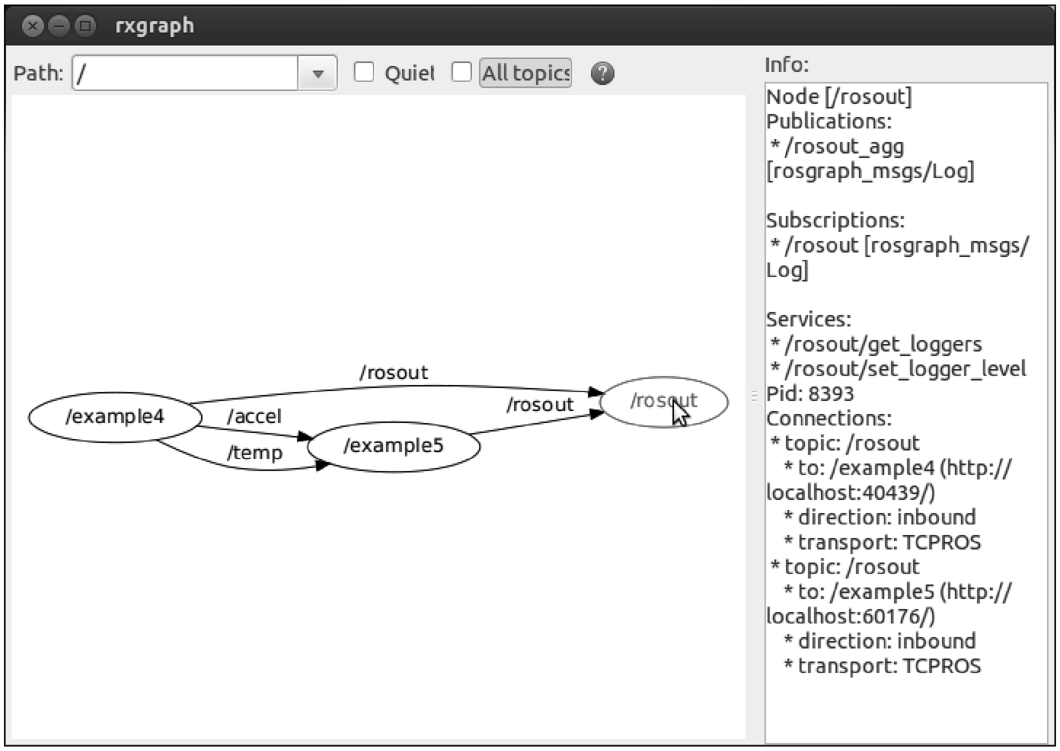
Inspecting the node's graph online with rxgraph

The simplest way to illustrate the current state of an ROS session is with a directed graph that shows the nodes running on the system and the publisher-subscriber connections among these nodes through the topics. The ROS framework provides some tools to generate such a node's graph. The main tool is rxgraph, which shows the node's graph during the system execution and allows us to see how a node appears or disappears dynamically.

To illustrate how to inspect the nodes, topics, and services with rxgraph, we are going to run the `example4` and `example5` nodes simultaneously with the following file:

```
roslaunch chapter3_tutorials example4_5.launch
```

The `example4` node publishes in two different topics and calls a service. Meanwhile, `example5` subscribes to those topics and also has the service server attend the request queries and provide the response. Once the nodes are running, we have the node's topology as shown in the following screenshot:



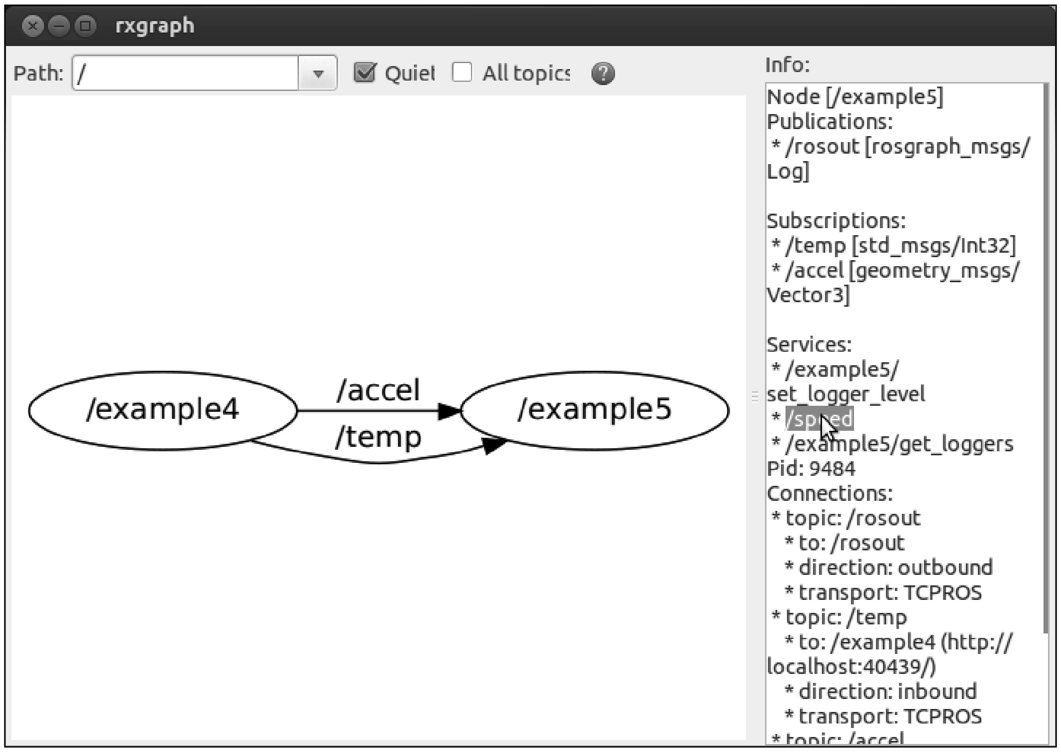
Node [/rosout]
Publications:
 * /rosout_agg
[rosgraph_msgs/Log]

Subscriptions:
 * /rosout [rosgraph_msgs/
Log]

Services:
 * /rosout/get_loggers
 * /rosout/set_logger_level
Pid: 8393
Connections:
 * topic: /rosout
   * to: /example4 (http://
localhost:40439/)
   * direction: inbound
   * transport: TCPROS
 * topic: /rosout
   * to: /example5 (http://
localhost:60176/)
   * direction: inbound
   * transport: TCPROS

In the preceding screenshot, we have nodes connected by topics. We also have the ROS server node as **rosout**, as well as the rosout topics that publish the log message for the diagnostic aggregator in the server as we have seen previously. There is also a panel to the right with information regarding the node selected. In the preceding screenshot, we have information regarding the server; that is, its IP and port for the remote nodes, topics, and connections.

