O'REILLY®

Early Release

RAW & UNEDITED

Programming Robots with ROS

A PRACTICAL INTRODUCTION TO THE ROBOT OPERATING SYSTEM

Morgan Quigley, Brian Gerkey
& William D. Smart

Programming Robots with ROS

Morgan Quigley, Brian Gerkey, and William D. Smart

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Programming Robots with ROS

by Morgan Quigley, Brian Gerkey, and William D. Smart

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Dawn Schanafelt and Meghan Blanchette
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition:

2015-05-29: Early release revision 1

2015-06-29: Early release revision 2

2015-07-22: Early release revision 3

See *http://oreilly.com/catalog/errata.csp?isbn=0636920024736* for release details.

ISBN: 063-6-920-02473-6

[?]

Table of Contents

Preface

ROS, the Robot Operating System, is an open-source framework for getting robots to do things. ROS is meant to serve as a common software platform for (some subset of) the people who were building and using robots. This common software lets people share code and ideas more readily and, perhaps more importantly, means that you do not have to spend years writing software infrastructure before your robots start moving!

ROS has been remarkable successful. At the time of writing, in the official distribution of ROS there are XXX software packages, written and maintained by XXX people. XXX commercially-available robots are supported, and we can find at least XXX academic papers that mention ROS. We no longer have to write everything from scratch, especially if we're working with one of the many robots that support ROS, and can spend more time thinging about *robotics*, rather than bit-fiddling and device drivers.

ROS consists of a number of parts:

1. A set of **drivers that let you read data from sensors and send commands to motors** and other actuators, in an abstracted, well-defined format. A wide variety of popular hardware is supported, including a growning number of commercially-available robot systems.
2. A large and growing collection of fundamental **robotics algorithms** that allow you to build maps of the world, navigate around it, represent and interpret sensor data, plan motions, manipulate objects, and a lot of other stuff. ROS has become very popular in the robotics research community, and a lot of cutting-edge algorithms are now available in ROS.
3. All of the **computational infrastructure** that allows you to move data around, to connect the various components of a complex robot system, and to incorporate your own algorithms. ROS is inherently distributed, and allows you to split the workload across multiple computers seamlessly.
4. A **large set of tools** that make it easy to visualize the state of the robot and the algorithms, debug faulty behaviors, and record sensor data. Debugging robot soft-

ware is notoriously difficult, and this rich set of tools are one of the things that make ROS as powerful as it is.

5. Finally, the larger ROS ecosystem includes an **extensive set of resources**, such as a wiki that documents many of the aspects of the framework, a question-and-answer site where you can ask for help and share what you've learned, and a thriving community of users and developers.

GOT TO HERE

So, why should you learn ROS? The short answer is because it will save you time. ROS provides all the parts of a robot software system that you would otherwise have to write. It allows you to focus on the parts of the system that you care about, without worry about the parts that you don't care about.

Why should you read this book? There's a lot of material on the ROS wiki, including detailed tutorials for many aspects of the framework. A thriving user community is ready to answer your questions on *http://answers.ros.org*. Why not just learn ROS from these resources? What we've tried to do in this book is to lay things out in a more ordered way, and to give comprehensive examples of how you can use ROS to do interesting things with real and simulated robots. We've also tried to include tips and hints about how to structure your code, how to debug your code when it causes the robot to do something unexpected, and how to become part of the ROS community.

There's a fair amount of complexity in ROS, especially if you're not a seasoned programmer; distributed computation, multi-threading, even-driven programming, and a host of other concepts lie at the heart of the system. If you're not already familiar with at least some of these, ROS can have a daunting learning curve. This book is an attempt to flatted out that curve a bit by introducing you to the basics of ROS, and giving you some practical examples of how to use it for real applications on real (and simulated) robots

Who Should Read This Book?

If you want to make your robots do things in the real world, but don't want to spend time reinventing the wheel, then this book is for you. ROS includes all of the computational infrastructure you'll need to get your robots up and running, and enough robotics algorithms to get them doing interesting things quickly.

If you're interested some aspect, like path planning, and want to investigate it in the context of a larger robot system, then this book is for you. We'll show you how to get your robot doing interesting things using the infrastructure and algorithms in ROS, and how to swap out some of the existing algorithms for your own.

If you want to get an introduction to the basic mechanisms of ROS and an overview of some of the things that are possible, but you're a bit daunted by the scale of the infor-

mation on the wiki, then this book is for you. We'll give you a tour of the basic mechanisms and tools in ROS and concrete examples of complete systems that you can build on and adapt.

Who Should Not Read This Book?

Although we don't want to exclude anyone from reading this book, it's probably not the right resource for everyone. We make certain implicit assumptions about the robots that you will be using. They are probably running Linux, and have decent computational resources (at least equivalent to a laptop computer). They have sophisticated sensors, such as a Microsoft Kinect. They are ground-based, and probably can move about the world. If your robots don't fall into at least some of these categories, the examples in this book might not be immediately relevant to you, although the material on the underlying mechanisms and tools should be.

This book is primarily about ROS, and not about robotics. While you will learn a bit about robotics here, we don't go into great depth about many of the algorithms in ROS. If you're looking for a broad introduction to robotics, then this book isn't the one you're looking for.

What You'll Learn

This book is meant to be a broad introduction to programming robots with ROS. We'll cover the important aspects of the basic mechanisms and tools that comprise the core of ROS, and show you how to use them to create software to control your robots. We'll show you concrete examples of how you can use ROS to do some interesting things with your robots, and give you advice on how to build on these examples to create your own systems.

In addition to the technical material, we'll also show you how to navigate the larger ROS ecosystem, such as the wiki and http:answers.ros.org, and how to become a part of the global ROS community, sharing your code and newly-found knowledge with other roboticists across the world.

Prerequisites

There are a few things that you need to know before you can really use the material in this book. Since ROS is a software framework, you really need to know how to program to properly understand it. Although it's possible to program in ROS in a variety of languages, in this book we're going to be using Python. If you don't know Python, then a lot of the code here isn't going to make much sense.

ROS works best in an Ubuntu Linux environment, and having some previous exposure to Linux will make your life a lot easier. We'll try to introduce the important parts of

Linux as we go, but having a basic understanding of the filesystem, the `bash` command shell, and at least one text editor will help you concentrate on the ROS-specific material.

A basic understanding of robotics, while not strictly necessary to learn ROS, will also be helpful. Knowing something about the underlying mathematics used by robotics, such as coordinate transforms and kinematic chains, will motivate some of the ROS mechanisms that we talk about. Again, we'll try to give a brief introduction to some of this material but, if you're not familiar with it, you might want to take a side-track and dig into the robotics literature to fill in some background.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://bitbucket.org/osrf/rosbook*.

This book is here to help you get your job done. To that end, the examples in the above-linked repository are available under the Apache 2.0 License, which permits very broad reuse of the code.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

CHAPTER 1
Introduction

The Robot Operating System (ROS) is a loosely-defined *framework* for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

Why? Because creating truly robust, general-purpose robot software is **hard**. From the robot's perspective, many problems which seem trivial to humans can actually encompass wild variations between instances of tasks and environments.

Consider a simple "fetch an item" task, where an office-assistant robot is instructed to retrieve a stapler. First, the robot must understand the request, either verbally or through some other user interface such as SMS, email, or a web portal. Then, the robot must start some sort of planner to coordinate the search for the item, which will likely require navigation to various rooms in a building, perhaps including elevators and doors. Once arriving in a room, the robot must search cluttered desks of similarly-sized objects, since all handheld objects are roughly the same size, and find a stapler. The robot must then retrace its steps and deliver the stapler to the desired location. Each of those subproblems can have arbitrary numbers of complicating factors. And this was a relatively simple task!

In our opinion, dealing with real-world variations in complex tasks and environments is so hard that no single individual, laboratory, or institution can hope to build a complete system from scratch. As a result, ROS was built from the ground up to encourage *collaborative* robotics software development. For example, in the "fetch a stapler" problem, one organization might have experts in mapping indoor environments, and could contribute a complex, yet easy-to-use, system for producing indoor maps. Another group might have expertise in using maps to robustly navigate indoor environments. Yet another group might have discovered a particular computer vision approach that works well for recognizing small objects in clutter. ROS includes many features specifically designed to simplify this type of large-scale collaboration.

Brief History

ROS is a large project that has many ancestors and contributors. The need for an open collaboration framework was felt by many people in the robotics research community. Various projects at Stanford University in the mid-2000s involving integrative, embodied AI, such as the STanford AI Robot (STAIR) and the Personal Robots (PR) program, created in-house prototypes of the types of flexible, dynamic software systems described in this book. In 2007, Willow Garage, a nearby visionary robotics incubator, provided significant resources to extend these concepts much further and create well-tested implementations. The effort was assisted by countless researchers who contributed their time and expertise to both the core ROS ideas and to its fundamental software packages. Throughout, the software was developed in the open using the permissive BSD open-source license, and gradually became widely used in the robotics research community.

From the start, ROS was being developed at multiple institutions and for multiple robots. At first, this seemed like a headache, since it would be far simpler for all contributors to place their code on the same servers. Ironically, over the years, this has emerged as one of the great strengths of the ROS ecosystem: any group can start their own ROS code repository on their own servers, and they will maintain full ownership and control of it. They don't need anyone's permission. If they choose to make their repository publicly visible, they can receive the recognition and credit they deserve for their achievements, and benefit from specific technical feedback and improvements like all open-source software projects.

The ROS ecosystem now consists of tens of thousands of users worldwide, working in domains ranging from tabletop hobby projects to large industrial automation systems.

Philosophy

All software frameworks seek to impose their development philosophy on their contributors, whether directly or indirectly. ROS follows the UNIX philosophy in several key aspects. This tends to make ROS feel more "natural" for developers coming from a UNIX background, but somewhat "cryptic" at first for those who have primarily used graphics development environments on Windows or Mac. The following paragraphs describe several philosophical aspects of ROS.

Peer to Peer: ROS systems consist of numerous small computer programs which connect to each other and continuously exchange *messages*. These messages travel directly from one program to another; there is no central routing service. Although this makes the underlying "plumbing" more complex, the result is a system that scales better as the amount of data increases.

Tools-based: As demonstrated by the enduring architecture of UNIX, complex software systems can be created from many small, generic programs. Unlike many other robotics

software frameworks, ROS does not have a canonical integrated development and runtime environment. Tasks such as navigating the source code tree, visualizing the system interconnections, graphically plotting data streams, generating documentation, logging data, etc., are all performed by separate programs. This encourages the creation of new, improved implementations, since (ideally) they can be exchanged for implementations better suited for a particular task domain.

Multi-Lingual: Many software tasks are easier to accomplish in "high-productivity" scripting languages such as Python or Ruby. However, there are times when performance requirements dictate the use of faster languages, such as C++. There are also various reasons that some programmers prefer languages such as LISP or MATLAB. Endless email flamewars have been waged, are currently being waged, and will always continue be waged, over which language is best suited for a particular task. In an attempt to sidestep these battles, ROS chose a *multi-lingual* approach. ROS software modules can be written in any language for which a *client library* has been written. At time of writing, client libraries exist for C++, Python, Java, JavaScript, Ruby, LISP, and MATLAB, among others. ROS client libraries can all communicate with each other by following a convention which describes how messages are "flattened" before being transmitted. This book will use the Python client library almost exclusively, to save space in the code examples and for its general ease of use. However, all of the tasks described in this book could be accomplished with any of the client libraries.

Thin: The ROS conventions encourage contributors to create stand-alone libraries and then *wrap* those libraries so they send and receive messages to/from other ROS modules. This extra layer is intended to allow the re-use of software outside of ROS for other applications, and it greatly simplifies the creation of automated tests using standard continuous-integration tools.

Free and Open-Source: The core of ROS is released under the permissive BSD license, which allows commercial and non-commercial use. ROS passes data between modules using inter-process communication (IPC), which means that systems buit using ROS can have fine-grained licensing of their various components. Commercial systems, for example, often have several closed-source modules communicating with a large number of open-source modules. Academic and hobby projects are often fully open-source. Commercial product development is often done completely behind a firewall. All of these use cases, and more, are common and perfectly valid under the ROS license.

Installation

Although ROS has been made to work on a wide variety of systems, in this book, we will be using Ubuntu Linux, a popular and relatively user-friendly Linux distribution. Ubuntu provides an easy-to-use installer that allows computers to dual-boot between the operating system they were shipped with (typically Windows or Mac OS X) and Ubuntu itself. That being said, it is important to back up your computer before installing

Ubuntu, in case something unexpected happens and the drive is completely erased in the process.

Although there are virtualization environments such as VirtualBox and VMWare that allow Linux to run concurrently with a host operating system such as Windows or Mac OS X, the simulator used in this book is rather compute- and graphics-intensive, and will likely run particularly well in virtualization. As such, we recommend running Ubuntu Linux natively by following the instructions on the Ubuntu website.

Ubuntu Linux can be downloaded freely from *http://ubuntu.com* The remainder of this book assumes that ROS is being run on Ubuntu 14.04, and will use the ROS Indigo distribution.

The ROS installation steps require a few shell commands that involve some careful typing. These can be either hand-copied from the following block, or via copy/paste from the ROS wiki, at time of writing: *http://wiki.ros.org/indigo/Installation/Ubuntu* These commands will add ros.org to the system's list of software sources, download and install the ROS packages, and set up the environment and ROS build tools.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu raring main" > \
  /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install ros-indigo-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ sudo apt-get install python-rosinstall
```

Throughout the book, we will refer to various operating-system features as "POSIX," such as, "POSIX processes," "POSIX environment variables," and so on. This is meant to indicate that much of ROS is written with *portability* in mind between POSIX-compliant systems, such as Linux or Mac OS X. That being said, in this book we will be focusing specifically on Ubuntu Linux, since it is the most popular Linux distribution for the desktop, and since the ROS build farm produces easy-to-install binaries for Ubuntu.

Summary

This chapter has provided a high-level overview of ROS and its guiding philosophical ideas: ROS is a *framework* for developing robotics software. The software is structured as a large number of small programs that rapidly pass messages to each other. This paradigm was chosen to encourage the re-use of robotics software outside the particular robot and environment that drove its creation. Indeed, this loosely-coupled structure allows for the creation of *generic* modules that are applicable to broad classes of robot

hardware and software pipelines, facilitating code sharing and re-use among the global robotics community.

CHAPTER 2
Preliminaries

Before we look at how to write code in ROS, we're going to take a moment to disucss some of the key concepts that underly the framework. ROS systems are organized as a *computation graph*, where a number of independent programs each perform some piece of computation, passing data and results to other programs, arranged in a pre-defined network. In this chapter we'll discuss this graph architecture, and look at the command-line tools that you're going to be using to interact with it. We'll also discuss the details of the naming schemes and namespaces used by ROS, and how these can be tailored to promote reuse of your code.

The ROS Graph

As mentioned in the previous chapter, one of the original "challenge problems" which motivated the design of ROS was fondly referred to as the "fetch a stapler" problem. Imagine a relatively large and complex robot with several cameras and laser scanners, a manipulator arm, and a wheeled base. In the "fetch a stapler" problem, the robot's task is to navigate a typical home or office environment, find a stapler, and deliver it to a person who needs one. Exactly why a person would be working in a building equipped with large, complex robots but lacking a sufficient stockpile of staplers is the logical first question, but will be ignored for the purposes of this discussion. There are several key insights which can be gained from the "fetch a stapler" problem:

- the task can be decomposed into many independent subsystems, such as navigation, computer vision, grasping, and so on;
- it should be possible to re-purpose these subsystems for other tasks, such as doing security patrols, cleaning, delivering mail, and so on; and
- with proper hardware and geometry abstraction layers, the vast majority of the software should be able to run on *any* robot.

These principles can be illustrated by the fundamental rendering of a ROS system: its *graph*. A ROS system is made up of many different programs, running simultaneously, which communicate with each other by passing *messages*. Visualizing this as a *graph*, the programs are the *nodes*, and programs that communicate with each other are connected by edges, as shown in Figure 2-1. We can visualize any ROS system, large or small, in this way. In fact, this visualization is so useful that we actually call all ROS programs *nodes*, to help remember that each program is just one piece of a much larger system.

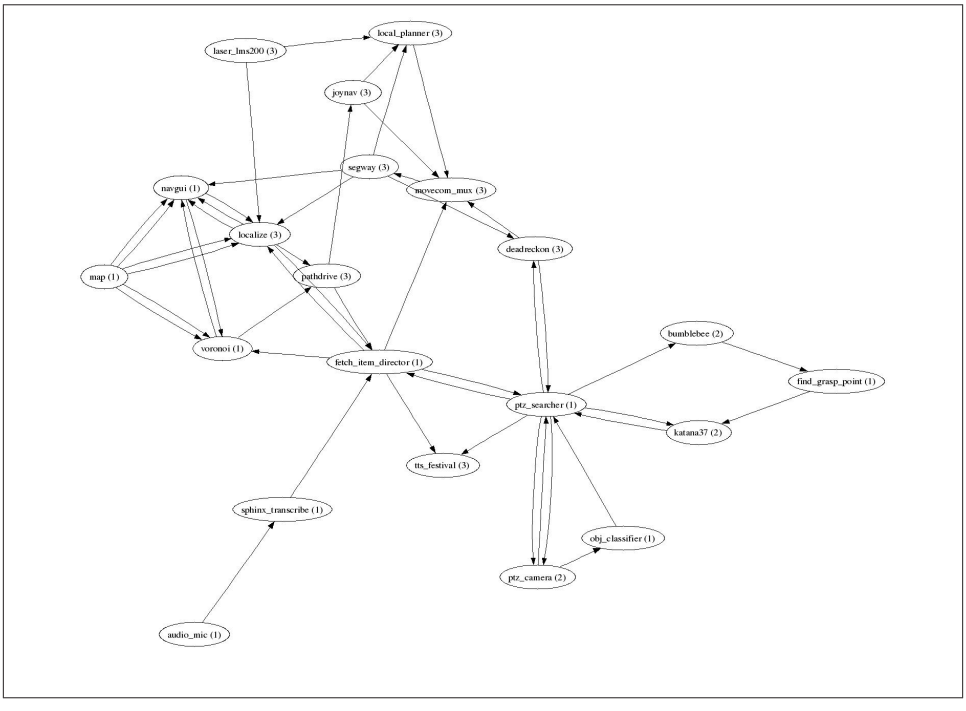


Figure 2-1. An hypothetical ROS graph for a fetch-an-item robot. Nodes in the graph represent individual programs; edges represent message streams.

To reiterate: a ROS graph *node* represents a software module that is sending or receiving messages, and a ROS graph *edge* represents a stream of messages between two nodes. Although things can get more complex, typically nodes are POSIX processes and edges are TCP connections.

To get started, consider the canonical first program, whose task is to just print "Hello, world!" to the console. It is instructive to think about how this is implemented. In UNIX, every program has a stream called "standard output," or `stdout`. When an interactive

terminal runs a "hello, world" program, its `stdout` stream is received by the terminal program, which (nowadays) renders it in a *terminal emulator* window.

In ROS, this concept is extended so that programs can have an arbitrary number of streams, connected to an arbitrary number of other programs, any of which can start up or shut down at any time. To re-create the minimal "Hello, world!" system in ROS requires two *nodes* with one stream of text messages between them. `talker` will periodically send "Hello, world!" as a text message. Meanwhile, `listener` is diligently awaiting new text messages, and whenever one arrives, it simply prints it to the console. Whenever both of these programs are running, ROS would connect them as shown in world graph.

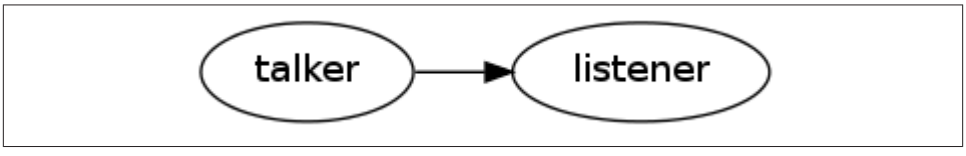


Figure 2-2. The simplest possible ROS graph: one program is sending messages to another program.

Indeed, we have created a gloriously complex "Hello, world!" system. It really doesn't do much! But we can now demonstrate some benefits of the architecture. Imagine that you wanted to create a log file of these important "Hello, world!" messages. In ROS, nodes have **no idea** who they are connected to, where their messages are coming from, or where they are going. We can thus create a generic `logger` program which writes all its incoming strings to disk, and tie that to `talker` as well, as shown in Figure 2-3.

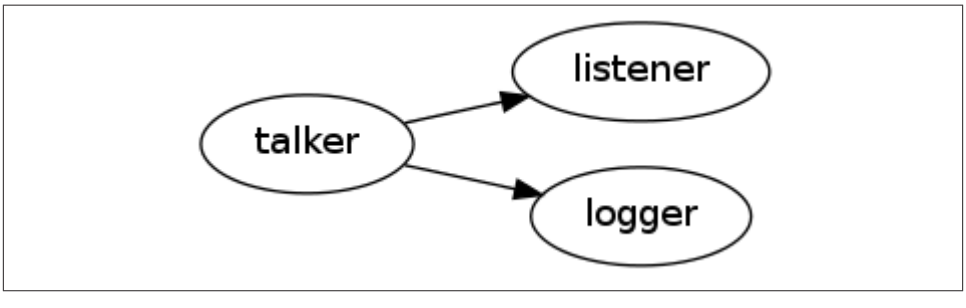


Figure 2-3. Hello, world! with a logging node.

Perhaps we wanted to run "Hello, world!" on two different computers, and have a single node receive both of their messages. Without having to modify any source code, we could just start `talker` twice, calling them "talker1" and "talker2," respectively, and ROS will connect them as shown in Figure 2-4.

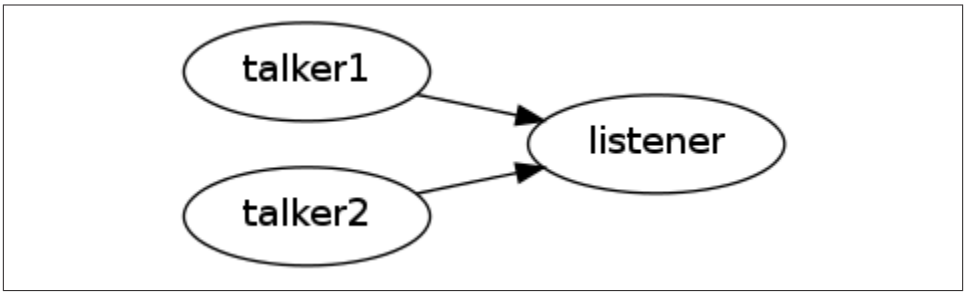


Figure 2-4. Instantiating two Hello, world! programs and routing them to the same receiver.

Perhaps we want to simultaneously log and print both of those streams? Again this can be accomplished without modifying any source code; we can command ROS to route the streams as shown in Figure 2-5.

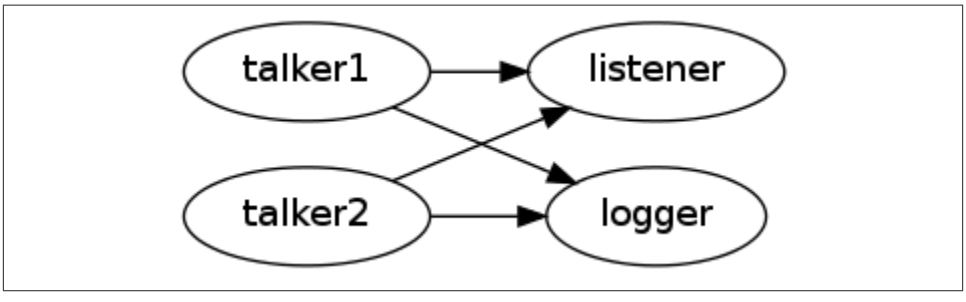


Figure 2-5. Now, there are two Hello, world! programs with two listeners. No source code has been modified.

"Hello, world!" programs are fun to write, but of course a typical robot is more complicated. For example, the "fetch a stapler" problem described at the beginning of this chapter was implemented in the early days of ROS using the exact graph previously shown as Figure 2-1. This system included 22 programs was running on four computers.

The navigation system is the upper half of Figure 2-1, and the vision and grasping systems are in the lower-right corner. It is interesting to note that this graph is *sparse*, meaning that most nodes connect to a very small number of other nodes. This property is commonly seen in ROS graphs, and can serve as a check on a software architecture: if the ROS graph starts looking like a star, where most nodes are streaming data to or from a central node, it is often worthwhile to re-assess the flow of data and separate functions into smaller pieces. The goal is to create small, manageable functional units, which ideally can be re-used in other applications on other robots.

Often, each graph node is running in its own POSIX process. This offers additional fault tolerance: a software fault will only take down its own process. The rest of the graph will stay up, passing messages as functioning as normal. The circumstances leading up to the crash can often be re-created by logging the messages entering a node, and simply playing them back at a later time inside a debugger.

However, perhaps the greatest benefit of a loosely-coupled, graph-based architecture is the ability to rapid-prototype complex systems with little or no software "glue" required for experimentation. Single nodes, such as the object-recognition node in the "fetch a stapler" example, can trivially be swapped by simply launching an entirely different process that accepts images and outputs labeled objects. Not only can a single node be swapped, but entire chunks of the graph can be torn down and replaced, even at runtime, with other large subgraphs. Real-robot hardware drivers can be replaced with simulators, navigation subsystems can be swapped, algorithms can be tweaked and recompiled, and so on. Since ROS is creating all the required network backend on-the-fly, the entire system is interactive and designed to encourage experimentation.

A ROS system is a graph of nodes, each doing some computation and sending messages to a small number of other nodes. Some nodes will be connected to sensors, and be sources of data for our system. Some will be connected to actuators, and will cause the robot to move. However, up until now, we have conveniently avoided the natural question: how do nodes find each other, so they can start passing messages? The answer lies in a program called `roscore`.

`roscore`

`roscore` is a broker that provides connection information to nodes so that they can transmit messages to each other. Nodes register details of the messages they provide, and those that they want to subscribe to, and this is stored in `roscore`. When a new node appears, `roscore` provides it with the information that it needs to form a direct peer-to-peer connection with other nodes with which it wants to swap messages. Every ROS system needs a running `roscore`. Without it, nodes cannot find other nodes to get messages from or send messages to.

When a ROS node starts up, it expects its process to have a POSIX environment variable named `ROS_MASTER_URI`. This is expected to be of the form `http://hostname:11311/`, which must point to a running instance of `roscore` somewhere on the network. Port 11311 was chosen as the default port for `roscore` simply because it was a palindromic prime. Different ports can be specified to allow multiple ROS systems to co-exist on a single LAN. With knowledge of the location of `roscore` on the network, nodes *register* themselves at startup with `roscore`, and can then query `roscore` to find other nodes and data streams by name. Each ROS node tells `roscore` which messages it provides, and which it would like to subscribe to. `roscore` then provides the addresses of the

relevant message producers and consumers in Viewed in a graph form, every node in the graph can periodically call on services provided by `roscore` to find their peers. This is represented by the dashed lines shown in Figure 2-6.

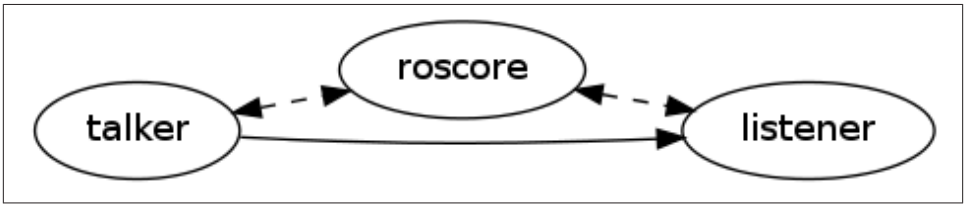


Figure 2-6. roscore connects only ephemerally to the other nodes in the system.

`roscore` also holds a *parameter server* which is used extensively by ROS nodes for configuration. The parameter server allows nodes to store and retrieve arbitrary data structures, such as descriptions of robots, parameters for algorithms, and so on. Like everything in ROS, there is a simple command-line tool to interact with the parameter server: `rosparam`, which will be used throughout the book.

We'll see examples of how to use `roscore` soon. For now, all you really need to remember about it is that it's a program that allows nodes to find other nodes. That brings is to the question of how you start a node in ROS, and to `rosrun`.

`rosrun`

Up until now, all command-line examples have assumed that the shell's working directory include the program of interest. Of course, this is not always the case, and it would be tiresome to have to continually `cd` all around the filesystem to chase down ROS programs. Since ROS has a large, distributed community, its software is organized into *packages*. The concept of a ROS *package* will be described in greater detail in subsequent chapters, but a package can be thought of as a collection of resources that are built and distributed together.

For example, the `talker` program lives in a packaged named `roscpp_tutorials`. ROS provides a command-line utility called `rosrun` which will search a package for the requested program, and pass it any and all other parameters supplied on the command line. The syntax is as follows:

