THE ROBOT OPERATING SYSTEM IN TRANSITION:

EXPERIMENTS AND TUTORIALS

by

JAMES STARKMAN

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May 2018

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

James Starkman

candidate for the degree of **Master of Science***.

Committee Chair

Dr. Wyatt Newman

Committee Member

Dr. M. Cenk Çavuşoğlu

Committee Member

Dr. Greg Lee

Date of Defense

2018-01-18

*We also certify that written approval has been obtained

for any proprietary material contained therein.

Contents

List of Tables **vi**

List of Figures **vii**

Acknowledgments **viii**

Abstract **ix**

Introduction **1**

I Experiments **4**

1 Tiny targets **5**

1.1 Introduction 5

1.1.1 Explanation of tiny target comparison table 7

1.2 Experiments 8

1.2.1 `jasadc` 8

1.2.1.1 Link to source 9

1.2.1.2 Link to pre-built ROS1 and ROS2 workspaces 10

1.2.2 Instrumentation 10

1.2.2.1 Explanation of sinusoid plots 12

1.2.3 Methods 12

1.2.3.1 ROS1 . . . 12
1.2.3.2 ROS2 . . . 14
1.2.4 Observations . . . 14
1.2.4.1 Dropouts . . . 14
1.2.4.2 Publication rate . . . 15
1.3 Analysis . . . 16
1.3.1 Publication rate . . . 16
1.3.1.1 Explanation of histograms . . . 18
1.3.2 Reconstructing input from samples . . . 18
1.3.2.1 Explanation of frequency recovery table . . . 21
1.4 Conclusion . . . 21
1.4.1 Once-off aspects: cost, setup, and installation . . . 22
1.4.2 Performance . . . 22
1.4.3 Overall . . . 23
1.4.4 Future work . . . 23

2 Windows **24**

2.1 Introduction . . . 24
2.1.1 Cygwin . . . 25
2.1.2 WSL . . . 25
2.2 Experiments . . . 26
2.2.1 Setup . . . 26
2.2.1.1 General . . . 26
2.2.1.2 ROS1 running via WSL . . . 26
2.2.1.3 ROS2 running natively . . . 27
2.2.2 Methods . . . 27
2.2.3 Observations . . . 28
2.2.3.1 ROS1 running via WSL . . . 28

2.2.3.2 ROS2 running natively 30

3 Miscellaneous **32**

3.1 Introduction 32

3.2 Programming languages and libraries 32

3.2.1 Platform differences 32

3.2.1.1 ROS1 32

3.2.1.2 ROS2 34

3.2.2 Language differences 35

3.2.2.1 Python 35

3.2.2.2 C++ 36

3.3 Security and encryption 39

3.3.1 Introduction 39

3.3.2 Enabling security (involves partial rebuilding) 40

3.3.2.1 How to enable security 40

3.3.2.2 Authentication and key management 40

3.4 Conclusion 42

II Tutorials **44**

4 How to port an existing ROS1 C++ program to ROS2 **45**

4.1 Introduction 45

4.2 Ancillary porting information 46

4.2.1 Introduction 46

4.2.2 Dependencies 46

4.2.2.1 Transitive limitations 47

4.2.3 Installing ROS2 itself 47

4.2.4 Preparing for the port 49

4.2.4.1 Removing deprecated ROS1 features 49
4.2.4.2 Setup for porting 51
4.2.5 List of useful statements to run on a command line 52
4.2.6 Toolchain 53
4.3 Porting your package 55
4.3.1 Introduction 55
4.3.2 Architectural changes 55
4.3.2.1 Launchers and parameters 55
4.3.2.2 Pointers 56
4.3.2.3 CMakeLists.txt 56
4.3.3 Programming changes 57
4.3.3.1 Message and service files 57
4.3.3.2 C++ files 58
4.3.4 Other changes 61
4.3.4.1 Documentation 61
4.4 Packages ported from `learning_ros2` 61

5 Miscellaneous addenda **64**
5.1 Introduction 64
5.2 How to debug ROS2 programs with Visual Studio 64
5.2.1 Introduction 64
5.2.2 Step-by-step instructions 65
5.2.3 Explanation of Visual Studio debugging screenshot 67
5.3 How to run ROS1 or ROS2 on C.H.I.P. 69

Concluding thoughts for tutorials **71**

Conclusion **74**

III Appendices **76**

A Abbreviations **77**

A.1 Table of abbreviations 77

B How to run ROS1 and ROS2 on C.H.I.P. **83**

B.1 ROS1 84

B.2 ROS2 84

B.2.1 Debian Jessie (8) versus Debian Stretch (9) 84

B.2.2 Dependencies 85

B.2.3 Cross-compiling 85

C Patch for `jasadc` from ROS1 to ROS2 **87**

References **98**

List of Tables

1.1 A comparison of tiny targets. See subsection 1.1.1 for details. 8

1.2 Frequency recovery estimates and errors. See subsubsection 1.3.2.1 for details. 21

List of Figures

1.1 Plots of 1 Hz and 10 Hz sinusoids captured with ROS1 running on C.H.I.P.. See subsubsection 1.2.2.1 for details. 13
1.2 Histogram, short abscissa. See subsubsection 1.3.1.1 for explanation. . 19
1.3 Histogram, long abscissa. See subsubsection 1.3.1.1 for explanation. . 20

2.1 The standard ROS1 introductory programs running unmodified on Windows via WSL. 29
2.2 A minimal subscriber in ROS2. See the miscellaneous addenda in the tutorials for an explanation of how to debug a ROS2 program in Visual Studio. 31

3.1 Comparison of captured packets displayed in Wireshark. Above: unsecured communication, clearly visible in plain text at the end of the packet. Below: same information, but encrypted. 43

5.1 Debugging the minimal subscriber. See subsection 5.2.3 for an explanation of what the red annotations mean. 68

Acknowledgments

The author would like to thank all of the people who have contributed to ROS over the years for the ecosystem that they have collectively created. The author would also like to thank his advisor, Professor Newman, for his suggestions during the planning, development, and writing of this thesis, as well as all of the other faculty members whose classes provided relevant background.

The Robot Operating System in Transition: Experiments and Tutorials

Abstract

by

JAMES STARKMAN

ROS, the Robot Operating System, was first made available in 2007. Since then, usage has grown considerably, along with the number of potential applications and use cases. Unfortunately, design decisions made when ROS was in its infancy still apply today and have begun to show their age. These decisions include: only being supported on the Ubuntu Linux distribution, assuming the use of powerful workstations, adhering to older versions of libraries and programming languages, and the lack of encrypted communications. Rather than addressing the limitations of ROS within the confines of the existing development framework, the Open Source Robotics Foundation decided in 2014 to develop an entirely new project, ROS2, which is backwards-incompatible with ROS1. This new project aims to address all of the above limitations and more. This thesis explores the trade-offs between ROS1 and ROS2.

Introduction

ROS, the Robot Operating System [1], was first made available in 2007 [2]. It was developed by a company called Willow Garage which was founded by an early developer at Google. They found that anyone working in research robotics "spent 90 percent of their time re-writing code others had written before and building a prototype test-bed", leaving only the remaining ten percent for innovation [3]. ROS was their solution to eliminate that wasted time. Three years later, in 2010, the first major version of ROS, Box Turtle, was released. Since then, usage and popularity have grown considerably[1], along with the number of potential applications and use cases. Unfortunately, design decisions made when ROS was in its infancy still apply today and have begun to show their age. These decisions include: only being officially supported on the Ubuntu Linux distribution (as of Lunar, the 2017 long-term support (LTS) release), assuming the use of powerful workstations, adhering to older versions of libraries and programming languages, and the lack of encrypted communications. Rather than addressing the limitations of ROS within the confines of the existing development framework, the Open Source Robotics Foundation (OSRF; a 2012 spinoff from Willow Garage that has since assumed the mantle of ROS maintenance) decided in 2014 to develop an entirely new project, ROS2, which is backwards-incompatible with the original project (which will hereafter be referred to as "ROS1"). For the purposes of this thesis, the two projects will collectively be referred to as "ROSn".

[1] As of this writing, [1] has 3866 citations on Google Scholar.

ROS2 aims to address all of the above limitations and more. The first alpha release of ROS2 was in August of 2015 and the first full, supported release was in December of 2017. Design goals of ROS2 include support for small embedded platforms, non-ideal networks, and a general shift from being focused on research applications to production enterprise environments. Furthermore, ROS1 included a significant amount of "middleware", software that implements the underlying data transfers, while ROS2 seeks to use a variety of new technologies that have been developed since the release of ROS1 [4]. Still, ROS2 is in its initial experimental stages, so the current state of affairs that is explored in this thesis will not necessarily be indicative of the future.

This thesis is not the first academic publication to explore ROS2. While this thesis is largely based on publically-available information from the OSRF employees who work on ROS2, it is in part based on prior academic works. For example, in 2016 a paper was published that explored the performance of ROS2 [5]. This paper found that, as of the third alpha of ROS2 (as declared in Table 2), ROS1 was substantially better at transporting a wide variety of data sizes than ROS2 with either the Connext or OpenSplice middlewares (as reported in Table 4). As such, this thesis only uses Fast-RTPS, the free and open-source middleware that is included with a default ROS2 installation (as of the beta and full releases). The paper also makes use of the `ros_bridge` (later renamed to `ros1_bridge`) program to connect the ROS1 topic space with its ROS2 equivalent. This program is also used in this thesis, albeit for capturing and analyzing network traffic in order to compute statistics about ROS2's performance on tiny targets.

Another paper implemented a proof-of-concept network module for ROS2 based on the specific type of network that they were studying [6]. They ran ROS2 on "low-end IoT communications modules", specifically the Atmel SAMR21 board, and did not mention any difficulties in their paper.

Another way to run ROS2 in an embedded system is to use NuttX, a real-time

operating system (RTOS) that can run on a variety of hardware, including microcontrollers with a word size of only eight bits. A repository prototyping ROS2 for NuttX is available under the ROS2 GitHub organization. Its last update was in 2014 [7].

This thesis is structured as follows. The first part empirically explores various aspects of the new ROS2 project, including tiny targets, Windows support, newer versions of programming languages and libraries, and security and encryption. The second part contains empirically-backed tutorials about developing various ROS2-related topics, including a detailed guide on how to port an existing ROS1 package to ROS2, how to use Visual Studio to debug a ROS2 program running on Windows, and how to run either ROS1 or ROS2 on the NTC C.H.I.P., a US$9 credit-card-sized computer that is similar to the perhaps more well-known Raspberry Pi. The next part concludes the thesis, and the final part contains the appendices and references. A particularly useful appendix is the glossary-like Abbreviations appendix, which contains within it a table of *every* abbreviation used in this thesis.

Part I

Experiments

Chapter 1

Tiny targets

1.1 Introduction

One of the claims of ROS2 is that it works better on smaller target computers than the "workstation-class" machines that ROS1 requires [8]. While the definitions of "tiny target" may vary, some traits remain constant; important considerations include size, price, and processing power. On robots running ROS1, there is generally one central computer running `roscore` and its attendant nodes[1]. This machine is sometimes referred to as the "brain" of the robot, and is generally constructed from laptop or desktop computer parts with prices and processing power availabilities to match. Even if the computer is only a small netbook, as in the case of Turtlebots [9], it is still orders of magnitude more expensive and powerful than some tiny targets are. For instance, the target that was explored in [8] costs about US$10, uses an ARM Cortex-M7, and can still communicate over Ethernet. At a similar price point, the C.H.I.P. computer [10] costs US$9 and uses an Allwinner R8 system-on-a-chip (SoC) [11]; this provides significantly more processing power and memory than the machine from [8]. Indeed, C.H.I.P. can even boot Debian Linux, albeit with modifications.

[1] In both ROS*n*, a *node* is the basic abstraction of a process. One can also write "nodelets", which can be run in the same process and thus can use more efficient techniques to move data (such as passing a pointer to the data instead of copying everything).

(For example, using UBIFS[2] instead of ext4[3] since C.H.I.P. is backed by NAND flash storage instead of a physical disk. Note that solid-state drives (SSDs) use different kinds of flash than C.H.I.P. and can handle disk-oriented file systems.) To see how the specifications of C.H.I.P. and other kinds of targets compare, see table 1.1. The rest of this chapter will explore C.H.I.P. in particular.

Why does ROS2 claim to work better on smaller targets? In part, this is due to its usage of DDS [13], which does not require the same support systems as ROS1. In particular, DDS does not necessarily need the full TCP/IP stack (only UDP), nor does it necessarily need to use dynamic memory allocation; that is, all of the memory that it needs can be statically allocated. Since memory allocation is often non-deterministic (due to the other processes on the system consuming unknown amounts and regions of memory), this can help a real-time system better meet its desired sample rate, as well as potentially speeding up normal operations and reducing the risk of a segfault. In other words, by no longer depending on these systems, one can target a broader array of operating systems and RTOSes[4], and even contemplate running on bare metal — a concept that might seem rather far-fetched under ROS1. This is the idealized future of ROS2: since all sensors, computers, and actuators would speak the same protocol, more processing would be able to happen closer to the sensor level, thus reducing the load on the central computer and providing for a more distributed system [8].