We can enable the quiet view so that the ROS server is omitted. This is useful for large systems because it is always there but does not provide any information on our system's topology itself. To see the node service, we turn on the node and will see the right-hand panel. In the following screenshot, we have highlighted the `speed` service of the `example5` node:



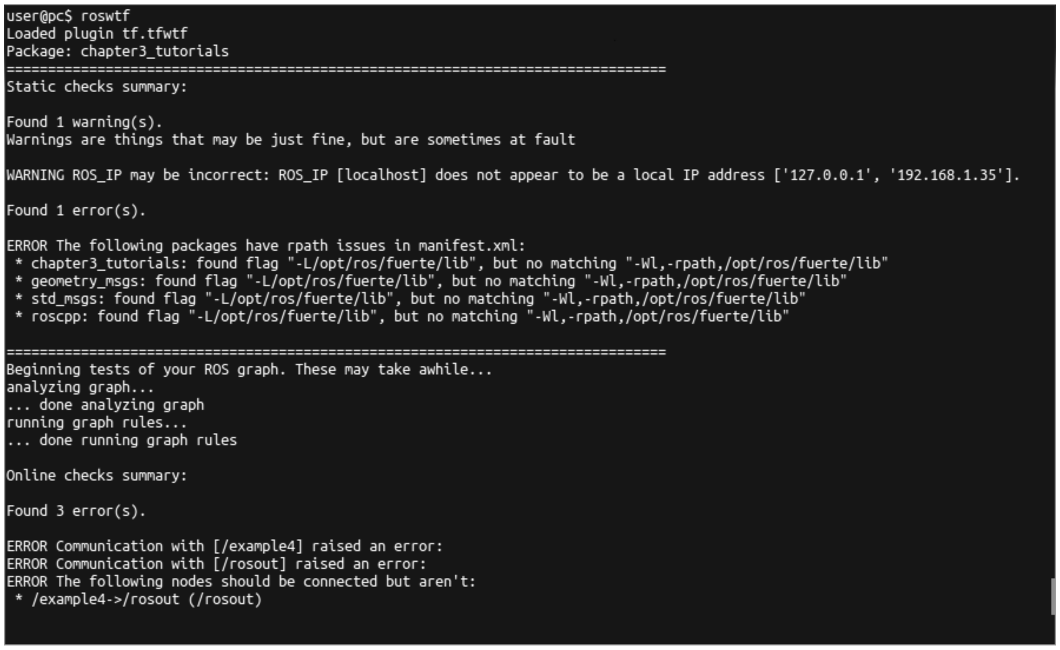
```
Node [/example5]
Publications:
 * /rosout [rosgraph_msgs/
Log]

Subscriptions:
 * /temp [std_msgs/Int32]
 * /accel [geometry_msgs/
Vector3]

Services:
 * /example5/
set_logger_level
 * /speed
 * /example5/get_loggers
Pid: 9484
Connections:
 * topic: /rosout
   * to: /rosout
   * direction: outbound
   * transport: TCPROS
 * topic: /temp
   * to: /example4 (http://
localhost:40439/)
   * direction: inbound
   * transport: TCPROS
 * topic: /accel
```

When there is a problem in the system, the nodes appear in red all the time (not just when we hover the mouse over). In such cases, it is useful to select **All topics** to show unconnected topics as well. Sometimes, the problems are a consequence of a misspelled topic name.

When running nodes in different machines, rxgraph shows its great high-level debugging capabilities since it shows whether the nodes see each other from one machine to the other.

When something weird happens – roswtf!

ROS also has another tool to detect potential problems in all the elements of a given package. Just move with `roscd` to the package you want to analyze, and then run `roswtf`. In the case of `chapter3_tutorials`, we have the following output:



```
user@pc$ roswtf
Loaded plugin tf.tfwtf
Package: chapter3_tutorials
================================================================================
Static checks summary:

Found 1 warning(s).
Warnings are things that may be just fine, but are sometimes at fault

WARNING ROS_IP may be incorrect: ROS_IP [localhost] does not appear to be a local IP address ['127.0.0.1', '192.168.1.35'].

Found 1 error(s).

ERROR The following packages have rpath issues in manifest.xml:
 * chapter3_tutorials: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"
 * geometry_msgs: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"
 * std_msgs: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"
 * roscpp: found flag "-L/opt/ros/fuerte/lib", but no matching "-Wl,-rpath,/opt/ros/fuerte/lib"

================================================================================
Beginning tests of your ROS graph. These may take awhile...
analyzing graph...
... done analyzing graph
running graph rules...
... done running graph rules

Online checks summary:

Found 3 error(s).

ERROR Communication with [/example4] raised an error:
ERROR Communication with [/rosout] raised an error:
ERROR The following nodes should be connected but aren't:
 * /example4->/rosout (/rosout)
```

Normally, we should expect no error or warning but some of them are innocuous. In the preceding screenshot, we see that `roswtf` has detected that it was not able to connect with the `example4` node. This happens because this node has a `sleep` instruction, and if analyzed, this might occur while sleeping. The other errors are a consequence of this one. The purpose of `roswtf` is to signal potential problems, and then we are responsible for checking whether they are real or meaningless ones, as in the previous case.

Plotting scalar data

Scalar data can easily be plotted with some generic tools already available in ROS. Scalar data cannot be plotted, rather each scalar field has to be plotted separately. This is the reason we talk about scalar data because most nonscalar structures are better represented with ad hoc visualizers, some of which we will see later; for instance, images, poses, and orientation/attitude.

Creating a time series plot with rxplot

In ROS, scalar data can be plotted as a time series over the time provided by the timestamps of the messages. Then, we will plot our scalar data in the y axis. The tool to do so is `rxplot`. It has a powerful argument syntax that allows us to specify several fields of a structured message (in a concise manner as well).

To show `rxplot` in action, we are going to use the `example4` node since it publishes a scalar and a vector (nonscalar) in two different topics, which are `temp` and `accel` respectively. The values put in these messages are synthetically generated, so they have no actual meaning but are useful for plotting demonstration purposes. So, start by running the node with:

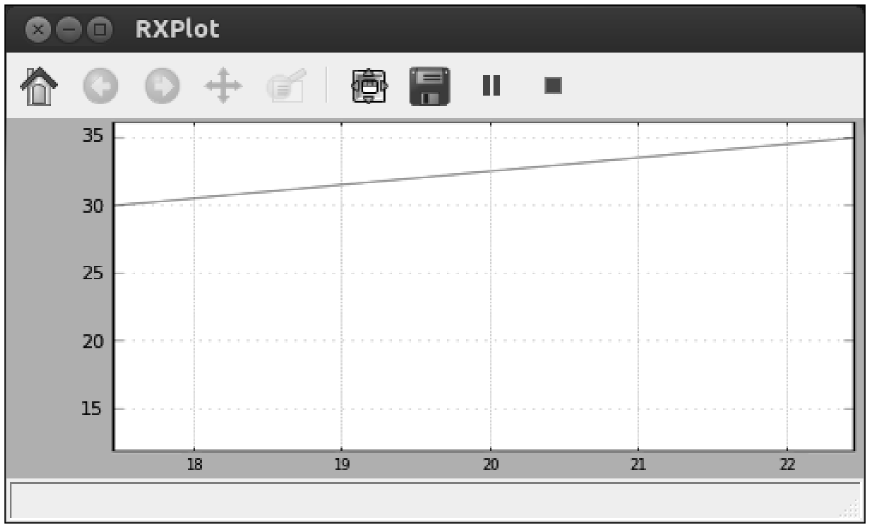
```
rosrun chapter3_tutorials example4
```

With `rostopic list`, you will see the topics `temp` and `accel` available. Now, instead of the typical `rostopic echo <topic>`, we will use `rxplot` so that we can see the values graphically over time.

To plot a message, we must know its format; use `rosmg show <msg type>` if you do not know it. In the case of scalar data, we always have a field called `data` that has the actual value. Hence, for the `temp` topic, which is of the type `Int32`, we will use:

```
rxplot /temp/data
```