```
$ rosrun PACKAGE EXECUTABLE [ARGS]
```

To run the `talker` program in the `rospy_tutorials` package, no matter where one happens to be in the filesystem, one could type:

```
$ rosrun rospy_tutorials talker
```

This is great for starting single ROS nodes, but real robot systems often consist of tens or hundreds of nodes, all running at the same time. Since it wouldn't be practical to call `rosrun` on each of these nodes, ROS includes a tool for starting collections of nodes, called `roslaunch`. However, before looking at `roslaunch`, we need to first talk about how things are named in ROS.

Names, Namespaces, and Remapping

Names are a fundamental concept in ROS. Nodes, data streams, and parameters must all have unique names. For example, the camera on a robot could be named `camera` and it could output a data stream named `image`, and read a parameter named `frame_rate` to know how fast to send images.

So far, so good. But, what happens when a fancy robot has two cameras? We wouldn't want to have to write a separate program for each camera, nor would we want the output of both cameras to be interleaved on the `image` topic.

More generally, namespace collisions are extremely common in robotic systems, which often contain several identical hardware or software modules that are used in different contexts. ROS provides two mechanisms to handle these situations: *namespaces* and *remapping*.

Namespaces are a fundamental concept throughout computer science. Following the convention of UNIX paths and Internet URI's, ROS uses the forward slash `/` to delimit namespaces. Just like how two files named `readme.txt` can exist in separate paths, such as `/home/username/readme.txt` and `/home/username/foo/readme.txt`, ROS can launch identical nodes into separate namespaces to avoid clashes.

In the previous example, a robot with two cameras could launch two camera drivers in separate namespaces, such as `left` and `right`, which would result in image streams named `left/image` and `right/image`.

This avoids a topic name clash, but how could we send these data streams to another program, which was still expecting to receive on topic `image` ? One answer would be to launch this other program in the same namespace, but perhaps this program needs to "reach into" more than one namespace. Enter *remapping*.

In ROS, any string in a program that defines a name can be *remapped* at run-time. As one example, there is a commonly-used program in ROS called `image_view` that renders a live video window of images being sent on the `image` topic. At least, that is what is written in the source code of the `image_view` program. Using remapping, we can instead cause the `image_view` program to render the `right/image` topic, or the `left/image` topic, without having to modify the source code of `image_view`!

The need to *remap* names is so common in ROS that there is a command-line shortcut for it. If the working directory contains the `image_view` program, one could type the following to map `image` to `right/image`:

```
$ ./image_view image:=right/image
```

This command-line *remapping* would produce the graph shown in Figure 2-7

Figure 2-7. The image topic has been renamed right/image using command-line remapping.

Pushing a node into a namespace can be accomplished with a special `__ns` remapping syntax (note the double underscore). For example, if the working directory contains the `camera` program, the following shell command would launch `camera` into the namespace `right`:

```
$ ./camera __ns:=right
```

Just as for filesystems, web URLs, and countless other domains, ROS names must be unique. If the same node is launched twice, `roscore` directs the older node to exit, to make way for the newer instance of the node. Earlier in this chapter, a graph was shown that had two nodes, `talker1` and `talker2`, sending data to a node named `listener`. To change the *name* of a node on the command line, the special `__name` remapping syntax can be used (again, note the double underscore). The following two shell commands would launch two instances of `talker`, one named `talker1` and one named `talker2`:

```
$ ./talker __name:=talker1
$ ./talker __name:=talker2
```

The previous examples demonstrated that ROS topics can be remapped quickly and easily on the command line. This is useful for debugging and for initially hacking systems together when experimenting with various ideas. However, after typing long command-line strings a few times, it's time to automate them! The `roslaunch` tool was created for this purpose.

roslaunch

`roslaunch` is a command-line tool designed to automate the launching of collections of ROS nodes. On the surface, it looks a lot like `rosrun`, needing a package name and a file name:

```
roslaunch PACKAGE LAUNCH_FILE
```

However, `roslaunch` operates on *launch files* which are XML files describing a combination of nodes, topic remappings, and parameters. By convention, these files have a suffix of `.launch`. For example, here is `talker_listener.launch` in the `rospy_tutorials` package:

```
<launch>
  <node name="talker" pkg="rospy_tutorials"
        type="talker.py" output="screen" />
  <node name="listener" pkg="rospy_tutorials"
        type="listener.py" output="screen" />
</launch>
```

Each `<node>` tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the *type* of node, which is simply the filename of the executable program. In this example, the `output="screen"` attributes indicate that the `talker` and `listener` nodes should dump their console outputs to the current console, instead of only to log files. This is a commonly-used setting for debugging; once things start working, it is often convenient to remove this attribute so that the console has less noise.

`roslaunch` has many other important features, such as the ability to launch programs on other computers across the network via `ssh`, automatically re-spawning nodes that crash, and so on. These features will be described throughout the book as they are necessary to accomplish various tasks. One of its most useful features is that it tears down the subgraph it launched when Ctrl+C is pressed in the console containing `roslaunch`. For example, the following command would cause `roslaunch` to spawn two nodes to form a talker-listener subgraph, as described in the `talker_listener.launch` file listed previously:

```
$ roslaunch rospy_tutorials talker_listener.launch
```

And, equally importantly, pressing Ctrl+C would tear down the subgraph! Virtually every time you use ROS, you will be calling `roslaunch` and pressing Ctrl+C in `roslaunch` terminals, to create and destroy subgraphs.

`roslaunch` will automatically instantiate a `roscore` if one does not exist when `roslaunch` is invoked. However, this `roscore` will exit when Ctrl+C is pressed in the `roslaunch` window. If one has more than one terminal open when launching ROS programs, it is often easier to remember to launch a `roscore` in a separate shell, which is left open the entire ROS session. Then, one can `roslaunch` and Ctrl+C with abandon in all other consoles, without risk of losing the `roscore` tying the whole system together.

Before we're start to look at writing some code with ROS, there's one more thing to cover, that will save you time and heartache as you try to remember the names of packages, nodes, and launch files: tab-completion.

The [TAB] key

The ROS command-line tools have tab-completion enabled. When running `rosrun`, for example, hitting the `[TAB]` key in the middle of typing that package name will auto-complete it for you, or present you with a list of possible completions. As with many other Linux commands, using tab-completion with ROS will save you a massive amount of time and avoid errors, especially when trying to type long package or message names. For example, typing

```
$ rosrun rospy_tutorials ta[TAB]
```

will auto-complete to

```
$rosrun rospy_tutorials talker
```

since no other programs in the `rospy_tutorials` package begin with `ta`. Additionally, `rosrun` (like virtually all ROS core tools) will auto-complete package names. For example, typing:

```
$ rosrun rospy_tu[TAB]
```

will auto-complete to

```
$ rosrun rospy_tutorials
```

since no other packages currently loaded begin with `rospy_tu`.

Summary

In this chapter, we looked at the ROS graph architecture, and introduced you to the tools that you're going to be using to interact with it, starting and stopping nodes. We also introduced the ROS namespace conventions, and showed how namespaces can be remapped to avoid collisions.

Now that you understand the underlying architecture of a ROS system, it's time to look at what sort of messages the nodes in this system might send to each other, how these messages are composed, sent, and received, and to think about some of the computations that the nodes might be doing. That brings us to *topics*, the fundamental communication method in ROS.

CHAPTER 3
Topics

As we saw in the previous chapter, ROS systems consist of a number of independent *nodes* that comprise a *graph*. These nodes by themselves are typically not very useful. Things only get interesting when nodes communicate with each other, exchanging information and data. The most common way to do that is through *topics*. A topic is a name for a stream of messages with a defined. For example, the data from a laser rangefinders might be send on a topic called `scan`, with a message type of `LaserScan`, while the data from a camera might be sent over a topic called `image`, with a message type of `Image`.

Before they can start to transmit data over topics, nodes must first announce, or *advertize*, both the topic name and the type of messages that are going to be sent. Then they can start to send, or *publish*, the actual data on the topic. Nodes that want to receive messages on a topic can *subscribe* to that topic by making a request to `roscore`. After subscribing, all messages on the topic are delivered to the node that make the request. Topics implement a *publish/subcribe communications mechanism*, one of the more common ways to exchange data in a distributed system. One of the main advantages to using ROS is that `roscore` and the ROS client libraries handle all the messy details of setting up the necessary connections when nodes advertise or subscribe to topics, so that you don't have to worry about it.

In ROS, all messages on the same topic **must** be of the same data type. Although ROS does not enforce it, topic names often describe the messages that are sent over them. For example, on the PR2 robot, the topic `/wide_stereo/right/image_color` is used for color images from the rightmost camera of the wide-angle stereo pair.

We'll start off by looking at how a node advertizes a topic and publishes data on it.

Publishing to a Topic

Example 3-1 shows the basic code for advertising a topic and publishing messages on it. This node publishes consecutive integers on the topic `counter` at a rate of 2Hz.

Example 3-1. topic_publisher.py

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')
import rospy

from std_msgs.msg import Int32


rospy.init_node('topic_publisher')

pub = rospy.Publisher('counter', Int32)

rate = rospy.Rate(2)

count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

The first three lines

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')
import rospy
```

should be familiar from (to come). The next line

```
from std_msgs.msg import Int32
```

imports the definition of the message that we're going to send over the topic. In this case, we're going to use a 32-bit integer, defined in the ROS standard message package, `std_msgs`. For the import to work as expected, we need to import from `<package name>.msg`, since this is where the package definitions are stored (more on this later). Since we're using a message from another package, we have to tell the ROS build system about this by adding a *dependency* to our `manifest.xml` file.

```
<depend package="std_msgs" />
```

Without this dependency, ROS will not know where to find the message definition and the node will not be able to run.

After initializing the node, we advertise it with a `Publisher`.

```
pub = rospy.Publisher('counter', Int32)
```

This gives the topic a name (`counter`), and specifies the type of message that will be sent over it (`Int32`). Behind the scenes, the publisher also sets up a connection to `roscore`, and sends some information to it. When another node tries to subscribe to the `counter` topic, `roscore` will share its list of publishers and subscribers, which the nodes will then use to create direct connections between all publishers and all subscribers of each topic.

At this point, the topic is advertised and is available for other nodes to subscribe to. Now we can go about actually publishing messages over the topic.

```
rate = rospy.Rate(2)

count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

First, we set the rate, in Hz, at which we want to publish. For this example, we're going to publish twice a second. The `is_shutdown()` function will return `True` if the node is ready to be shutdown, and `False` otherwise, so we can use this to determine if it is time to exit the `while` loop.

Inside the `while` loop, we publish the current value of the counter, increment its value by 1, and then sleep for a while. The call to `rate.sleep()` will sleep for long enough to make sure that we run the body of the `while` loop at approximately 2Hz.

And that's it. We now have a minimalist ROS node that advertises the `counter` topic, and publishes integers on it.

Checking that Everything Works as Expected

Now that we have a working node, let's verify that it works. We can use the `rostopic` command to dig into the currently available topics. Open a new terminal, and start up `roscore`. Once it's running, you can see what topics are available by running in another terminal.

```
user@hostname$ rostopic list
/rosout
/rosout_agg
```

These topics are used by ROS for logging and debugging; don't worry about them. Now, run the node we've just looked at in (yet another) terminal.

```
user@hostname$ rosrun basics topic_publisher.py
```

Remember that the `basics` directory has to be in your `ROS_PACKAGE_PATH`, and that, if you typed in the code for the node yourself, the file will need to have its execute permissions set using `chmod`. Once the node is running, you can verify that the `counter` topic is advertised by running `rostopic list` again.

```
user@hostname$ rostopic list
/counter
/rosout
/rosout_agg
```

Even better, you can see the messages being published to the topic by running `rostopic echo`.

```
user@hostname$ rostopic echo counter -n 5
data: 681
---
data: 682
---
data: 683
---
data: 684
---
data: 685
---
```

The `-n 5` flag tells `rostopic` to only print out 5 messages. Without it, it will happily go on printing messages forever, until you stop it with a `ctrl-c`. We can also use `rostopic` to verify that we're publishing at the rate we think we are.

```
user@hostname$ rostopic hz counter
subscribed to [/counter]
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00000s window: 2
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00004s window: 4
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00006s window: 6
average rate: 2.000
        min: 0.500s max: 0.500s std dev: 0.00005s window: 7
```

`rostopic hz` has to be stopped with a `ctrl-c`. Similarly, `rostopic bw` will give information about the bandwidth being used for by the topic.

You can also find out about an advertised topic with `rostopic info`.

```
user@hostname$ rostopic info counter
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://hostname:39964/)

Subscribers: None
```

This reveals that `counter` carries messages of type `std_msgs/Int32`, that it is currently being advertised by `topic_publisher`, and than no-one is currently subscribing to it. Since it's possible for more than one node to publish to the same topic, and for more than one node to be subscribed to a topic, this command can help you make sure things

are connected in the way that you think they are. The publisher `topic_publisher` is running on the computer `hostname` and is communicating over TCP port 39964+.[1] `rostopic type` works similarly, but only returns the message type for a given topic.

Finally, you can find all of the topics that publish a certain message type using `rostopic find`.

```
user@hostname$ rostopic find std_msgs/Int32
/counter
```

Note that you have to give both the package name (`std_msgs`) and the message type (`Int32`) for this to work.

So, now we have a node that's happily publishing consecutive integers, and we can verify that everything is as it should be. Now, let's turn our attention to a node that subscribes to this topic and uses the messages it is receiving.

Subscribing to a Topic

Example 3-2 shows a minimalist node that subscribes to the `counter` topic, and prints out the values in the messages as they arrive.

Example 3-2. topic_subscriber.py

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')

import rospy
from std_msgs.msg import Int32


def callback(msg):
    print msg.data


rospy.init_node('topic_subscriber')

sub = rospy.Subscriber('counter', Int32, callback)

rospy.spin()
```

The first interesting part of this code is the *callback* that handles the messages as they come in.

1. Don't worry if you don't know what a TCP port is. ROS will generally take care of this for you without you having to think about it.

```
def callback(msg):
    print msg.data
```

ROS is an event-driven system, and it uses callback functions heavily. Once a node has subscribed to a topic, every time a message arrives on it, the associated callback function is called, with the message as its parameter. In this case, the function simply prints out the data contained in the message (see "Defining Your Own Message Types" on page 25 for more details about messages and what they contain).

After initializing the node, as before, we subscribe to the `counter` topic.

```
sub = rospy.Subscriber('counter', Int32, callback)
```

We give the name of the topic, the message type on the topic, and the name of the callback function. Behind the scenes, the subsciber passes this information on to `roscore`, and tries to make a direct connection with the publishers of this topic. If the topic does not exist, or if the type is wrong, there are no error messages: the node will simply have nothing to do until messages start being published on the topic.

Once the subscription is made, we give control over to ROS by running `rospy.spin()`. This function will only return when the node is ready to shut down. This is just a useful shortcut to avoid having to define a top-level `while` loop like we did in Example 3-1; ROS does not necessarily need to "take over" the main thread of execution.

Checking that Everything Works as Expected

First, make sure that the publisher node is still running, and that it is still publishing messages on the `counter` topic. Now, in another terminal, start up the subscriber node.

```
user@hostname$ rosrun basics topic_subscriber
355
356
357
358
359
360
```

It should start to print out integers published to the `counter` topic by the publisher node. Congratulations! You're now running your first ROS system: Example 3-1 is sending messages to Example 3-2. You can visualize this system by typing `rqt_graph`, which will attempt to draw the publishers and subscribers in a logical manner.

We can also publish messages to a topic from the command line using `rostopic pub`. Run the following command, and watch the output of the subscriber node.

```
user@hostname$ rostopic pub counter std_msgs/Int32 1000000
```

We can use `rostopic info` again to make sure things are the way we expect them to be.

```
user@hostname$ rostopic info counter
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://hostname:46674/)

Subscribers:
 * /topic_subscriber (http://hostname:53744/)
```

Now that you understand how basic topics work, we can talk about special types of topics, designed for nodes that only publish data infrequently, called *latched topics*.

Latched Topics

Messages in ROS are fleeting. If you're not subscribed to a topic when a message goes out on it, you'll miss it and will have to wait for the next one. This is fine if the publisher sends out messages frequently, since it won't be long until the next message comes along. However, there are cases where sending out frequent messages is a bad idea.

For example, the `map_server` node advertises a map (of type `nav_msgs/OccupancyGrid`) on the `map` topic. This represents a map of the world that the robot can use to determine where it is, such as the one shown in Figure 3-1. Often, this map never changes, and is published once when the `map_server` loads it from disk. However, if another node needs the map, but starts up after `map_server` publishes it, it will never get the message.

Figure 3-1. An example map.

We could periodically publish the map, but we don't want to publish the message more often than we have to, since it's typically huge. Even if we did decide to republish it, we would have to pick a suitable frequency, which might be tricky to get right.

Latched topics offer a simple solution to this problem. If a topic is marked as latched when it is advertised, subscribers automatically get **the last message sent** when they subscribe to the topic. In our example of `map_server`, this means that we only need to mark it as latched, and publish it once. Topics can be marked as latched with the optional `latch` argument.

```
pub = rospy.Publisher('map', nav_msgs/OccupancyGrid, latched=True)
```

Now that we know how to send messages over topics, it's time to think about what to do if we want to send a message that isn't already defined by ROS.

Defining Your Own Message Types

ROS offers a rich set of built-in message types. The `std_msgs` package defines the primitive message types (booleans, integers, floating point numbers, strings, and arrays). `sensor_msgs` defines data structures returned by common sensors, such as laser range-fingers and cameras. `geometry_msgs` defines data structures dealing with positions, rotations, and their derivatives.

These common message types are part of what gives ROS its power. Since (most) laser range-finder sensors publish `sensor_msgs/LaserScan` messages, we can write control code for our robots without having to know the specific details of the laser range-finder hardware. Furthermore, most robots can publish their estimated location in a standard way. By using standardized message types for laser scans and location estimates, nodes can be written that provide navigation and mapping (among many other things) for a wide variety of robots.

However, there are times when the built-in message types are not enough, and we have to define our own messages.

Defining a New Message

ROS messages are defined by special message definition files in the `msg` directory of a package. These files are then compiled into language-specific implementations that can be used in your code. This means that, even if you're using an interpreted language such as Python, you need to run `catkin_make` if you're going to define your own message types. Otherwise, the language-specific implementation will not be generated, and Python will not be able to find your new message type. Furthermore, if you don't rerun `catkin_make` after you change the message definion, Python will still be using the older version of the message. Although this sounds like an extra layer of complexity, there are good reasons to do things this way: it allows us to define a message once, and have it automatically available in all languages that ROS supports, without having to manually write the (extremely tedious) code which "deflates" and "inflates" messages as they come across the network.

Message definition files are typically quite simple and short. Each line specifies a type and a field name. Types can be build-in ROS primitive types, message types from other packages, arrays of types (either primitive or from other packages, and either fixed- or variable-length), or the special `Header` type. `Header` contains some common information that many messages include: a sequence number (used internally by ROS), a timestamp, and a coordinate frame identifier.

As a concrete example, suppose we want to modify Example 3-1 to publish random complex numbers, instead of integers. A complex number has two parts, real and imag-

inary, both of which are floating point numbers. The message definition file for our new type, called `Complex` is shown in Example 3-3.

Example 3-3. Complex.msg

```
float32 real
float32 imaginary
```

The file `Complex.msg` is in the `msg` directory of the `basics` package. It defines two values, `real` and `imaginary`, both with the same type (`float32`).[2]

Once the message is defined, we need to run `catkin_make` to generate the language-specific code that will let us use it. This code includes a definition of the type, and code to marshall and unmarshall it for transmission down a topic. This allows us to use the message in all of the languages that ROS supports; nodes written in one language can subscribe to topics from nodes written in another. Moreover, it allows us to use messages to communicate seamlessly between computers with different architectures. We'll talk about this more in [Link to Come].

To get ROS to generate the language-specific message code, we need to make sure that we tell the build system about the new message definitions. We can do this by adding the lines shown in ??? to our `package.xml` file.

```
[source,xml]
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

are in your `package.xml` file (and that they are not commented out). You also need to add a line to the `CMakeLists.txt` file so that `catkin` knows to look for the `message_generation` package. To do this, you add `message_generation` to the end of the `find_package` call in `CMakeLists.txt`.

```
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation   # Add message_generation here, after the other packages
)
```

You also need to tell `catkin` that you're going to use messages at runtime, by adding `message_runtime` to the `catkin_package` call (again, in the `CMakeLists.txt` file)

```
catkin_package(
  CATKIN_DEPENDS message_runtime   # This will not be the only thing here
)
```

2. The two primitive floating point types, `float32` and `float64` both map to the Python `float` type.

Tell `catkin` the message files you want to compile, by adding them to the `add_message_files` call.

```
add_message_files(
  FILES
  Complex.msg
)
```

Finally, make sure the `generate_messages` call is uncommented, and contains all of the dependencies that are needed by your messages.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Now that you've told `catkin` everything that it needs to know about your messages, you're ready to compile them. Go to the root of your `catkin` workspace, and run `catkin_make`. This will generate a message type with the same name as the message definition file, with the `.msg` extension removed. By convention, ROS types are capitalized and contain no underscores.

Using Your New Message

Once your message is defined and compiled, you can use it just like any other message in ROS, as you can see in Example 3-4.

Example 3-4. message_publisher.py

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')
import rospy

from basics.msg import Complex

from random import random


rospy.init_node('message_publisher')

pub = rospy.Publisher('complex', Complex)

rate = rospy.Rate(2)

while not rospy.is_shutdown():
    msg = Complex()
    msg.real = random()
    msg.imaginary = random()
```

```
    pub.publish(msg)
    rate.sleep()
```

Importing your new message type works just like including a standard ROS message type, and allows you to create a message instance just like any other Python class. Once you've created the instance, you can fill in the values for the individual fields. Any fields that are not explicitly assigned a value should be considered to have an undefined value.

Subscribing to, and using, your new message is similarly easy.

Example 3-5. message_subscriber.py

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')

import rospy
from basics.msg import Complex


def callback(msg):
    print 'Real:', msg.real
    print 'Imaginary:', msg.imaginary
    print


rospy.init_node('message_subscriber')

sub = rospy.Subscriber('complex', Complex, callback)

rospy.spin()
```

The `rosmsg` command lets you look at the contents of a message type.

```
user@hostname$ rosmsg show Complex
[basics/Complex]:
float32 real
float32 imaginary
```

If a message contains other messages, they are displayed recursively by `rosmsg`. For example, `PointStamped` has a Header and a Point, each of which is a ROS type.

```
user@hostname$ rosmsg show PointStamped
[geometry_msgs/PointStamped]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Point point
  float64 x
  float64 y
  float64 z
```

`rosmsg list` will show all of the messages available in ROS. `rosmsg packages` will list all of the packages that define messages. Finally, `rosmsg package` will list of the messages defined in a particular package.

```
user@hostname$ rosmsg package basics
basics/Complex

user@hostname$ rosmsg package sensor_msgs
sensor_msgs/CameraInfo
sensor_msgs/ChannelFloat32
sensor_msgs/CompressedImage
sensor_msgs/FluidPressure
sensor_msgs/Illuminance
sensor_msgs/Image
sensor_msgs/Imu
sensor_msgs/JointState
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
sensor_msgs/LaserScan
sensor_msgs/MagneticField
sensor_msgs/MultiEchoLaserScan
sensor_msgs/NavSatFix
sensor_msgs/NavSatStatus
sensor_msgs/PointCloud
sensor_msgs/PointCloud2
sensor_msgs/PointField
sensor_msgs/Range
sensor_msgs/RegionOfInterest
sensor_msgs/RelativeHumidity
sensor_msgs/Temperature
sensor_msgs/TimeReference
```

When Should You Make a New Message Type?

The short answer is "only when you absolutely have to". ROS already has a rich set of message types, and you should use one of these if you can. Part of the power of ROS is the ability to combine nodes together to form complex systems, and this can only happen if nodes publish and receive messages of the same type. So, before you go and create a new message type, you should use `rosmsg` to see if there is already something there that you can use instead.

Mixing Publishers and Subscribers

The examples above showed nodes that have a single publisher and a single subscriber, but there's no reason why a node can't be both a publisher and a subscriber, or have multiple publications and subscriptions. In fact, one of the most common things nodes

in ROS do is to transform data by performing computations on it. For example, a node migh t subscribe to a topic containining camera images, identify faces in those images, and publish the positions of those faces in another topic. Example 3-6 shows an example of a node like this.

Example 3-6. doubler.py

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')
import rospy

from std_msgs.msg import Int32


rospy.init_node('doubler')


def callback(msg):
    doubled = Int32()
    doubled.data = msg.data * 2

    pub.publish(doubled)


sub = rospy.Subscriber('number', Int32, callback)
pub = rospy.Publisher('doubled', Int32)

rospy.spin()
```

The subscriber and publisher are set up as before, but now we're going to publish data in the callback, rather than periodically. The idea behind this is that we only want to publish when we have new data coming it, since the prupose of this node is to transform data, in this case by doubling the number that comes in on the subscribed topic.