The default DDS used in ROS2 is Fast-RTPS[5], although others can be used as

[2]The *Unsorted Block Image File System* is intended for use with flash storage and has to: track "bad blocks" (regions of flash that have degraded beyond the point of usefulness), offer wear levelling (flash is not infinitely re-writeable), and work nicely with the Linux kernel (merged since version 2.6.27, in the year 2008 [12]).

[3]The *fourth extended file system* (for Linux). This is the default file system for numerous Linux distributions, including Debian and Debian-derivatives like Ubuntu.

[4]A *real-time operating system* is distinguished from more mainstream consumer operating systems by allowing more stringent deadlines to be set and consistently achieved. One common usage of RTOSes is for sampling signals.

[5]*RTPS* is an abbreviation for "Real-Time Publish-Subscribe". Like TCP, it is a standard protocol, although with much greater flexibility; for example, one can tune the quality of service (QoS) to match the application, *e.g.*, by not requiring guaranteed delivery of packets for sensor data.

well. Fast-RTPS started out as an independent DDS implementation by eProsima[6], a Spanish company. Recently [14], eProsima agreed to work with the Open-Source Robotics Foundation (OSRF), the developers of ROS2. The Fast-RTPS developers' willingness to work with OSRF and implement requested features, along with Fast-RTPS being open-source, is a contributing factor to why Fast-RTPS is the default DDS for ROS2. However, Fast-RTPS is not the only supported DDS; other supported DDSes include RTI[7] Connext (which is proprietary and not free, although a free trial is available) and Vortex[8] OpenSplice (which is open source and available under multiple licenses). Despite the availability of these alternatives, the author of this thesis could find no compelling reason to use them; accordingly, this thesis only uses Fast-RTPS.

1.1.1 Explanation of tiny target comparison table

All values are approximations of orders of magnitude. Speed is in megahertz, RAM in megabytes, power in watts, and cost in United States dollars. Baxter cost is an estimate of only the computer (excluding the rest of Baxter). The OnePlus One was chosen as a representative of "modern" flagship-grade smartphones; additionally, it is what this author uses. The first iPhone is provided for historical interest. "CCC" is an abbreviation for "credit-card-sized computers", such as the Raspberry Pi family, C.H.I.P., and the Intel Edison family, which all use the ARM architecture (usually armv7l(hf), which is 32-bit). The last two columns use values from the ROSCON 2015 presentation. Note that ROS2 does not yet work on Arduino-class systems. All targets use the ARM instruction set unless otherwise specified.

[6]eProsima is an abbreviation for "e-PROyectos y SIstemas de MAntenimiento".

[7]RTI is the name of the company and is an abbreviation for "Real-Time Innovations".

[8]The company used to be named "PrismTech" and is now named "ADLINK Technology". The brand is "Vortex".

Table 1.1: A comparison of tiny targets. See subsection 1.1.1 for details.

Class	Desktop	Phone, 2014	Phone, 2007	CCC	MCU	8-bit
Example	Baxter	OnePlus One	First iPhone	C.H.I.P.	PX4	Arduino
Speed	1000 i7	1000	100	1000	100	10 AVR
RAM	1000	1000	100	512	$< \frac{1}{4}$	$\frac{1}{32}$
Power	100	10	5	$< \frac{1}{2}$	0.1	0.01
Cost	1000	250	600	9	10	2
Tiny?	No	Not really	Yes	Yes	Yes	Yes

1.2 Experiments

1.2.1 `jasadc`

All of the experiments run on C.H.I.P. were run with a custom program called "jasadc" (JAS's Analog-to-Digital Converter), available for both ROS1 and ROS2. This program reads values from the C.H.I.P.'s ADC via the sysfs interface. The ADC is part of the C.H.I.P. board and is electrically connected via an internal I2C bus. Values from the ADC are sent out to ROSn via `int32` messages (defined in the `std_msgs` package). Besides that package and the general ROSn C++ API package, `jasadc` has no other ROS dependencies.

The node included in the package publishes on two topics, `adc` and `microvolts`. The former publishes the raw ADC reading as an integer from zero to 4095 inclusive, while the latter is computed from the former via the following formula, which is defined in the function `convert_adc_to_microvolts()` in `src/main.cpp`.

$$\left\lfloor adc \times \frac{1\,000\,000}{2^{11}} \right\rfloor + (OFFSET \times 700\,000)$$

where adc is the value from the ADC and $OFFSET$ is `#defined`[9] to be zero. If it is set to one (and the program re-built), then the ADC will measure over the interval 0.7 V to 2.7 V instead of 0.0 V to 2.0 V.

[9]Defined with the C `#define` macro.

Both of these topics appear under the same namespace, `/chip_adc_MAC_address`[10], where "`MAC_address`" is the address of C.H.I.P.'s `wlan0`[11] interface[12]. This is the interface used for general device WiFi, and should also appear in the administrator page of the router to which C.H.I.P. is connected. Other C.H.I.P.s will be different. Note that these topics can be renamed or remapped like any other ROS*n* topic[13]. The end user can also directly modify `main.cpp`, the only source file. This naming scheme was chosen to ensure that multiple C.H.I.P.s could run in the same ROS namespace (*i.e.*, on the same robot) and not collide with each other. Additionally, the MAC address does not change when the system is rebooted, so one can manually map each C.H.I.P. to the physical place on the robot where it is mounted and then not have to change the mapping later.

With regard to sampling, by default `jasadc` samples as fast as it can. However, one can uncomment a line in the main loop of `main.cpp` to introduce a sampling rate and allow it to be set via a command-line argument.

1.2.1.1 Link to source

`https://github.com/jstarkman/jasadc`

This link points to a Git repository of the code for `jasadc`. Note that ROS1-compatible code is on a branch[14] called "ros1", while ROS2-compatible code is on a branch called "master". By default, the above link will show the "master" version.

[10] For example, the C.H.I.P. used to develop this package publishes under the topics named `/chip_adc_cc_79_cf_23_bf_71/adc` and `/chip_adc_cc_79_cf_23_bf_71/microvolts`.

[11] This changed in Debian Stretch. However, devices `dist-upgrade`d from Jessie retain their original interface names. Accordingly, if one day the producer of C.H.I.P., NTC, releases a build of Stretch for C.H.I.P., then the program will have to be updated to accodomate.

[12] In this context, an *interface* is a structure in the Linux kernel through which networking packets can travel. Usually, one interface is created per physical connection to a network (*e.g.*, per Ethernet port, per WiFi radio, &c.) as well as a few others (*e.g.*, the local loopback interface where 127.0.0.0/8 packets are routed).

[13] Topic remapping is not yet implemented in ROS2 as of the third beta.

[14] In Git, a *branch* is an easily-updated pointer to a commit. A *commit* represents a snapshot of the state of the repository at the given time, and is analogous to a "revision" in other version-control tools, including Subversion (svn).

The ROS1 version can be accessed through the "branches" section of the page; alternatively, append `/tree/ros1` to the base repository URL above. One can also clone[15] the repository to one's local machine and check out the branch directly.

1.2.1.2 Link to pre-built ROS1 and ROS2 workspaces

`https://github.com/jstarkman/jasadc/releases`

To further facilitate usage, under the "releases" section (linked above) one can find tarballs[16] of pre-built ROS1 and ROS2 workspaces. As of this writing, the ROS1 tarball uses the "bare bones" install of Kinetic (2016 LTS), while the ROS2 tarball uses the third beta (September 2017). ROS1 is available for both Debian Jessie and Stretch, while ROS2 is only provided for Stretch.

The "releases" section also contains pre-configured overlay workspaces for each ROSn. The overlay workspaces each contain an appropriate branch of `jasadc` and nothing else. If one chooses to use a pre-built overlay, then it is recommended that one update the version of `jasadc` present in the tarball to the latest release via running `git pull`. After doing so, one will then need to build the newest version with `catkin` or `ament`.

1.2.2 Instrumentation

The function generator used in these experiments is a 33120A 15 MHz function generator from HP[17] [15]. It was set to produce sinusoids with a +500 mV DC offset and peak-to-peak voltage (VPP) of 1.000 V. Due to the load of the C.H.I.P., these values are doubled, giving a sinusoid spanning the full two-volt range of C.H.I.P.'s ADC.

[15] In Git, *cloning* a repository means creating a full local copy of it, complete with full history. Since Git is decentralized, this copy is identical to the one on the server. This is analogous to a checkout in Subversion.

[16] A *tarball* is a single file containing all of the files of a directory and its child directories. It usually has the file extension `.tar`, often followed by `.gz` or `.bz2` (`tar` does not compress files, only hold them together). It is analogous to a `.zip` file, which is more commonly found on Windows.

[17] Also branded as "Agilent" and "Keysight", two HP spinoffs.

To move the data off of C.H.I.P., both C.H.I.P. and a laptop (representing the robot) were connected to the same WiFi network. For ROS1, this was the university's public WiFi network, CaseGuest. For ROS2, this was a private router with only C.H.I.P. and the laptop connected (no other devices nor Internet access). This discrepancy is due to CWRU's policy of blocking multicast[18] traffic on their network. While Fast-RTPS has an option to use unicast instead of multicast, that option was not found to work when using the XML configuration provided both in a presentation from ROSCON 2017 [16] and on eProsima's website [17]; Wireshark[19] always showed at least one multicast packet. For ROS1, it is not known how much load CaseGuest was under, or whether that had an influence on the data gathering. For ROS2, since only two devices were connected to the router, the network was clearly not under load.

The laptop captured the published messages to bagfiles[20] on its hard drive for future processing. Each bagfile is approximately ten seconds long and consists of a single frequency from the function generator. The following frequencies were sampled: 1 Hz, 3 Hz, 5 Hz, 10 Hz, 30 Hz, 50 Hz, 100 Hz, 200 Hz, 300 Hz, and 400 Hz. Plots created with `rqt_plot` of two of these frequencies can be seen in figure 1.1. Additionally, 90 Hz was captured for ROS2. The bagfiles were named "only-1Hz.bag", where "1" is the frequency from the generator in hertz. For each frequency, the ADC on C.H.I.P. always sampled at 200 Hz. The main loop of the program that sampled the ADC was allowed to run freely, meaning that there were no calls to `nanosleep()` nor

[18] *Multicast* is a feature that allows packets sent to certain IP addresses to be broadcast to all devices on the same network (with different addresses for different definitions of "same", *e.g.*, local, universal, &c.); this is in contrast to *unicast*, where the packet is only sent to a single recipient. Multicast is commonly used by NTP, the Network Time Protocol, to disseminate timing information more efficiently. Large corporate networks tend to block multicast traffic to avoid being flooded by spurious packets, although they can selectively allow NTP.

[19] *Wireshark* is a popular tool for sniffing and analyzing packets. It works on all major operating systems. It needs superuser/administrator rights to sniff but does not need them to analyze.

[20] A *bagfile* is a file containing a log of all of the traffic on a given set of ROS topics. Here, the only topic that was captured was the one that carried the reading from the ADC. This file is named after the `rosbag` program that generates it. This program gives the filename of the bagfile a `.bag` extension.

related functions. The only other programs running on C.H.I.P. were system defaults and the shell from which `jasadc` was run. In a deployment environment, this could be relegated to a startup script, negating the need for a terminal.

1.2.2.1 Explanation of sinusoid plots

Figure 1.1 shows plots of 1 Hz and 10 Hz sinusoids captured with ROS1 running on C.H.I.P.. The plots were created by playing the recorded bagfiles such that `rqt_plot` could plot their contents. Note the jaggedness from the irregular sampling. Plots and bagfiles from ROS2 were no better in this regard.

1.2.3 Methods

1.2.3.1 ROS1

Only one command was issued on C.H.I.P.:

```
ROS_REMOTE_URI=<IP address of laptop> rosrun jasadc main
```

This was allowed to run continuously. Separately, these commands were all running concurrently on the laptop:

```
$ roscore
```

$ `rqt_plot` (set to show incoming data from C.H.I.P.) (not necessary)

```
$ rosbag record <C.H.I.P. ADC topic> -O only-1Hz.bag
```

Each `rosbag` command was allowed to run for about ten seconds (the exact value is unimportant; the purpose is to record enough samples that future processing will not be adversely impacted). It was then interrupted (by pressing `^C` (control-C)), the frequency on the function generator changed, and restarted with a different output name (*e.g.*, "only-3Hz.bag") corresponding to the new frequency from the function generator.