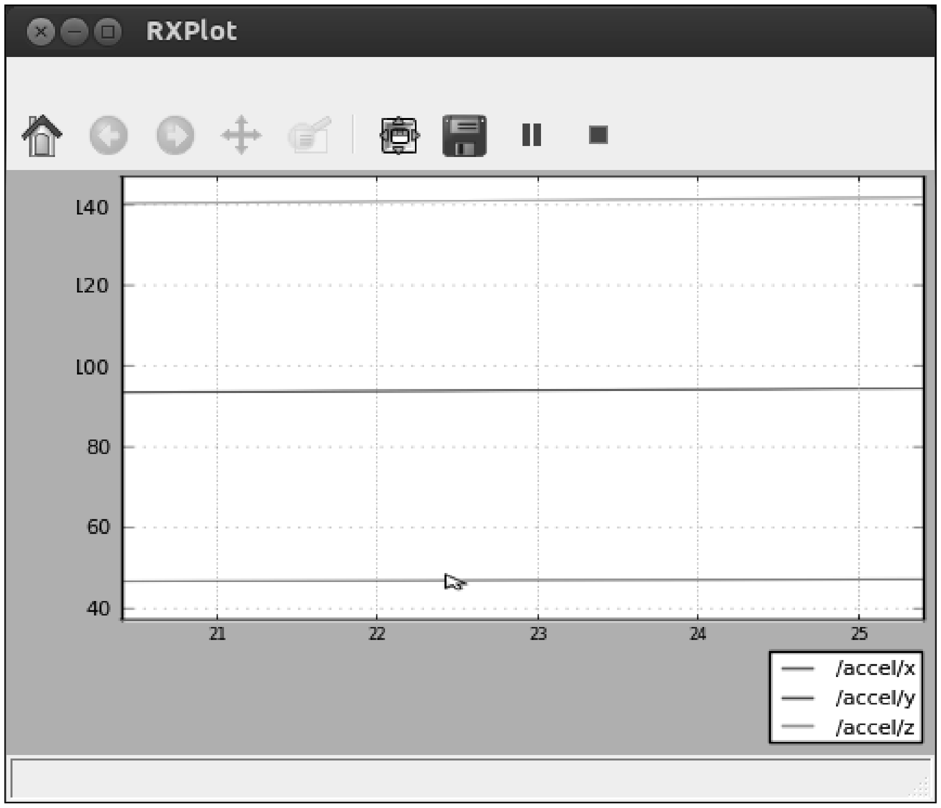
With the node running, we will see a plot that changes over time with incoming messages as shown in the following screenshot:



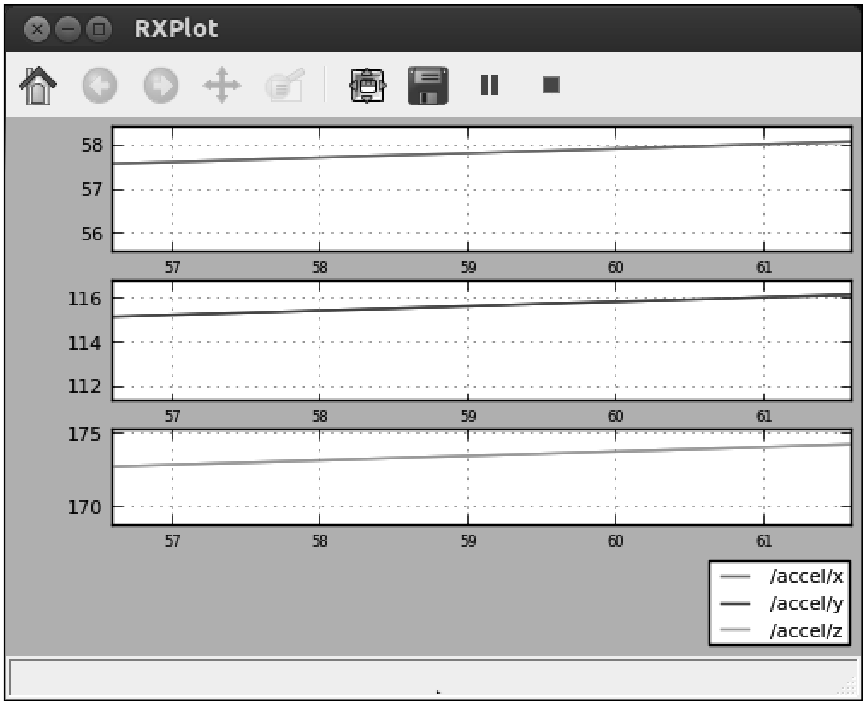
For the `accel` topic provided by the example node, in which we have a `Vector3` message (as you can check with `rostopic type /accel`), we can plot three fields of the vector in a single plot, which is a great feature of `rxplot`. The `Vector3` message has the fields x, y, and z. We can specify the fields separated by commas (,) or in a more concise manner as follows:

```
rxplot /accel/x:y:z
```

The plot will look like this:



We can also plot each field in separate axes as shown in the next screenshot. To do so, we separate each field by a blank space; remember that when you use commas, you must not insert any spaces. Therefore, if we run `rxplot /accel/x /accel/y /accel/z`, the plot will show like this:



Other plotting utilities – rxtools

The `rxplot` tool is part of the `rxtools` package along with other tools. You might go to this package to see more GUI or batch tools that can help in the development of robotic applications and the process of debugging, monitoring, and introspecting. It is also important to know that being a node (inside this package), these tools can also be run from a `launch` file.

In the case of `rxplot`, in order to run it from a `launch` file, we must put the following code inside it:

```
<node pkg="rxtools" type="rxplot" name="accel_plot"
            args="/accel/x:y:z"/>
```

Note that we use the `args` argument of the `node` element in the `launch` file to pass `rxplot` the arguments.

Visualization of images

In ROS, we have a node that allows us to show images coming from a camera on the fly. You only need a camera to do this. It is also possible to reproduce a video with a simple node in ROS but here we are going to use your laptop's webcam. The `example6` node implements a basic camera capture program using OpenCV and ROS bindings to convert `cv::Mat` images into ROS image messages that can be published in a topic. This node publishes the camera frames in the `/camera` topic.

We are only going to run the node with a launch file created to do so. The code inside the node is still new for the reader, but in the upcoming chapters, we will cover how to work with cameras and images in ROS so that we can come back to this node and understand every bit of the code:

```
roslaunch chapter3_tutorials example6.launch
```

Once the node is running, we can list the topics (`rostopic list`) and see that the `/camera` topic is there. A straightforward way to see that we are actually capturing images is to see at which frequency we are receiving images in the topic with `rostopic hz /camera`. It should be something like 30 Hz usually, but at least some value must be seen:

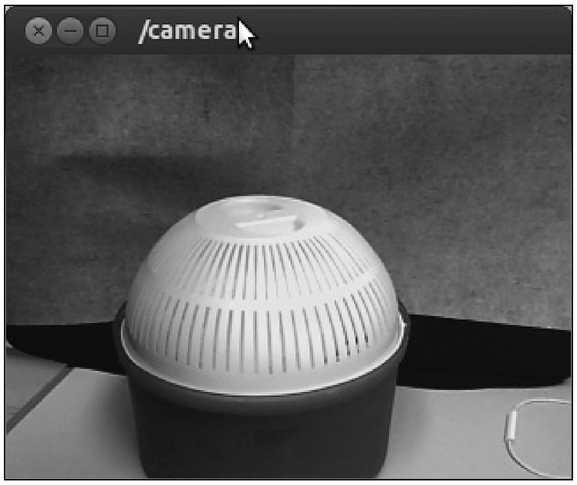
```
subscribed to [/camera]
average rate: 30.131
min: 0.025s max: 0.045s std dev: 0.00529s window: 30
```

Visualizing a single image

Being an image, we cannot use `rostopic echo /camera` because the amount of information in plain text would be very huge and also difficult to analyze. Hence, we are going to use the following code:

```
rosrun image_view image_view image:=/camera
```

This is the `image_view` node, which shows the images in the given topic (the `image` argument) in a window, as shown in the following screenshot:



This way we can visualize every image or frame published in a topic in a very simple and flexible manner, even over a network. If you right-click on the window, you can save the current frame in the disk, usually in your home directory or `~/.ros`.