Summary

In this chapter we covered *topics*, the fundamental ROS communication mechanism. You should now know how to advertise a topic and publish messages over it, how to subscribe to a topic and receive messages from it, how to define your own messages, and how to write simple nodes that interact with topics. You should also know to write nodes that transform data that comes in on one topic and republish it on another. This sort of node is the backbone in many ROS systems, performing computation to transform one sort of data into another, and we'll be seeing examples of this throughout the book.

Topics are probably the communication mechanism that you will use most often in ROS. Whenever you have a node that generates data that other nodes can use, you should

consider using a topic to publish those data. Whenever you need to transform data from one form to another, a node like the one shown in Example 3-6 is often a good choice.

While we covered most of what you can do with topics in this chapter, we didn't cover everything. For the full API documentation, you should look at the topic documentation.

Now that you've got the hang of topics, it's time to talk about the second main communication mechanism in ROS: *services*.

CHAPTER 4
Services

Services are another way to pass data between nodes in ROS. Services are really just synchronous remote procedure calls; they allow one node to call a function in another node. We define the inputs and outputs of this function similarly to the way we define new message types. The server (which provides the service) specifies a callback to deal with the service request, and advertises the service. The client (which calls the service) then accesses this service through a local proxy.

Although there are several services already defined by packages in ROS, we'll start by looking at how to define and implement our own service, since this gives some insight into the underlying mechanisms of service calls. As a concrete example in this chapter, we're going to show how to create a service that counts the number of words in a string.

Defining a Service

The first step in creating a new service is to define the service call inputs and outputs. This is done in a *service definition file*, which has a similar structure to the message definition files we've already seen. However, since a service call has both inputs and outputs, it's a bit more complicated than a message.

Our example service counts the number of words in a string. This means that the input to the service call should be a `string` and the output should be an integer. WordCount.srv shows a service definition for this.

WordCount.srv.

```
string words
---
uint32 count
```

The inputs to the service call come first. In this case we're just going to use the ROS built-in `string` type. Three dashes (`---`) mark the end of the inputs and the start of the

output definition. We're going to use a 32-bit unsigned integer (`uint32`) for our output. The file holding this definition is called `WordCount.srv`, and it must be in a directory called `srv` in the main package directory.

Once we've got the definition file in the right place, we need to run `catkin_make` to create the code and class definitions that we will actually use when interacting with the service, just like we did for new messages. To get `catkin_make` to generate this code, we need to make sure that the `find_package` call in `CMakeLists.txt` contains `message_generation`, just like we did for new messages.

```
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation   # Add message_generation here, after the other packages
)
```

Then, we need to tell `catkin` which service definition files we want compiled, using the `add_service_files` call in `CMakeLists.txt`.

```
add_service_files(
  FILES
  WordCount.srv
)
```

Make sure that the dependencies for the service definition file are declared (again in `CMakeLists.txt`, using the `generate_messages` call.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

With all of this in place, running `catkin_make` will generate three classes: `WordCount`, `WordCountRequest`, and `WordCountResponse`. These classes will be used to interact with the service, as we will see below.

We can verify that the service call definition is what we expect by using the `rossrv` command.

```
user@hostname$ rossrv show WordCount
[basics/WordCount]:
string words
---
uint32 count
```

You can see all available services using `rossrv list`, all packages offering services with `rossrv packages`, and all the services offered by a particular package with `rossrv package`.

Implementing a Service

Now that we have a definition of the inputs and outputs for the service call, we're ready to write the code that implements the service. As with topics, services are a callback-based mechanism. The service-provider specifies a callback that will be run when the service call is made, and then waits for requests to come in. service_server.py shows a simple server that implements our word-counting service call.

service_server.py.

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')
import rospy

from basics.srv import WordCount,WordCountResponse


def count_words(request):
    return WordCountResponse(len(request.words.split()))


rospy.init_node('service_server')

service = rospy.Service('word_count', WordCount, count_words)

rospy.spin()
```

We first need to import the code generated by `catkin`.

```
from basics.srv import WordCount,WordCountResponse
```

Notice that we need to import both `WordCount` and `WordCountResponse`. Both of these are generated in a python module with the same name as the package, with a `.srv` extension (`basics.srv` in our case).

The callback function takes a single argument, of type `WordCount`, and returns a single argument, of type `WordCountResponse`.

```
def count_words(request):
    return WordCountResponse(len(request.words.split()))
```

The constructor for `WordCountResponse` takes parameters that match those in the service definition file. For us, this means an unsigned integer. By convention, services that fail, for whatever reason, should return `None`.

After initializing the node, we advertise the service, giving it a name (`word_count`), a type (`WordCount`), and specifying the callback that will implement it.

```
service = rospy.Service('word_count', WordCount, count_words)
```

Finally, we make a call to `rospy.spin()`, which gives control of the node over to ROS, and waits for incoming service requests.

Checking that Everything Works as Expected

Now that we have the service defined and implemented, we can verify that everything is working as expected with the `rosservice` command. Start up a `roscore` and run the service node.

```
user@hostname$ rosrun basics service_server.py
```

First, let's check that the service is there.

```
user@hostname$ rosservice list
/rosout/get_loggers
/rosout/set_logger_level
/service_server/get_loggers
/service_server/set_logger_level
/word_count
```

In addition to the logging services provided by ROS, our service seems to be there. We can get some more information about it with `rosservice info`.

```
user@hostname$ rosservice info word_count
Node: /service_server
URI: rosrpc://hostname:60085
Type: basics/WordCount
Args: words
```

This tells us the node that provides the service, where it's running, the type that it uses, and the name of the arguments to the service call. We can also get some of this information using `rosservice type word_count` and `roservice args word_count`

Other Ways of Returning Values from a Service

In the example above, we have explicitly created a `WordCountResponse` object, and returned it from the service callback. There are a number of other ways to return values from a service callback that you can use. In the case where there is a single return argument for the service, you can simply return that value.

```
def count_words(request):
    return len(request.words.split())
```

If there are multiple return arguments, you can return a tuple or a list. The values in the list will be assigned to the values in the service definition in order. This works even if there's only one return value.

```
def count_words(request):
    return [len(request.words.split())]
```

You can also return a dictionary, where the keys are the argument names (given as strings).

```
def count_words(request):
    return {'count': len(request.words.split())}
```

In both of these cases, the underlying service call code in ROS will translate these return types into a `WordCountResponse` object, and return it to the calling node, just as in the initial example code.

Using a Service

The simplest way to use a service is to call it using the `rosservice` command. For our word-counting service, the call looks like this.

```
user@hostname$ rosservice call word_count 'one two three'
count: 3
```

The command takes the `call` argument, the service name, and the arguments. While this lets us call the service and make sure that it's working as expected, it's not as useful as calling it from another running node. service_client.py shows how to call our service programmatically.

service_client.py.

```
#!/usr/bin/env python

import roslib; roslib.load_manifest('basics')
import rospy

from basics.srv import WordCount

import sys


rospy.init_node('service_client')

rospy.wait_for_service('word_count')

word_counter = rospy.ServiceProxy('word_count', WordCount)

words = ' '.join(sys.argv[1:])

word_count = word_counter(words)

print words, '->', word_count.count
```

First, we wait for the service to be advertised by the server.

```
rospy.wait_for_service('word_count')
```

If we try to use the service before it's advertised, the call will fail with an exception. This is a major difference between topics and services. We can subscribe to topics that are not yet advertised, but we can only use advertised services. Once the service is advertised, we can set up a local proxy for it.

```
word_counter = rospy.ServiceProxy('word_count', WordCount)
```

We need to specify the name of the service (`word_count`) and the type (`WordCount`). This will allow us to use `word_counter` like a local function which, when called, will actually make the service call for us.

```
word_count = word_counter(words)
```

Checking that Everything Works as Expected

Now that we've defined the service, built the support code with `catkin`, and implemented both a server and a client, it's time to see if everything works. Make sure that your server is still running, and run the client node.

```
user@hostname$ rosrun basics service_client.py these are some words
these are some words -> 4
```

Now, stop the server, and re-run the client node. It should stop, waiting for the service to be advertised. Starting the server node should result in the client completing normally, once the service is available.

Other Ways to Call Services

In our client node, we are calling the service through the proxy as if it was a local function. The arguments to this function are used to fill in the elements of the service request in order. In our example, we only have one argument (`words`), so we are only allowed to give the proxy function one argument. Similarly, since there is only one output from the service call, the proxy function returns a single value. If, on the other hand, our service definition looked like this

```
string words
int min_word_length
---
uint32 count
uint32 ignored
```

then the proxy function would take two arguments, and return two values

```
c,i = word_count(words, 3)
```

The arguments are passed in the order they are defined in the service definition. It is also possible to explicitly construct a service request object, and use that to call the service.

```
request = WordCountRequest('one two three', 3)
count,ignored = word_counter(request)
```

Note that, if you choose this mechanism, you will have to also import the definition for `WordCountRequest` in the client code.

```
from basics.srv import WordCountRequest
```

Finally, if you only want to set some of the arguments, you can use keyword arguments to make the service call.

```
count,ignored = word_counter(words='one two three')
```

While this mechanism can be useful, you should use it with care, since any arguments that you do not explicitly set will remain undefined. If you omit arguments that the service needs to run, you might get strange return values. You should probably steer clear of this calling style, unless you actually *need* to use it.

Summary

Now you know all about services, the second main communication mechanism in ROS. Services are really just synchronous remote procedure calls, and allow explicit two-way communication between nodes. You should now be able to use services provided by other packages in ROS, and also to implement your own services.

Service calls are well-suited to things that you only need to do occasionally, and that take a bounded amount of time to complete. Common computations, which you might want to distribute to other computers, are a good example. Discrete actions that the robot might do, such as turning on a sensor or taking a high-resolution picture with a camera, are also good candidates for a service call implementation.

Once again, we didn't cover all of the details of services. To get more detail on more sophisticated uses of services, you should look at the service API documentation.

Now that you've mastered topics and services, it's time to look at the third, and final, communication mechanism in ROS: *actions*.

CHAPTER 5
Actions

The previous chapter described ROS *services*, which are useful for *synchronous* request / response interactions, for those cases where asynchronous ROS *topics* don't seem like the best fit. However, services aren't always the best fit, either, in particular when the request that's being made is more than a simple "get (or set) the value of X."

While services are handy for simple get / set interactions like querying status and managing configuration, they don't work well when you need to initiate a long-running task. For example, imagine commanding a robot to drive to some distant location; call it `goto_position`. The robot will require significant time (seconds, minutes, perhaps longer) to do so, with the exact amount of time impossible to know in advance, since obstacles may arise that result in a longer path.

Imagine what a service interface to `goto_position` might look like to the caller: you send a request containing the goal location, then you wait for an indeterminate amount of time to receive the response that tells you what happened. While waiting, your calling program is forced to block, you have no information about the robot's progress toward the goal, and you can't cancel or change the goal. To address these shortcomings, ROS provides *actions*.

ROS *actions* are the best way to implement interfaces to time-extended goal-oriented behaviors like `goto_position`. While services are synchronous, actions are asynchronous (actions are implemented atop topics). Similar to the request and response of a service, an action uses a *goal* to initiate a behavior and sends a *result* when the behavior is complete. But the action further uses *feedback* to provide updates on the behavior's progress toward the goal and also allows for goals to be canceled.

Using an action interface to `goto_position`, you send a goal, then move on to other tasks while the robot is driving. Along the way, you receive periodic progress updates (distance traveled, estimated time to goal, etc.), culminating in a result message (did the

robot make it to the goal or was it forced to give up?). And if something more important comes up, you can at any time cancel the goal and send the robot somewhere else.

Actions require only a little more effort to define and use then do services, and they provide a lot more power and flexibility. Let's see how they work.

Defining an Action

The first step in creating a new action is to define the *goal*, *result*, and *feedback* message formats in an *action definition file*, which by convention has the suffix `.action`. The `.action` file format is similar to the the `.srv` format used to define services, just with an additional field.

As a simple example, let's define an action that acts like an timer (we'll come back to the more useful `goto_position` behavior in a later chapter). We want this timer to count down, signaling us when the specified time has elapsed. Along the way, it should tell us periodically how much time is left. And when it's done, it should tell us how much time actually elapsed.

Shown in Timer.action is an action definition that will satisfy these requirements.

Timer.action.

```
# This is an action definition file, which has three parts: the goal, the
# result, and the feedback.
#
# Part 1: the goal, to be sent by the client
#
# The amount of time we want to wait
duration time_to_wait
---
# Part 2: the result, to be sent by the server upon completion
#
# How much time we waited
duration time_elapsed
# How many updates we provided along the way
uint32 updates_sent
---
# Part 3: the feedback, to be sent periodically by the server during
# execution.
#
# The amount of time that has elapsed from the start
duration time_elapsed
# The amount of time remaining until we're done
duration time_remaining
```

Just like with service definition files, we use three dashes (`---`) as the separator between the parts of the definition. While service definition have two parts (request and response), action definitions have three parts (goal, result, and feedback).

The action file `Timer.action` should be placed in a directory called `action` within a ROS package. As with our previous examples, this file is already present in the `basics` package.

With the definition file in the right place, we need to run `catkin_make` to create the code and class definitions that we will actually use when interacting with the action, just like we did for new services. To get `catkin_make` to generate this code, we need to add some lines to the `CMakeLists.txt` file. First, add `actionlib_msgs` to the `find_package` call (in addition to any other packages that are already there).

```
find_package(catkin REQUIRED COMPONENTS
  actionlib_msgs
)
```

Then, use the `add_action_files` call to tell `catkin` about the action files you want to compile.

```
add_action_files(
  DIRECTORY action
  FILES Timer.action
)
```

Make sure you list the dependencies for your actions. You also need to explictly list `actionlib_msgs` as a dependency in order for actions to compile properly.

```
generate_messages(
  DEPENDENCIES
  actionlib_msgs
  std_msgs
)
```

Finally, add `actionlib_msgs` as a dependency for `catkin`.

```
catkin_package(
  CATKIN_DEPENDS
  actionlib_msgs
)
```

With all of this information in place, running `catkin_make` in the top level of your `catkin` workspace does quite a bit of extra work for us: our `Timer.action` file is processed to produce several message definition files: `TimerAction.msg`, `TimerActionFeedback.msg`, `TimerActionGoal.msg`, `TimerActionResult.msg`, `TimerFeedback.msg`, `TimerGoal.msg`, and `TimerResult.msg`. These messages are used to implement the action client / server protocol, which, as mentioned previously, is built on top of ROS topics. The generated message definitions are in turn processed by the message generator to produce corresponding class definitions. Most of the time, you'll use only a few of those classes, as you'll see in the following examples.

Implementing a Basic Action Server

Now that we have a definition of the goal, result, and feedback for the timer action, we're ready to write the code that implements it. As with topics and services, actions are a callback-based mechanism, with your code being invoked as a result of receiving messages from another node.

The easiest way to build an action server is to use the `SimpleActionServer` class from the `actionlib` package. We'll start by defining only the callback that will be invoked when a new goal is sent by an action client. In that callback, we'll do the work of the timer, then return a result when we're done. We'll add feedback reporting in the next step. simple_action_server.py shows the code for our first action server.

simple_action_server.py.

```
import roslib; roslib.load_manifest('basics')
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
    server.set_succeeded(result)

rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

Let's step through the key parts of the code. First we import the standard Python `time` package, which we'll use for the timer functionality of our server. We also import the ROS `actionlib` package that provides the `SimpleActionServer` class that we'll be using. We also import some of the messages classes that were autogenerated from our `Timer.action` file.

```
import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult
```

Next we define `do_timer()`, the function will be invoked when we receive a new goal. In this function, we handle the new goal in place and set a result before returning. The type of the `goal` argument that is passed to `do_timer()` is `TimerGoal`, which corresponds to the goal part of `Timer.action`. We save the current time, using the standard Python

`time.time()` function, then we sleep for the time requested in the goal, converting the `time_to_wait` field from a ROS duration to seconds.

```
def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
```

Next we build up the result message, which will be of type `TimerResult`, which corresponds to the result part of `Timer.action`. We fill in the `time_elapsed` field by subtracting our saved start time from the current time, and converting the result to a ROS duration. We set `updates_sent` to zero, because we didn't send any updates along the way (we'll add that part shortly).

```
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
```

Our final step in the callback is to tell the `SimpleActionServer` that we successfully achieved the goal by calling `set_succeeded()` and passing it the result. The `server` variable is declared below.

```
    server.set_succeeded(result)
```

Back in the global scope, we initialize and name our node as usual, then create a `SimpleActionServer`. The first constructor argument for `SimpleActionServer` is the server's name, which will determine the namespace into which its constituent topics will be advertised; we'll use `timer`. The second argument is the type of the action that the server will be handling, which in our case is `TimerAction`. The third argument is the goal callback, which is the function `do_timer()` that we defined above. Finally, we pass `False` to disable auto-starting the server. Having created the action server, we explicitly `start()` it, then go into the usual ROS `spin()` loop to wait for goals to arrive.

Auto-starting should *always* be disabled on action servers, because it can allow a race condition that leads to puzzling bugs. It was an oversight in the implementation of `actionlib` to make auto-starting the default, but by the time the problem was discovered, there was too much existing code that relied on that default behavior to change it.

```
rospy.init_node('timer_action_server')
server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
server.start()
rospy.spin()
```

Checking that Everything Works as Expected

Now that we have the action server implemented, we can do a couple of checks to ensure that it's working as expected. Start up a `roscore` and then run the action server:

```
user@hostname$ rosrun basics simple_action_server.py
```

Let's check that the expected topics are present:

```
user@hostname$ rostopic list
/rosout
/rosout_agg
/timer/cancel
/timer/feedback
/timer/goal
/timer/result
/timer/status
```

That looks good: we can see the five topics in the `timer` namespace that are used under the hood to manage the action. Let's take a closer look at the `/timer/goal` topic, using `rostopic`:

```
user@hostname$ rostopic info /timer/goal
Type: basics/TimerActionGoal

Publishers: None

Subscribers:
 * /timer_action_server (http://localhost:63174/)
```

What's a `TimerActionGoal`? Let's dig in further, now with `rosmsg`:

```
user@hostname$ rosmsg show TimerActionGoal
[basics/TimerActionGoal]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
basics/TimerGoal goal
  duration time_to_wait
```

Interesting; we can see our goal definition in there, as the `goal.time_to_wait` field, but there are also some extra fields that we didn't specify. Those extra fields are used by the action server and client code to keep track of what's happening. Fortunately, that bookkeeping information is automatically stripped away before our server code sees a goal message. While a `TimerActionGoal` message is sent over the wire, what we see in our goal execution is a bare `TimerGoal` message, which is just what we defined in our `.action` file:

```
user@hostname$ rosmsg show TimerGoal
[basics/TimerGoal]:
duration time_to_wait
```

In general, if you're using the libraries in the `actionlib` package, you should not need to access the auto-generated messages with `Action` in their type name. The bare `Goal`, `Result`, and `Feedback` messages should suffice.

However, if you like, you can publish and subscribe directly to an action server's topics, using the autogenerated `Action` message types. This is a nice feature of ROS actions: they are just a higher level protocol built on top of ROS messages. But for most applications (including everything that we'll cover in this book), the `actionlib` libraries will do the job, handling the underlying messages for you behind the scenes.

Using an Action

The easiest way to use an action is via the `SimpleActionClient` from the `actionlib` package. simple_action_client.py shows a simple client that sends a goal to our action server and waits for the result.

simple_action_client.py.

```
import roslib; roslib.load_manifest('basics')
import rospy

import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

Let's step through the key parts of the code. Following the usual imports and initialization of our ROS node, we create a `SimpleActionClient`. The first constructor argument is the name of the action server, which the client will use to determine the topics that it will use when communicating with the server. This name must match the one that we used in creating the server, which is `timer`. The second argument is the type of the action, which must also match the server: `TimerAction`.

Having created the client, we tell it to wait for the action server to come up, which it does by checking for the five advertised topics that we saw earlier when testing the server. Similar to `rospy.wait_for_service()`, which we used to wait for a service to be ready, `SimpleActionClient.wait_for_server()` will block until the server is ready.

```
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
```

Now we create a goal of type `TimerGoal` and fill in the amount of time we want the timer to wait, which is five seconds. Then we send the goal, which causes the transmission of the goal message to the server.

```
goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
client.send_goal(goal)
```

Next we wait for a result from the server. If things are working properly, we expect to block here for about five seconds. After the result comes in, we use `get_result()` to retrieve it from within the client object and print out the `time_elapsed` field that was reported by the server.

```
client.wait_for_result()
print('Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
```

Checking that Everything Works as Expected

Now that we have the action client implemented, we can get to work. Make sure that your `roscore` and action server are still running, then run the action client:

```
user@hostname$ rosrun basics simple_action_client.py
Time elapsed: 5.001044
```

Between the invocation of the client and the printing of the result data, you should see a delay of approximately five seconds, as requested. The time elapsed should be slightly more than five seconds, because a call to `time.sleep()` will always take a little longer than requested.

Implementing a More Sophisticated Action Server

So far, actions look a lot like services, just with more configuration and setup. Now it's time to exercise the asynchronous aspects of actions that set them apart from services. We'll start on the server side, making some changes that demonstrate how to abort a goal, how to handle a goal preemption request, and how to provide feedback while pursuing a goal. fancy_action_server.py shows the code for our improved action server.

fancy_action_server.py.

```
import roslib; roslib.load_manifest('basics')
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_timer(goal):
```

```
        start_time = time.time()
        update_count = 0

        if goal.time_to_wait.to_sec() > 60.0:
            result = TimerResult()
            result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
            result.updates_sent = update_count
            server.set_aborted(result, "Timer aborted due to too-long wait")
            return

        while (time.time() - start_time) < goal.time_to_wait.to_sec():

            if server.is_preempt_requested():
                result = TimerResult()
                result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
                result.updates_sent = update_count
                server.set_preempted(result, "Timer preempted")
                return

            feedback = TimerFeedback()
            feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
            feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed
            server.publish_feedback(feedback)
            update_count += 1

            time.sleep(1.0)

        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        result.updates_sent = update_count
        server.set_succeeded(result, "Timer completed successfully")

    rospy.init_node('timer_action_server')
    server = actionlib.SimpleActionServer('timer', TimerAction, do_timer, False)
    server.start()
    rospy.spin()
```

Let's step through the changes with respect to simple_action_server.py. Because we will be providing feedback, we add `TimerFeedback` to the list of message types that we import.

```
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
```

Stepping inside our `do_timer()` callback, we add a variable that will keep track of how many times we published feedback.

```
        update_count = 0
```

Next we add some error checking. We don't want this timer to be used for long waits, so we check whether the requested `time_to_wait` is greater than 60 seconds and if so, we explicitly abort the goal by calling `set_aborted()`. This call sends a message to the client notifying it that the goal has been aborted. Like with `set_succeeded()`, we include

a result; doing this is optional, but a good idea if possible. We also include a status string to help the client understand what happened; in this case we aborted because the requested wait was too long. Finally we return from the callback because we're done with this goal.

```
    if goal.time_to_wait.to_sec() > 60.0:
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Timer aborted due to too-long wait")
        return
```

Now that we're past the error check, instead of just sleeping for the requested time in a one shot, we're going to loop, sleeping in increments. In this way we can do things while we're working toward the goal: check for preemption and provide feedback.

```
    while (time.time() - start_time) < goal.time_to_wait.to_sec():
```

In the loop, we first check for preemption by asking the server `is_preempt_requested()`. This function will return `True` if the client has requested that we stop pursuing the goal. If so, similar to the abort case, we fill in a result and provide a status string, this time calling `set_preempted()`.

```
        if server.is_preempt_requested():
            result = TimerResult()
            result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted")
            return
```

Next we send feedback, using the type `TimerFeedback`, which corresponds to the feedback part of `Timer.action`. We fill in the `time_elapsed` and `time_remaining` fields then call `publish_feedback()` to send it to the client. We also increment `update_count` to reflect the fact that we sent another update.

```
        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
        feedback.time_remaining = goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1
```

Then we sleep a little and loop. Sleeping for a fixed amount of time here is not the right way to implement a timer, as we could easily end up sleeping longer than requested. But it makes for a simpler example.

```
        time.sleep(1.0)
```

Exiting the loop means that we've successfully slept for the requested duration, so it's time to notify the client that we're done. This step is very similar to the simple action server, except that we fill in the `updates_sent` field and add a status string.

```
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, "Timer completed successfully")
```

The rest of the code is unchanged from simple_action_server.py: initialize the node, create and start the action server, then wait for goals.

Using the More Sophisticated Action

Now we'll modify the action client to try out the new capabilities that we added to the action server: we'll process feedback, we'll preempt a goal, and we'll trigger an abort. fancy_action_client.py shows the code for our improved action client.

fancy_action_client.py.

```
import roslib; roslib.load_manifest('basics')
import rospy

import time
import actionlib
from basics.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def feedback_cb(feedback):
    print('[Feedback] Time elapsed: %f'%(feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f'%(feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()

goal = TimerGoal()
goal.time_to_wait = rospy.Duration.from_sec(5.0)
# Uncomment this line to test server-side abort:
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
client.send_goal(goal, feedback_cb=feedback_cb)

# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()

client.wait_for_result()
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))
```

Let's step through the changes with respect to simple_action_client.py. We define a callback, `feedback_cb()`, that will be invoked when we receive a feedback message. In this callback we just print the contents of the feedback.

```
def feedback_cb(feedback):
    print('[Feedback] Time elapsed: %f'%(feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f'%(feedback.time_remaining.to_sec()))
```

We register our feedback callback by passing it as the `feedback_cb` keyword argument when calling `send_goal()`.

```
client.send_goal(goal, feedback_cb=feedback_cb)
```

After receiving the result, we print a little more information to show what happened. The `get_state()` function returns the state of the goal, which is an enumeration that is defined in `actionlib_msgs/GoalStatus`. While there are ten possible states, in this example, we'll encounter only three: `PREEMPTED=2`, `SUCCEEDED=3`, and `ABORTED=4`. We also print the status text that was included by the server with the result.