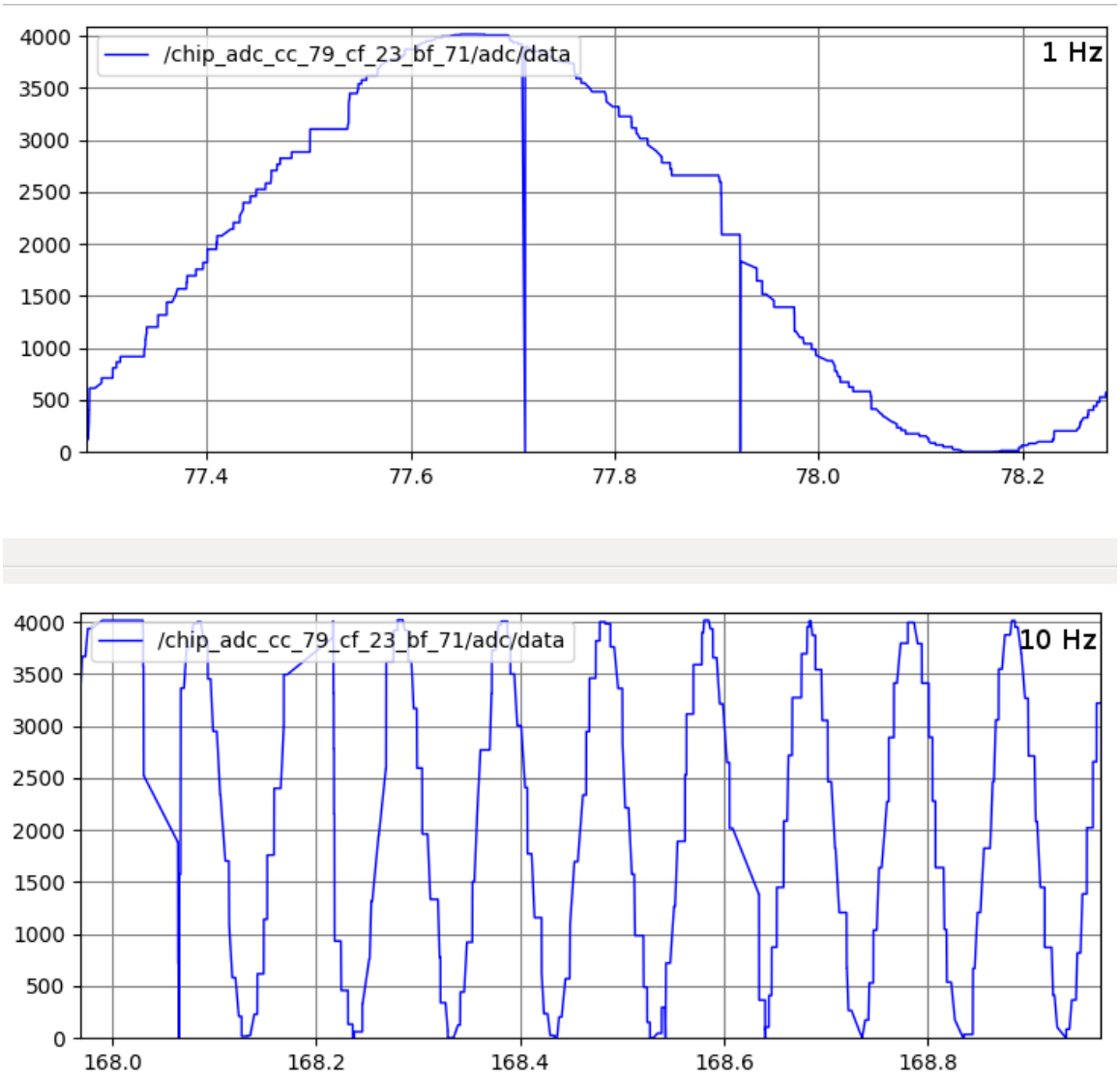

Figure 1.1: Plots of 1 Hz and 10 Hz sinusoids captured with ROS1 running on C.H.I.P.. See subsubsection 1.2.2.1 for details.

1.2.3.2 ROS2

Only one command was issued on C.H.I.P.:

```
root@chip# nice -n -10 ros2 run jasadc main
```

This was allowed to run mostly continuously, although it was restarted occasionally when traffic seemed to slow (see observations below). Separately, these commands were all running concurrently on the laptop:

- The other commands from ROS1

```
$ ros2 run ros1_bridge dynamic_bridge -- --bridge-all-topics
```

The bridge was required because, as of this writing and experimenting, ROS2 does not have an equivalent to `rosbag`. Apart from the bridge and the different program on C.H.I.P., everything else was the same as for ROS1. Note that, since Fast-RTPS uses multicast by default, one does not need to specify the IP address of the master computer (laptop) like one does with ROS1. This would allow for easier deployment and might be a reason to use ROS2 over ROS1.

1.2.4 Observations

1.2.4.1 Dropouts

The values displayed in `rqt_plot` showed a nontrivial amount of dropouts, jaggedness, and other sampling issues, as can be seen in figure 1.1. This motivated the "missed updates" component of the analysis.

Setting the function generator to produce a sinusoid at 200 Hz (the sampling rate of the ADC) would ideally have resulted in a flat line since the ADC would sample the same point on each cycle of the sinusoid. However, the curve that appeared on `rqt_plot` (with C.H.I.P. under ROS1) looked like the 1 Hz signal (the signal under

ROS2 was less coherent; see next subsubsection). Other multiples of the Nyquist rate[21] behaved similarly. This is why those values are in table 1.2.

1.2.4.2 Publication rate

Under ROS1, the ADC was sampled at a rate of about 780 Hz, as measured by `rostopic hz`. This is less than the 1000 Hz that one might want out of a real time system, but still shows well under an order of magnitude[22] difference.

Under ROS2 with Fast-RTPS, measuring the rate of publication to the ADC topic in the same way as with ROS1 produced a significantly smaller value, often in the range of 100 Hz to 200 Hz. This value was also subject to much larger variations: values as high as 350 Hz and as low as 50 Hz could sometimes be seen. When the rate was too low for too long, the sampling process was restarted, which seemed to speed up publication temporarily. Since at no point during the experiment were the laptop's eight CPU cores fully utilized (according to `htop`, which was running simultaneously in another window), the performance of the program that bridged ROS2 to ROS1 should not be a major factor. Since the ADC samples at 200 Hz, the ROS2 sampling rate may be low enough to cause nontrivial problems.

The poor publication rate of ROS2 — specifically the error term associated with sampling the 100 Hz sinusoid, as compared to ROS1 — is what motivated the sample at 90 Hz, in an attempt to see how bad reconstruction was near but below the Nyquist rate. See the reconstruction table in the next section for numeric values.

[21] The *Nyquist rate* of a signal is twice the highest frequency present in that signal (all signals can be represented as a summation of sinusoids; to convert, one uses the (inverse) Fourier transform). The Nyquist rate is the minimum frequency at which one must sample the signal in order to fully reconstruct it without error. Accordingly, if one always samples at a given frequency (*e.g.*, 200 Hz), then reconstructions of signals that contain sinusoids whose frequencies are above half of the sampling frequency will have errors while lower-frequency signals will be fine; following the previous example, under ideal conditions, a 99 Hz sine wave would be recovered properly while a 101 Hz sine wave would be an error-laden mess.

[22] $\log_{10}(780) \approx 2.9$, which is close to $\log_{10}(1000) = 3$.

1.3 Analysis

The full analysis for this chapter was carried out in an IPython notebook[23] via Jupyter[24]. This notebook file, `only-foo-analysis.ipynb`, is available with all of the other source code used in this thesis. The only difference in the analysis between ROS1 and ROS2 is in the path to the bagfiles in one of the earlier cells[25], and even that could be avoided by restarting the IPython kernel from different working directories (since the path is relative); no changes are needed to any other part of the file, although all cells will still need to be re-run with the new data since they do not automatically update.

1.3.1 Publication rate

The ADC on C.H.I.P. samples at 200 Hz, meaning that five milliseconds pass between each update. Thus, any dropout in the data publications of over five milliseconds is at risk of missing an update. To determine this, we can take the difference between the timestamps of the samples and count how many are over the threshold. Since C.H.I.P. was running more-or-less continuously through all cases, there is no reason to segregate the different frequencies from one another. Doing this showed that for ROS1 about 5.81% of updates happened more than five milliseconds after their predecessors, while for ROS2 about 31.23% of updates were at least as late.

However, five milliseconds is a rather arbitrary choice, as it was chosen because of the particular ADC used on C.H.I.P., not because of anything ROS-specific. Accordingly, a "1% line" was added to each plot to show the gap size that encompasses 99% of all updates. That is, 1% of all updates for a given ROS version are to the right of

[23] IPython is an abbreviation for "Interactive Python". It is a program that can run Python code in snippets, much like the Mathematica kernel. It is sometimes referred to as the IPython kernel.

[24] Jupyter is a suite of programs that provide graphical user interfaces to kernels of various languages, such as the IPython kernel for the Python language.

[25] In these notebooks, a *cell* is a snippet of code that is treated as a unit. It is analogous to a cell in Mathematica.

each 1% line, while 99% are to the left.

The distributions of timestamp separation times for both ROSn can be seen in figure 1.2. Note the long tails on the distributions, as well as the logarithmic vertical axes and the deliberately-matched scales. The full axes for both ROSn can be seen in figure 1.3. Note the even longer tail on ROS2, as well as the complete lack of dropout delays for ROS1 after about 260 ms.

Why the disparity? After all, ROS2 was seemingly given every advantage: ROS2 packets ran over a dedicated network instead of a (potentially) crowded public guest network; the `ros2 run` process was given a lower niceness (-10), thus having a higher priority to the Linux kernel than what the ROS1 process had (0); and ROS2 is said to work better on smaller targets. Why did the ROS2 program only publish new messages at less than half of the rate of the ROS1 program?

In part, this is due to running full ROS2 in user space[26] on a slightly-modified Debian installation. This means that numerous non-critical programs were running concurrently with each sampling program. If a more pared-down real-time system were used, then perhaps publication would happen at a more consistent rate. As shown in [8], if one uses Fast-RTPS directly from such an environment, then one can achieve higher publication rates than ROS1 achieved in the experiments presented earlier.

However, some potential impacts are less likely. For example, the only difference between the two programs running on the laptop was the addition of the bridge for ROS2. Since the laptop's CPU never ran near full capacity during the experiments, the bridge is unlikely to have caused the deterioration in publication rate.

[26] *User space* refers to the part of memory where user programs run. It is in contrast to *kernel space*, where protected operating-system-critical processes run. To exemplify, drivers run in kernel space, while shell commands run in user space.

1.3.1.1 Explanation of histograms

This subsubsection applies to both figures 1.2 and 1.3. These histograms show times between updates for ROS1 and ROS2 on C.H.I.P.. The percentages near the vertical lines show how many values are to the right of the respective line. The left line is set to five milliseconds. The right line was chosen such that 1% of samples are to the right of it. It is at fifteen milliseconds for both histograms.

For the first figure, note that the horizontal axis of the histogram for ROS2 has been cropped to make it use the same scale as the histogram for ROS1. The second figure uses the same histograms as in figure 1.2, but with the horizontal axis changed to show all of ROS2.

1.3.2 Reconstructing input from samples

In spite of the above issues with sampling, the signal can still be recovered from the samples.

First, the input values are resampled to force them to occur at five-millisecond intervals. Gaps in coverage can then be recovered via a variety of interpolation methods. Linear interpolation was chosen as it is cheap to compute, makes conceptual sense for low-frequency sinusoids, and appears to work well (for ROS1, at least).

After cleaning the input, the absolute value of a real-value FFT can be found. The largest element of the transformed data corresponds to the strongest frequency present in the input signal. Since the signal has a DC component (one volt) that we are not interested in, we deliberately avoid checking the value of the FFT at 0 Hz when searching for the peak.

After finding the peak, we can compare it against the known value from the function generator and compute an error term. The results of these calculations are in table 1.2.