FireWire cameras

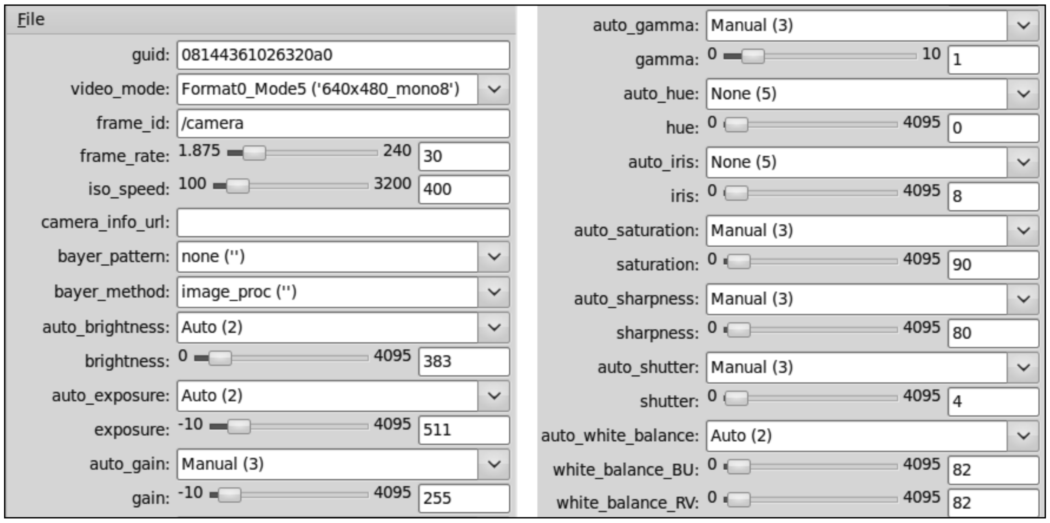
In the case of FireWire cameras, ROS also provides a larger set of tools that support calibration, both mono and stereo, as well as a way to change the camera parameters dynamically with the `reconfigure_gui` node. Usually, FireWire cameras allow changing some configuration parameters of the sensor, such as the frame rate, shutter speed, and brightness. ROS already comes with a driver for FireWire (IEEE 1394, a and b) cameras that can be run using the following command:

```
rosrun camera1394 camera1394_node
```

Once the camera is running, we can configure its parameters with the `reconfigure_gui` node, in which the first thing we do is the selection of the node we want to configure. We only have to run this:

```
rosrun dynamic_reconfigure reconfigure_gui
```

We will see an interface with all the configuration parameters and a series of slider or comboboxes, depending on the data type, to set its value within the valid limits. The following screenshot illustrates this for a particular FireWire camera:


File

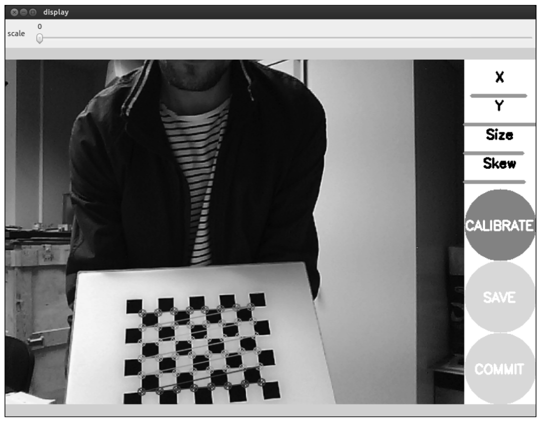
guid:	08144361026320a0		
video_mode:	Format0_Mode5 ('640x480_mono8')		
frame_id:	/camera		
frame_rate:	1.875	240	30
iso_speed:	100	3200	400
camera_info_url:			
bayer_pattern:	none ('')		
bayer_method:	image_proc ('')		
auto_brightness:	Auto (2)		
brightness:	0	4095	383
auto_exposure:	Auto (2)		
exposure:	-10	4095	511
auto_gain:	Manual (3)		
gain:	-10	4095	255
auto_gamma:	Manual (3)		
gamma:	0	10	1
auto_hue:	None (5)		
hue:	0	4095	0
auto_iris:	None (5)		
iris:	0	4095	8
auto_saturation:	Manual (3)		
saturation:	0	4095	90
auto_sharpness:	Manual (3)		
sharpness:	0	4095	80
auto_shutter:	Manual (3)		
shutter:	0	4095	4
auto_white_balance:	Auto (2)		
white_balance_BU:	0	4095	82
white_balance_RV:	0	4095	82

Note that we will cover how to work with cameras in later chapters. Also, note that the parameters reconfiguration, from the developer point of view, will be explained in detail in *Chapter 6, Computer Vision*.

Another important utility that ROS gives to the user is the possibility to calibrate the camera. It has a calibration interface built on top of the OpenCV calibration API. We will also cover this in *Chapter 6, Computer Vision*, when we see how to work with cameras. This tool is very simple to use; so, once the camera is running, we only have to show some views of a calibration pattern (usually a checkerboard) using this:

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108
image:=/camera/image_raw camera:=/camera
```

You can see the checkerboard pattern in the following screenshot:



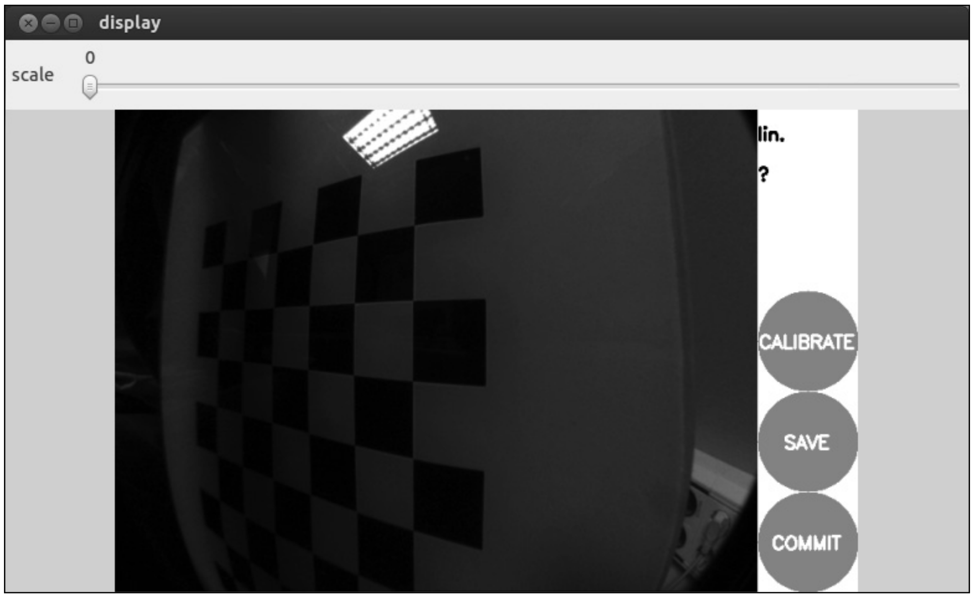
After the calibration, the result will be the so-called camera matrix and distortion coefficients along with the views used to compute it. Although we will see this later in the book, it is important to say that in the case of the FireWire camera, all the calibration information is saved in a file that is pointed by the camera configuration. Hence, the ROS system allows seamless integration so that we can use the `image_proc` tool to rectify the images; that is, to correct the distortion as well as to de-Bayer the raw images if they were in Bayer.

Working with stereo vision

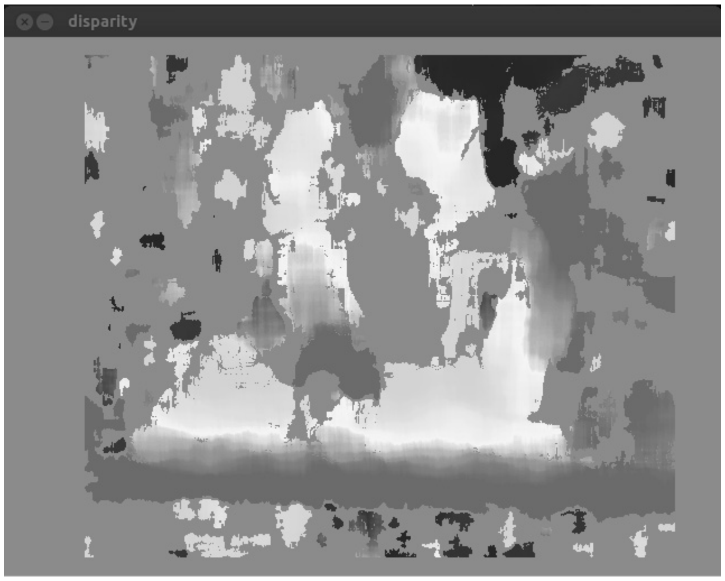
To some extent, ROS also supports stereo vision. If you have a stereo pair, you can calibrate both cameras simultaneously as well as the baseline between them. For this, we will use:

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108
right:=/my_stereo/right/image_raw left:=/my_stereo/left/image_raw right_
camera:=/my_stereo/right left_camera:=/my_stereo/left
```

This command runs the Python camera calibrator node and receives two initial parameters that indicate the type of calibration pattern. In particular, the size specified is the number of inner corners (8 x 6 in the example) and the dimensions of the square cells. Then, the topics that publish the right and left raw images and the respective camera information topics are given. In this case, the interface will show the images from the left and right cameras and the results will be for each as well. They will include the baseline as well that is useful for some stereo tools.