```
print('[Result] State: %d'%(client.get_state()))
print('[Result] Status: %s'%(client.get_goal_status_text()))
print('[Result] Time elapsed: %f'%(client.get_result().time_elapsed.to_sec()))
print('[Result] Updates sent: %d'%(client.get_result().updates_sent))
```

Checking that Everything Works as Expected

Let's try out our new server and client. As before, start up a `roscore`, then run the server:

```
user@hostname$ rosrun basics fancy_action_server.py
```

In another terminal, run the client:

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001626
[Feedback] Time remaining: 3.998374
[Feedback] Time elapsed: 2.003189
[Feedback] Time remaining: 2.996811
[Feedback] Time elapsed: 3.004825
[Feedback] Time remaining: 1.995175
[Feedback] Time elapsed: 4.006477
[Feedback] Time remaining: 0.993523
[Result] State: 3
[Result] Status: Timer completed successfully
[Result] Time elapsed: 5.008076
[Result] Updates sent: 5
```

Everything works as expected: while waiting, we receive one feedback update per second, then receive a successful result (`SUCCEEDED=3`).

Now let's try preempting a goal. In the client, following the call to `send_goal()`, uncomment these two lines, which will sleep briefly, then request that the server preempt the goal:

```
# Uncomment these lines to test goal preemption:
#time.sleep(3.0)
#client.cancel_goal()
```

Run the client again:

```
user@hostname$ rosrun basics fancy_action_client.py
[Feedback] Time elapsed: 0.000044
[Feedback] Time remaining: 4.999956
[Feedback] Time elapsed: 1.001651
[Feedback] Time remaining: 3.998349
[Feedback] Time elapsed: 2.003297
[Feedback] Time remaining: 2.996703
[Result] State: 2
[Result] Status: Timer preempted
[Result] Time elapsed: 3.004926
[Result] Updates sent: 3
```

That's the behavior we expect: the server pursues the goal, providing feedback, until we send the cancellation request, after which we receive the result confirming the preemption (`PREEMPTED=2`).

Now let's trigger a server-side abort. In the client, uncomment this line to change the requested wait time from five seconds to 500 seconds:

```
# Uncomment this line to test server-side abort:
#goal.time_to_wait = rospy.Duration.from_sec(500.0)
```

Run the client again:

```
user@hostname$ rosrun basics fancy_action_client.py
[Result] State: 4
[Result] Status: Timer aborted due to too-long wait
[Result] Time elapsed: 0.000012
[Result] Updates sent: 0
```

As expected, the server immediately aborted the goal (`ABORTED=4`).

Summary

In this chapter we covered *actions*, a powerful communications tool that is commonly used in ROS systems. Similar to ROS *services*, actions allow you to make a request (for actions, a *goal*) and receive a response (for actions, a *result*). But actions offer much more control to both the client and the server than do services. The server can provide feedback along the way while it's servicing the request, the client can cancel a previously issued request, and because they're built atop ROS *messages*, actions are asynchronous, allowing for non-blocking programming on both sides.

Taken together, these features of actions make them well-suited to many aspects of robot programming. It's common in a robotics application to implement time-extended goal-seeking behaviors, whether it's `goto_position` or `clean_the_house`. Any time you need

to be able trigger a behavior, actions are probably the right tool for the job. In fact, any time that you're using a service, it's worth considering replacing it with an action; actions require a bit more code to use, but in return they're much more powerful and extensible than services. We'll see many examples in future chapters where actions provide rich but easy-to-use interfaces to some pretty complex behaviors.

As usual, we did not cover the entire API in this chapter. There are more sophisticated uses of actions that can be useful in situations where you need more control over how the system behaves, such as what to do when there are multiple clients and/or multiple simultaneous goals. For full API documentation, consult actionlib documentation.

At this point, you know all of the basics of ROS: how nodes are organized into a graph, how to use the basic command-line tools, how to write simple nodes, and how to get these nodes to communicate with each other. Before we look at our first complete robot application in Chapter 7, let's take a moment to talk about the various parts of a robot system, for both real and simulated robots, and how they relate to ROS.

CHAPTER 6

Robots and Simulators

The previous chapters discussed many fundamental concepts of ROS. They may have seemed rather vague and abstract, but those concepts were necessary to describe how data moves around in ROS and how its software systems are organized. In this chapter, we will introduce common robot subsystems and describe how the ROS architecture handles them. Then, we will introduce the robots that we will use throughout the remainder of the book, and describe the simulators in which we can most easily experiment with them.

Subsystems

Like all complex machines, robots are most easily designed and analyzed by considering one subsystem at a time. In this section, we will introduce the main subsystems commonly found on the types of robots considered in this book. Broadly speaking, they can be divided into three categories: actuation, sensing, and computing. In the ROS context, actuation subsystems are interacting directly with how the robot's wheels or arms move. Sensing subsystems interact directly with sensor hardware, such as cameras or laser scanners. Finally, the computational subsystems are what tie actuators and sensing together, with (ideally) some relatively intelligent processing that allows the robot to perform useful tasks. We will introduce these subsystems in the next few sections. Note that we are not attempting to provide an exhaustive discussion; rather, we are trying to describe these subsystems just deeply enough to convey the issues typically faced when interacting with them from a software-development standpoint.

Actuation: Mobile Platform

The ability to move around, or *locomote*, is a fundamental capability of many robots. It is surprisingly nuanced: there are many books written entirely on this subject! However,

broadly speaking, a mobile base is a collection of actuators which allow a robot to move around. They come in an astonishingly wide variety of shapes and sizes.

Although legged locomotion is popular in some domains in the research community, and camera-friendly walking robots have seen great progress in recent years, most robots drive around on wheels. This is because of two main reasons: first, wheeled platforms are often simple to design and manufacture. Second, for the very smooth surfaces that are common in artificial environments, such as indoor floors or outdoor pavement, wheels are the most energy-efficient way to move around.

The simplest possible configuration of a wheeled mobile robot is called *differential drive*. It consists of two independently-actuated wheels, often located on the centerline of a round robot. In this configuration, the robot moves forward when both wheels turn forward, and spins in place when one wheel drives forward and one drives backward. Differential-drive robots often have one or more *casters*, which are unpowered wheels that spin freely to support the front and back of the robot, just like the wheels on the bottom of a typical office chair. This is an example of a *statically stable* robot, which means that, when viewed from above, the center of mass of the robot is inside a polygon formed by the points of contact between the wheels and the ground. Statically stable robots are simple to model and control, and among their virtues is the fact that power can be shut off to the robot at any time, and it will not fall over.

However, *dynamically stable*, or *balancing* wheeled mobile robots are also possible, with the term *dynamic* implying that the actuators must constantly be in motion (however slight) to preserve stability. The simplest dynamically stable wheeled robots look like (and often are literally built upon) Segway platforms, with a pair of large differential-drive wheels supporting a tall robot above. Among the benefits of balancing wheeled mobile bases is that the wheels contacting the ground can have very large diameter, which allows the robot to smoothly drive over small obstacles: imagine the difference between running over a pebble with an office-chair wheel versus a bicycle wheel (this is, in fact, precisely the reason why bicycle wheels are large). Another advantage of balancing wheeled mobile robots is that the footprint of the robot is as small as possible, which can be useful in tight quarters.

The differential-drive scheme can be extended to more than two wheels, and is often called *skid-steering*. Four-wheel and six-wheel skid-steering schemes are common, in which all of the wheels on the left side of the robot actuate together, and all of the wheels on the right side actuate together. As the number of wheels extends beyond six, typically the wheels are connected by external *tracks*, as exemplified by excavators or tanks.

As is typically the case in engineering, there are tradeoffs with the skid-steering scheme, and it makes sense for some applications, but not all. One advantage is that skid-steering provides maximum traction while preserving mechanical simplicity (and thus controlling cost), since all contact points between the vehicle and the ground are being actively

driven. However, skid-steering is, as its name states, constantly skidding when it is not driving exactly forwards or backwards.

In the most extreme case, when trying to turn in place with one set of wheels turning fowards and the other turning backwards, the wheels are skidding dramatically, which can tear up gentle surfaces and wear tires quickly. This is why excavators are typically towed to a construction site on a trailer! In some situations, however, traction and the ability to surmount large obstacles are valued so highly that skid-steering platforms are used extensively. However, all this traction comes at a cost: the constant skidding is tremendously inefficient, since massive energy is spent tearing up the dirt (or heating up the wheels) whenever the robot turns at low speeds.

The inefficiencies and wear-and-tear of skid-steering are among the reasons why passenger cars use more complex (and expensive) schemes to get around. They are often called *Ackerman* platforms, in which the rear wheels are always pointed straight ahead, and the front wheels turn together. Placing the wheels at the extreme corners of the vehicle maximizes the area of the *supporting polygon*, which is why cars can turn sharp corners without tipping over, and (when not driven in action movies), car wheels do not have to skid when turning. However, the downside of Ackerman platforms is that they cannot drive sideways, since the rear wheels are always facing forwards. This is why parallel parking is a dreaded portion of any drivers' license examination, since elaborate planning and sequential actuator maneuvers are required to move an Ackerman platform sideways.

All of the platforms described thus far can be summarized as being *non-holonomic*, which means that they cannot move in *any* direction at any given time. For example, neither differential-drive platforms nor Ackerman platforms can move sideways. To do this, a *holonomic* platform is required, which can be built using *steered casters*. Each steered caster actuator has two motors: one motor rotates the wheel forwards and backwards, and another motor steers the wheel about its vertical axis. This allows the platform to move in any direction while spinning arbitrarily. Although significantly more complex to build and maintain, these platforms simplify motion planning. Imagine the ease of parallel parking if you could drive sideways into a parking spot!

As a special case, when the robot only needs to move on very smooth surfaces, a low-cost holonomic platform can be built using *Mecanum* wheels. These are clever contraptions in which each wheel has a series of rollers on its rim, angled 45 degrees to the plane of the wheel. Using this scheme, motion in any direction, with any rate of rotation, is possible at all times, using only four actuators, without skidding. However, due to the small diameter of the roller wheels, it is only suitable for very smooth surfaces such as hard flooring or extremely short carpets.

This section has described the fundamental types of wheeled mobile platforms, but we must re-emphasize that this is a massive subject which is treated in great detail in many textbooks.

Since one of the design goals of ROS was to allow the software re-use across a variety of robots, ROS software which interacts with mobile platforms virtually always uses a `Twist` message. A *twist* is a way to express general linear and angular velocities in three dimensions. Although it may seem easier to express mobile-base motions simply by expressing its wheel velocities, using the linear and angular velocities of the center of the vehicle allows the software to abstract away the kinematics of the vehicle.

For example, high-level software can command the vehicle to drive forward at 0.5 meters/second while rotating clockwise at 0.1 meters/second. From the standpoint of the high-level software, whether the mobile platform's actuators are arranged as differential-drive, Ackerman steering, or Mecanum wheels is irrelevant, just as the transmission ratios and wheel diameters are irrelevant to high-level behaviors.

The robots we describe in this book only be navigating on flat two-dimensional surfaces. However, expressing velocities in three dimensions allows path-planning or obstacle-avoidance software to be used by vehicles capable of more general motions, such as aerial, underwater, or space vehicles. It is important to recognize that even for vehicles designed for two-dimensional navigation, the general three-dimensional Twist methodology is necessary to express desired or actual motions of many types of actuators, such as grippers, since they are often capable of three-dimensional motions when flying on the end of a manipulator arm. Manipulators are the other main class of robot actuators, which we will discuss in the next section.

Actuation: Manipulator Arm

Many robots need to *manipulate* objects in their environment. For example, packing or palletizing robots sit on the end of production lines, grab items coming down the line, and place them into boxes or stacks. There is an entire domain of robot manipulation tasks called *pick and place*, in which manipulator arms grasp items and place them somewhere else. Security robot tasks include handling suspicious items, for which a strong manipulator arm is often required. An emerging class of *personal robots* hope to be genuinely useful in home and office applications, performing manipulation tasks including cleaning, delivering items, preparing meals, and so on.

Just as for mobile bases, the manipulator-arm subsystem has astonishing variety across robots, in response to the many tradeoffs made to support particular application domains and price points.

Although there are exceptions, the majority of manipulator arms are formed by a *chain* of rigid *links* connected by *joints*. The simplest kind of joints are single-axis revolute joints (also called "pin" joints), where one link has a shaft which serves as the axis around which the next link rotates, in the same way that a typical residential door rotates around its hinge pins. However, *linear* joints (also called *prismatic* joints) are also common, in which one link has a *slide* or tube along which the next link travels, just as a sliding door runs sideways back and forth along its track.

A fundamental characteristic of a robot manipulator is the number of degrees of freedom (DOF) of its design. Often, the number of joints is equal to the number of actuators; when those numbers differ, typically the DOF is taken to be the lower of the two numbers. Regardless, the number of DOF is one of the most significant drivers of manipulator size, mass, dexterity, cost, and reliability. Adding DOF to the *distal* (far) end of a robot arm typically increases its mass, which requires larger actuators on the *proximal* (near) joints, which further increases the mass of the manipulator.

In general, six DOF are required to position the wrist of the manipulator arm in any location and orientation within the workspace, providing that each joint has full range of motion. However, achieving full (360-degree) range of motion on all joints of a robot is often difficult to achieve at reasonable cost, due to constraints in mechanical structures, electrical wiring, and so on. As a result, seven-DOF arms are often used. The seventh DOF provides an "extra" degree of freedom which can be used to move the links of the arm while maintaining the position and orientation of the wrist, much as a human arm can move its elbow through an arc segment while maintaining the wrist in the same position.

Robots intended for manipulation tasks in human environments often have human-scale, 7-DOF arms, quite simply because the desired *workspaces* are human-scale surfaces, such as tables, countertops, and so on. In contrast, robots intended for industrial applications have wildly varying dimensions depending on the task they are to perform.

So far, we have discussed the two main classes of robot actuators: those used for locomotion, and those used for manipulation. The next major class of robot hardware is its sensors. We'll start with the sensor head, a common mounting scheme, and then describe the sub-components found in many robot sensor heads.

Sensor Head

Robots must sense the world around them in order to react to variations in tasks and environments. Although many successful industrial deployments use surprisingly little sensing, the applications we will describe in this book are reliant on sensor data. Any configuration of sensing hardware is possible (and has likely been tried), but for convenience, aesthetics, and to preserve line-of-sight with the center of the workspace, it is common for robots to have a *sensor head* on top of the platform. Typically, these heads sit atop a *pan/tilt* assembly, so that they can rotate to a bearing of interest and look up or down as needed. The following several sections will describe sensors commonly found in robot sensor heads and on other parts of their bodies.

Visual Camera

Higher-order animals tends to rely on visual data to react to the world around them. If only robots were as smart as animals! The use of camera data is surprisingly difficult,

as we will describe in later chapters of this book. However, cameras are cheap and often useful for teleoperation, so it is common to see them on robot sensor heads.

Interestingly, it is often more mathematically robust to describe robot tasks and environments in three dimensions (3D) rather than to work with 2D camera images. This is because the 3D shape of tasks and environments are *invariant* to changes in scene lighting, shadows, occlusions, and so on. In fact, in a surprising number of application domains, the visual data is largely ignored; the algorithms are interested in 3D data. As a result, intense research efforts have been expended on producing 3D data of the scene in front of the robot.

When two cameras are (very!) rigidly mounted to a common mechanical structure, they form a *stereo camera*. Each camera sees a slightly different view of the world, and these slight differences can be used to estimate the distances to various features in the image. This sounds simple, but as always, the devil is in the details. The performance of a stereo camera depends on a large number of factors, such as the quality of the camera's mechanical design, its resolution, lens type and quality, and so on. Equally important are the qualities of the scene being imaged: a stereo camera can only estimate the distances to mathematically discernable *features* in the scene, such as sharp, high-contrast corners. A stereo camera cannot, for example, estimate the distance to a featureless wall, although it most likely can estimate the distance to the corners and edges of the wall, if they intersect a floor, ceiling, or other wall of a different color. However, many natural outdoor scenes possess sufficient texture that stereo vision can be made to work quite well for depth estimation. Indoor scenes, however, can often be quite difficult.

Several conventions have emerged in the ROS community for handling cameras. The canonical ROS message type for images is `sensor_msgs::Image` and it contains little more than the size of the image, its pixel encoding scheme, and the pixels themselves. To describe the *intrinsic* distortion of the camera resulting from its lens and sensor alignment, the `sensor_msgs::CameraInfo` message is used. Often, these ROS images need to be sent to and from OpenCV, a popular computer vision library. The `cv_bridge` package is intended to simplify this operation, and will be used throughout the book.

Depth Camera

As discussed in the previous section, even though visual camera data is intuitively appealing, and seems like somehow it should be useful, many perception algorithms work much better with 3D data. Fortunately, the past few years have seen massive progress in low-cost *depth cameras.* Unlike the passive stereo cameras described in the previous section, *depth cameras* are **active** devices. They illuminate the scene in various ways, which greatly improves the system performance. For example, a completely featureless indoor wall or surface is essentially impossible to detect using passive stereo vision. However, many depth cameras will shine a texture pattern on the surface, which is subsequently imaged by its camera. The texture pattern and camera are typically set to

operate in near-infrared wavelengths to reduce the system's sensitivity to the colors of objects, as well as to not be distracting to people nearby.

Some common depth cameras, such as the Microsoft Kinect, project a *structured light* image. The device is projecting a precisely known pattern into the scene, its camera is observing how this pattern is deformed as it lands on the various objects and surfaces of the scene, and finally a *reconstruction algorithm* estimates the 3D structure of the scene from this data.

Many other schemes are possible. For example, *unstructured light* depth cameras perform "standard" stereo vision on a scene with random texture injected by some sort of projector. This scheme has been shown to work far better than passive stereo systems in feature-scarce environments, such as many indoor scenes.

A different approach is used by *time-of-flight* depth cameras. These imagers rapidly blink an infrared LED or laser illuminator, while using specially-designed pixel structures in their image sensors to estimate the time required for these light pulses to fly into the scene and bounce back to the depth camera. Once this "time of flight" is estimated, the (constant) speed of light can be used to convert the estimates into a *depth image*.

Intense research and development is occuring in this domain, due to the enormous markets (and potential markets) for depth cameras in video games and other mass-market user-interaction scenarios. It is not yet clear which (if any) of the schemes discussed previously will end up being best-suited for robotics applications. At time of writing, cameras using all of the previous modalities are in common usage in robotics experiments.

Just like visual cameras, depth cameras produce an enormous amount of data. This data is typically in the form of *point clouds*, which are the 3D points estimated to lie on the surfaces facing the camera. The fundamental point cloud message is `sensor_msgs::PointCloud2` (so named purely for historical reasons). This message allows for unstructured point cloud data, which is often advantageous, since depth cameras often cannot return a valid depth estimate for each pixel in their camera. As such, depth images often have substantial "holes" in the image, which processing algorithms must handle gracefully.

Laser Scanner

Although depth cameras have greatly changed the depth-sensing market in the last few years due to their simplicity and low cost, there are still some applications in which *laser scanners* are widely used due to their superior accuracy and longer sensing range. There are many types of laser scanners, but one of the most common schemes used in robotics is constructed by shining a laser beam on a rotating mirror spinning around 10 to 80 times per second (typically 600 to 4800 RPM). As the mirror is rotating, the laser light

is pulsed rapidly and the reflected waveforms are correlated with the outgoing waveform to estimate the *time of flight* of the laser pulse for a series of angles around the scanner.

Laser scanners used for autonomous vehicles are considerably different from those used for indoor or slow-moving robots. Vehicle laser scanners made by companies such as Velodyne must deal with the significant aerodynamic forces, vibrations, and temperature swings common to the automotive environment. Since vehicles typically move much faster than smaller robots, vehicle sensors must have considerably longer range so that sufficient reaction time is possible. Additionally, many software tasks for autonomous driving, such as detecting vehicles and obstacles, work much better when multiple laser *scanlines* are received each time the device rotates, rather than just one. These extra scanlines can be extremely useful when distinguishing between classes of objects, such as between trees and pedestrians. To produce multiple scanlines, automotive laser scanners often have multiple lasers mounted together in a rotating structure, rather than simply rotating a mirror. All of these additional features naturally add to the complexity, weight, size, and thus the cost of the laser scanner.

The complex signal processing steps required to produce range estimates are virtually always handled by the firmware of the laser scanner itself. The devices typically output a vector of ranges several dozen times per second, along with the starting and stopping angles of each measurement vector. In ROS, laser scans are stored in `sensor_msgs::LaserScan` messages, which map directly from the output of the laser scanner. Each manufacturer, of course, has their own raw message formats, but ROS drivers exist to translate between the raw output of many popular laser scanner manufacturers and the `sensor_msgs::LaserScan` message format.

Shaft Encoders

Estimating the motions of the robot is a critical component of virtually all robotic systems, ranging from low-level control schemes to high-level mapping, localization, and manipulation algorithms. Although estimates can be derived from many sources, the simplest and often most accurate estimates are produced simply by counting how many times the motors or wheels have turned.

Many different types of *shaft encoders* are designed expressly for this purpose. Shaft encoders are typically constructed by attaching a marker to the shaft, and measuring its motion relative to another frame of reference, such as the chassis of the robot or the previous link on a manipulator arm. The implementation may be done with magnets, optical discs, variable resistors, or variable capacitors, among many other options, with tradeoffs including size, cost, accuracy, maximum speed, and whether or not the measurement is *absolute* or *relative* to the position at power-up. Regardless, the principle remains the same: the angular position of a marker on a shaft is measured relative to an adjacent *frame of reference*.

Just like automobile speedometers and odometers, shaft encoders are used to count the precise number of rotations of the robot's wheels, and thereby estimate how far the vehicle has traveled and how much it has turned. Note that *odometry* is simply a count of how many times the drive wheels have turned. It is *not* a direct measurement of the vehicle position. Minute differences in wheel diameters, tire pressures, carpet weave direction (really!), axle misalignments, minor skidding, and countless other sources of error are **cumulative** over time. As a result, the raw odometry estimates of **any** robot will drift; the longer the robot drives, the more error accumulates in the estimate. For example, a robot traveling down the middle of a long, straight corridor will **always** have odometry that is a gradual curve. Put another way, if both tires of a differential-drive robot are turned in the same direction at the exact same *wheel velocity*, the robot will **never** drive in a truly straight line. This is why mobile robots need additional sensors and clever algorithms to build maps and navigate.

Shaft encoders are also used extensively in robot manipulators. The vast majority of manipulator arms have at least one shaft encoder for every rotary joint, and the vector of shaft encoder readings is often called the *manipulator configuration*. When combined with a geometric model of each link of a manipulator arm, the shaft encoders allow higher-level collision-avoidance, planning, and trajectory following algorithms to control the robot.

Because the mobility and manipulation uses of shaft encoders are quite different, the ROS conventions for each use are also quite different. Although the raw encoder counts may also be reported by some mobile-base device drivers, odometry estimates are most useful when reported as a *spatial transformation* represented by a `geometry_msgs::Transform` message. This concept will be discussed at great length throughout the book, but in general, a spatial transform describes one frame of reference relative to another frame of reference. In this case, the odometry transform typically describes the shaft-encoder's odometric estimate relative to the position of the robot at power-up, or where its encoders were last reset.

In contrast, the encoder readings for manipulator arms are typically broadcast by ROS manipulator device drivers as `sensor_msgs::JointState` messages. The `JointState` message contains vectors of angles in radians, and angular velocities in radians per second. Since typical shaft encoders have thousands of discrete states per revolution, the ROS device drivers for manipulator arms are required to scale the encoders as required, accounting for transmissions and linkages, to produce a `JointState` vector with standard units. These messages are used extensively by ROS software packages, as they provide the minimal complete description of the state of a manipulator.

That about covers it for the physical parts of a robot system. We now turn our attention to the "brains", where the robot interprets sensor data and determines how to move it's body, and where we'll be spending most of our time in this book.

Computation

Impressive robotic systems have been implemented on computing resources ranging from large racks of servers down to extremely small and efficient 8-bit microcontrollers. Fierce debates have raged throughout the history of robotics as to exactly how much computer processing is required to produce robust, useful robot behavior. Insect brains, for example, are extremely small and power-efficient, and yet insects are arguably the most successful life forms on the planet. Biological brains process data very differently from "mainstream" systems-engineering approaches of human technology, which has led to large and sustained research projects which study and try to replicate the success of bio-inspired computational architectures.

ROS takes a more traditional software-engineering approach to robotic computational architecture; as described in the first few chapters of this book, ROS uses a dynamic message-passing graph to pass data between software nodes which are typically isolated by the POSIX process model. This does not come for free. It certainly requires additional CPU cycles to serialize a message from one node, send it over some interprocess or network communications method to another node, and deserialize it for another node. However, it is our opinion that the rapid-prototyping and software-integration benefits of this architecture outweigh its computational overhead.

Because of this messaging overhead and the emphasis on module isolation, ROS is not currently intended to run on extremely small microcontrollers. ROS can be (and has been) used to emulate and rapid-prototype minimalist processing paradigms. Typically, however, ROS is used to build systems that include considerable perceptual input and complex processing algorithms, where its modular and dynamically-extensible architecture can simplify system design and operation.

ROS currently must run on top of a full-featured operating system such as Linux or Mac OS X. Fortunately, the continuing advance of Moore's Law and mass-market demand for battery-powered devices has led to ever-smaller and more power-efficient platforms capable of running full operating systems. ROS can run on small-form-factor embedded computer systems such as Gumstix, Raspberry Pi, or the BeagleBone, among many others. Going up the performance and power curve, ROS has been widely used on a large range of laptops, desktops, and servers. Human-scale robots often carry one or more standard PC motherboards running Linux headless, which are accessed over a network link.

Actual Robots

The previous sections have described subsystems commonly found on many types of robots running ROS. Many of these robots used in research settings are custom-built to investigate a particular research problem. However, there are a growing number of standard products which can be purchased and used "out of the box" for research,

development, and operations in many domains of robotics. The following sections will describe several of these platforms which will be used for examples throughout the rest of the book.

PR2

The PR2 robot was one of the original ROS target platforms. In many ways, it was the ultimate research platform for robotics software at the time of its design in 2010. Its mobile base is actuated by four steerable casters and has a laser scanner for navigation. Atop this mobile base, the robot has a telescoping torso which carries two human-scale 7-DOF arms. The arms have a unique passive mechanical counterbalance, which permits the use of surprisingly low-power motors for human-scale arms.

The PR2 has a pan/tilt head equipped with a wide range of sensors, including a "nodding" laser scanner which can tilt up and down independently of the head, a pair of stereo cameras for short and long distances, and a Kinect depth camera. Additionally, each forearm of the robot has a camera, and the gripper fingertips have tactile sensors. All told, the PR2 has two laser scanners, six cameras, a depth camera, four tactile arrays, and 1 kHz encoder feedback. All of this data is handled by a pair of computers in the base of the robot, with an onboard gigabit network connecting them to a pair of WiFi radios.