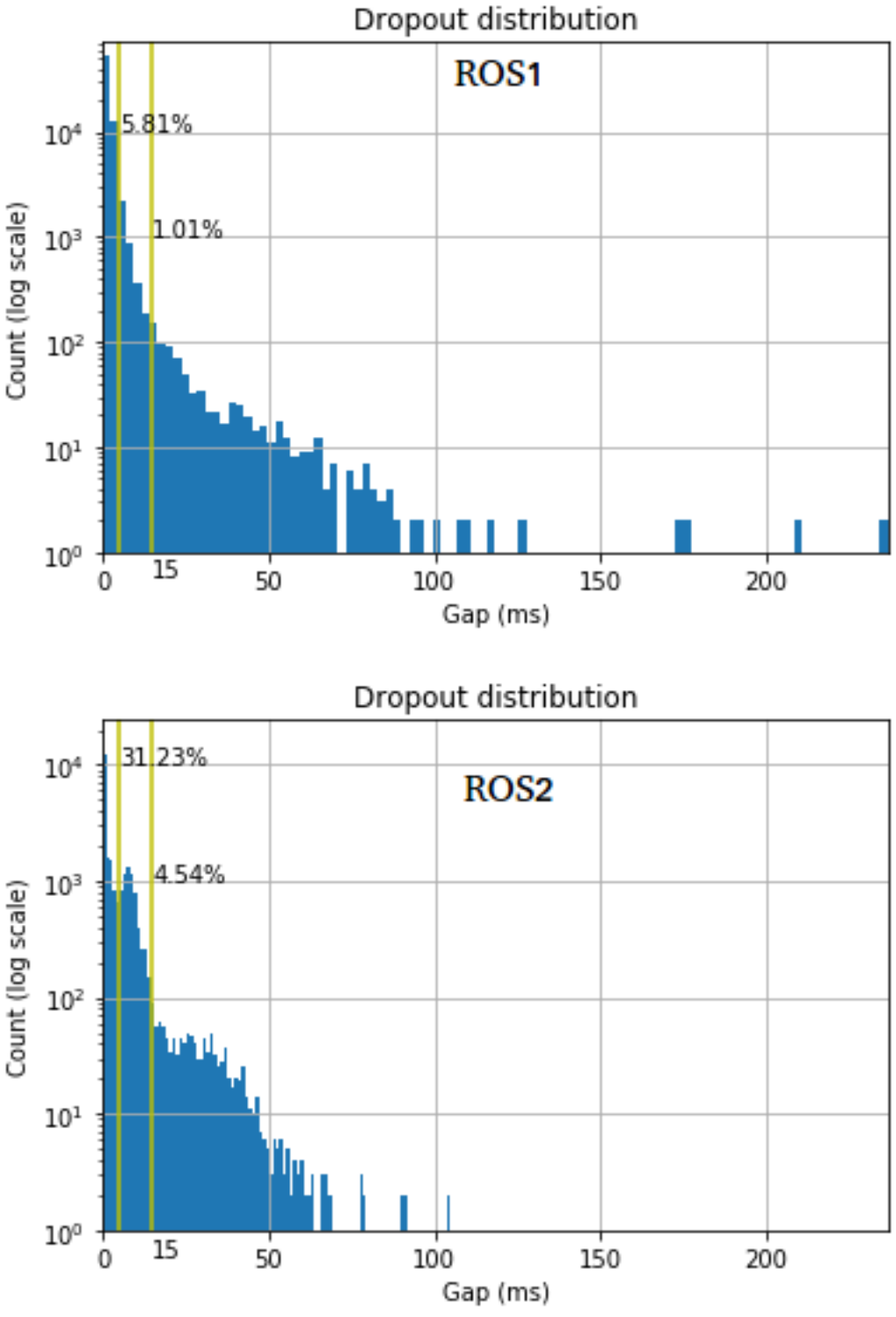

Figure 1.2: Histogram, short abscissa. See subsubsection 1.3.1.1 for explanation.

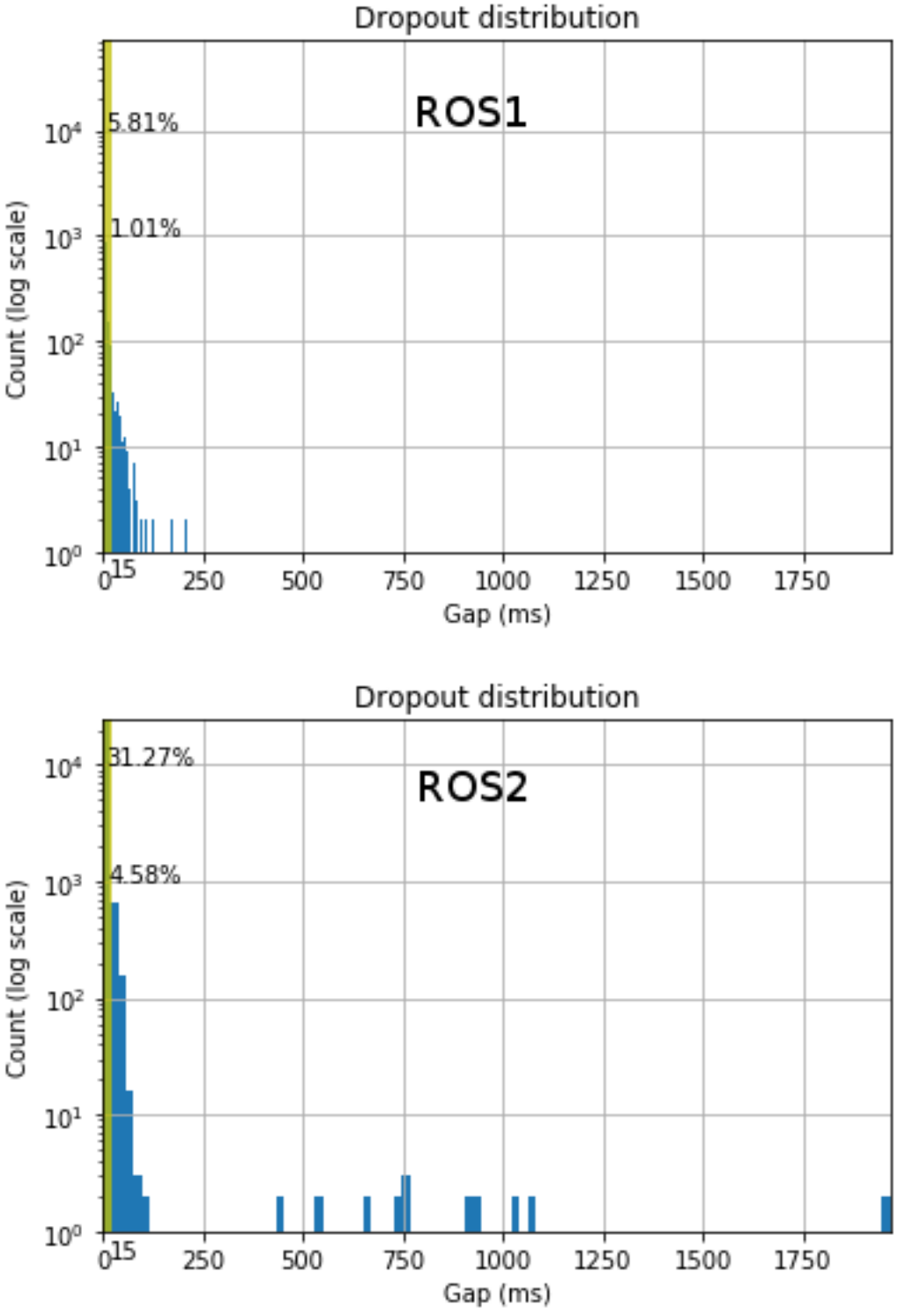

Figure 1.3: Histogram, long abscissa. See subsubsection 1.3.1.1 for explanation.

Table 1.2: Frequency recovery estimates and errors. See subsubsection 1.3.2.1 for details.

f_t	f_1	e_1	f_2	e_2
1	0.953	4.71%	0.955	4.46%
3	2.983	0.55%	3.022	0.74%
5	4.960	0.79%	4.952	0.97%
10	9.989	0.11%	10.021	0.21%
30	30.042	0.14%	30.000	0.00%
50	50.000	0.00%	50.025	0.05%
100	97.152	2.85%	5.508	94.49%
200	2.929	98.54%	3.376	98.31%
300	94.396	68.53%	9.766	96.74%
400	5.514	98.62%	6.794	98.30%

1.3.2.1 Explanation of frequency recovery table

All frequencies are in hertz. f_t is the theoretical frequency of the sinusoid supplied by the signal generator. f_n is the empirical frequency for ROSn as determined by the ADC on C.H.I.P. and subsequent processing. e_n is the corresponding error term. The horizontal line represents the Nyquist rate, after which sampling becomes unreliable. The estimate and error for ROS2 at 90 Hz are 23.177 and 74.2%, respectively.

1.4 Conclusion

As can be seen from the preceding experiments and analyses, if one tries to use ROS2 like ROS1 on a tiny target then one will not get the same results. This can be seen throughout the work; for example, building the ROS2 workspace took much longer than building the ROS1 "bare bones" workspace did, and provides different features. Once installed, ROS2 had markedly worse performance in terms of publication rate, although performance was minimally different after acausal signal recovery. Both of these points are addressed in the subsections below.

For those who wish to experiment with both versions of ROS, the full patch needed

to convert `jasadc` from ROS1 to ROS2 can be found in Appendix C[27]. Note that in `main.cpp` only fourteen lines changed, all in straightforward ways. More information about porting can be found in the second part of this thesis.

1.4.1 Once-off aspects: cost, setup, and installation

Neither ROSn was particularly difficult to install on C.H.I.P.; furthermore, since tarballs of the workspaces that were built are provided freely on the Internet to anyone who wishes to use them, installation difficulties should not be a deciding factor when choosing the version of ROS that one wants to run on one's C.H.I.P., nor should it be a factor when weighing the US$9 C.H.I.P. against other small ARM Linux targets, such as members of the US$20–40 Raspberry Pi family. As for cost, one's budget will determine whether or not that is relevant to the decision.

1.4.2 Performance

When running ROSn on C.H.I.P. in user space, ROS1 performs better than ROS2. As seen in table 1.2, for many cases below the Nyquist rate ROS1 has a lower error rate than ROS2, although most of the error terms are so low that the difference can probably be neglected. Perhaps a better deciding factor would be from the observations of the publication rate, where it was noted that the ROS1 version of `jasadc` sent packets at a more consistent rate than did the ROS2 version, as can be seen in the histograms referenced in subsubsection 1.3.1.1. That being said, in spite of these issues the sinusoids could still be reconstructed about equally well under both ROSn; as such, for any real system where one wants to use the ADC on C.H.I.P. at a low sample rate, one should use the version of ROS that is most compatible with the rest of the robot.

[27] Also available online at `https://github.com/jstarkman/jasadc/commit/8b85326`.

1.4.3 Overall

Overall, if one wants to sample an ADC and send the values to ROS, `jasadc` on C.H.I.P. is not a bad choice. It can easily be added to a robot's existing WiFi network and one can use pre-made workspaces to set up a new device fairly quickly. While the sampling may involve non-trivial dropouts, it should still be good enough for some applications.

1.4.4 Future work

One possible avenue to explore in future iterations on this subject might include changing the niceness of sampling process; for ROS1, this analysis only used the default value of 0 (resulting in a priority of 20). By lowering the niceness, the ADC could be read more often and fewer updates might be missed. Another area of exploration is the signal processing done to the sampled values. In this analysis, missing values were filled in with linear interpolation since the signal was known to be sinusoidal; how would other strategies fare under different circumstances? Futhermore, the interpolation was done offline and used future values to predict missing ones, resulting in an acausal system. This is clearly not feasible for a real system, but might be acceptable for after-the-fact reporting.

Chapter 2

Windows

2.1 Introduction

Microsoft Windows is a highly widespread desktop operating system. Its usage is no less prevalent among large corporations, including those that are interested in robotics. However, ROS1 does not run natively — *i.e.*, being compiled into `.exe` and `.dll` files — on Windows; rather, it only runs on Unix-like operating systems, and while it can be made to work (to an extent) via various make-Windows-act-like-Unix schemes (such as Cygwin and WSL; see below), these approaches are not free of issues, particularly with lower-level hardware acceleration such as GPU access. Furthermore, as of the 2016 release of ROS1, Kinetic, ROS1 is only supported on Ubuntu and Debian, although there are experimental builds for other Linux distributions (such as Gentoo and Yocto) and even operating systems created by Apple Inc. (both desktop and mobile). Clearly, this is not an optimal situation, as limited operating system support deters adoption and usage. ROS2 attempts to address this by running natively on Windows, which means that one can compile packages with the Visual Studio x64 (or x86) Native Tools command prompt.

There are two existing methods for running ROS1 on Windows. Both of them

amount to using Windows to mimic Unix-like operating systems and letting ROS1 think that it is being run on a Unix-like machine. The following subsections will also provide an example of the difference between the two.

2.1.1 Cygwin

Cygwin is a system for running Unix-dependent programs (*e.g.*, Bash and Git[1]) on Windows [18]. One of the major components of Cygwin is a dynamically-linked library (`.dll` file) that provides a Unix-like interface. This interface is implemented with the corresponding Windows equivalents from the Win32 API. For example, the system-level Unix function `sched_yield()` is implemented with the public Win32 function `SwitchToThread()` from `kernel32.dll` [19].