The stereo-specific tools allow you to compute the disparity image (refer to the following image), which is actually a way to obtain a 3D point cloud that represents the depth of each pixel in the real world. Therefore, the calibration of the camera and their baseline gives a 3D point cloud, up to some error and noise distribution that represents the real 3D position of each pixel in the world along with its color (or texture).



Similar to monocular cameras, we can generate the disparity image using stereo along with the rectified left and right images; in this case, using `stereo_image_proc`.

3D visualization

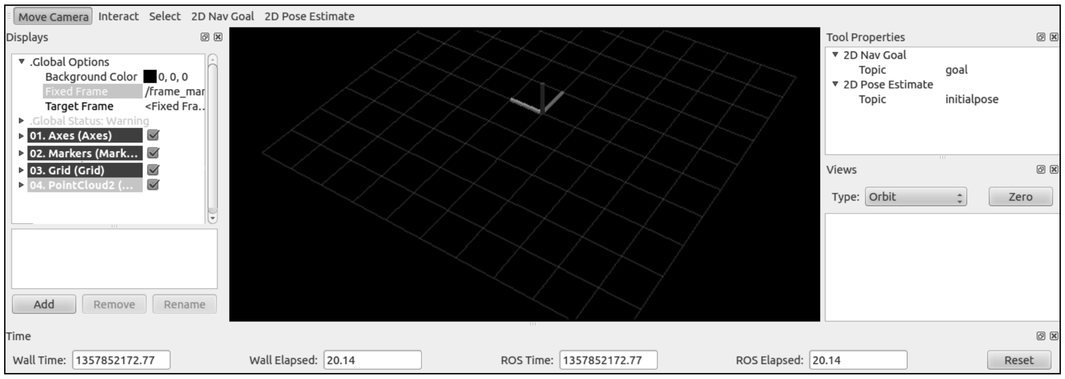
As we have seen in the previous section, there are some devices (such as stereo cameras, 3D laser, and the Kinect sensor) that provide 3D data, usually in the form of point clouds (organized or not). For this reason, it is extremely useful to have tools that visualize this type of data. In ROS, we have rviz, which we will see in the following section, that integrates an OpenGL interface with a 3D world that represents sensors' data in a modeled world. To do so, we will see that, for complex systems, the frame of each sensor and the transformations among them is of crucial importance.

Visualizing data on a 3D world using rviz

With `roscore` running, we only have to execute the following code to start rviz:

```
rosrun rviz rviz
```

We will see the graphical interface in the following screenshot:



Background Color 0, 0, 0
Fixed Frame /frame_mar
Target Frame <Fixed Fra..
Topic goal
Topic initialpose
Type: Orbit
Wall Time: 1357852172.77
Wall Elapsed: 20.14
ROS Time: 1357852172.77
ROS Elapsed: 20.14

To the left, we have the **Displays** pane, in which we have a tree list of the different elements in the world, which appears in the middle. In this case, we have some elements that are already loaded. Indeed, this configuration or layout is saved in the `config/example7.cvg` file, which can be loaded by navigating to **File** | **Open Config**.

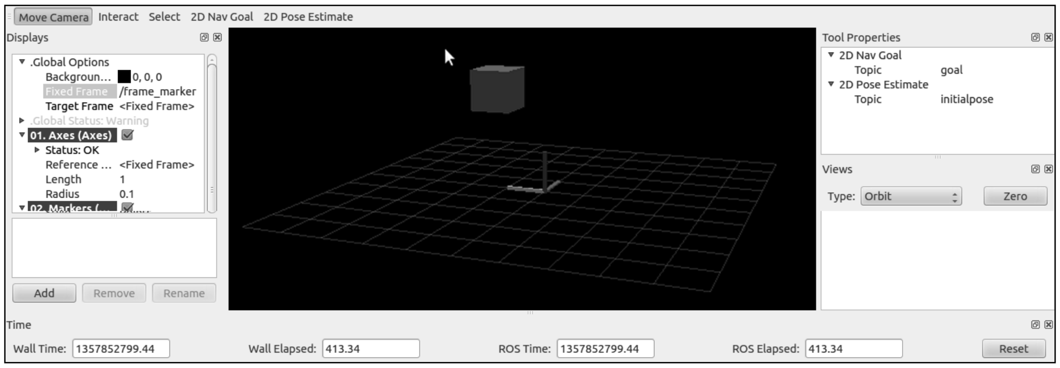
Below the **Displays** area, we have the **Add** button that allows the addition of more elements. Also, note that there are some global options, which are basically tools to set the fixed frame in the world with respect to which others might move. Then, we have **Axes (Axes)** and **Grid (Grid)** as a reference for the rest of the elements. In this case, for the `example7` node, we are going to see **Markers (Markers)** and **PointCloud2 (PointCloud2)**.

Finally, at the status bar, we have information regarding time, and to the right are the menus for the way to navigate in the world and select and manipulate elements.

Now we are going to run the `example7` node:

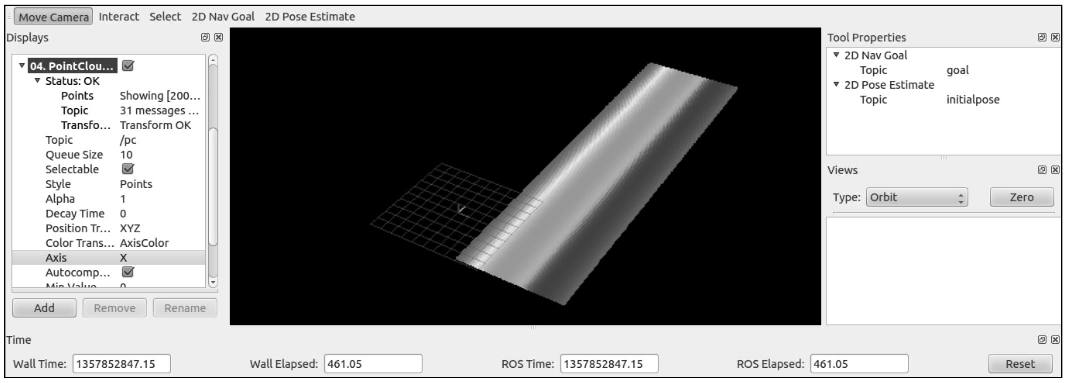
```
roslaunch chapter3_tutorials example7.launch
```

In rviz, we are going to set `frame_id` of the marker, that is `frame_marker`, in the fixed frame. We will see a red cube marker moving as shown in the following screenshot:



```
.Global Options
    Backgroun...   0, 0, 0
    Fixed Frame    /frame_marker
    Target Frame   <Fixed Frame>
  .Global Status: Warning
01. Axes (Axes)
    Status: OK
    Reference ...  <Fixed Frame>
    Length         1
    Radius         0.1
02. Markers (...

2D Nav Goal
    Topic          goal
2D Pose Estimate
    Topic          initialpose

Type: Orbit

Wall Time: 1357852799.44   Wall Elapsed: 413.34   ROS Time: 1357852799.44   ROS Elapsed: 413.34
```

Similarly, if we set the fixed frame to `frame_pc`, we will see a point cloud that represents a plane of 200 x 100 points as shown in the following screenshot:



```
04. PointClou...
    Status: OK
        Points       Showing [200...
        Topic        31 messages ...
        Transfo...   Transform OK
    Topic            /pc
    Queue Size       10
    Selectable       ✓
    Style            Points
    Alpha            1
    Decay Time       0
    Position Tr...   XYZ
    Color Trans...   AxisColor
    Axis             X
    Autocomp...      ✓
    Min Value        0

2D Nav Goal
    Topic          goal
2D Pose Estimate
    Topic          initialpose

Type: Orbit

Wall Time: 1357852847.15   Wall Elapsed: 461.05   ROS Time: 1357852847.15   ROS Elapsed: 461.05
```

The list of supported built-in types in rviz also includes cameras and images, which are shown in a window similar to **image_view**. In the case of the camera, its calibration is used and in the case of stereo images it allows us to overlay the point cloud. We can also see laser scan data from range lasers and range cone values from IR/SONAR sensors.