All of this functionality came at a price, since the PR2 was not designed for low cost. When it was commercially available, PR2 listed for about $400,000.[1] Despite this financial hurdle, its fully-integrated "out-of-the-box" experience was a landmark for research robots, and is why PR2 robots are being actively used in dozens of research labs around the world.

Robonaut 2

The NASA/GM Robonaut 2 is a human-scale robot designed with the extreme reliability and safety systems necessary for operation aboard the International Space Station. At time of writing, the Robonaut 2 aboard the space station is currently running ROS for high-level task control. Much more information is available at *http://robonaut.jsc.nasa.gov*

Turtlebot

Turtlebot was designed in 2011 as a minimalist platform for ROS-based mobile robotics education and prototyping. It has a small differential-drive mobile base with an internal battery, power regulators, and charging contacts. Atop this base is a stack of laser-cut "shelves" that provide space to hold a netbook computer, depth camera, and lots of open

1. All prices are approximate, as of the time of writing, and quited in US Dollars.

space for prototyping. To control cost, Turtlebot relies on a depth camera for range sensing; it does not have a laser scanner. Despite this, mapping and navigation can work quite well for indoor spaces. Turtlebots are available from several manufacturers for less than $2,000. More information is available at *http://turtlebot.org*

Because the shelves of Turtlebot are covered with mounting holes, many owners have added additional subsystems to Turtlebot, such as small manipulator arms, additional sensors, or upgraded computers. However, the "stock" Turtlebot is an excellent starting point for indoor mobile robotics. Many similar systems exist from other vendors, such as the Pioneer and Erratic robots and thousands of custom-built mobile robots around the world. The examples in this book will use Turtlebot, but any other small differential-drive platform could easily be substituted.

Baxter

The Baxter Robot was released in 2012 by Rethink Robotics. It is a stationary, low-cost, dual-arm manipulation platform intended for flexible industrial automation. With its list price of $22,000, Baxter is intended to create a new segment of the industrial robotics market, where low cost, ease of (re)configuration, and intrinsic human-safety are more important than raw speed, accuracy, or decades-long durability.

Baxter uses ROS internally to manage its various hardware and software subsystems. Although the internal installation of ROS is closed in the industrial-certified version of the robot, Rethink Robotics sells a version of Baxter with a public ROS interface, called the Baxter Research Robot. This platform is currently the most affordable human-scale dual-arm manipulator on the market.

Simulators

Although the preceding list of robots includes platforms which we consider to be remarkably low cost compared to prior robots of similar capabilities, they are still significant investments. In addition, real robots require logistics including lab space, recharging batteries, and operational quirks which often become part of the institutional knowledge of the organization operating the robot. Sadly, even the best robots break periodically due to various combinations of operator error, environmental conditions, manufacturing or design defects, and so on.

Many of these headaches can be avoided by using *simulated* robots. At first glance, this seems to defeat the whole purpose of robotics; after all, the very definition of a robot involves perceiving and/or manipulating the environment. Software robots, however, are extraordinarily useful. In simulation, we can model as much or as little of reality as we desire. Sensors and actuators can be modeled as ideal devices, or they can incorporate various levels of distortions, errors, and unexpected faults. Although data logs can be used in automated test suites to verify that sensing algorithms produce expected results,

automated testing of control algorithms typically requires simulated robots, since the algorithms under test need to be able to experience the consequences of their actions.

Simulated robots are the ultimate low-cost platforms. They are free! They do not require complex operating procedures; you simply spawn a `roslaunch` script, wait a few seconds, and a shiny new robot is created. At the end of the experimental run, a quick `Ctrl-C` and the robot vaporizes. For those of us who have spent many long nights with the pain and suffering caused by operating real robots, the benefits of simulated robots are simply magical.

Due to the isolation provided by the messaging interfaces of ROS, the vast majority of the robot's software graph can be run identically whether it is controlling a real robot or a simulated robot. At runtime, as the various nodes are launched, they simply find each other and connect. Simulation input and output streams connect to the graph in the place of the device drivers of the real robot. Although some parameter tuning is often required, ideally the *structure* of the software will be the same, and often times the simulation can be modified to reduce the amount of parameter tweaks required when transitioning between simulation and reality.

As alluded to in the previous paragraphs, there are many use cases for simulated robots, ranging from algorithm development to automated software verification. This has led to the creation of a large number of robot simulators, many of which have integrate nicely with ROS. The following sections describe two simulators which will be used in this book.

Stage

For many years, the two-dimensional simultaneous localization and mapping (SLAM) problem was one of the most heavily-research topics in the robotics community. A number of 2D simulators were developed in response to the need for repeatable experiments as well as the many practical annoyances of gathering long datasets of robots driving down endless office corridors. Canonical laser rangefinders and differential-drive robots were modeled, often using simple *kinematic* models which enforce that, for example, the robot stays plastered to a 2D surface and its range sensors only interact with vertical walls, creating worlds that vaguely resemble Pac-Man. Although limited in scope, these 2D simulators are very fast computationally, and they are generally quite simple to interact with.

Stage is an excellent example of this type of 2D simulator. It has a relatively simple modeling language which allows the creation of planar worlds with simple types of objects. Stage was designed from the outset to support multiple robots simultaneously interacting with the same world. It has been wrapped with a ROS integration package which accepts velocity commands from ROS, and outputs an odometric transformation as well as the simulated laser rangefinders from the robot(s) in the simulation.

Gazebo

Although Stage and other 2D simulators are computationally efficient and excel at simulating planar navigation in office-like environments, it is important to note that planar navigation is only one aspect of robotics. Even when only considering robot navigation, a vast array of environments require non-planar motion, ranging from outdoor ground vehicles to aerial, underwater, and space robotics. Three-dimensional simulation is necessary for software development in these environments.

In general, robot motions can be divided into *mobility* and *manipulation*. The mobility aspects can be handled by 2- or 3-dimensional simulators in which the environment around the robot is *static*. Simulating manipulation, however, requires a significant increase in the complexity of the simulator to handle the dynamics of not just the robot, but also the *dynamic* models in the scene. For example, at the moment that a simulated household robot is picking up a handheld object, contact forces must be computed between the robot, the object, and the surface the object was previously resting upon.

Simulators often use *rigid-body* dynamics, in which all objects are assumed to be incompressible, as if the world was a giant pinball machine. This assumption drastically improves the computational performace of the simulator, but often requires clever tricks to remain stable and realistic, since many rigid-body interactions become *point contacts* which do not accurately model the true physical phenomena. The art and science of managing the tension between computational performance and physical realism is highly nontrivial. There are many approaches to this tradeoff, with many well-suited to some domains but ill-suited to others.

Like all simulators, Gazebo is the product of a variety of tradeoffs in its design and implementation. Historically, Gazebo has used the Open Dynamics Engine for rigid-body physics, but recently it has gained the ability to choose between physics engines at startup. For the purposes of this book, we will be using Gazebo with either the Open Dynamics Engine or with the Bullet Physics library, both of which are capable of real-time simulation with relatively simple worlds and robots, and with some care, can produce physically plausible behavior.

ROS integrates closely with Gazebo through the `gazebo_ros` package. This package provides a Gazebo *plugin* module which allows bidirectional communication between Gazebo and ROS. Simulated sensors and physics data can stream from Gazebo to ROS, and actuator commands can stream from ROS back to Gazebo. In fact, by choosing consistent names and data types of these data streams, it is possible for Gazebo to *exactly* match the ROS API of a robot. When this is achieved, all of the robot software above the device-driver level can be run identically on both the real robot, and (after parameter tuning) in the simulator. This is an enormously powerful concept, and will be used extensively throughout this book.

Summary

In this chapter, we've looked at the subsystems of a typical robot, focusing on the types of robots that ROS is most concerned with: mobile manipulation platforms. By now, you should have a pretty good idea of what a robot looks like, and you should be starting to figure out how ROS might be used to control one, reading data from the sensors, figuring out how to interpret it and what to do, and sending commands to the acuators to make it move.

The next chapter ties together all of the material you've already read, and shows you how to write code that will make a real (or simulated) robot wander aimlessly around a real (or simulated) world.

CHAPTER 7
Wander-bot

The first chapters of this book introduced many of the abstract ROS concepts used for communication between modules, such as Topics, Services, and Actions. Then, the previous chapter introduced many of the sensing and actuation subsystems commonly found in modern robots. In this chapter, we will put these concepts together to create a robot which can wander around its environment. This might not sound terribly earth-shattering, but such a robot is actually capable of doing meaningful work: there is an entire class of tasks which are accomplished by driving across it. For example, many vacuuming or mopping tasks can be accomplished by cleverly designed and carefully-tuned algorithms where the robot, carrying its cleaning tool, traverses the environment somewhat randomly. The robot will eventually drive over all parts of the environment, completing its task.

In this chapter, we will go step-by-step through the process of writing minimalist ROS-based robot control software, including creating a ROS package and testing it in simulation.

Creating a package

As described in Chapter 2, ROS-based software is organized into *packages*, each of which contains some combination of code, data, and documentation. The ROS ecosystem includes thousands of publicly-available packages in open repositories, and many thousand more packages are certainly lurking behind organizational firewalls.

Packages are used inside *workspaces* of the ROS build system. A ROS workspace is simply a directory which contains a few subdirectories which are described in the following list:

- `src`: the parent directory of the actual package directories
- `build`: the build products: libraries, executables, and so on

- `devel`: auto-generated files used to describe the workspace to the build system
- `install`: useful to test the build scripts, to ensure they will install all required files

This sounds complex, but it is surprisingly easy and efficient to use, thanks to the many scripts which automate the ROS build system.

Creating a workspace and a package is very simple. As always, be sure that the shell includes the ROS environment variables:

```
source /opt/ros/indigo/setup.bash
```

Now, create the workspace, which we will place in `~/workspace` :

```
mkdir -p ~/workspace/src
cd ~/workspace
catkin_init_workspace
```

That's it! The workspace is ready to use. It is just one more command to create a package in the new workspace. To create a package called `wanderbot` that uses `rospy` (python ROS client), and a few standard ROS message packages, we will use the `catkin_create_pkg` command:

```
cd ~/workspace/src
catkin_create_pkg wanderbot rospy std_msgs geometry_msgs sensor_msgs
```

The first argument, `wanderbot`, is the name of the new package we want to create. The following arguments are the names of packages that the new package depends on. Why? Because the ROS build system needs to know the package dependencies in order to efficiently keep the builds up to date when source files change, and to generate any required installation dependencies when packages are released.

After running the `catkin_create_pkg` command, there will be a package directory called `wanderbot` inside the workspace, including the following files:

- `~/workspace/src/wanderbot/CMakeLists.txt`: a starting point for the build script for this package
- `package.xml`: a machine-readable description of this package, including its name, description, author, license, and which other packages it depends on to build and run.

Now that we have our `wanderbot` package created, we can create a minimal ROS *node* inside of it. In the previous chapters, we were just sending generic messages between nodes, such as strings or integers. Now, we can send something robot-specific. The following code will send a stream of motion commands 10 times per second, which alternates every three seconds between driving and stopping. When driving, the program will send forward velocity commands of 0.5 meters per second. When stopped, it

will send commands of zero meters per second. This program is shown in red_light_green_light.py.

red_light_green_light.py.

```
import rospy
from geometry_msgs.msg import Twist

cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('red_light_green_light')

red_light_twist = Twist()
green_light_twist = Twist()
green_light_twist.linear.x = 0.5 # drive straight ahead at 0.5 m/s

driving_forward = False
rate = rospy.Rate(10)
light_change_time = rospy.Time.now()

while not rospy.is_shutdown():
  if driving_forward:
    cmd_vel_pub.publish(green_light_twist)
  else:
    cmd_vel_pub.publish(red_light_twist)
  if light_change_time > rospy.Time.now():
    driving_forward = not driving_forward
    light_change_time = rospy.Time.now() + rospy.Duration(3)
  rate.sleep()
```

A lot of red_light_green_light.py is just setting up the system and its data structures. The most important behavior of this program is to change behavior every 3 seconds from driving to stopping. This is performed by the three-line block copied beow, which uses `rospy.Time` to measure the duration since the last change of behavior:

```
if light_change_time > rospy.Time.now():
  driving_forward = not driving_forward
  light_change_time = rospy.Time.now() + rospy.Duration(3)
```

Like all Python scripts, it is convenient to make it an executable:

```
chmod +x red_light_green_light.py
```

Now, we can use our program to control a simulated robot. But first, we need to make sure that the Turtlebot simulation stack is installed:

```
sudo apt-get install ros-indigo-turtlebot-gazebo
```

We are now ready to instantiate a Turtlebot in the simulator. We'll use a simple world to start, by typing this in a new terminal window (remember to hit the [TAB] key often when typing ROS shell commands for auto-completion):

```
roslaunch turtlebot_gazebo turtlebot_playground.launch
```

In a second terminal, let's fire up our control node:Now we can fire up our node:

```
./red_light_green_light.py cmd_vel:=cmd_vel_mux/input/teleop
```

The `cmd_vel` remapping is necessary so that we are publishing our `Twist` messages to the topic that the Turtlebot software stack is expecting. Although we could have declared our `cmd_vel_pub` to publish to this topic in the `red_light_green_light.py` source code, our usual goal is to write ROS nodes that are as generic as possible, and in this case, we can easily remap `cmd_vel` to whatever is required by any robot's software stack.

When `red_light_green_light.py` is running, you should now see a Turtlebot alternating every second between driving forwards and stopping. Progress! When you are bored with it, just give a `Ctrl-C` to the newly-created node as well as the turtlebot simulation.

Reading sensor data

Blindly driving around is fun, but we typically want robots to use sensor data. Fortunately, streaming sensor data into ROS nodes is quite easy. Whenever we want to receive a topic in ROS, it's often helpful to first just echo it to the console, to make sure that it is actually being published under the topic name we expect, and to confirm that we understand the datatype.

In the case of Turtlebot, we want to see something like a laser scan: a linear vector of ranges from the robot to the nearest obstacles in various directions. To save cost, sadly, the Turtlebot does not have a real laser scanner. It does, however, have a Kinect depth camera, and the Turtlebot software stack extracts the middle few rows of the Kinect's depth image, does a bit of filtering, and then publishes the data as `sensor_msgs/LaserScan` messages on the `/scan` topic. This means that from the standpoint of the high-level software, the data shows up exactly like "real" laser scans on more expensive robots. The field of view is just more narrow, and the maximum detectable range is quite a bit shorter than typical laser scanners.

To start, we can just dump the `scan` topic to console to verify that the simulated laser scanner is working. First, fire up a Turtlebot simulation, if one isn't already running:

```
roslaunch turtlebot_gazebo turtlebot_playground.launch
```

Then, in another console, we can use `rostopic`

```
rostopic echo scan
```

This will print a continuous stream of text representing the `LaserScan` messages. When you're bored, press `Ctrl-C` to stop it. Most of the text is the `ranges` member of the `LaserScan` message, which is exactly what we are interested in: the `ranges` array contains the range from the Turtlebot to the nearest object at bearings easily computed from the `ranges` array index. Specifically, if the message instance is named `msg`, we can compute

the bearing for a particular range estimate as follows, where `i` is the index into the `ranges` array:

```
bearing = msg.angle_min + i * msg.angle_max / len(msg.ranges)
```

To retrieve the range to the nearest obstacle directly in front of the robot, we will select the middle element of the `ranges` array:

```
range_ahead = msg.ranges[len(msg.ranges)/2]
```

or, to return the range of the closest obstacle detected by the scanner:

```
closest_range = min(msg.ranges)
```

This signal chain is deceptively complex: we are picking out elements of an *emulated* laser scan, which is itself produced by picking out a few of the middle rows of the Turtlebot's Kinect depth camera, which is itself generated in Gazebo by backprojecting rays into a simulated environment! It's hard to over-emphasize the utility of simulation for robot software development.

The following listing is a complete ROS node which prints the distance to an obstacle directly in front of the robot:

range_ahead.py.

```
import rospy
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
  range_ahead = msg.ranges[len(msg.ranges)/2]
  print "range ahead: %0.1f" % range_ahead

rospy.init_node('range_ahead')
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
rospy.spin()
```

This little program shows how easy it is to connect to data streams in ROS and process them in Python. The `scan_callback` function is called each time a new message arrives on the `scan` topic. This callback function then prints the range measured to the object directly in front of the robot, by picking the middle element of the `ranges` field of the LaserScan message:

```
def scan_callback(msg):
  range_ahead = msg.ranges[len(msg.ranges)/2]
  print "range ahead: %0.1f" % range_ahead
```

We can experiment with this program in Gazebo by dragging and rotating the Turtlebot around the world. First, click the Move icon in the Gazebo toolbar to enter the Move mode, and then click and drag the Turtlebot around the scene. The terminal running `range_ahead.py` will print a continually changing stream of numbers indicating the

range (in meters) from the Turtlebot to the nearest obstacle (if any) directly in front of it.

Gazebo also has a Rotate tool which will (by default) rotate a model about its vertical axis. Both the Move and Rotate tools will immediately affect the output of the `range_ahead.py` program, since the simulation (by default) stays running while models are being dragged and rotated.

Sensing and Actuation: Wander-bot!

We have now written `red_light_green_light.py`, which causes Turtlebot to drive *open-loop*, and we have also written `range_ahead.py`, which uses the Turtlebot's sensors to estimate the range to the nearest object directly in front of the Turtlebot. We can put these two capabilities together and write `wander.py`, which will cause Turtlebot to drive straight ahead until it sees an obstacle within 0.8 meters or times out after 30 seconds. Then, Turtlebot will stop and spin to a new heading. It will continue doing those two things until the end of time or Ctrl+C, whichever comes first.

wander.py.

```
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

def scan_callback(msg):
  global g_range_ahead
  g_range_ahead = min(msg.ranges)

g_range_ahead = 1 # anything to start
scan_sub = rospy.Subscriber('scan', LaserScan, scan_callback)
cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
rospy.init_node('wander')
state_change_time = rospy.Time.now()
driving_forward = True
rate = rospy.Rate(10)

while not rospy.is_shutdown():
  if driving_forward:
    if (g_range_ahead < 0.8 or rospy.Time.now() > state_change_time):
      driving_forward = False
      state_change_time = rospy.Time.now() + rospy.Duration(5)
  else: # we're not driving_forward
    if rospy.Time.now() > state_change_time:
      driving_forward = True # we're done spinning, time to go forwards!
      state_change_time = rospy.Time.now() + rospy.Duration(30)
  twist = Twist()
  if driving_forward:
    twist.linear.x = 1
  else:
```

```
    twist.angular.z = 1
  cmd_vel_pub.publish(twist)

  rate.sleep()
```

As will always be the case with ROS python programs, we start by importing rospy and the ROS message types we'll need: the Twist and LaserScan messages. Since this program is so simple, we'll just use a global variable called `g_range_ahead` to store the minimum range that our (simulated) laser scanner detects in front of the robot. This makes the `scan_callback` function very simple; it just copies out the range to our global variable. And yes, this is horrible programming practice in complex programs, but for this small example, we'll pretend it's OK.

We start the actual program by creating a subscriber to `scan` and a publisher to `cmd_vel`, as we did previously. We also set up two variables which we'll use in our controller logic: `state_change_time` and `driving_forward`. The `rate` variable is a helpful construct in `rospy`: it helps create loops that run at a fixed frequency. In this case, we'd like to run our controller at 10 Hz, so we construct a `rospy.Rate` object by passing 10 to its constructor. Then, we call `rate.sleep()` at the end of our main loop, and each time through, `rospy` will adjust the amount of actual sleeping time so that we run at something close to 10 Hz on average. The actual amount of sleeping time will depend on what else is being done in the control loop, and the speed of the computer; we can just call `rospy.Rate.sleep()` and not worry about it.

The actual control loop is kept as simple as possible. The robot is in one of two states: `driving_forward` or `not driving_forward`. When in the `driving_forward` state, the robot keeps driving until it either sees an obstacle within 0.8 meters, or it times out after 30 seconds, after which it transitions to the `not driving_forward` state:

```
if (g_range_ahead < 0.8 or rospy.Time.now() > state_change_time):
  driving_forward = False
  state_change_time = rospy.Time.now() + rospy.Duration(5)
```

When the robot is in the `not driving_forward` state, it simply spins in place for five seconds, then transitions back to the `driving_forward` state:

```
if rospy.Time.now() > state_change_time:
  driving_forward = True # we're done spinning, time to go forwards!
  state_change_time = rospy.Time.now() + rospy.Duration(30)
```

As before, we can quickly test our program in a Turtlebot simulation. Let's start one up:

```
roslaunch turtlebot_gazebo turtlebot_playground.launch
```

Then, in a separate console, we can make `wander.py` executable and run it:

```
chmod +x red_light_green_light.py
./wander.py cmd_vel:=cmd_vel_mux/input/teleop
```

The turtlebot will wander around aimlessly, while avoiding collisions with obstacles it can see. Hooray!

Summary

In this chapter, we first created an open-loop control system in `red_light_green_light.py` that started and stopped the Turtlebot based on a simple timer. Then, we saw how to read the information from Turtlebot's depth camera. Finally, we closed the loop between sensing and actuation by creating `wander-bot`, which causes the Turtlebot to avoid obstacles and randomly wander around its environment. This brought together all of the aspects of the book thus far: the streaming data-transport mechanisms of ROS, the discussion of robot sensors and actuators, and the simulation framework of Gazebo. In the next chapter, we will start making things more complex by listening to user input, as we create Teleop-bot.

CHAPTER 8

Teleop Bot

This chapter will describe how to drive a robot around via *teleoperation*. Although the term "robot" often brings up images of *fully autonomous* robots which are able to make their own decisions in all situations, there are many domains in which close human guidance is standard practice due to a variety of factors. Since teleoperated systems are, generally speaking, simpler than autonomous systems, they make a natural starting point. In this chapter, we will construct progressively more complex teleoperation systems.

As discussed in the previous "Wander Bot" chapter, we drive Turtlebot by publishing a stream of `Twist` messages. Although the `Twist` message has the ability to describe full 3-d motion, when operating differential-drive planar robots, we only need to populate two members: the linear (forward/backward) velocity, and the angular velocity about the vertical axis, which is often called *yaw*. From those two fields, it is then an exercise in trigonometry to compute the required wheel velocities of the robot as a function of the spacing of the wheels and their diameter. This calculation is usually done at low levels in the software stack, either in the robot's device driver or in the firmware of a microcontroller onboard the robot. From the teleoperation software's perspective, we simply command the linear and angular velocities in meters per second and radians per second, respectively.

Given that we need to produce a stream of velocity commands to move the robot, the next question is, how can we elicit these commands from the robot operator? There are a wide variety of approaches to this problem, and naturally we should start with the simplest approach to program: keyboard input.

Development pattern

Throughout the remainder of the book, we will encourage a development pattern which makes use of the ROS debugging tools wherever possible. Since ROS is a distributed

system with topic-based communications, we can quickly create testing environments to help our debugging, so that we are only starting and stopping a single piece of the system every time we need to tweak a bit of code. By structuring our software into a collection of very small message-passing programs, it becomes easier and more productive to insert ROS debugging tools into these message flows.

In the specific case of producing teleop-bot velocity commands, we will write two programs: one which listens for keystrokes and then broadcasts them as ROS messages, and one program which listens for keystroke ROS messages and outputs `Twist` messages in response. This extra layer of indirection helps isolate the two functional pieces of this system, as well as makes it easier for us, or anyone else in the open-source community, to re-use the individual pieces in a completely different system. Creating a constellation of small ROS nodes often will simplify the creation of manual and (especially) automated software tests. For example, we can feed a pre-canned sequence of keystroke messages to the node which translates between keystrokes and motion commands, comparing the output motion command with the previously-defined "correct" responses. Then, we can set up automated testing to verify the correct behavior as the software evolves over time.

Once we have decided the highest-level breakdown of how a task should be split into ROS nodes, the next task is to write them! As is often the case with software design, sometimes it helps to create a *skeleton* of the desired system which prints console messages or just publishes dummy messages to other nodes in the system. However, our preferred approach is to build the required collection of new ROS nodes incrementally, with a strong preference towards writing *small* nodes.

Keyboard driver

The first node we need to write for keyboard-teleop-bot is a keyboard driver which listens for keystrokes and publishes them as `std_msgs::String` messages on the `key` topic. There are many ways to perform this task. This example uses the Python `termios` and `tty` libraries to place the terminal in raw mode and capture keystrokes, which are then published as `std_msgs::String` messages.

key_publisher.py.

```
import sys, select, tty, termios
import rospy
from std_msgs.msg import String

if __name__ == '__main__':
  key_pub = rospy.Publisher('keys', String, queue_size=1)
  rospy.init_node("keyboard_driver")
  rate = rospy.Rate(100)
  old_attr = termios.tcgetattr(sys.stdin)
  tty.setcbreak(sys.stdin.fileno())
```

```
print "Publishing keystrokes. Press Ctrl-C to exit..."
while not rospy.is_shutdown():
  if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
    key_pub.publish(sys.stdin.read(1))
  rate.sleep()
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

This program uses the `termios` library to capture raw keystrokes, which requires working around some quirks of how Unix consoles operate. Typically, consoles buffer an entire line of text, only sending it to programs when [ENTER] is pressed. In our case, we want to receive the keys on the our program's standard input stream as soon as they are pressed. To alter this behavior of the console, we first need to save the attributes:

```
old_attr = termios.tcgetattr(sys.stdin)
tty.setcbreak(sys.stdin.fileno())
```

Now, we can continually poll the `stdin` stream to see if any characters are ready. Although we could simply block on `stdin`, that would cause our process to not fire any ROS callbacks, should we add any in the future. Thus, it is good practice to instead call `select()` with a timeout of zero, which will return immediately. We will then spend the rest of our loop time inside `rate.sleep()`, as shown in this snippet:

```
  if select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]:
    key_pub.publish(sys.stdin.read(1))
  rate.sleep()
```

Finally, we need to put the console back into standard mode before our program exits:

```
termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

To test if the keyboard driver node is operating as expected, three terminals are needed. In the first terminal, run `roscore`. In the second terminal, run the `key_publisher.py` node. In the third terminal, run `rostopic echo keys` , which will print any and all messages that it receives on the `keys` topic to the console. Then, set focus back to the second terminal by clicking on it or using window manager shortcuts such as `ALT-TAB` to switch between terminals. Keystrokes in the second terminal should cause `std_msgs::String` messages to print to the console of the third terminal. Progress! When finished testing, press `Ctrl-C` in all terminals to shut everything down.