2.1.2 WSL

The Windows Subsystem for Linux (WSL) is another system for running Unix-dependent programs on Windows. WSL can be installed on any machine running Windows 10.1607 or later; there are no minimum system specifications beyond what is needed to run Windows itself. In particular, as of Windows 10.1709 WSL cannot access the GPU; as such, it does not matter what GPU(s) the system has installed.

WSL is implemented via a kernel driver and works by translating system-level functions from the Linux kernel into their Windows kernel equivalents. For example, the `sched_yield()` function is implemented with `ZwYieldExecution()` from `ntdll.dll` [20]. Note that this function is identical in functionality to the aforementioned `SwitchToThread()` function.

[1]Git for Windows version 2.x is implemented with MSYS2, a stripped-down version of Cygwin.

2.2 Experiments

The headings in this section are structured as follows: "general" means both ROSn, while the other two subsubsections are for specific changes for the given version of ROS.

2.2.1 Setup

2.2.1.1 General

All experiments for this chapter were run on the same desktop computer. The hostname[2] of this machine is "THESEUS", as can be seen in the various screenshots that follow. The machine has 16 GiB of RAM, a modern Intel processor, and a mechanical disk drive. The machine boots into Windows 10.1703.

2.2.1.2 ROS1 running via WSL

This subsubsection contains instructions for installing ROS1 2017 (Lunar) in Windows via WSL. The instructions are taken from [21].

First, enable WSL with Ubuntu 16.04 (Xenial). The details on doing this vary with the different versions of Windows 10; for best results, consult the official Microsoft documentation [22]. Note that this requires Windows 10.1703 Creators Update or later.

Then, install ROS1 Lunar from Ubuntu Xenial binaries (`.deb` files). Details on how to do this can be found on the ROS wiki [23]. The procedure works the same way as installing ROS1 Lunar on an Ubuntu machine.

Historically, after installing the binaries one had to update the "ros_comm" package by checking out the latest revision on the "master" branch and re-running the

[2] A *hostname* is the name a computer uses on its local network. For reference, the typical command prompt on Unix-like systems is of the form `username@hostname`.

build tool, `catkin_make`. However, as of this writing the latest official ROS1 Lunar releases contain these changes, thus rendering this step obsolete.

2.2.1.3 ROS2 running natively

ROS2 was installed as per the wiki on the ros2/ros2 GitHub repository. This involved manual installation of various programs (*e.g.*, OpenSSL[3], Visual Studio[4], and OpenCV 2[5]), as well as automated installation via Chocolatey [24]. Chocolatey is a package manager for Windows that works similarly to the more Linux-friendly apt, yum, pacman, and others, allowing one to install multiple software packages entirely from the command line with no Web browser, GUI, or monitor required. It is based on NuGet, specifically `Nuget.Core.dll`. Chocolatey packages can contain arbitrary Windows installers (*e.g.*, MSI files, zipped archives, and self-extracting executables; since "Windows has over 20 different installer formats" [25], this is not an exhaustive list). These installers can be run completely from the command line without the use of a GUI nor human interaction of any kind.

2.2.2 Methods

Each applicable feature was tested under both ROS1 and ROS2. Lists of what works and what does not work were made. Screenshots depicting these features were taken to provide evidence to support their respective existences (or non-existences). The following lists itemize the subset of these features that are (or should be) present in both ROSn, as well as the ones that only apply to a single ROSn and the ones that do not work in either ROSn under Windows:

- Features present in both ROS1 and ROS2 and working on Windows

[3]For building features related to security and encryption.

[4]For the build tools for Windows Native, including a C++ compiler, API bindings, and the integrated development environment itself.

[5]For building features related to cameras and image processing. Note that, as of Ardent, the default publish-subscribe middleware, Free-RTPS, does not yet handle large payloads efficiently.

  - Minimal working examples:
    * Publish/subscribe
    * Services and parameters[6]
  - RViz
  - `learning_ros` [26]

- ROS1-only features (working on Windows via WSL)
  - Action servers
  - `rosbag`

- ROS2-only features (working on Windows natively)
  - Multicast
  - No need to install WSL

- ROS1-only features that do not work on Windows
  - Gazebo

2.2.3 Observations

2.2.3.1 ROS1 running via WSL

See figure 2.1. It shows an example of ROS1 Lunar running on Windows 10.1703 via the Windows Subsystem for Linux. The following programs are depicted running concurrently (clockwise from top-right): `roscore`, `turtlesim/turtlesim_node`, `turtlesim/turtle_teleop_key`, and `Xming` (showing the turtle simulator). This demonstrates Publishers, Subscribers, and running X programs on Windows.

[6] In ROS2, parameters are quite clearly a special case of a service.

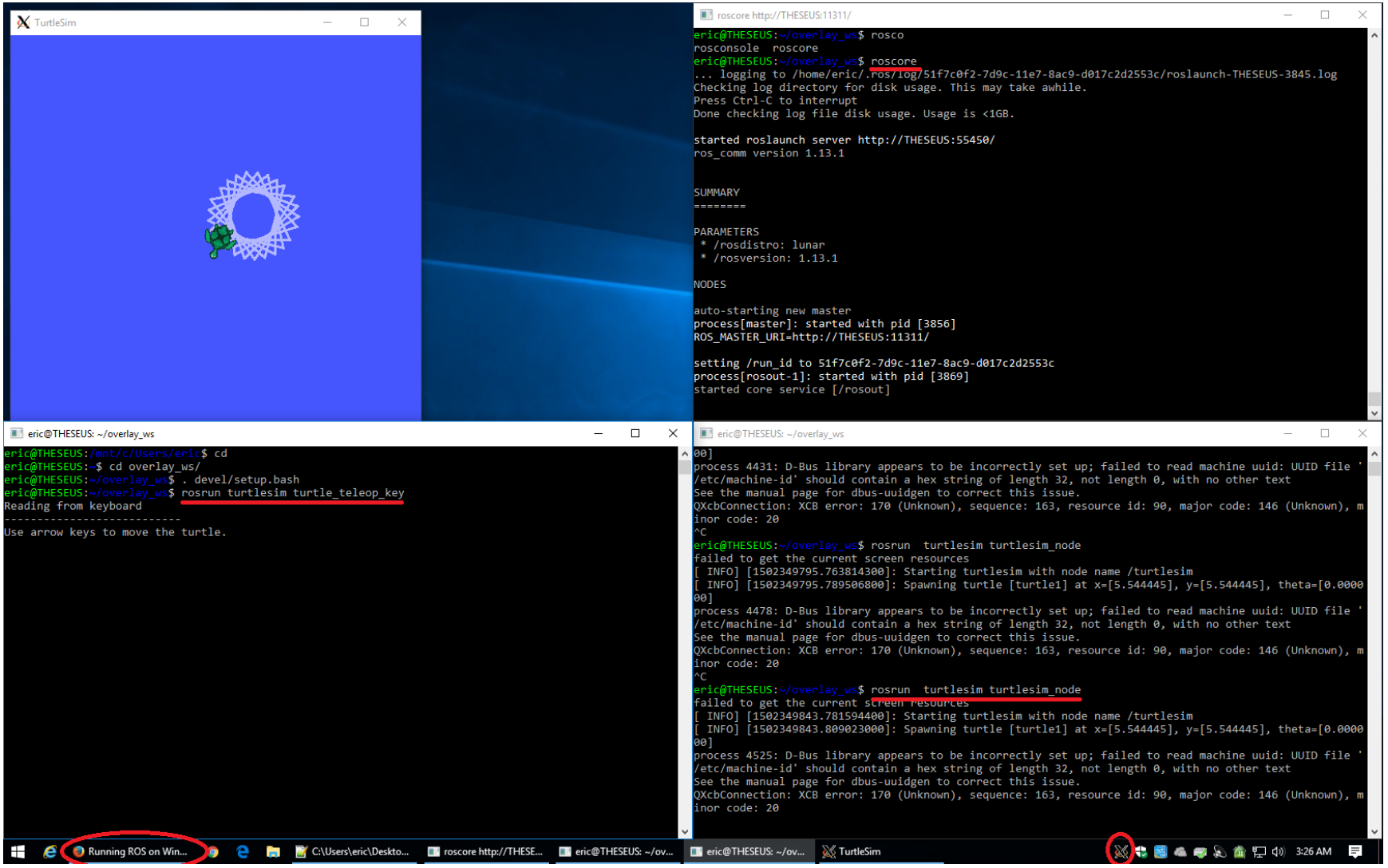

Figure 2.1: The standard ROS1 introductory programs running unmodified on Windows via WSL.

2.2.3.2 ROS2 running natively

See figure 2.2. It shows an example of ROS2 beta one running on Windows 10.1703 natively. The subscriber in the image is being debugged in Visual Studio; see the miscellaneous addenda in the tutorials for an explanation of how to debug a ROS2 program in Visual Studio. This demonstrates Publishers and Subscribers working in Windows.

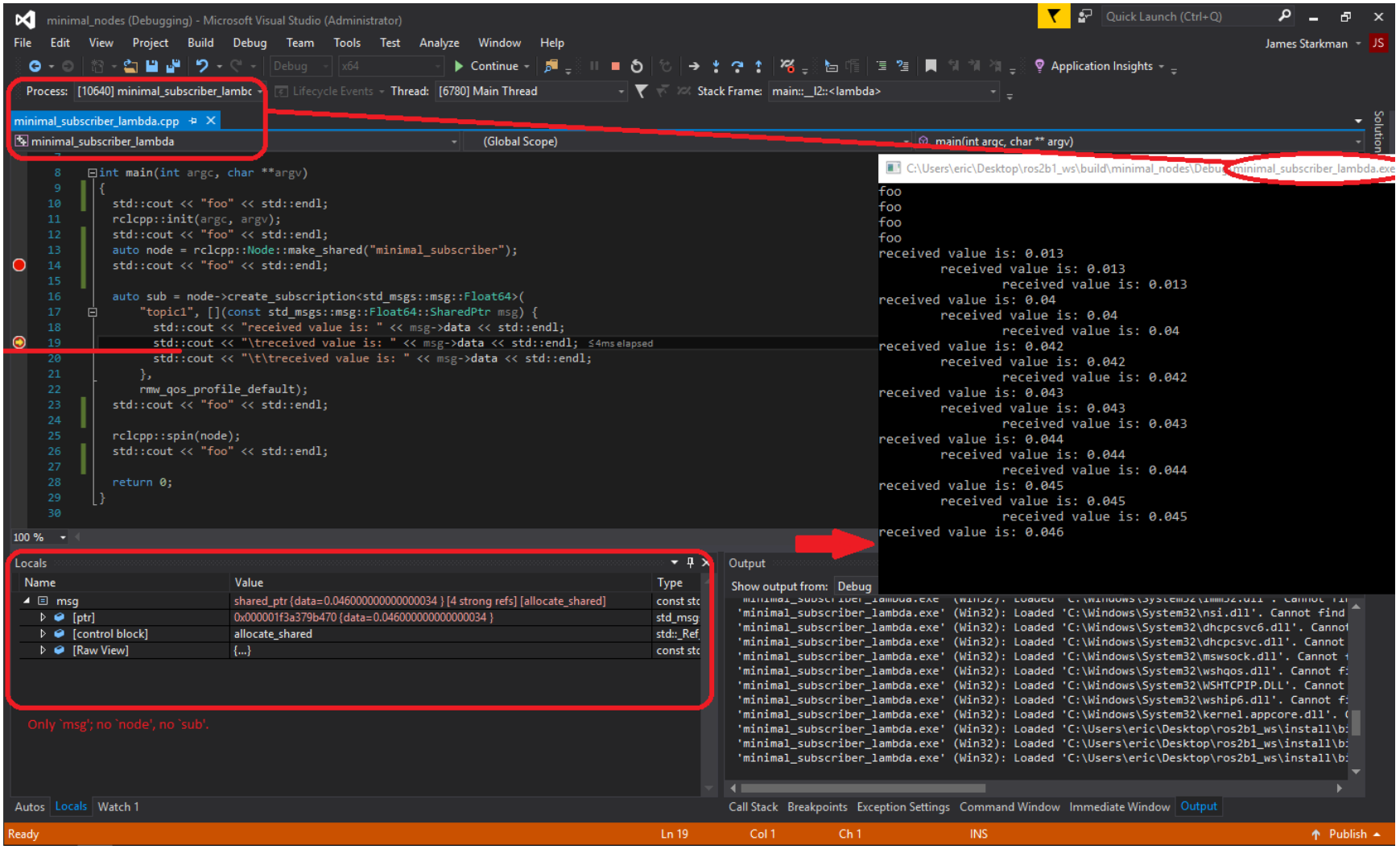

Figure 2.2: A minimal subscriber in ROS2. See the miscellaneous addenda in the tutorials for an explanation of how to debug a ROS2 program in Visual Studio.