Basic elements can also be represented, such as a polygon, several kinds of markers, a map (usually a 2D occupancy grid map), and even interactive marker objects, which allow users to set a pose (position and orientation) in the 3D world.

For the navigation stack that we will cover in the next chapters, we have several data types that are also supported such as odometry (plots the robot odometry poses), and path (draws the path followed by the robot), which is a succession of pose objects. Among other types, it is also worth mentioning the robot model, which shows the CAD model of all the robot parts, taking into account the transformation among the frame of each element. Indeed, **TF** (**transform frame**) elements can also be drawn, which is very useful for debugging the frames in the system; we will see an example in the next section.

This 3D graphical interface can also be embedded in the new `rqt_gui` GUI. We can also develop plugins for new types, and much more. However, we believe that the information given here is usually enough, and we recommend you consult the rviz documentation for further details and advanced topics.

The relationship between topics and frames

All topics must have a frame if they are publishing data from a particular sensor that have a physical location in the real world; for example, an accelerometer that is located in some position with respect to the mass center of the robot. If we integrate the accelerations to estimate the robot's velocities or its pose, we must take the transformation between the base (mass center) and the accelerometer frames. In ROS, the messages with a header, apart from the timestamp (also extremely important to put or synchronize different messages), can be assigned `frame_id`, which gives a name to the frame it belongs to.

But the frames itself are not very useful when we have more than a single device in our robot, each in a different frame/pose. We need the transformation among them. Actually, we have a frame transformation tree that usually has the base frame as its root. Then, we can see in rviz how this and other frames move with respect to the world frame.

Visualizing frame transformations

To illustrate how to visualize the frame transformations, we are going to use the turtlesim example. Run the following `launch` file then:

```
roslaunch turtle_tf turtle_tf_demo.launch
```

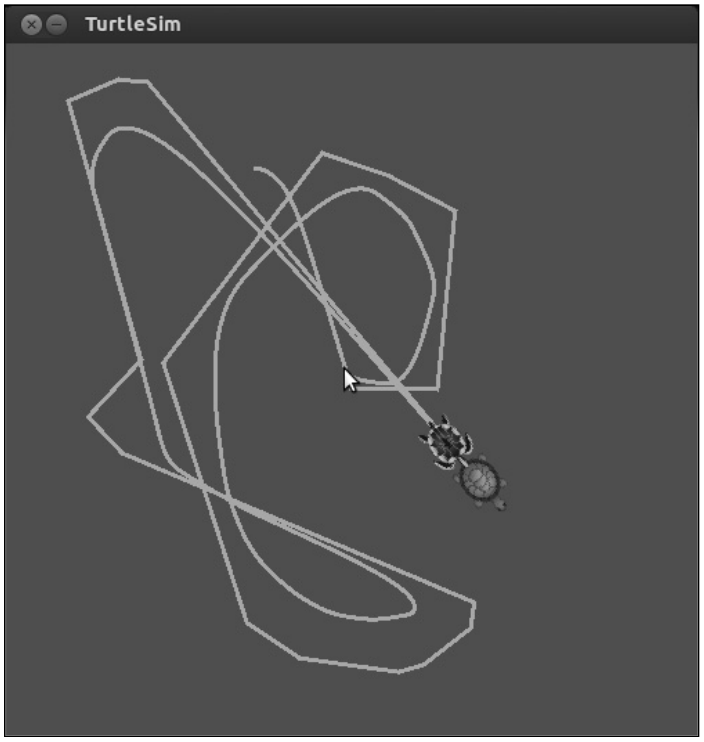
This is a very basic example with the purpose of illustrating the TF visualization in rviz. Note that for the different possibilities offered by the TF API, you should refer to later chapters of this book, in particular *Chapter 7, Navigation Stack – Robot Setups* and *Chapter 8, Navigation Stack – Beyond Setups*. For now, it is enough to know that they allow making the computations in one frame and then transforming them to another, including time delays.

It is also important to know that TFs are published at a certain frequency in the system, so it is like a subsystem where we can traverse the TF tree to obtain the transformation between any frames in it, and we can do it in any node of our system just by consulting TF.

If you receive an error, it is probably because the listener died on the launch startup, as another node that was required was not yet ready; so, please run the following on another terminal to start it again:

```
rosrun turtle_tf turtle_tf_listener
```

Now you should see a window with two turtles (the icon might differ) where one follows the other. You can control one of the turtles with the arrow keys but with the focus on the terminal for which the `launch` file is run. The following screenshot shows how one turtle has been following the other, after moving the one we can control for some time:

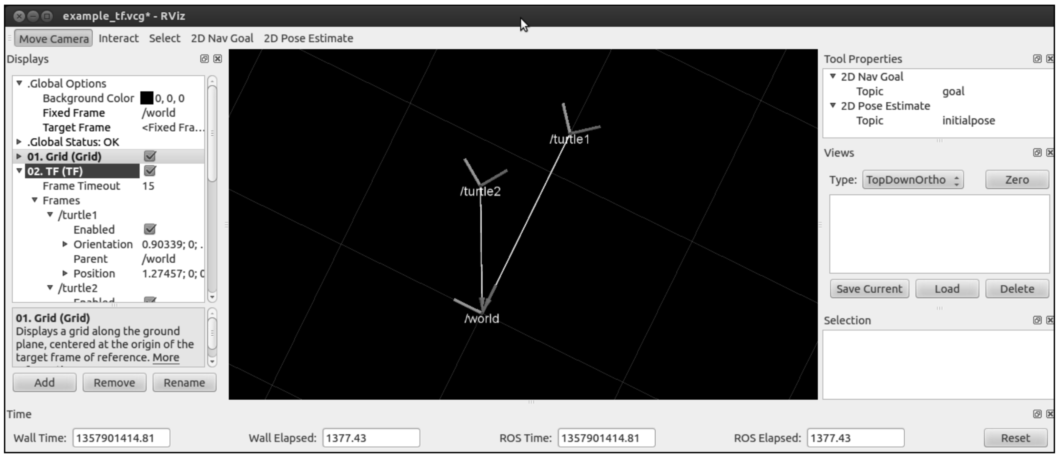


Each turtle has its own frame. We can see them in `rviz`:

```
rosrun rviz rviz
```

Now, instead of **TurtleSim**, we are going to see how the turtles' frames move in `rviz` while we move our turtle with the arrow keys. We have to set the fixed frame to `/world`, and then add the TF tree to the left area. We will see that we have the `/turtle1` and `/turtle2` frames, both as children of the `/world` frame. In the world representation, the frames are shown as axes. The `/world` frame is fixed because we configured it as such in `rviz`. It is also the root frame and the parent of the turtles' frames. This is represented with a yellow arrow that has a pink end. Also, set the view of the world to `TopDownOrtho` because this makes it easier to see how the frames move in this case, as they move only on the ground (2D plane). Also, you may find it useful to translate the world center, which is done with the mouse, as you do to rotate, but with the *Shift* key pressed.