You'll notice that "normal" keys, such as the alphabet, numerals, and simple punctuation, work as expected. However, "extended" keys, such as the arrow keys, result in `std_msgs::String` messages that are either weird symbols or multiple messages (or both). That is expected, since our minimalist `key_publisher.py` node is just pulling characters one at a time from `stdin`, and improving `key_publisher.py` is an exercise left to the motivated reader! For the remainder of this chapter, we will use just alphabetic characters.

Motion generator

In this section, we will use the common keyboard mapping of W,X,A,D,S to express, respectively, that we want the robot to go forwards, backwards, turn left, turn right, and stop.

As a first attempt at this problem, we could make a ROS node which outputs a Twist message every time it receives a `std_msgs::String` message that starts with a character it understands:

keys_to_twist.py.

```
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }

def keys_cb(msg, twist_pub):
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key.
  vels = key_mapping[msg.data[0]]
  t = Twist()
  t.angular.z = vels[0]
  t.linear.x  = vels[1]
  twist_pub.publish(t)

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  rospy.spin()
```

This program uses a Python dictionary to store the mapping between keystrokes and the target velocities:

```
key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
```

In the callback function for the `keys` topic, incoming keys are looked up in this dictionary. If a key is found, the target velocities are extracted from the dictionary:

```
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key.
  vels = key_mapping[msg.data[0]]
```

In an effort to prevent runaway robots, most robot device drivers will automatically stop the robot if no messages are received in a few hundred milliseconds. The program in the previous listing would work, but only if it had a continual stream of key presses to

continually generate `Twist` messages for the robot driver. That would be exciting for a few seconds, but once the euphoria of "hey, the robot is moving!" wears off, we'll be searching for improvements!

Issues such as robot firmware timeouts can be tricky to debug. As with everything in ROS (and complex systems in general), the key for debugging is to find ways to divide the system into smaller pieces and discover where the problem lies. The `rostopic` tool can help in several ways. As in the previous section, start three terminals: one with `roscore`, one with `key_publisher.py`, and one with `keys_to_twist.py`. Then, we can start a fourth terminal for various incantations of `rostopic`.

First, we can see what topics are available:

```
$ rostopic list
```

whic provides the following output:

```
/cmd_vel
/keys
/rosout
/rosout_agg
```

The last two items, `/rosout` and `/rosout_agg`, are part of the general-purpose ROS logging scheme, and are always there. The other two, `/cmd_vel` and `/keys` , are what our programs are publishing. Now, let's dump the `cmd_vel` data stream to the console:

```
$ rostopic echo cmd_vel
```

Each time a valid key is pressed in the console with `key_publisher.py`, the `rostopic` console should print the contents of the resulting `Twist` message published by `keys_to_twist.py`. Progress! As always with ROS console tools, simply press `Ctrl-C` to exit. Next, we can use `rostopic hz` to compute the average rate of messages. Running this command:

```
$ rostopic hz cmd_vel
```

Will estimate the rate of messages on a topic every second, and print the estimated rate to the console. With `keys_to_twist.py`, this estimate will almost always be zero, with minor bumps up and down each time a key is pressed in the keyboard driver console.

To make this node useful for robots that require a steady stream of velocity commands, we will output a `Twist` message every 100 milliseconds, or at a rate of 10 Hz, by simply repeating the last motion command if a new key was not pressed. Although we could do something like this by using a `sleep(0.1)` call in the `while` loop, this would only ensure that the loop runs *no faster* than 10 Hz; the timing results would likely have quite a bit of variance since the scheduling and execution time of the loop itself is not taken into account. Since each different computers run different speeds, have different numbers of processors, and have different (and varying) loads on those processors, the exact amount of time the loop needs to sleep is not knowable ahead of time. Looping tasks

are better accomplished with the `ROS::Rate` construct, which continually estimates the time spent processing the loop to obtain more consistent results:

keys_to_twist_using_rate.py.

```
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
g_last_twist = None

def keys_cb(msg, twist_pub):
  global g_last_twist
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key.
  vels = key_mapping[msg.data[0]]
  g_last_twist.angular.z = vels[0]
  g_last_twist.linear.x  = vels[1]
  twist_pub.publish(g_last_twist)

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  rate = rospy.Rate(10)
  g_last_twist = Twist() # initializes to zero
  while not rospy.is_shutdown():
    twist_pub.publish(g_last_twist)
    rate.sleep()
```

Now, when the `keys_to_twist_using_rate.py` node is running, we will see a quite consistent 10 Hz output of messages when we run `rostopic hz cmd_vel`. This can be seen using a separate console running `rostopic echo cmd_vel`, as in the previous section. The key difference between this program and the previous one is the use of `rospy.Rate`:

```
  rate = rospy.Rate(10)
  g_last_twist = Twist() # initializes to zero
  while not rospy.is_shutdown():
    twist_pub.publish(g_last_twist)
    rate.sleep()
```

When debugging low-dimensional data, such as the velocity commands send to a robot, it is often useful to plot the data stream as a time series. ROS provides a command-line tool called `rqt_plot`, which can accept *any* numerical data message stream and plot it graphically in real time.

To create an `rqt_plot` visualization, we need to send rqt_plot the exact message field that we want to see plotted. To find this field name, we can use several methods. The simplest is to look at the output of `rostopic echo`. This is always printed as YAML, a simple whitespace-based markup format. For example, `rostopic echo cmd_vel` will print a series of records of this format:

Example 8-1. rostopic_echo_cmd_vel

```
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Nested structures are indicated by whitespace: first, the `linear` field structure has field names `x`, `y`, `z`, followed by the `angular` field structure, with the same members.

Alternatively, we can discover the topic datatype using `rostopic`:

```
$ rostopic info cmd_vel
```

This will print quite a bit of information about the topic publishers and subscribers, as well as stating that the `cmd_vel` topic is of type `geometry_msgs/Twist`. With this datatype name, we can use the `rosmsg` command to print the structure:

```
$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

This console output showed us that the `linear` and `angular` members of the `Twist` message are of type `geometry_msgs/Vector3`, which has fields named `x`, `y`, and `z`. Granted, we already knew that from the `rostopic echo` output, but `rosmsg show` is sometimes a useful way of obtaining this information when we don't have a data stream available to print to the console.

Now that we know the topic name and the names of the fields, we can generate streaming plots of the linear velocity that we are publishing, by using slashes to descend into the message structure and select the fields of interest. As mentioned previously, for planar differential-drive robots, the only non-zero fields in the `Twist` message will be the x-

axis linear (forward/backward) velocity, and the z-axis (yaw) angular velocity. We can start streaming those fields to a plot with a single command:

```
$ rqt_plot cmd_vel/linear/x cmd_vel/angular/z
```

This plot will look something like Figure 8-1 as keys are pressed and the stream of velocity commands changes.

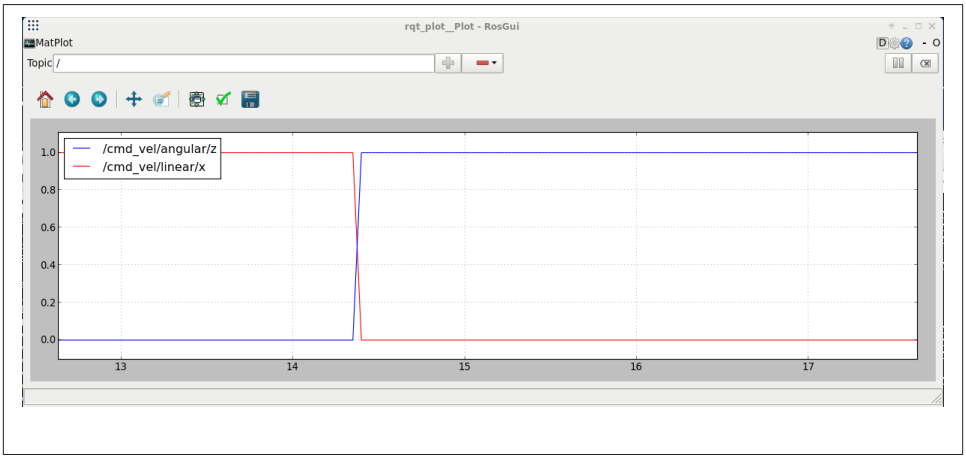


Figure 8-1. A live plot rendered by rqt_plot which shows the linear and angular velocity commands over time.

We now have a pipeline built where pressing letters on the keyboard will send velocity commands to a robot, and we can view those velocities in a live plot. That's great! But it has a lot of room for improvement. First, notice in the previous plot that our velocities are always either 0, -1, or +1. ROS uses SI units throughout, which means that we are asking our robot to drive forwards and backwards at one meter per second, and turn at one radian per second. Unfortunately, robots run at greatly varying speeds in different applications: for a robotic car, one meter per second is very slow, but for a small indoor robot navigating a corridor, one meter per second is actually quite fast. We need a way to *parameterize* this program, so that it can be used with multiple robots. We'll do that in the next section.

Parameter Server

We can improve the `keys_to_twist_using_rate.py` program by using ROS *parameters* to specify the linear and angular velocity scales. Of course, there are countless ways that we can give parameters to programs. When developing robotic systems, it is often useful to set parameters in a variety of ways: at the command line when debugging, in `roslaunch` files, from graphical interfaces, from other ROS nodes, or even in separate

parameter files to cleanly define behavior for multiple platforms or environment. The ROS master, often called roscore, includes a *parameter server* which can be read or written by all ROS nodes and command-line tools. The parameter server can support quite sophisticated interactions, but for our purposes in this chapter, we will only be setting parameters at the command line when running our teleop_bot nodes.

The parameter server is a generic key/value store. There are many strategies for how to name parameters, but for our teleop_bot node, we want a *private* parameter name. In ROS, a private parameter name is still publicly accessible, the notion of "private" simply means that its full name is formed by appending the parameter name to the node's name. This ensures that no name clashes can occur. For example, if our node name is keys_to_twist, we can have private parameters named keys_to_twist/linear_scale and keys_to_twist/angular_scale.

To set private parameters on the command line at the time the node is launched, the parameter name is prepended by an underscore and set using := syntax, as follows:

```
./keys_to_twist_parameterized.py _linear_scale:=0.5 _angular_scale:=0.4
```

This would set the keys_to_twist/linear_scale parameter to 0.5, and the keys_to_twist/angular_scale parameter to 0.4, immediately before the node is launched, which we will then use accordingly:

keys_to_twist_parameterized.py.

```
import rospy
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [0, -1],
                'a': [-1, 0], 'd': [1,  0],
                's': [ 0, 0] }
g_last_twist = None
g_vel_scales = [0.1, 0.1] # default to very slow

def keys_cb(msg, twist_pub):
  global g_last_twist, g_vel_scales
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key.
  vels = key_mapping[msg.data[0]]
  g_last_twist.angular.z = vels[0] * g_vel_scales[0]
  g_last_twist.linear.x  = vels[1] * g_vel_scales[1]
  twist_pub.publish(g_last_twist)

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  g_last_twist = Twist() # initializes to zero
  if rospy.has_param('~linear_scale'):
```

```
  g_vel_scales[1] = rospy.get_param('~linear_scale')
else:
  rospy.logwarn("WARNING: linear scale not provided. Defaulting to %.1f" % g_vel_scales[1])

if rospy.has_param('~angular_scale'):
  g_vel_scales[0] = rospy.get_param('~angular_scale')
else:
  rospy.logwarn("WARNING: angular scale not provided. Defaulting to %.1f" % g_vel_scales[0])

rate = rospy.Rate(10)
while not rospy.is_shutdown():
  twist_pub.publish(g_last_twist)
  rate.sleep()
```

At startup, this program queries the parameter server using `rospy.has_param()` and `rospy.get_param()`, and outputs a warning if the parameter was not set:

```
if rospy.has_param('~linear_scale'):
  g_vel_scales[1] = rospy.get_param('~linear_scale')
else:
  rospy.logwarn("WARNING: linear scale not provided. Defaulting to %.1f" % g_vel_scales[1])
```

If a parameters was not set, warning text will be printed to the console:

```
[WARN] [WallTime: 1429164125.989120] WARNING: linear scale not provided. Defaulting to 0.1
[WARN] [WallTime: 1429164125.989792] WARNING: angular scale not provided. Defaulting to 0.1
```

When we run `keys_to_twist_parameterized.py` with command-line parameters:

```
./keys_to_twist_parameterized.py _linear_scale:=0.5 _angular_scale:=0.4
```

The resulting stream of `Twist` messages is scaled as expected: for example, pressing *w* (move forwards) in the console running `key_publisher.py` will result in a stream of these messages appearing in the `rostopic echo cmd_vel` output:

```
linear:
  x: 0.5
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

Each time we launch `keys_to_twist_parameterized.py` we can specify the desired maximum velocities for our robot. Even more conveniently, we can put these parameters into *launch files* so that we don't have to remember them! But first, we need to deal with the physics problem of finite acceleration, which we will address in the next section.

Velocity ramps

Unfortunately, like all objects with mass, robots cannot start and stop instantaneously. Physics dictates that robots accelerate gradually over time. As a result, when a robot's wheel motors try to instantly jump to a wildly different velocity, typically something bad happens, such as skidding, belts slipping, "shuddering" as the robot repeatedly hits electrical current limits, or something can even break in the mechanical driveline. To avoid these problems, we should *ramp* our motion commands up and down over a finite amount of time. Often lower levels of robot firmware will enforce this, but in general, it's considered good practice not to send impossible commands to robots. The following version of *keys_to_twist.py* applies *ramps* to the outgoing velocity stream, to limit the instantaneous accelerations that we are asking of the motors:

keys_to_twist_with_ramps.py.

```
import rospy
import math
from std_msgs.msg import String
from geometry_msgs.msg import Twist

key_mapping = { 'w': [ 0, 1], 'x': [ 0, -1],
                'a': [ 1, 0], 'd': [-1,  0],
                's': [ 0, 0] }
g_target_twist = None
g_last_twist = None
g_last_send_time = None
g_vel_scales = [0.1, 0.1] # default to very slow
g_vel_ramps = [1, 1] # units: meters per second^2

def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate):
  # compute maximum velocity step
  step = ramp_rate * (t_now - t_prev).to_sec()
  sign = 1 if (v_target > v_prev) else -1
  error = math.fabs(v_target - v_prev)
  if error < step: # we can get there within this timestep. we're done.
    return v_target
  else:
    return v_prev + sign * step  # take a step towards the target

def ramped_twist(prev, target, t_prev, t_now, ramps):
  tw = Twist()
  tw.angular.z = ramped_vel(prev.angular.z, target.angular.z, t_prev, t_now, ramps[0])
  tw.linear.x = ramped_vel(prev.linear.x, target.linear.x, t_prev, t_now, ramps[1])
  return tw

def send_twist(twist_pub):
  global g_last_twist_send_time, g_target_twist, g_last_twist, g_vel_scales, g_vel_ramps
  t_now = rospy.Time.now()
  g_last_twist = ramped_twist(g_last_twist, g_target_twist, g_last_twist_send_time, t_now, g_vel_r
  g_last_twist_send_time = t_now
```

```
  twist_pub.publish(g_last_twist)

def keys_cb(msg, twist_pub):
  global g_target_twist, g_last_twist, g_vel_scales
  if len(msg.data) == 0 or not key_mapping.has_key(msg.data[0]):
    return # unknown key.
  vels = key_mapping[msg.data[0]]
  g_target_twist.angular.z = vels[0] * g_vel_scales[0]
  g_target_twist.linear.x  = vels[1] * g_vel_scales[1]
  send_twist(twist_pub)

def fetch_param(name, default):
  if rospy.has_param(name):
    return rospy.get_param(name)
  else:
    print "parameter [%s] not defined. Defaulting to %.3f" % (name, default)
    return default

if __name__ == '__main__':
  rospy.init_node('keys_to_twist')
  g_last_twist_send_time = rospy.Time.now()
  twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
  rospy.Subscriber('keys', String, keys_cb, twist_pub)
  g_target_twist = Twist() # initializes to zero
  g_last_twist = Twist()
  g_vel_scales[0] = fetch_param('~angular_scale', 0.1)
  g_vel_scales[1] = fetch_param('~linear_scale', 0.1)
  g_vel_ramps[0] = fetch_param('~angular_accel', 1.0)
  g_vel_ramps[1] = fetch_param('~linear_accel', 1.0)

  rate = rospy.Rate(20)
  while not rospy.is_shutdown():
    send_twist(twist_pub)
    rate.sleep()
```

The code is a bit more complex, but the main lines of interest are in the `ramped_vel` function, where the velocity is computed under the acceleration constraint provided as a parameter. Each time it is called, this function takes a step towards the target velocity, or, if the target velocity is within one step away, it goes directly to it:

```
def ramped_vel(v_prev, v_target, t_prev, t_now, ramp_rate):
  # compute maximum velocity step
  step = ramp_rate * (t_now - t_prev).to_sec()
  sign = 1 if (v_target > v_prev) else -1
  error = math.fabs(v_target - v_prev)
  if error < step: # we can get there within this timestep. we're done.
    return v_target
  else:
    return v_prev + sign * step  # take a step towards the target
```

At the command line, the following incantation of our teleop program will produce reasonable behavior for the Turtlebot:

```
$ ./keys_to_twist_with_ramps.py _linear_scale:=0.5 _angular_scale:=1.0 _linear_accel:=1.0 _angular
```

The motion commands we are sending the Turtlebot are now physically possible to achieve, as shown in Figure 8-2, since they take non-zero time to ramp up and down. Using the `rqt_plot` program as shown previously, we can generate a live plot of the system:

```
$ rqt_plot cmd_vel/linear/x cmd_vel/angular/z
```

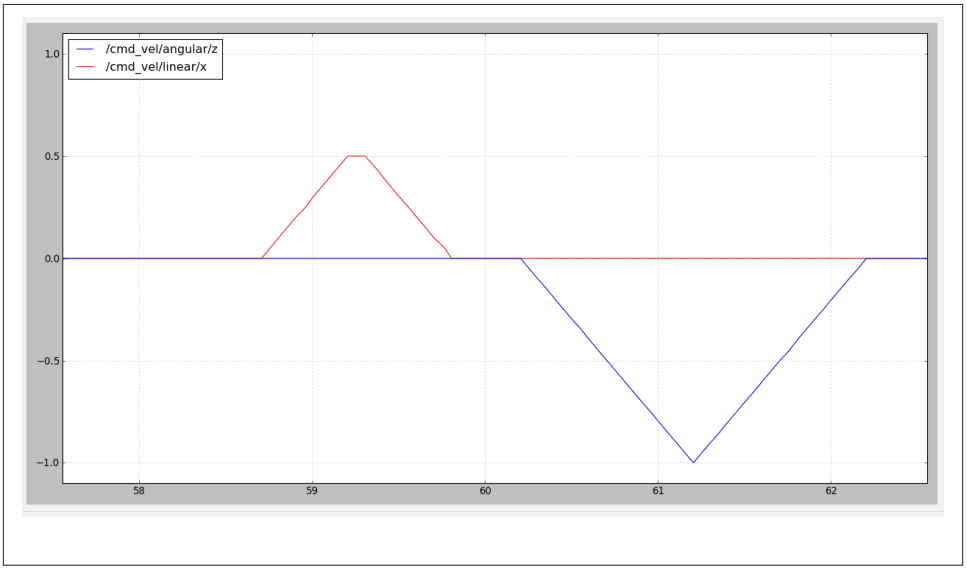


Figure 8-2. The velocity commands in this plot ramp up and down over finite time, allowing this trajectory to be physically achievable.

To reiterate: even if we were to give instanteously-changing or "step" commands to the Turtlebot, somewhere in the signal path, or in the physics of the mechanical system, the "step" command would be slowed into a ramp. The advantage to doing this in higher-level software is that there is simply more visibility into what is happening, and hence a better understanding of the behavior of the system.

Let's Drive!

Now that we have reasonable `Twist` messages streaming from our teleop program over the `cmd_vel` topic, we can now drive some robots. Let's start by driving a Turtlebot. Thanks to the magic of robot simulation, we can get a Turtlebot up and running with a single command:

```
$ roslaunch turtlebot_gazebo turtlebot_playground.launch
```

This will launch a Gazebo instance with a world similar to Figure 8-3, as well as emulating the software and firmware of the Turtlebot behind the scenes.

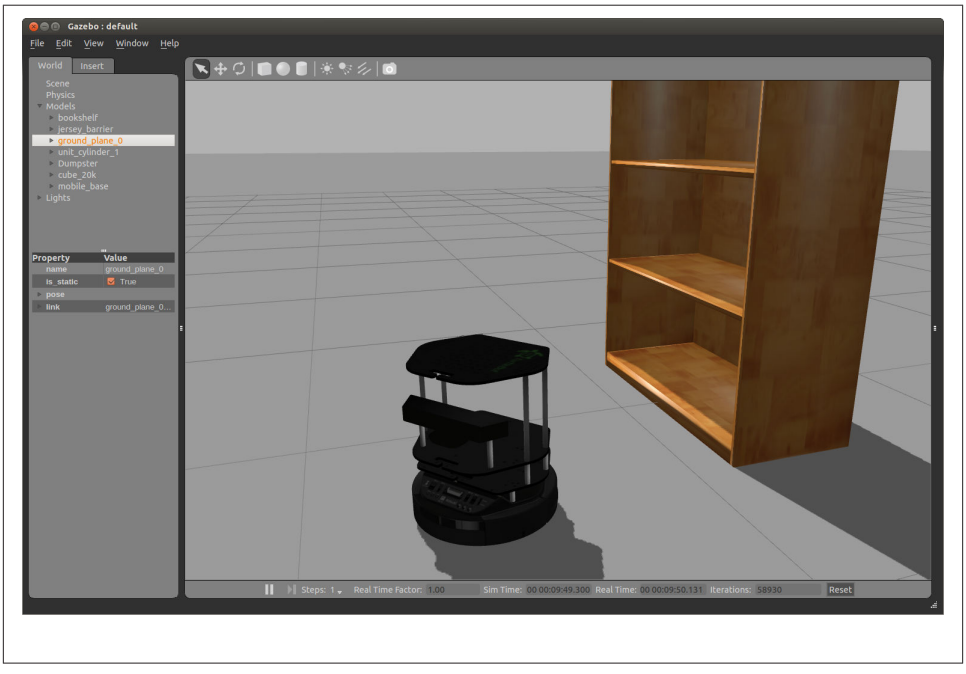


Figure 8-3. A snapshot of a simulated Turtlebot in front of a bookcase in the Gazebo simulator.

Next, we want to run our teleop program which broadcasts `Twist` messages on the `cmd_vel` topic:

```
$ ./keys_to_twist_with_ramps.py
```

However, if we do this, it won't work! Why? Because the Turtlebot looks for its `Twist` motion messages on a different topic. This is an **extremely** common problem to debug in distributed robotic software systems, or in any large software system, for that matter. We will describe a variety of tools for debugging these types of problems in a later chapter. For now, however, to make the Turtlebot simulator work, we need to publish `Twist` messages to a topic named `cmd_vel_mux/input/teleop` . To fix this, we need to *remap* our `cmd_vel` messages, so that they instead are published on the `cmd_vel_mux/input/teleop` topic. We can use the ROS remapping syntax to do this on the command line, without changing our source code:

```
$ ./keys_to_twist_with_ramps.py cmd_vel:=cmd_vel_mux/input/teleop
```

We can now drive the Turtlebot around in Gazebo using the W,A,S,D,X buttons on the keyboard. Hooray!

This style of teleoperation is similar to how remote-control cars work: the teleoperator maintains line-of-sight with the robot, sends motion commands, observes how they affect the robot and its environment, and reacts accordingly. However, it is often impossible or undesirable to maintain line-of-sight contact with the robot. This requires the teleoperator to visualize the robot's sensors and see the world through the "eyes" of the robot. ROS provides several tools to simplify development of such systems, including *rviz*, which will be described in the following section.

RViz

RViz stands for **ROS Visualization**. It is a general-purpose 3D visualization environment for robots, sensors, and algorithms. Like most ROS tools, it can be used for any robot and rapidly configured for a particular application. For teleoperation, we want to be able to see the camera feed of the robot. First, we will start from the configuration described in the previous section, where we have four terminals open: one for roscore, one for the keyboard driver, one for the `keys_to_teleop_with_rates.py` program, and one that ran a roslaunch script to bring up `gazebo` and a simulated turtlebot. Now, we'll need a fifth console to run `rviz`, which is in its own package, also called `rviz` :