Chapter 3

Miscellaneous

3.1 Introduction

There are many differences between ROS1 and ROS2. This chapter looks at two major additional considerations beyond those explored in previous chapters. One of these considerations is differences in available programming languages and libraries. ROS2, being a newer platform, has correspondingly greater platform availability, although language availability is reduced due to the relatively young age of the project as many user-contributed language bindings have not yet been ported. Another consideration is the availability of security and encryption features, which are largely non-existant in ROS1.

3.2 Programming languages and libraries

3.2.1 Platform differences

3.2.1.1 ROS1

ROS1 started in the year 2007 [2] with the first major releases (Box Turtle and C Turtle) in 2010 [27]. Then as now, the dominant approach to writing performance-

sensitive programs and libraries was and is to use C++. At the time, the latest version of C++ was C++03 (ISO/IEC 14882:2003), and to this day that is the most recent version of C++ that all supported distributions of ROS1[1] are able to use [27]. Note that while as of Kinetic C++11 may be used, packages that wish to support older versions of ROS1 must either restrict themselves to C++03 or provide macros[2] to make the code work if C++11 features are not available[3]. As such, the new features available in C++11 and later are of limited availability in ROS1.

For programs, libraries, and tools that are not sensitive to performance, ROS1 originally decided to use two interpreted languages: Python and Common Lisp, specifically CPython[4] and Steel Bank Common Lisp (SBCL). Python is used for many core ROS1 tools, including rosmaster and roslaunch, while Lisp "is currently being used for the development of planning libraries" [28]. Since Python 3 would not be released until 2008, and would not be in popular use until somewhat later, ROS1 uses Python 2. Furthermore, since Python 3 is not completely backwards compatible with Python 2[5], ROS1 still uses the older version, although since ROS1 Indigo (2014 LTS) there has been a standing recommendation that users test against the latest minor version of Python 3 (*e.g.*, Python 3.3 for Indigo and Python 3.5 for Lunar) [27]. With regard to Lisp, ROS1 tracks a single, stable version of a single implementation of Common Lisp and nothing else. This compiler is included in ROS as "roslisp" and is not a system dependency like Python and C++ are.

[1] As of late 2017, the supported distributions are Indigo, Kinetic, and Lunar.

[2] A *macro* is a feature found in various programming languages (including C++) that allows code generation (also called "expansion") based on lexical rules. Macros are expanded by a *preprocessor*, which processes the source code before giving the result to the compiler. Since the expansion happens based on the text of the program (without checking types, syntax, or even what language the program is in), one can cause the code to do *anything*, from creating functions to conditionally including entire blocks of code.

[3] *e.g.*, by checking the language version and only including code that uses newer features if the language version is high enough to provide them. If not, code that is compatible with lower versions would be included instead.

[4] The reference implementation of Python, available at `https://www.python.org/`.

[5] While some programs can be written that function correctly with both versions of the interpreter, this cannot be expected of an arbitrary, unmodified program.

In addition to the three main client libraries, there also exist numerous "experimental" client libraries of varying stability for many other programming languages, including Java (with Android support), Lua, and Haskell [28], along with not-as-well-supported (but still functional) clients for other languages, including JavaScript in a web browser [29] (Node.js is experimental) and Rust [30].

3.2.1.2 ROS2

In constrast, ROS2 started in the mid-2010s with the first major release near the end of 2017. At this point in time, support for Python 3 was much more prevalent[6], and the official end-of-life (EOL) date[7] for Python 2 — sometime in the year 2020 [33] — was much closer. Accordingly, ROS2 uses Python 3 for all scripting purposes; notably, Common Lisp support is absent. With regard to performance-critical code, ROS2 still uses C++ like its predecessor. However, since C++11 was several years old when work began on ROS2, ROS2 can be designed and written with new features like lambdas[8], `auto`, and improved multithreading and asynchronous code throughout, as well as expecting client code (*i.e.*, what end users write) to take full advantage of these same features. Furthermore, ROS2 works with newer versions of C++ (*e.g.*, C++14 and C++17) than 11. Thus, not only does ROS2 start with more recent language versions, it provides a more clear upgrade path than was the case under ROS1 (which still uses Python 2) [34].

[6]For example, NumPy (Numeric Python) was ported to Python 3 in 2010 [31]. This is revelant because, among code publically-available on GitHub, NumPy "might be the single most-imported non-stdlib [*i.e.*, third-party] module in the entire Pythonverse", more so than even the `django` and `lxml` modules [32].

[7]The *end-of-life date* is the date on which a given product is no longer supported. EOL'ed software will no longer received bug fixes, security patches, or other changes of any kind.

[8]A *lambda expression* is an anonymous function, meaning that it has no identifier. While lambdas are sometimes assigned to local variables, the important distinction is that a newly-constructed lambda can access the same scope (*scope* refers to the variables to which a given expression has access) as was available where it was defined, while a standard, named function has no scope beyond global values and, if the function is a non-static method of a class, the object to which it is attached. See following subsubsections for reasons why one might use this construct.

3.2.2 Language differences

3.2.2.1 Python

While Python 2 and Python 3 have much in common, they are not compatible. While the languages themselves are quite similar, their internals are quite different and many major libraries (*e.g.*, NumPy[9]) were not immediately ported when Python 3 first became available in 2008 [36] (NumPy was ported to Python 3 in 2010 [31]). The new version of Python broke compatibility in order to permit more extensive design changes, so that the BDFL[10] and maintainers could "clean up Python 2.x properly" [37]. Due to this backwards incompatibility with the older version of Python, ROS1 still uses Python 2 and as of Lunar (2017 LTS) has not fully transitioned to Python 3 yet. Since ROS2 is a new project, it does not have legacy code to port and can start requiring Python 3 directly from the initial pre-alpha releases[11] [34]. In terms of what the cleanup entailed, besides internal speed improvements and bug fixes, some of the major ROS-relevant changes that occured are as follows:

- Strings are now all in Unicode (UTF-8[12]) and byte arrays are their own type. In Python 2, strings and byte arrays were of the same type, and there was another type for Unicode. This change in typing may reduce confusion when dealing with binary data in ROS messages, as well as being more accomodating of non-English languages.

- Integer division now requires an explicitly-doubled solidus, `//`. A single solidus will return a floating-point number. In Python 2, a single solidus would only

[9] *Numeric Python (NumPy)* is a Python package for controlling large-scale computations with Python. By "controlling", it is meant that most of the actual element-by-element matrix operations are written in C and Python is only used to wrap the code to present a Python API. This is similar to the approach taken by MATLAB®, Julia, and other higher-levelled languages, albeit with marginally faster results [35].

[10] The *Benevolent Dictator for Life* (*i.e.*, the person in charge) of the Python project is Guido van Rossum.

[11] Specifically, ROS2 started using Python 3.4 (according to the initial revision of [34]) and has since upgraded to Python 3.5 for the first major release.

[12] A particularly popular and comparatively space-efficient encoding of Unicode.

return a float if at least one of the inputs was a float. Since programs for controlling robots may require the use of division or averages, this may be a potential source of bugs when porting a Python library from ROS1 to ROS2.

- The `xrange` function has been renamed `range`. This function works by generating a list of sequential values from the given start value (defaulting to zero) up to the given stop value (required argument) (does not return the stop value itself), differing by the increment or decrement (defaulting to an increment of one) each time. This is also called "lazy evaluation". The old `range` function from Python 2 is no longer present in Python 3. Since these functions are the idiomatic way to iterate over a loop a given number of times, existing ROS1 code that depends on them will break when porting.

- As of Python 3.5 (the minimum requirement for ROS2 [34]), there is now a dedicated infix operator for matrix multiplication: `A @ B`[13] [32]. This is supported by NumPy version 1.10 and higher when installed in Python 3.5 or higher [38] and may allow for simplified notation in ROS programs that use matrix operations, thus making it easier to develop and prototype matrix-dependent algorithms in Python.

(Citation for above itemized list: [39].)

3.2.2.2 C++

Unlike with Python, the newer version of C++ that is used by ROS2 is backwards compatible with its predecessor. This means that existing C++ executables and libraries should still build correctly under a newer compiler, although since some libraries depend on ROS1 they cannot be used without being ported. A notable

[13] This operator can be overridden by defining these methods: `__matmul__`, `__rmatmul__`, and, for the in-place (`@=`) case, `__imatmul__` . This follows the same pattern that other operators in Python use for overriding.

example is the library "tf": it was deprecated in ROS1 Indigo (2014 LTS), and while it is still available in ROS1 Lunar (2017 LTS) it is *not* available in ROS2. Instead, ROS2 code must use the library "tf2", which has been ported to ROS2. Since this library is also available for ROS1, one should port one's existing "tf" code to use "tf2" before contemplating porting from ROS1 to ROS2. Alternatively, one could port "tf".

With regard to the C++ language itself, the new features in C++11 (and newer) are generally designed to allow one to write code at a higher level of abstraction while preserving run-time performance (also called "zero-cost abstractions"[14]). Here are a few that would be particularly useful for ROSn programs:

- Many parts of the Boost library have been incorporated and improved upon as part of the standard library, including:
  - Foreach loops (removes need for explicit indexing of loop iterations)
  - Lambda expressions (defining a function locally; used to use `_1`, now uses dedicated syntax. Good for callbacks[15])
  - Regular expressions (defaulting to ECMAScript-style, which is broadly similar to PCRE. These terms refer to the syntax used in constructing the expressions; a full explanation is beyond the scope of this thesis)
  - Function binding (also called partial application of a function, wherein a new function is created with fewer arguments than the old function; the other arguments are "baked in" to the new definition)

[14]For example, one kind of zero-cost abstraction is inlining, where a function's body is safely inserted where the function call appears. This allows one to avoid the overhead of calling a function while retaining the benefit of separating out that block of code.

[15]A *callback* is a function that is not run immediately, but rather to handle some other event when it occurs. The event is said to "call [the function] back", hence the name. For example, when a new message is received by a ROSn Subscriber, this event causes the Subscriber to execute the callback function with which it was initialized.

- One can now use the `auto` keyword to automatically set the type of a variable to the type of the right-hand side of the equals sign. This is particularly useful for long type names with nested namespaces, a common occurrence in both ROSn. For example, one can simplify the first of the following two lines into `auto msg =`

  - `std::shared_ptr<std_msgs::msg::String> msg =`
  - `    std::make_shared<std_msgs::msg::String>();`

- One can now use lambda expressions to capture code locally. This is ideal for writing function callbacks, as one can spatially locate the implementation near the function call that consumes the callback. For example, one could use a class to define a logger and capture a pointer to an instance of that class for use inside the lambda, which itself could be attached to a Subscriber. This entire pipeline could happen inside the same function body, possibly even in the class constructor. See listing 1 for what this looks like in code. That example is taken from the official ROS2 examples repository (as of Ardent).

(Citation for above itemized list: [40].)

```
// inside the constructor of a subclass of rclcpp::Node
auto sub = this->create_subscription<std_msgs::msg::String>(
      "topic",
      [this](std_msgs::msg::String::UniquePtr msg) {
            RCLCPP_INFO(this->get_logger(),
                        "I heard: '%s'", msg->data.c_str())
      });
// Reference: https://github.com/ros2/examples/blob/
//                e2ab494/rclcpp/minimal_publisher/lambda.cpp
```

Listing 1: Example of a C++11 lambda expression being used as a Subscriber callback. The lambda is the part beginning with `[this]` and includes the following arguments and code block.

3.3 Security and encryption

3.3.1 Introduction

In ROS1, security and encryption[16] are largely nonexistant. Most communications are transmitted in cleartext[17] over unencrypted TCP sockets[18]. While there do exist some efforts towards securing communications in ROS1, they are still experimental and are not suitable for production code [41].

In ROS2, security is dependent on the middleware via DDS-Security. DDS-Security is defined by OMG, the same group that defined DDS itself. If correctly implemented, DDS-Security will prevent third-party attackers[19] from reading and modifying the contents of the RTPS transmissions. See figure 3.1 for a visual comparison of what a packet sniffer would see with and without encryption.

Security in ROS2 is provided by the SROS2 package [42]. The next few subsections will summarize the installation and usage of SROS2 as of ROS2 beta three. Note that this will require OpenSSL to be installed. Ubuntu users can acquire this by installing `libssl-dev` through their package managers, while (as of this writing) Windows users must download an installer from a web page and manually set the necessary environment variables. Note that the version of OpenSSL in Chocolatey[20] is not the version that is recommended by the ROS2 installation guide.