In the following screenshot, you can see how the two frames of each turtle are shown with respect to the `/world` frame. We advise the user to play with the example to see it in action, in real time. Also, you might change the fixed frame. Note that `config/example_tf.vcg` is provided to give the basic `rviz` configuration used in this example.



```
▼ .Global Options
    Background Color   0, 0, 0
    Fixed Frame        /world
    Target Frame       <Fixed Fra...
▶ .Global Status: OK
▶ 01. Grid (Grid)      ✓
▼ 02. TF (TF)          ✓
    Frame Timeout      15
  ▼ Frames
    ▼ /turtle1
        Enabled        ✓
      ▶ Orientation    0.90339; 0; .
        Parent         /world
      ▶ Position       1.27457; 0; 0
    ▼ /turtle2

01. Grid (Grid)
Displays a grid along the ground plane, centered at the origin of the target frame of reference. More

▼ 2D Nav Goal
    Topic     goal
▼ 2D Pose Estimate
    Topic     initialpose

Type: TopDownOrtho

Wall Time: 1357901414.81   Wall Elapsed: 1377.43   ROS Time: 1357901414.81   ROS Elapsed: 1377.43
```

Saving and playing back data

Usually, when we work with robotic systems, the resources are shared, not always available, or the experiments cannot be done regularly because of the cost or time required to prepare and perform them. For this reason, it is good practice to record the data of the experiment session for future analysis and to work, develop, and test our algorithms. However, the process of saving good data so that we can reproduce the experiment offline is not trivial. Fortunately, we have powerful tools in ROS that already solve this problem.

ROS can save all messages published by the nodes through the topics. It has the ability to create a bag file that contains the messages as they are with all their fields and timestamps. This allows reproducing the experiment offline and simulating the real condition, which is the latency of message transmission. Moreover, ROS tools do all this efficiently with a high bandwidth and an adequate manner to organize the saved data.

In the next section, we explain the tools provided by ROS to save and playback the data stored in bag files, which use a binary format designed for and by ROS developers. We will also see how to manage these files; that is, inspect the content (number of messages, topics, and so on), compress, and split or merge several of them.

What is a bag file?

A bag file is a container of messages sent by topics that are recording during a session using a robot or some nodes. In brief, they are the logging files for the messages transferred during the execution of our system and allow us to playback everything even with the time delays, since all messages are recorded with a timestamp; not only for the timestamp in the header but also for the packets that have it. The difference between the timestamp used for recording and the one in the header is that the first one is set by the message that is recorded, while the other is set by the producer/publisher of the message.

The data stored in a bag file is in the binary format. The particular structure of this container allows for an extremely fast recording bandwidth, which is the most important concern while saving data. Also, the size of the bag file is relevant but is usually at the expense of speed. Anyway, we have the option to compress the file on the fly with the **bz2** algorithm; just use the `-j` parameter when you record with `rosbag record`, as you will see in the following section.

Every message is recorded along with the topic that published it. Therefore, we can specify which topics to record or just mention all (with `-a`). Later, when we play the bag file back, we can also select a particular subset of topics of all the ones in the bag file by indicating the names of the topics we want to be published.

Recording data in a bag file with rosbag

The first thing we have to do to start is simply record the data. We are going to use a very simple system, our `example4` node, as an example. Hence, we first run the node:

```
rosrun chapter3_tutorials example4
```

Now we have two options. First, we can record all the topics:

```
rosbag record -a
```

Or, second, record only some specific (user-defined) topics. In this case, it will make sense to record only the `example4` topics, so we will use the following:

```
rosbag record /temp /accel
```

By default, when we run the preceding command, the `rosbag` program subscribes to the node and starts recording the message in a bag file in the current directory with data as the name. Once you have finished the experiment or you want to stop recording, you only have to hit *Ctrl* + *C*. The following is an example of recording the data and the resulting bag file:

```
[ INFO] [1357815371.768263730]: Subscribing to /temp
[ INFO] [1357815371.771339658]: Subscribing to /accel
[ INFO] [1357815371.774950563]: Recording to 2013-01-10-10-56-11.bag.
```

You can see more options with `rosbag help record` that includes things such as the bag file size, the duration of the recording, and options to split the files into several ones of a given size. As we have mentioned before, the file can be compressed on the fly (using the `-j` option). In our honest opinion, this is only useful for small bandwidths because it also consumes some CPU time and might produce some message dropping. Also, we can increase the buffer (`-b`) size for the recorder in MB, which defaults to 256 MB, but it can be increased to some GB if the bandwidth is very high (especially with images).

It is also possible to include the call to `rosbag record` into a launch file. To do so, we must add a node like this:

```
<node pkg="rosbag" type="record" name="bag_record"
            args="/temp /accel"/>
```

Note that the topics and other arguments to the command are passed using the `args` argument. Also, it is important to say that when running from the `launch` file, the bag file is created by default in `~/.ros`, unless we give the name of the file with `-o` (prefix) or `-O` (full name).

Playing back a bag file

Now that we have a bag file recorded, we can use it to play back all the messages of the topics inside it. We need `roscore` running and nothing else. Then, we move to the folder with the bag file we want to play (there are two examples in the `bag` folder of this chapter's tutorials) and do this:

```
rosbag play 2013-01-10-10-56-11.bag
```

We will see the following output:

```
[ INFO] [1357820252.241049890]: Opening bag/2013-01-10-10-56-11.bag


Waiting 0.2 seconds after advertising topics... done.


Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time: 1357815375.147705   Duration: 2.300787 / 39.999868
```

In the terminal where we are playing the bag file, we can pause (hit Space bar) or move step by step (hit *S*), and, as usual, use *Ctrl* + *C* to finish it immediately. Once we reach the end of the file, it will close, but there is an option to loop (`-l`) that sometimes might be useful.

Automatically, we will see the topics with `rostopic list`:

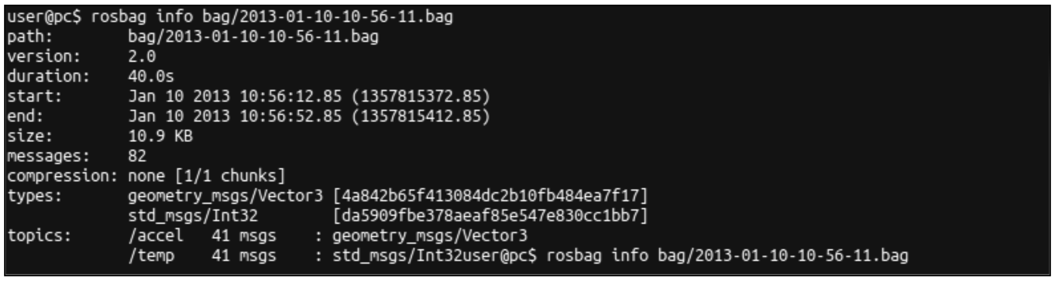
```
/accel
/clock
/rosout
/rosout_agg
/temp
```

The `/clock` topic is part of the fact that we can instruct the system clock to simulate a faster playback. This can be configured using the `-r` option. In the `/clock` topic, the time for simulation at a configurable frequency with the `--hz` argument (it defaults to 100 Hz) is published.

Also, we could specify a subset of the topics in the file to be published. This is done with the `--topics` option. In order to see what we have inside the file, we would use `rosbag info <bag_file>`, which we will explain in the next section.