```
$ rosrun rviz rviz
```

`rviz` can plot a variety of data types streaming through a typical ROS system, with heavy emphasis on the three-dimensional nature of the data. In ROS, all forms of data are attached to a *frame of reference* where the data was taken. For example, the camera on a Turtlebot is attached to the turtlebot reference frame. The *odometry* reference frame, often called `odom`, is taken by convention to have its origin at the location where the robot was powered on, or where its odometers were most recently reset. Each of these frames can be useful for teleoperation, but it is often useful to have a "chase" perspective, which is immediately behind the robot and looking over its "shoulders." This is because simply viewing the robot's camera frame can be deceiving, since the field-of-view of a camera is often much more narrow than we are used to as humans, and thus it is easy for teleoperators to bonk the robot's "shoulders" when turning corners.

Like many complex graphical user interfaces (GUIs), `rviz` has a number of *panels* and *plugins* which can be configured as needed for a given task. Configuring `rviz` can take some time and effort, so the *state* of the visualization can be saved to *configuration files* for later re-use. Additionally, when closing `rviz`, by default the program will save its configuration to a special local file, so that the next time `rviz` is run, it will instantiate and configure the same panels and plugins.

A default, unconfigured `rviz` will appear as shown in Figure 8-4.

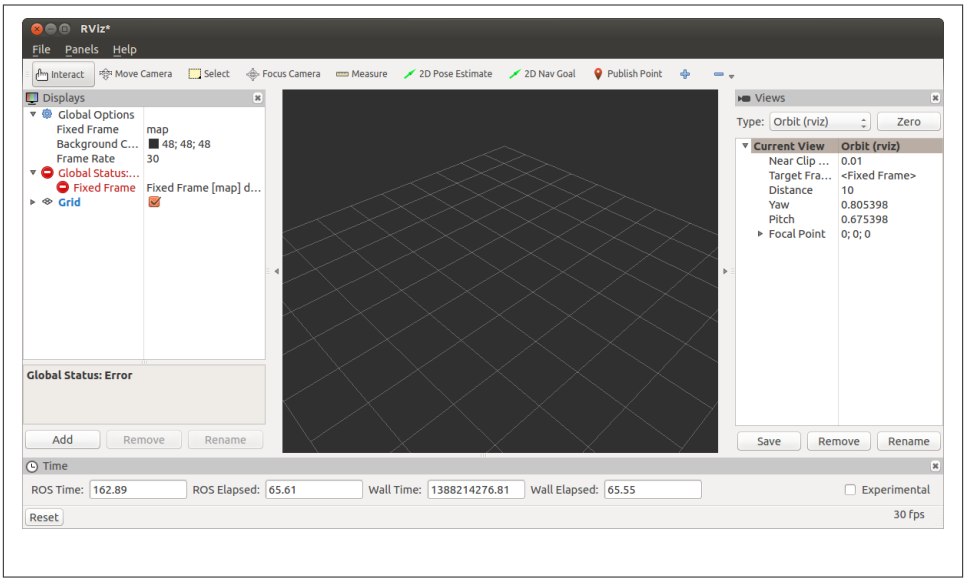


Figure 8-4. The initial state of rviz, before any visualization panels have been added to the configuration.

The first task is to choose the *frame of reference* for the visualization. In our case, we want a visualization perspective which is attached to the robot, so we can follow the robot as it drives around. On any given robot, there are many possible frames of reference, such as the center of the mobile base, a wheel (note that this frame would continually flip around and around, making it rather dizzying as a vantage point for `rviz`), various links of the robot's structure, and so on. For purposes of teleoperation, here we will select a frame of reference attached to the optical center of the Kinect depth camera on the Turtlebot. To do this, click in the table cell to the right of the "Fixed Frame" row in the upper-left panel of `rviz`. This will pop up the menu shown in the following screenshot, which contains all transform frames currently being broadcast in this ROS system. For now, select `camera_depth_frame` in the popup menu, as shown in Figure 8-5.

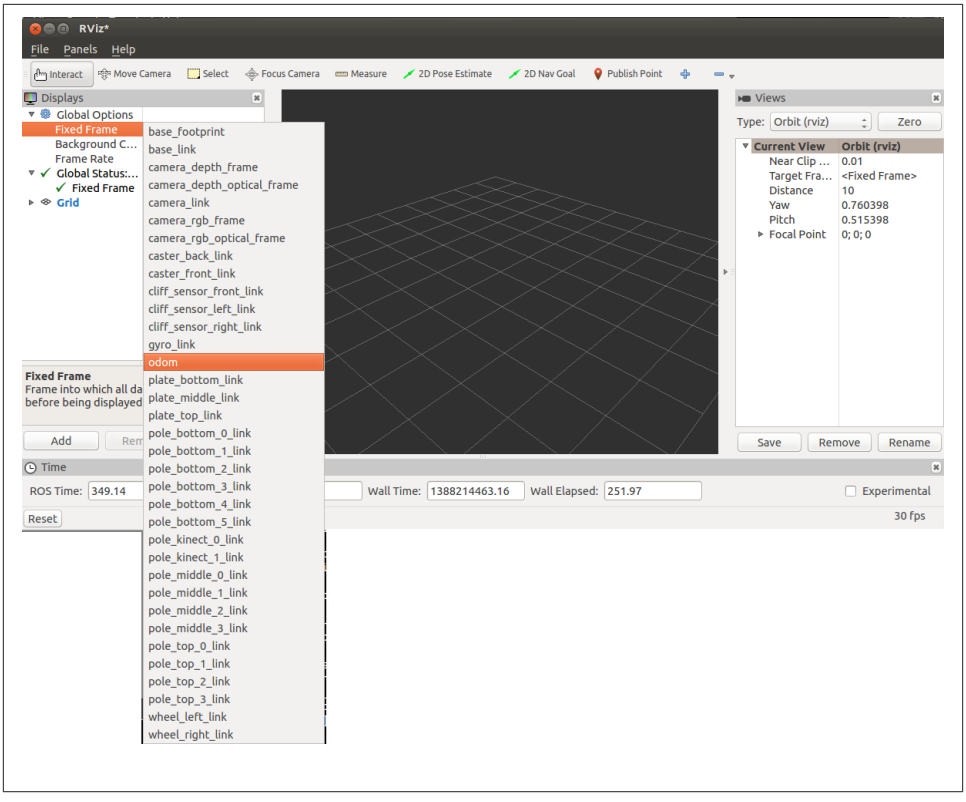


Figure 8-5. The fixed-frame popup menu. Selecting the fixed-frame for visualization is one of the most important configuration steps of rviz.

Next, we want to view the 3D model of the robot. We will insert an instance of the *robot model* plugin. Although Turtlebot has no moving parts (other than its wheels) that we need to visualize, it is still useful to see a rendering of the robot to improve situational awareness and a sense of scale for teleoperation. To add the robot model to the `rviz` scene, click the `Add` button on the left-hand side of the `rviz` window, approximately halfway down. This will bring up a dialog box, shown in Figure 8-6, that contains all of the available `rviz` plugins for various datatypes.

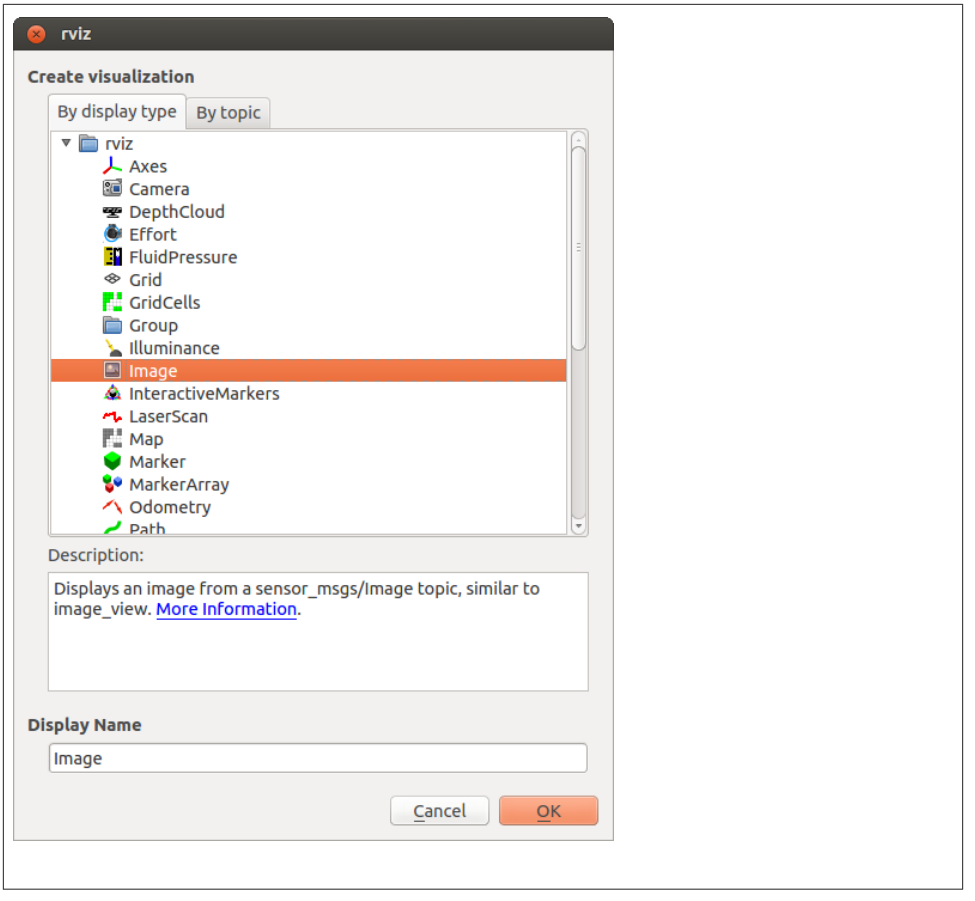


Figure 8-6. This rviz dialog box is used to select the datatype that is currently being added to the visualization.

From this dialog box, select `Robot Model` and then `OK`. Plugin instances appear in the tree-view control at left of the `rviz` window. To configure a plugin, ensure it is expanded in the tree view on the left pane. Its configurable parameters can then be edited. For the *robot model* plugin, the only configuration typically required is to enter the name of the robot model on the parameter server. However, since the ROS convention is for this to be called `robot_description`, this is auto-filled and typically "just works" for single-robot applications. This will produce an `rviz` visualization similar to that shown in Figure 8-7, which is centered on a model of the Turtlebot.

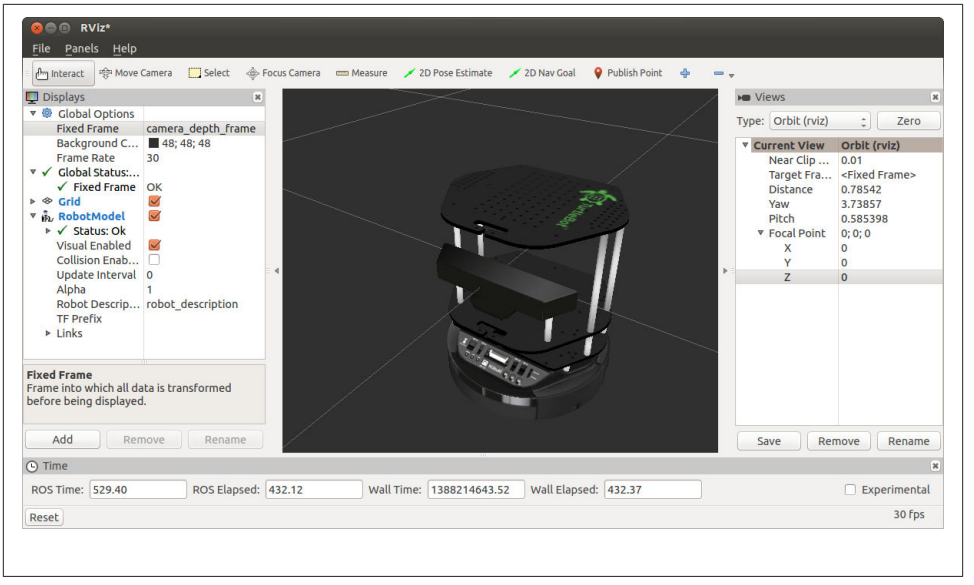


Figure 8-7. A Turtlebot model has been added to rviz.

In order to teleoperate the Turtlebot reasonably, we need to plot its sensors. To plot the depth image from the Kinect camera on the Turtlebot, click `Add` and then select `PointCloud2` from the plugin dialog box, near the lower-left corner of `rviz`. The `PointCloud2` plugin has quite a few options to configure in the tree view control in the left pane of `rviz`. Most importantly, we need to tell the plugin which topic should be plotted. Click the space to the right of the `Topic` label, and a drop-down box will appear, showing the `PointCloud2` topics currently visible on the system. Select `/camera/depth/points`, and the Turtlebot's point cloud should be visible, as shown in Figure 8-8.

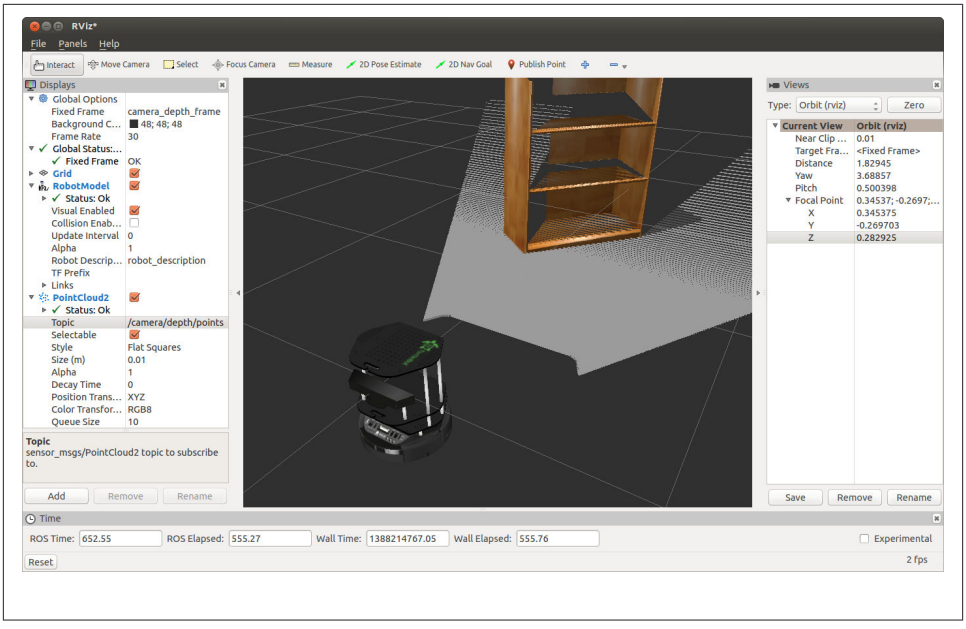


Figure 8-8. The Turtlebot's depth-camera data has been added to the visualization.

The Kinect camera on the Turtlebot also produces a color image output, in addition to its depth image. Sometimes it can be useful for teleoperation to render both the image and the point cloud. `rviz` provides a plugin for this. Click `Add` in the near the lower-left corner of `rviz`, and then select `Image` from the plugin dialog box. As usual, this will instantiate the plugin, and now we need to configure it. Click on the white space to the right of the `Image Topic` label of the image plugin property tree, and then select `/camera/rgb/image_raw`. The camera stream from the Turtlebot should then be plotted in the left pane of `rviz`, as shown in Figure 8-9.

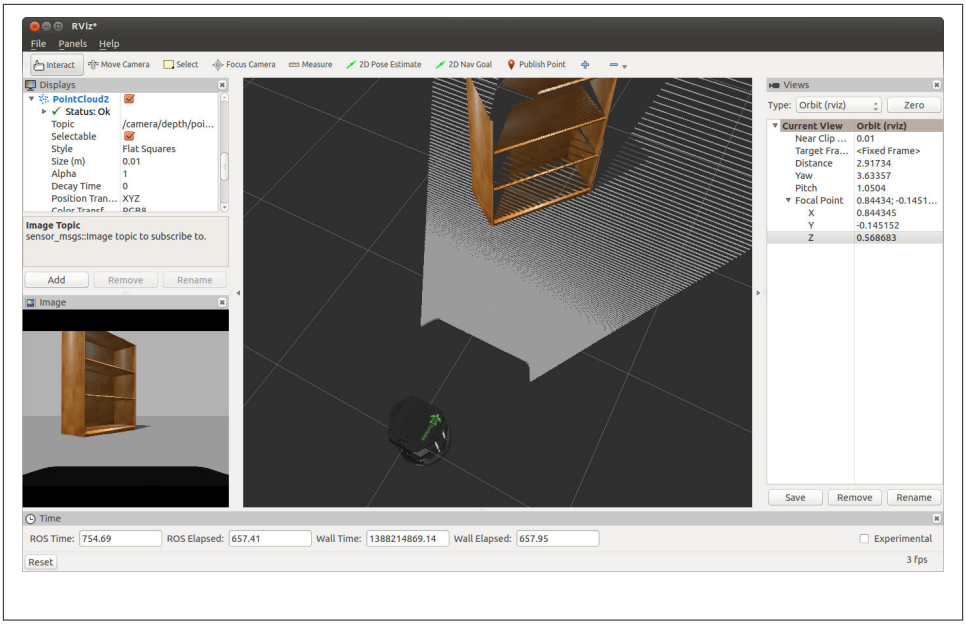


Figure 8-9. The camera image in the lower-left corner has been added to the visualization, allowing teleoperators to see the first-person perspective as well as the third-person perspective of the main window.

The `rviz` interface is panelized and thus can be easily modified to suit the needs of the application. For example, we can drag the image panel to the right-hand column of the `rviz` window, and resize it so that the depth image and camera image are similarly sized. We can then rotate the 3D visualization so that it is looking at the point-cloud data from the side, which could be useful in some situations. An example panel configuration is shown in Figure 8-10.

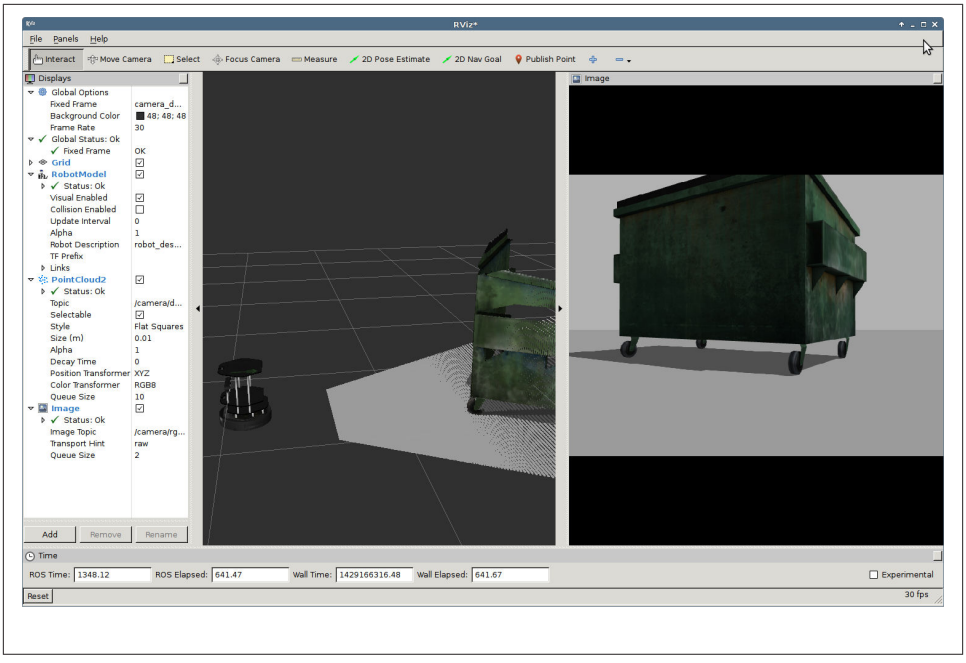


Figure 8-10. Rviz panels can be dragged around to create different arrangements. In this screenshot, the left panel has the third-person renderings of the depth camera data, and the visual camera is shown in the right panel.

Alternatively, we can rotate the 3D scene so that it has a top-down perspective, which can be useful for driving in tight quarters. An example of this "bird's-eye" perspective is shown in Figure 8-11.

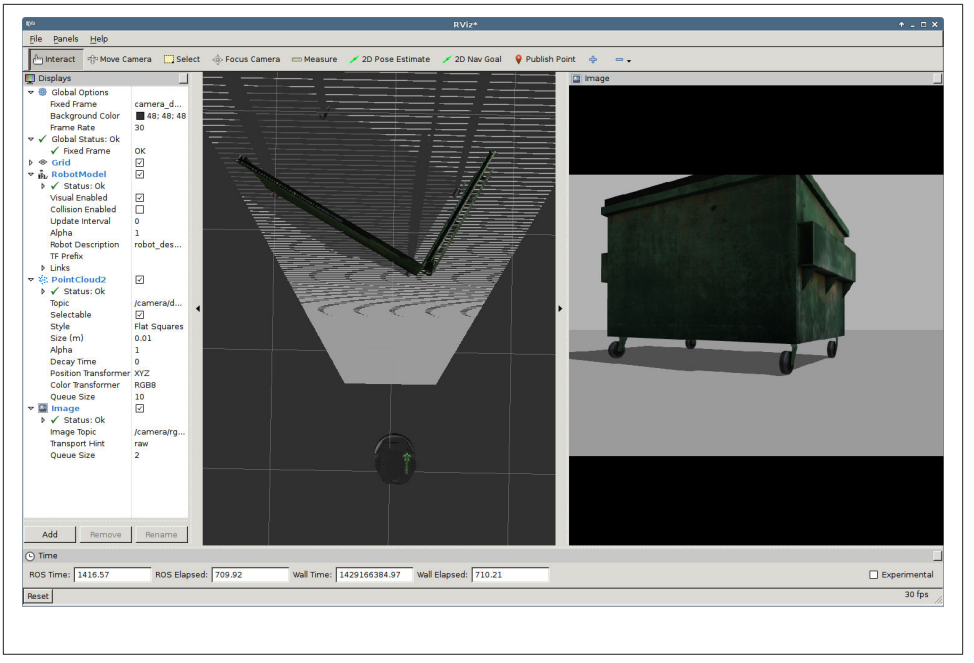


Figure 8-11. The perspective of the 3D view can be rotated and zoomed as desired. In this screenshot, a "bird's-eye" top-down perspective has been created. This perspective can be particularly useful in teleoperation.

These examples just scratch the surface of what rviz can do! It is an extremely flexible tool that we will use throughout the remainder of the book.

Summary

This chapter developed a progressively more-complex keyboard-based teleoperation scheme, and then showed how to connect the resulting motion commands to a Turtlebot. Finally, this chapter introduced `rviz` and showed how to quickly configure `rviz` to render point cloud and camera data, to create a teleoperation interface for a mobile robot.

CHAPTER 9

Building Maps of the World

Now that you know how ROS works, and have moved your robot around a bit, it's time start looking at how to get it to navigate around the world on its own. In order to do this, the robot needs to know where it is, and where you want it to go to. For most robots, this means that it needs to have a map of the world, and to know where it is in this map. In this chapter, we're going to see how to build a high-quality map of the world, using data from your robot's sensors. We'll then use these maps in the next chapter when we talk about how to make the robot move about in the world.

If your robot had perfect sensors and knew exactly how it was moving, then building a map would be simple: take the objects detected by your sensors, transform them into some global coordinate frame (using the robot's position and some geometry), and them record them in a map (in this global coordinate frame). Unfortunately, in the real world, it's not quite that easy. The robot doesn't know exactly how it's moving, since it's interacting with an uncertain world. No sensor is perfect, and you'll have to deal with noisy measurements. How can you combine all this error-laden information together to produce a usable map?

Luckily, ROS has a set of tools that will do this for you. The tools are based of some quite advanced mathematics but, luckily, you don't have to understand everything that's going on under the hood in order to use them. We'll describe these tools in this chapter but, first, let's talk a bit about exactly what we mean by "map".

Maps in ROS

Navigation maps in ROS are represented by a 2d grid, where each grid cell contains a value that corresponds to how likely it is to be occupied. Figure 9-1 shows an example of a map learned directly from the sensor data on a robot. White is open space, black is occupied, and the greyish color is unknown.

Figure 9-1. An example of a map used by ROS.

Map files are stored as images, with a variety of common formats being supported (such as PNG, JPG, and PGM). Although color images can be used, they are converted to greyscale images before being interpreted by ROS. This means that maps can be displayed with any image display program. Associated with each map is a YAML file that holds additional information, such as the resolution (the length of each grid cell in meters), where the origin of the map is, and thresholds for deciding if a cell is occupied or unoccupied. An example of a map YAML file is shown in ???.

```
image: map.pgm
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 1
```

This map is stored in the file `map.png`, has cells that represent 10cm squares of the world, and has an origin at (0, 0, 0). A cell is considered to be occupied if the value in it is more than 65% of the total range allowed by the image format. A cell is unoccupied if it has a value lss than 19.6% of the allowable range. This means that occupied cells will have a large value, and would appear lighter in color in the image. Unoccupied cells will have a lower value, and would appear darker. Since it is more intuitive for open space to be represented by white and occupied space by black, the `negate` flag allows for the values in the cells to be inverted before being used by ROS. So, for the example in ???, if we assume that each cell holds a single byte (an integer from 0 to 255), the values will first be inverted, by subtracting the original value from 255. Then, all cells with a value less that 49 (255 * 0.196 = 49.98) will be considered free, and all those with a value greater than 165 (255 * 0.65 = 165.75) will be considered to be occupied. All other cells will be classified as "unknown". These classifications will be used by ROS when we try to plan a path through this map for the robot to follow.

Since maps are represented as image files, you can edit them in your favorite image editor. This allows you to tidy up any maps that you create from sensor data, removing things that shouldn't be there, or adding in fake obstacles to influence path planning. A common use of this is to stop the robot from planning paths through certain areas of the map by, for example, drawing a line across a corridor you don't want to the robot to drive through, as you can see in Figure 9-2. Notice that the added lines don't have to be all that neat; as long as they completely block the corridor, the ROS navigation system (which we'll talk about in the next chapter) will not be able to plan a path through them. This allows you to control where there robot can and cannot go as it wanders around the world.

Before we start to talk about how we're going to build maps in ROS, we're going to take a short detour to talk about `rosbag`. This is a tool that allows you to record and replat published messages, and is especially useful when building large maps of the world.

Recording Data with `rosbag`

`rosbag` is a tool that lets us record messages and replay them later. This is really useful when debugging new algorithms, since it lets you present the same data to the algorithm over and over, which will help you isolate and fix bugs. It also allows you to develop algorithms without having to use a robot all the time. You can record some sensor data from the robot with `rosbag`, then use these recorded data to work on your code. `rosbag` can do more then record and play back data, but that's whare we're going to focus on for now.

To record messages, we use the `record` functionality and a list of topic names. For example, to record all the messages sent over the `scan` and `tf` topics, you would run

```
user@hostname$ rosbag record scan tf
```

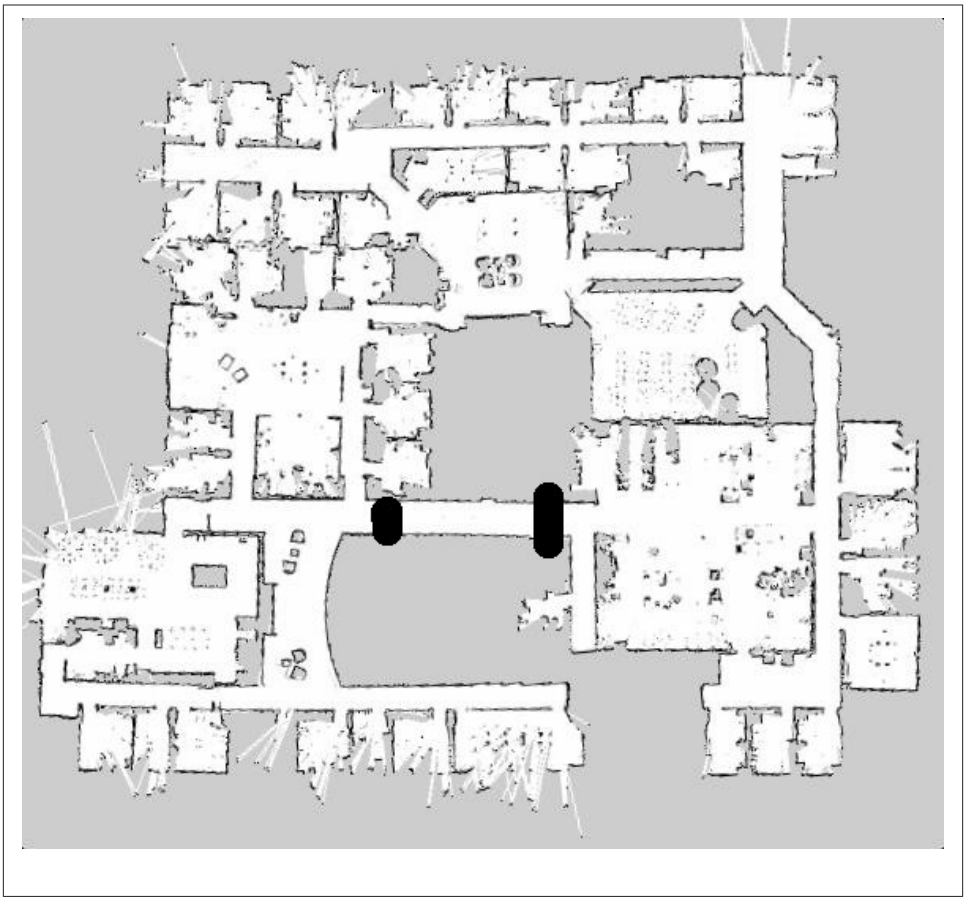


Figure 9-2. A hand-modified map. The black lines were added to stop the robot from planning paths down the corridor in the middle of the map.

This will save all of the messages in a file with the format `YYYY-MM-DD-HH-mm-ss.bag`, corresponding to the time that `rosbag` was run. This should give each bag file a unique name, assuming you don't run rosbag more than once a second. You can change the name of the output file using the `-O` or `--output-name` flags, and add a prefix with the `-o` and `--output-prefix` flags. For example

```
user@hostname$ rosbag record -O foo.bag scan tf
user@hostname$ rosbag record -o foo scan tf
```

would create bags named `foo.bag` and `foo_2015-03-05-14-29-30.bag`, respectively (obviously, with appropriate values for the current date and time). We can also record *all* of the topics that are currently publishing with the `-a` flag

```
user@hostname$ rosbag record -a
```

While this is often useful, it can also record **a lot** of data, especially on robots with a lot of sensors, like the PR2. There are also flags that let you record topics that match a regular expression, which are described in detail on the `rosbag` wiki page. `rosbag` will record data until you stop it with a `ctrl-c`.

You can play back a pre-recorded bag file with the `play` functionality. There are a number of command-line parameters that allow you to manipulat how fast you play back the bag, where you start in the file, and other things (all of which are documented on the wiki), but the basic usage is straightforward.