[16] Encryption is a subset of security. However, as of the first release of ROS2 (Ardent), the only forms of security that work with the default middleware, Fast-RTPS, are encryption and authentication (access control is available for another middleware, Connext). As such, encryption will be the focus of this section.

[17] *Cleartext* is raw data that has not been obfuscated, encoded, encrypted, or otherwise hidden in any way.

[18] A *TCP socket* is similar to a shell pipe in that data are transferred directly and without changes. A primitive chat program can be implemented by connecting two terminals together with a TCP socket and sending keystrokes over the protocol. A simple way to do this with a pair of almost any Unix-like systems is to run `netcat` on each one and point them at each other.

[19] Such as *packet sniffers*, programs that read network traffic of all kinds. They usually dump this traffic to a file for future analysis, although some tools allow for live viewing. Sniffers will require root/superuser/Administrator privileges to run since they will capture *every* packet on the targeted network bus.

[20] *Chocolatey* is a package manager for Windows. See the Windows chapter for details.

3.3.2 Enabling security (involves partial rebuilding)

3.3.2.1 How to enable security

For Fast-RTPS, the default middleware that ships with ROS2, security can be enabled by setting the CMake flag[21] `-DSECURITY=ON` when building ROS2. This can be passed to ament as follows:

```
$ ament build --only fastrtps --cmake-args -DSECURITY=ON --
```

Other ament-related flags (*e.g.*, `--symlink-install`) must be positioned after the `build` argument and before the `--cmake-args` argument.

Once security is enabled in the DDS provider of choice (here, Fast-RTPS), one must then rebuild the corresponding `rmw_foo` package (where "foo" is replaced with the name of the middleware). One can also include this directly into the above command, thus building, *e.g.*, `--only fastrtps rmw_fastrtps`. These are the only packages that need rebuilding as empirically determined on a stock Ubuntu source installation of ROS2 beta three.

3.3.2.2 Authentication and key management

Upon enabling security, one must have the appropriate keys in a single directory ("keystore") where the middleware will know to look for them. These can be generated via the `ros2 security` subcommand (which internally uses OpenSSL). The middleware can then be informed via shell environment variables.

Note that the name of the key must match the name of the *node*, not the name of the executable. This allows clients to authenticate with completely different programs and not be aware of the difference, as can be seen by switching between the

[21] For readers unfamiliar with how CMake interacts with C macros, this acts as as though one had placed `#define SECURITY ON` in the beginning of each compilation unit. A *compilation unit* roughly corresponds to a C source file with all of the macros expanded, including `#include` and `#ifdef`.

"demo_nodes_cpp" and "demo_nodes_py" packages (which use the same node and topic names).

If one executable provides multiple nodes, then all nodes need their own keys. For example, the executable `ros1_bridge` has a single ROS2 node named `ros_bridge`, so (as of ROS2 beta three) the command to generate a key for that node would look like this:

```
$ ros2 security create_key keystore_foo ros_bridge
```

Once the keys have been generated, the following shell variables should be set to enforce usage of the keys:

- `ROS_SECURITY_ROOT_DIRECTORY=/path/to/keystore`
- `ROS_SECURITY_ENABLE=true`
- `ROS_SECURITY_STRATEGY=Enforce`

Linux users will want to `export` these values in their shell run commands files (*e.g.*, `~/.bashrc` for Bash or `~/.zshrc` for Zsh), while Windows users will want to define them as global environment variables via the Control Panel[22]. Alternatively[23], Windows users can also define the variables locally in each shell (via `set`), but that could quickly become tiresome unless one modifies one's registry to automatically run a batch file when starting the command processor. The relevant registry key is a String Value under `HKCU\Software\Microsoft\Command Processor\AutoRun` containing the absolute path to your custom definitions file[24]. This file is also a good place to put `doskey` aliases.

[22] Control Panel → System → Advanced system settings → Advanced tab → Environment Variables → either User or System (if your account is the only user, there is little practical difference) → New... → (define the variable).

[23] Less-advanced users may freely skip the rest of this paragraph.

[24] *e.g.*, `%USERPROFILE%\autorun.cmd`. Note that environment variables are allowed in registry key paths; that is, this is not a metasyntactic variable, but rather the literal text that one might use for the key.

At this point, all of the nodes for which keys exist can be run. If the system default middleware implementation does not match the middleware implementation for which security was enabled, then the `RMW_IMPLEMENTATION` environment variable must be set so as to override the system default.

3.4 Conclusion

As can be seen from the preceding discussions, the many differences between ROS1 and ROS2 that are looked at in this chapter show that developing for ROS2 will not be the same as developing for ROS1. The availability of newer languages and libraries allows for new, different, and possibly more efficient programming styles, which in turn can allow for improved legibility, maintainability, efficiency in writing. Meanwhile, the ease with which the new security and encryption features can be used to secure any communication channel could make ROS2 more suitable for production environments.

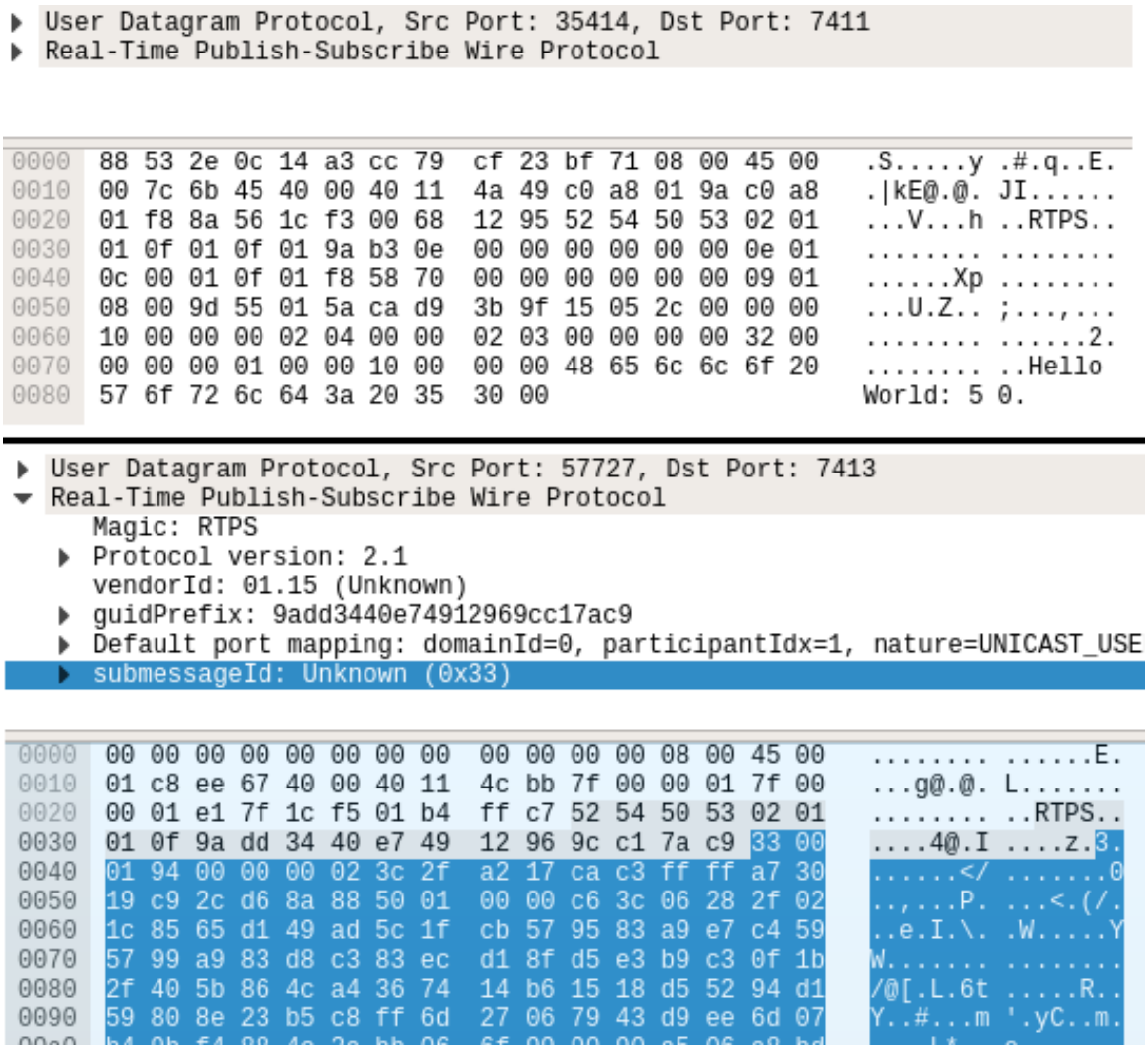

Figure 3.1: Comparison of captured packets displayed in Wireshark. Above: unsecured communication, clearly visible in plain text at the end of the packet. Below: same information, but encrypted.

Part II

Tutorials

Chapter 4

How to port an existing ROS1 C++ program to ROS2

4.1 Introduction

Suppose that, after much consideration, you decide that you want to use ROS2 for your existing system. However, your existing system has a non-trivial amount of ROS1 client code and you neither wish to throw it all away nor incur the runtime cost of bridging all of the old nodes to the new nodes and vice versa. The only remaining option at that point is to port each existing package to ROS2 and have the entire system run ROS2. This chapter contains information regarding how you can port your own packages.

The content below is based on the first and second betas of ROS2. Many of the lessons were learned empirically from porting the "learning_ros" repository[1] that accompanies this author's thesis advisor's recent book [26]. The differences against that repository can be found in this author's fork, available here:

`https://github.com/jstarkman/learning_ros2`

[1]Link: `https://github.com/wsnewman/learning_ros`

Slightly more detailed explanations can be found inlined with the code by switching to the "pedant" branch.

4.2 Ancillary porting information

4.2.1 Introduction

Porting a package to ROS2 is not terribly different from general ROS2 development. Both rely on similar knowedge and similar setup steps and only differ in that when porting a package one has an existing codebase on which to base one's new work. Accordingly, the common elements have been placed in this section while the porting-specific elements have been placed in the next section.

This section is based on empirical lessons learned when porting `learning_ros2`. The porting effort was primarily done on Linux, specifically Ubuntu 16.04 Xenial. However, in order to better accommodate the Windows users who are reading this, instructions for Windows have been approximated and are mentioned next to their Linux counterparts.

Note that not every directory in a given repository is necessarily a package, and not every package has any ROS-specific dependencies. If a given package can be reworked to remove any dependence on ROS, then it will not need porting and will work equally well with both ROS1 and ROS2. This was the case with several packages in `learning_ros2` that related to forwards and inverse kinematics.

4.2.2 Dependencies

Ensure that all necessary dependencies are available under ROS2. If this is not the case, port the dependency first and ensure that it works so as to avoid having problems when building your original package. Note that there is no limit to how many dependencies through which one may have to recurse. One might benefit from

fully discovering the entire dependency DAG[2] and checking every unique package for ROS2 availibility. Only if they are all available in ROS2 might porting be feasible.

4.2.2.1 Transitive limitations

Some dependencies are not currently available in ROS2 (as of the first ROS2 release) but are planned for future releases. These limitations are thus transitory: if they prevent one's existing package from being ported then, after a waiting period, one could revisit the state of the ROS2 ecosystem to see if the problems have been resolved. If so, then the limitation would no longer apply and porting could proceed as normal.

As of Ardent, the first ROS2 release, limitations include:

- No action servers
- No Gazebo (although `ros1_bridge` allows partial usage).
- FastRTPS performance for large payloads

4.2.3 Installing ROS2 itself

Check if your platform is supported by ROS2. In general, the latest versions of the Microsoft Windows, Apple OS, and Ubuntu Linux operating systems will likely be supported, while support for older versions is dependent on the release of ROS2. If your platform is not supported, consider using a virtual machine or container[3]. Note that an Ubuntu VM will tend to require less disk space than the Windows equivalent[4].

[2]Directed Acyclic Graph; a graph (as in graph theory, not as in plotting) where each edge has a direction associated with it and where, if one starts from any node/vertex and travels along the edges in their directions, one can never return to the node where one started.

[3]A *container* is a means of specifying the environment in which a program will be deployed. One popular containerization tool is *Docker*, and since ROS1 builds are already available as Docker containers, it would not be unreasonable to expect ROS2 equivalents to be made available eventually. One can also build one's own.

[4]A Windows installation seems to need about 30 GiB to 40 GiB of space while Ubuntu should need maybe half of that. These numbers are by no means exact and are provided for estimation purposes only.

Also note that running a Linux Docker container on a bare-metal Linux system will not be a true virtual machine but rather a special process, not unlike a chroot jail[5]. A Linux Docker container on a Linux host will run with the same kernel as normal userspace programs (*e.g.*, a web browser) use and will be visible in any system process explorer (such as `htop`[6]). When the same container runs under a not-Linux host, then the Docker engine will run it as a virtual machine.