Inspecting all the topics and messages in a bag file using rxbag

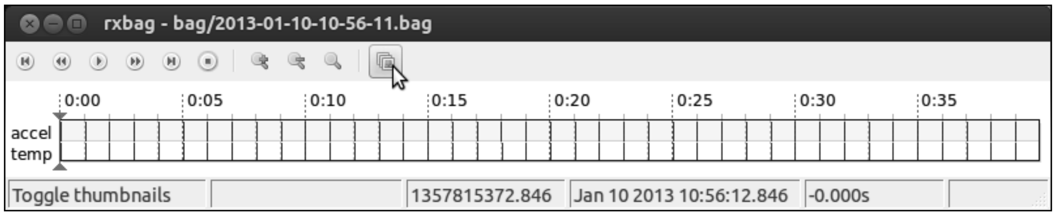
There are two main ways to see what we have inside a bag file. The first one is very simple. We just type `rosbag info <bag_file>` and the result is something like this:



```
user@pc$ rosbag info bag/2013-01-10-10-56-11.bag
path:        bag/2013-01-10-10-56-11.bag
version:     2.0
duration:    40.0s
start:       Jan 10 2013 10:56:12.85 (1357815372.85)
end:         Jan 10 2013 10:56:52.85 (1357815412.85)
size:        10.9 KB
messages:    82
compression: none [1/1 chunks]
types:       geometry_msgs/Vector3 [4a842b65f413084dc2b10fb484ea7f17]
             std_msgs/Int32        [da5909fbe378aeaf85e547e830cc1bb7]
topics:      /accel    41 msgs    : geometry_msgs/Vector3
             /temp     41 msgs    : std_msgs/Int32user@pc$ rosbag info bag/2013-01-10-10-56-11.bag
```

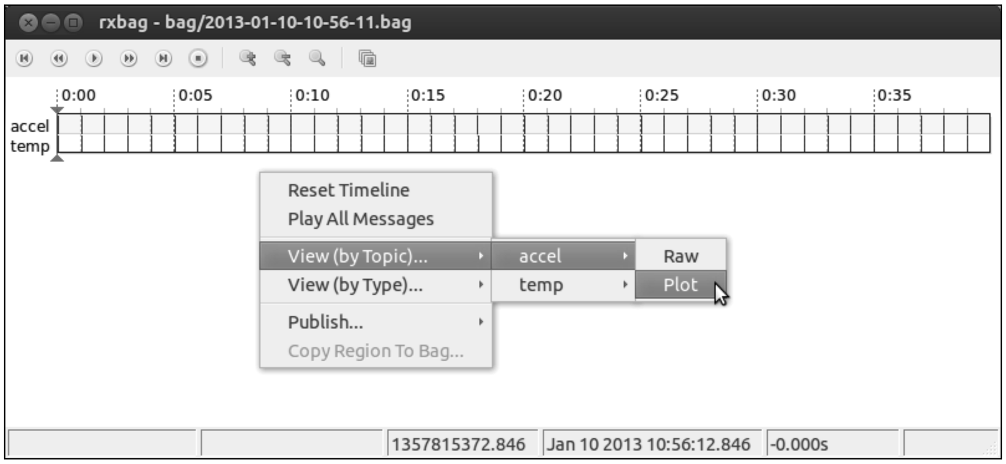
We have information about the bag file itself, such as the creation date, duration, size, as well as the number of messages inside, and the compression (if any). Then, we have the list of data types inside the file, and finally the list of topics with their corresponding name, number of messages, and type.

The second way to inspect a bag file is extremely powerful. It is a graphical interface named **rxbag** that also allows playing back the files, viewing the images (if any), plotting scalar data, and also the raw structure of the messages. We only have to pass the name of the bag file, and we will see something like the following screenshot (for the previous bag file):

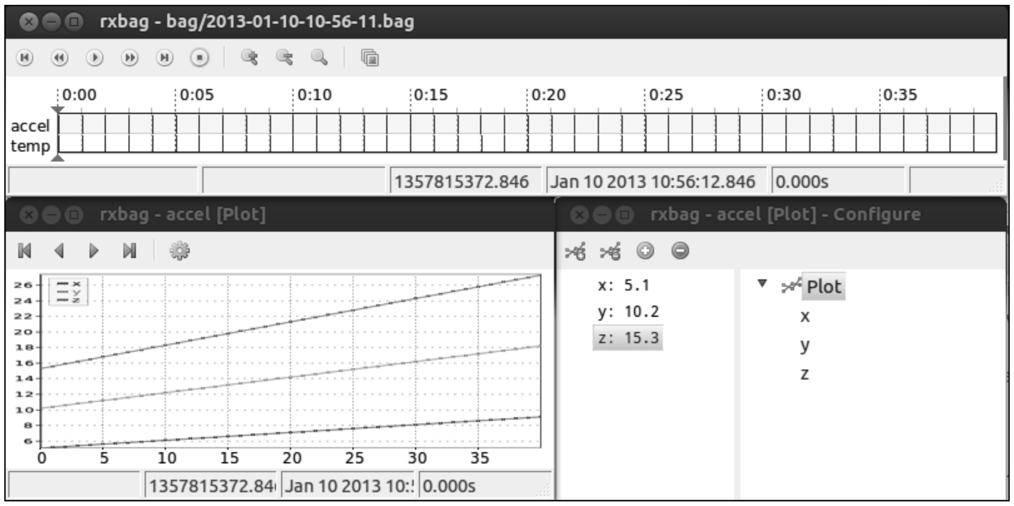


We have a timeline for all the topics where each message appears with a mark. In the case of images, we can enable the thumbnails to see them in the timeline (marked with the mouse pointer).

In the following screenshot, we can see how to access the **Raw**, **Plot**, and **Image** (if the topic is of the type Image) views for the topics in the file. This pop-up menu appears with a right-click over the timeline.



For /accel, we can plot all the fields in a single axis. To do so, once we are in the Plot view, we click on the gear button/icon and then select every field. Note that we can remove them later or create a different axis (in the bottom-right window). The plot is generated for all the values in the file, and a vertical line shows the current position in the playback.



Note that we must have clicked on the **Play** button at least once to be able to plot the data. Then we can play, pause, stop, and move to the beginning or the end of the file.

The images are straightforward, and a simple window appears with the current frame with options to save them as image files in the disk.

rqt plugins versus rx applications

Since ROS Fuerte, the `rx` applications or tools are deprecated and we should instead use the `rqt` nodes. They are basically the same, only with a few of them incorporated with small updates, bug fixes, and new features. Also, they can be loaded as plugins into a single window/application, which is `rqt_gui`. We show the equivalent for the tools shown in this chapter in the following list:

- `rxconsole` is replaced by `rosrun rqt_console rqt_console`
- `rxgraph` is replaced by `rosrun rqt_graph rqt_graph`
- `rxplot` is replaced by `rosrun rqt_plot rqt_plot`
- `rxbag` is replaced by `rosrun rqt_bag rqt_bag`

Furthermore, being plugins that can also be run standalone, there exist more tools, such as a shell, a topic publisher, and a message type viewer. Even rviz has a plugin named `rqt_rviz` that can be integrated in the new `rqt_gui` interface; all this is fully integrated in ROS Groovy and Hydro where `rx` tools are deprecated but still in the bundle. The same happens for ROS Fuerte, which was the first release to incorporate the `rqt` tools.

Summary

After reading and running the code of this chapter, you will have learned to use many tools that will enable you to develop robotic systems faster, debug errors, and visualize your results so you can evaluate their quality or validate them. Some of the specific concepts and tools you will exploit the most in your life as a robotic developer are summarized as follows:

- Now you know how to include logging messages in your code with different levels of verbosity, which will help you debug errors in your nodes. For this purpose, you could also use the powerful tools included in ROS, such as the rxconsole interface. Additionally, you can also inspect or list the nodes running, the topics published, and the services provided in the whole system while running. This includes the inspection of the node graph using rxgraph.
- Regarding the visualization tools, you should be able to plot scalar data using `rxplot` for a more intuitive analysis of certain variables published by your nodes. Similarly, you can view more complex types (nonscalar ones). This includes images and 3D data using rviz.
- Finally, recording and playing back the messages of the topics available is now in your hands with `rosbag`. And you also know how to view the contents of a bag file with `rxbag`. This allows you to record the data from your experiments and process them later with your AI or robotics algorithms.