```
user@hostname$ rosbag play --clock foo.bag
```

Will replay the messages recorded in the bag file `foo.bag`, as if they were being generated live from ROS nodes. Giving more than one bag file name will result in the bag files being played sequentially. The `--clock` flag causes `rosbag` to publish the clock time, which will be important when we come to build our maps.

You can find out information about a bag file with the `info` functionality.

```
user@hostname$ rosbag info laser.bag
path:        laser.bag
version:     2.0
duration:    1:44s (104s)
start:       Jul 07 2011 10:04:13.44 (1310058253.44)
end:         Jul 07 2011 10:05:58.04 (1310058358.04)
size:        8.2 MB
messages:    2004
compression: none [11/11 chunks]
types:       sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]
topics:      base_scan   2004 msgs    : sensor_msgs/LaserScan
```

This give you information about how much time the bag covers, when it started and stopped recording, how large it is, how many messages it has in it, and what those messages (and topics) are.

Building Maps

Now we're going to look at how you can build a map like the one shown in Figure 9-1 using the tools in ROS. One thing to note about the map in Figure 9-1 is that it's quite "messy". Since it was created from sensor data taken from a robot, it include some things you might not expect. Along the bottom edge of the map, the wall seems to have holes in it. These are caused by bad sensor readings, possibly the result of clutter under the desks in those rooms. The strange blob in the largish room towards to top in the middle is a pool table. The grey spots in the larger room diagonally down and right as chair legs (this was a conference room). The walls are not always perfectly straight, and there are sometimes "unknown" areas in the middle of rooms if the sensors never made meas-

urements there. When you start to make maps of your own with your robot, you should be prepared for them to look like this. Generally speaking, using more data to create the map will result in a better map. However, no map will be perfect. Even though the maps might not look all that great to you, they're still perfectly useful to the robot, as we will see below.

You can build maps with the `slam_gmapping` node from the `gmapping` package. This node uses an implementation of the `GMapping` algorithm, written by Giorgio Grisstti, Cyrill Stachniss, and Wolfram Burgard. `Gmapping` uses a Rao-Blackwellized particle filter to keep track of the likely positions of the robot, based on it's sensor data and the parts of the map that have already been built. If you're interested in the details of the algorithm they're described in these two papers:

- "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters", Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007.
- "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling", Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pages 2432-2437, 2005.

First, we're going to generate some data to build the map from. Although you can build a map using live sensor data, as the robot moves about the world, we're going to take another approach. We're going to drive the robot around, and save the sensor data to a file using called `rosbag`. We're then going to replay these sensor data, and use `slam_gmapping` to build a map for us. First, let's record some data. First, start up a simulator with a Turtlebot in it.

```
user@hostname$ roslaunch turtlebot_stage turtlebot_in_stage.launch
```

This launch file starts up the `Stage` robot simulator and an instance of `rviz`. Zoom out a bit in the simulator (using the mouse wheel), and you should see something like Figure 9-3.

Now, start up the the `keyboard_teleop` node from `turtlebot_teleop` package.

```
user@hostname$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

This will let you drive the robot around in the simulated world using the keys shown by the node when it started:

```
Control Your Turtlebot!
//--------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .
```

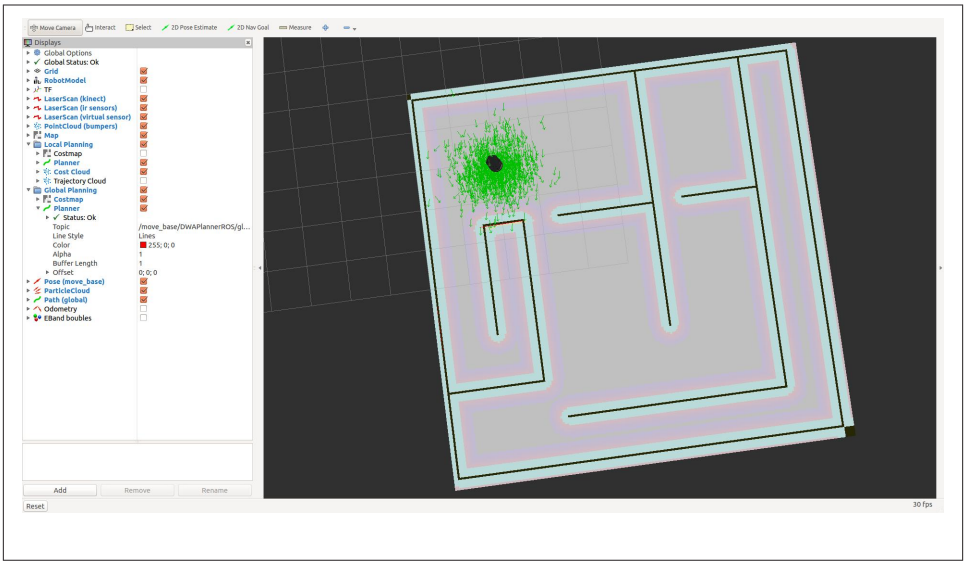


Figure 9-3. The `rviz` visualizer showing a simple world with a Turtlebot in it.

```
q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
space key, k : force stop
anything else : stop smoothly

CTRL-C to quit

currently:      speed 0.2       turn 1
```

Practice driving the robot around for a bit. Once you got the hang of it, we can get started collecting some data. `slam_gmapping` builds maps from data from the laser range-finder and the odometry system, as reported by `tf`. Although the Turtlebot doesn't actually have a laser range-finder, it creates `LaserScan` messages from it's Kinect data, and sends them over the `scan` topic. With the simulator still running, in a new terminal window, start recording some data.

```
user@hostname$ rosbag record -O data.bag /scan /tf
```

Now, drive the robot around the world for a while. Try to cover as much of the map as possible, and make sure you visit the same locations a couple of times. Doing this will result in a better final map. If you get to the end of this section and your map doesn't look very good, try recording some new data and drive the robot around the simulated world for longer, or a bit more slowly.

Once you've driven around for a while, use ctrl-c to stop `rosbag`. Verify that you have a data bag called `data.bag`. You can find out what's in this bag by using the `rosbag info` command.

```
user@hostname$ rosbag info data.bag
path:        data.bag
version:     2.0
duration:    3:15s (195s)
start:       Dec 31 1969 16:00:23.80 (23.80)
end:         Dec 31 1969 16:03:39.60 (219.60)
size:        14.4 MB
messages:    11749
compression: none [19/19 chunks]
types:       sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]
             tf2_msgs/TFMessage    [94810edda583a504dfda3829e70d7eec]
topics:      /scan   1959 msgs    : sensor_msgs/LaserScan
             /tf     9790 msgs    : tf2_msgs/TFMessage    (3 connections)
```

Once you have a bag that seems to have enough data in it, stop the simulator with a ctrl-c in the terminal you ran `roslaunch` in. It's important to stop the simulator before starting the mapping process, because it will be publishing `LaserScan` messages that will conflict with those that are being replayed by `rosbag`. Now it's time to build a map. Start `roscore` in one of the terminals. In another terminal, we're going to tell ROS to use the timestamps recorded in the bag file, and start the `slam_gmapping` node.

```
user@hostname$ rosparam set use_sim_time true
user@hostname$ rosrun gmapping slam_gmapping
```

If your robot's laser scan topic is not called `scan`, you will need to tell `slam_gmapping` what it is by adding `scan:=laser_scan_topic` when you start the node. The mapper should now be running, waiting to see some data.

We're going to use `rosbag play` to replay the data that we recorded from the simulated robot.

```
user@hostname$ rosbag play --clock data.bag
```

When it starts receiving data, `slam_gmapping` should start printing out diagnostic information. Sit back and wait until `rosbag` finishes replaying the data, and `slam_gmapping` has stopped printing diagnostics. At this point, your map has been build, but it hasn't been saved to disk. Tell `slam_gmapping` to do this by using the `map_saver` node from the `map_server` package. Without stopping `slam_gmapping`, run the `map_saver` node in another terminal.

```
user@hostname$ rosrun map_server map_saver
```

This will save your map to disk in the file `map.pgm`, and a YAML file describing it in `map.yaml`. Take a look, and make sure you can see these files. You can view the map file using any standard image viewer, such as `eog`.

```
user@hostname$ eog map.pgm
```

The map shown in Figure 9-4 was generated by slowly rotating the turtlebot in place for a little more than one revolution, without moving from it's starting position in the simulator. The first thing to notice about this map is that the actual mapped part of the world is tiny compared to the rest of the map. This is because the default size of ROS maps are 200m by 200m, with a cell size of 5cm (that means that the image size is 2,000 by 2,000 pixels). Figure 9-5 shows a zoomed-in version of the interesting part of the map. This map isn't very good: the walls are not at right angles to each other, the open space extends beyond one of the walls, and there are notable gaps in several of the walls. As Figure 9-5 shows, getting a good map is not just a simple matter or running `slam_gmapping` on any old set of data. Building a good map is hard and can be time-consuming, but it's worth the investment, since a good map will make navigating around the world and knowing where you are a lot easier, as we'll see in the next chapter.

The YAML file describing the map shown in Figure 9-4. shows the YAML file generated by `slam_gmapping`.

The YAML file describing the map shown in Figure 9-4.

```
image: map.pgm
resolution: 0.050000
origin: [-100.000000, -100.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Why is the map *so* bad? One of the reasons is that the sensors on the Turtlebot are not great for creating maps. `slam_gmapping` expects `LaserScan` messages, and the Turtlebot doesn't have a laser range-finder. Instead, it uses a Microsoft Kinect sensor, and uses the data from this to synthesize a `LaserScan` message, which can be used by `slam_gmapping`. The problem is that this fake laser range-finder data has a shorter range and a narrower field of view than a typical laser sensor does. `slam_gmapping` uses the laser data to estimate how the robot is moving, and this estimation is better with long-range data over a wide field of view.

We can improve mapping quality by setting some of the `gmapping` parameters to different values.

```
user@hostname$ <userinput>/slam_gmapping/angularUpdate 0.1</userinput>
user@hostname$ <userinput>/slam_gmapping/linearUpdate 0.1</userinput>
user@hostname$ <userinput>/slam_gmapping/lskip 10</userinput>
user@hostname$ <userinput>/slam_gmapping/xmax 10</userinput>
user@hostname$ <userinput>/slam_gmapping/xmin -10</userinput>
user@hostname$ <userinput>/slam_gmapping/ymax 10</userinput>
user@hostname$ <userinput>/slam_gmapping/ymin -10</userinput>
```

These change how far the robot has to rotate (`angularUpdate`) and move (`linearUpdate`) before a new scan is considered for inclusion in the map, how many beams to

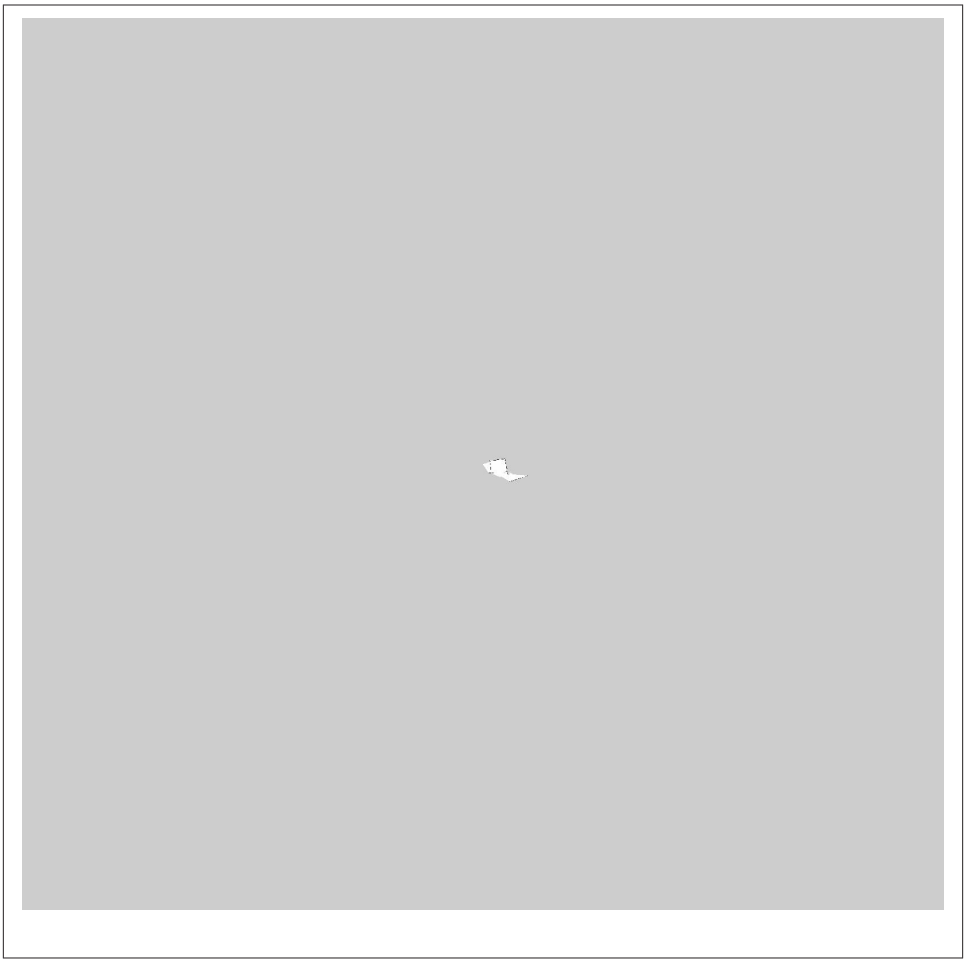


Figure 9-4. A map generated from a Turtlebot spinning in place

skip when processing each `LaserScan` message (`lskip`), and the extent of the map(`xmin`, `xmax`, `ymin`, `ymax`)

We can also improve the quality of the maps by driving around slowly, especially when turning the robot. Make the parameter changes above, collect a new bag of data by driving your robot around slowly, and you get a map that looks more like Figure 9-6. It's still not perfect (no map built from sensor data ever is), but it's certainly better than the previous one.

You can also build your maps directly from published messages, without saving them to a bag file first. To do this, you just need to start the `slam_gmapping` node while you're

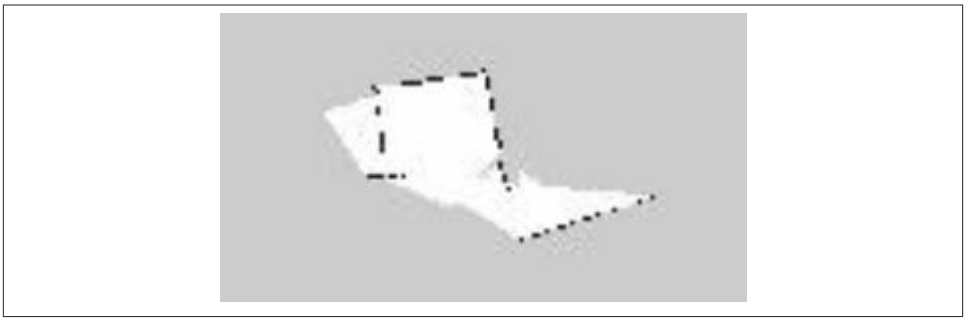


Figure 9-5. A map generated from a Turtlebot spinning in place

driving your robot around. We prefer to record the data first, since there's less computational load on the robot when you're driving it around that way. In the end, however, you should end up with similar maps, regardless of whether or not you saved the data with `rosbag` first or not.

Starting a Map Server and Looking at a Map

Once we have a map, you need to make it available to ROS. We do this by running the `map_server` node from the `map_server` package, and pointing it at the YAML file for a map we have already made. This YAML file has the file name for the image that represents the map and additional information about it, like the resolution (meters per pixel), where the origin is, what the thresholds for occupied and open space are, and whether the image uses white for open space or occupied space (we talked about this in more detail in the mapping chapter). With `roscore` running, you can start a map server like this:

```
user@hostname$ rosrun map_server map_server willow.yaml
```

where `map.yaml` is the map YAML file. Running the map server will result in two topics being published. `map` contains messages of type `nav_msgs/OccupancyGrid`, corresponding to the map itself. `map_metadata` contains messages of type `nav_msgs/MapMetaData`, corresponding to the data in the yaml file.

```
user@hostname$ rostopic list
/map
/map_metadata
/rosout
/rosout_agg
user@hostname$ rostopic echo map_metadata
map_load_time:
  secs: 1427308667
  nsecs: 991178307
resolution: 0.0250000003725
```

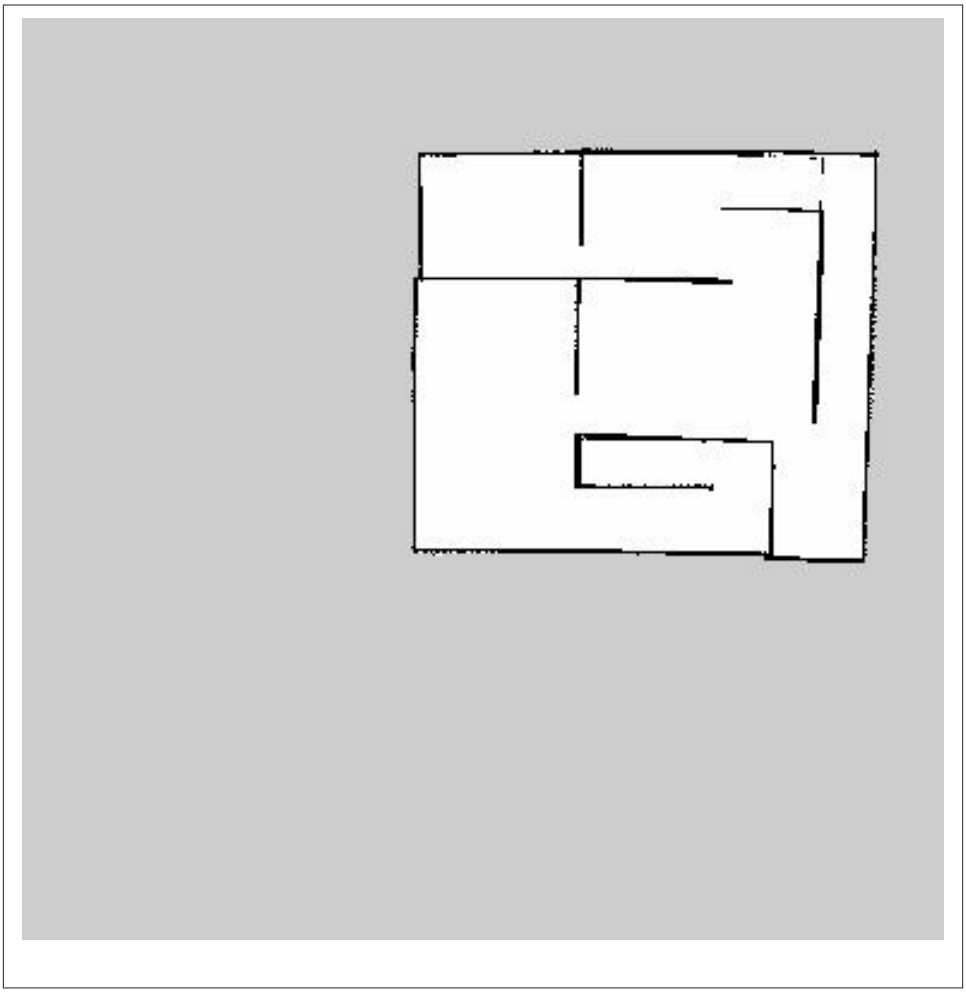


Figure 9-6. A better map, built from more carefully-collected data.

```
width: 2265
height: 2435
origin:
  position:
    x: 0.0
    y: 0.0
    z: 0.0
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
```

This map has 2265 by 2435 cells in it, with a resolution of 2.5cm per cell. The origin of the world coordinate frame is a the origin of the map, with the same orientation.

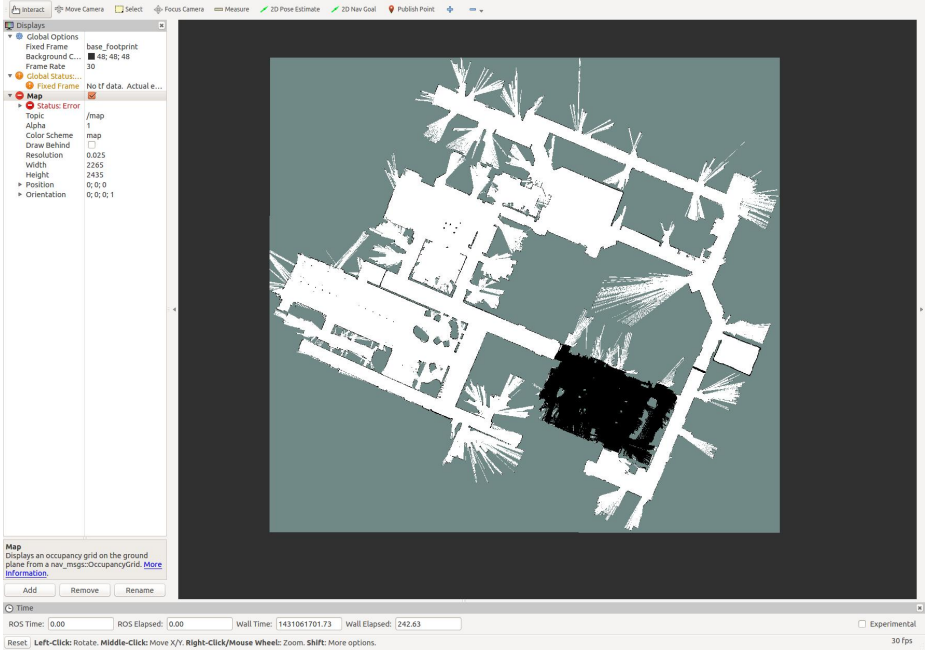
Now, let's have a look at a map in `rviz`. Start up a map server like this

```
user@hostname$ roscd mapping/maps
user@hostname$ rosrun map_server map_server willow.yaml
```

Now, in another terminal, start up an instance of `rviz`

```
user@hostname$ rosrun rviz rviz
```

Add a display of type `Map`, and set the topic name to `/map`. Make sure that the fixed frame is also set to `/map`. You should see something like ???.



This map was built using a PR2 robot with a laser range-finder and `slam_gmapping`, and illustrates a number of things you often see in maps built from sensor data. First, it is not axis-aligned. When the robot was collecting data to build the map, the odometry data coordinate frame was aligned with the starting position of the robot, which means that the final map is rotated a bit. We can fix this in the YAML file if we want to, although it doesn't affect the robot's ability to navigate.

Second, the map is quite "messy". Although the corridors and open spaces are quite clean, there seem to be a lot of long, skinny open spaces coming off these open spaces. These are actually rooms that the robot did not drive into. As the robot drove past these

rooms, the laser range-finder made some measurements into the room, but there weren't enough data to reconstruct a decent map of the room. Again, this won't affect the ability of the robot to localize itself, but it does mean that we might not be able to get the robot to navigate into these rooms autonomously, since they're technically not in the map.

Finally, there's a big black blob in the lower right of the map. This is a room that the robot should not go into, even though it's on the map. After the map was made, someone loaded the image file into a graphics program, like `gimp`, and painted the pixels in the room black. The means that, when the robot tries to plan a path in the map, these areas will be considered to be occupied, and it will not plan a path through them. It will affect the robot's ability to localize itself a bit, especially when it is near the doorway to this space. Localization involves comparing the current sensor readings to the map, to make sure the robot is seeing what it expects to in a given location. Since there's an obstacle in the map (the big black blob) that doesn't match up with something in the real world, the robot's confidence in where it is will be lower. However, as long as it can see enough of the world that *does* match up with its map (which is can do, since the laser range-finder on the PR2 has a wide field-of-view), the localization algorithm is robust enough to cope.

Summary

In this chapter, we looked at how to use the `slam_gmapping` package to learn a high-quality map of the robot's environment. We also introduced you to `rosbag`, which can let you save published messages to a file, and replay them later. We'll be seeing `rosbag` again later on in this book, since it's a useful tool.

One of the important things to remember about building maps is that, although many roboticists consider it to be a "solved problem", it is often tricky to do in practice, especially with cheaper robots and less capable sensors. It often take a

We're really just scratched the surface of the ROS mapping system. There are a huge number of parameters you can set to alter the mapping behavior. These are all documented at http://wiki.ros.org/gmapping[the `gmapping` wiki page) and described in the papers mentioned above. However, unless you know that the effects of changing these parameters are, we'd recommend that you don't fiddle with them too much. Find some settings that work for your robot, and then don't change them.

Once you've build maps a few times, and you have a feel for it, it shouldn't take too long to make a new one when you find yourself in a new environment. Once you have a map, then you're ready to have your robot start to autonomously navigate about, which is the subject of the next chapter.