Install ROS2 in your chosen environment, optionally doing so from source. Instructions for how to do this vary by platform; see the official instructions[7] for details. Continue all the way through running the C++ demo nodes.

If you have decided to build ROS2 from source, then you may find that the build might occasionally crash. When this happens, one possible cause is a missing system dependency. If this is indeed the case, find the missing package and install it. While this is not terribly likely for the base ROS2 installation (due to the official installation guide spelling out its dependencies) it is somewhat more likely for those who choose to install OpenCV from source. Depending on what packages have already been installed on your system, the exact list of new packages to install may vary.

[5]A *chroot jail* is a way of running a process such that it thinks that the root directory of its filesystem is not the system root (`/`) but rather somewhere else in the file system tree (*e.g.*, `/home/me/fakeRoot/`). The name is an abbreviation of "*ch*anged *root*" This directory would have the usual Unix directories inside (*e.g.*, usr, bin, var, &c.) with the subset of their usual contents that the process will need (since if a given file will never be opened then it does not need to exist). Conceptually, since the process will think of the fake root as the real one, attempts to escape the jail via the ".." file should fail because, on a regular, bare-metal Unix system, "/.." means the same thing as "/". However, processes run as root can easily escape the jail, so this is not a universal solution.

[6]`htop` is a terminal-based tool that shows various statistics about running processes, as well as overall system resource utilization. It can be thought of as a Unix-like equivalent to the Windows Task Manager.

[7]Link: `https://github.com/ros2/ros2/wiki`

4.2.4 Preparing for the port

4.2.4.1 Removing deprecated ROS1 features

Some existing ROS1 packages use deprecated features that were never made available for ROS2. As such, to minimize the difficulty of porting, existing packages should replace deprecated features with forwards-compatible features.

One such feature is the library "tf": it was deprecated around ROS1 Indigo (2014 LTS), and while it is still available in ROS1 Lunar (2017 LTS) it is *not* available in ROS2. Instead, ROS2 code must use the library "tf2", which has been ported to ROS2. Since this library is also available for ROS1, one should port one's existing "tf" code to use "tf2" before contemplating porting from ROS1 to ROS2. Alternatively, one could port "tf" itself.

Another difference is in the format of the "package.xml" file in the root of the package: while the original format of this file is still valid even under ROS1 Lunar, since at least ROS1 Groovy [43] there have been recommendations to migrate to a newer format defined in REP 140 [44]. This newer format is supported by both ROS1's catkin and ROS2's ament. Since the original format is not supported by ROS2's ament, if your existing package still uses the original format it should be updated to the newer format before the package source code itself is ported. In this way, you eliminate a possible source of bugs and confusion before they happen. A guide for migrating the file can be found at the link in this citation: [43]. This guide also exists for newer versions of ROS1; to find them, replace the word "groovy" in the URL with the desired version of ROS1 (*e.g.*, "lunar"). Note that there are no significant differences between various versions of this file. To provide a concrete example of how to implement the changes from this guide, the approach taken here was to copy the "package.xml" file from a working package into the editor for the target package's "package.xml" file (such that the one editor window contained the

contents of both files, one after another) and modify the copied file as needed, using the old contents as reference and deleting them when they were no longer needed. Working in this way allows one to avoid duplicating minor changes[8]. Note that with the newer format of this file, the <build_depend/> tag and the <exec_depend/> tag can be merged into a single <depend/> tag, further reducing duplication.

Another difference relates to the image processing library OpenCV: ROS1 uses OpenCV version 2, while ROS2 uses version 3. The two are not perfectly compatible. Guides on how to port between the two (for arbitrary C++ programs, not just ROS-related ones) are widely available on the public Internet. Note that OpenCV 3 has been available since ROS1 Indigo.

Yet another possible improvement relates to the numerical library Eigen. This difference is dependent on how one initially wrote the package. Some existing ROS1 packages (like "learning_ros") depended upon a package called "Eigen3". Since this package is not available in ROS2 (as of the second beta), one should include Eigen in the usual C++ way:

```
set(EIGEN3_INCLUDE_DIR "$ENV{EIGEN3_INCLUDE_DIR}")
if(NOT EIGEN3_INCLUDE_DIR)
message(FATAL_ERROR "Point environment variable EIGEN3_INCLUDE_DIR to the
include directory of your Eigen3 installation.")
endif()
include_directories("${EIGEN3_INCLUDE_DIR}")
```

(Reference: https://stackoverflow.com/a/12258855)

Note that on a stock installation of Ubuntu, "the include directory of your Eigen3 installation" is /usr/include/eigen3/. If Eigen3 is missing, then it can be installed via Apt[9]. Other distributions may vary.

[8] *e.g.*, including <build_type>ament_cmake</build_type> inside of the <export/> tag or making the dependencies on the build tools point to ament instead of catkin.

[9] *i.e.*, $ sudo apt install libeigen3-dev

4.2.4.2 Setup for porting

Make the source of your package (the examples will use the various packages under `learning_ros2`) available under a directory that is not under the ROS2 workspace. Create parent directories `name_of_overlay/src/` between your package and the chosen resting site. For example, if your ROS2 workspace is `~/ros2_ws/`, then you might want to check your package out from version control under your home directory[10]. If your ROS2 workspace is under `%USERPROFILE%\ros2_ws`, then there might be a folder on your Windows desktop named "ros2_overlay", a folder inside that named "src", and inside of that is where you would put your package. Once the directory has been copied (or moved), create an ament overlay pointing to your ROS2 installation. You can do this by sourcing the setup script (or calling the batch file) of the base ROS2 workspace installation and then running ament in the overlay workspace root. Since the relevant environment variables were last modified by the setup script, the workspace from which that script was run (*i.e.*, `ros2_ws`) will be used as the base layer over which this new directory is an overlay. Thus, future invocations of the build tool from the overlay workspace will only try to build the packages present in the overlay, and will not try to re-build ROS2 itself.

What if you have a package in your ROS2 workspace but do not want to build it? Maybe it is being more troublesome than expected (*e.g.*, by depending on a third-party package that has not been ported yet) and you do not want to deal with it right now. This may also occur if you have multiple packages under your version control root, as is the case for `learning_ros2`. In any case, to cause ament to ignore a set of packages, you should create an empty file named "AMENT_IGNORE" next to each package.xml file in each unwanted package. To make ament pay attention to that package again, simply delete the file. Users of Unix-like systems (*e.g.*, Linux) can create these files next to all package.xml files under the present working directory

[10] *i.e.*, `~/ros2_overlay/src/your_package/`

with a single command:

```
$ find .  -name "package.xml" -execdir touch AMENT_IGNORE \;
```

This command assumes that it is being run from your package root directory. If not, either change into that directory or replace the dot with the path to the package root directory[11]. Due to the lack of a simple equivalent in Windows, Windows users might wish to consider any of: doing it manually (best for small numbers of packages); working out the corresponding PowerShell command; installing Cygwin or WSL; or finding a Linux system (or a friend with a Linux system), running the command there, putting the new files under version control, and making them appear on the Windows system by synchronizing the repositories.

4.2.5 List of useful statements to run on a command line

Each invocation of the build tool has the potential to be much more complicated than a simple `ament build`. Here are several options, each with an explanation. Note that these are the exact commands (quoted verbatim with particular values replaced with metasyntactic equivalents) used when porting `learning_ros2`.

```
$ ament build -s
```

Meaning: The last flag means "install with `ln -s`, not `cp`"[12]. This is useful for files that are directly copied including Python source files and data files as they will always be up to date without needing to re-run the build tool.

```
$ ament build -s --only package_name
```

[11] This includes the directory itself, *e.g.*, `~/ros2_overlay/src/your_package/`

[12] On a Unix-like system, `ln -s` means "make a symbolic link" (commonly abbreviated to "symlink" and used as both a noun and a transitive verb), while `cp` means "copy". A symlink is stub of a file that points to another file in the file system. It allows the file to appear to exist in both places at once while really only existing in one place. It can also be thought of as a copy that is always up to date. As such, a symlink is very similar to a pointer in C or C++.

Meaning: This flag only builds the one package, "package_name". It is useful for debugging builds because it allows one to skip every other package in the workspace, including packages that depend on the package being built. Note that this may lead to misleading behavior in the dependent packages. Since multiple packages can be specified after the `--only` argument, if dependent packages are a concern then they should be included in the above statement.

```
$ VERBOSE=1 ament build -s --only package_name
```

Meaning: Setting this environment variable will cause the build process and its child processes to produce more verbose output. This can be an excellent tool to use when debugging a stubborn package.

4.2.6 Toolchain

This subsection lists the primary toolchain used for porting `learning_ros2`. Other toolchains would also work; this subsection is provided for completeness.

- Ubuntu 16.04 (Xenial) for an operating system installed on bare metal (*i.e.*, not in a VM). This was done purely for speed purposes.
- Git for version control.
- VS Code[13] with plugins for C/C++, Python, XML files, and CMake files. If one opens the ROS2 workspace directory instead of the project directory then one can easily see (and navigate to) built-in ROS2 classes, commands, and macros. Usefulness of this feature is why an overlay was not used for this particular porting effort.

[13]Visual Studio Code; the executable is named `code`. It is an open-source "editor with a debugger" from Microsoft. It is not related to Visual Studio itself; only the branding is shared.

- Emacs, Vi, Vim, and an sh-compatible shell (here, bash) for miscellaneous editing and file system operations.
- Apt for package management; ROS2 itself was built from source, although binary Debian releases exist and should work.

A toolchain popular with Windows users might be:

- The latest version of Windows 10.
- Any version control system with Tortoise support. This list includes Git, Mercurial, Bazaar, Subversion, and CVS. Note that regardless of which version control tool is used one still needs Git installed in order to clone other packages, including the ROS2 source code itself (if building from source).
- Visual Studio for C++ and Windows Native development. Note that regardless of which editor is used one still needs to install VS in order to compile and build code. One can also use VSCode, as on Linux.
- Notepad++ and cmd.exe (or PowerShell) for miscellaneous editing and file system operations.
- Chocolatey for package management for dependencies; ROS2 itself is not available in the default Chocolatey repository as of this writing, although some of its dependencies are only available through .nupkg files.

For `learning_ros2`, Windows work was done with all of the above. The only exceptions are the use of command-line Git instead of TortoiseGit for version control and using Notepad++ for all editing. Visual Studio was only used as a command-line build tool and occasionally as a debugger.

4.3 Porting your package

4.3.1 Introduction

The Open Source Robotics Foundation has released an official reference guide [45]. To summarize this guide, every ROS-related line of code must be changed. Also, due to the transition to C++11 dependencies on Boost can largely be removed and replaced with their equivalents from the standard C++ library.

Note that many packages are not (yet) available for ROS2, so some existing packages cannot easily be ported. Although programs like Gazebo may function via `ros1_bridge`, performance will not be improved, particularly since Fast-RTPS cannot yet efficiently handle large messages like images and pointclouds.

4.3.2 Architectural changes

4.3.2.1 Launchers and parameters

One particularly large change is the decision for launchers to move from being XML files to Python scripts. While this offers more flexibility in terms of what can be done at launch time, it also imposes a higher cost on simple launchers. A relatively simple example of launching a parameter server with several parameters from a YAML file can be found under `learning_ros2`[14]. This file also includes the original, three-line launch file in a comment, as well as several spurious imports[15] that might be of use to people who might want to modify the file in the future.

One common use case for launch files is to insert certain values into the parameter server (and that is what the example file in the above paragraph does). Since ROS2 does not have an equivalent to `roscore`, there is no standard system-wide parameter server; instead, the user must create a node that will act as one. The parameter

[14] Path: `Part_1/example_parameter_server/launch/launch_example_param_server.py`

[15] `import` is the Python equivalent to `#include` C++.

server node used by the above ROS2 launch file is about as simple of a parameter node as one could write.

4.3.2.2 Pointers

Another major change is that in ROS1 messages and service components were copied by value while in ROS2 they are shared by pointer. This reduces the overhead associated with composing nodes (as each node in the process can point to the same struct instead of each having their own copy), as well as allowing ROS2 system functions to control memory allocation in order to better support custom allocators. From the perspective of writing a client program, this amounts to making heavier usage of the C++ pointer templates in the standard library and remembering to use an arrow instead of a dot to access the fields in the struct.

4.3.2.3 CMakeLists.txt

Yet another large difference in ROS2 is that `CMakeLists.txt` has to be largely rewritten since, as of the second beta release of ROS2, `catkin_simple` is no longer available and there does not yet exist an "ament simple" to replace it[16]. Fortunately, most of this file is boilerplate and can be copied from an existing working project with minimal changes. If one copies the file that is associated with the example parameter server in Part_1 of `learning_ros2`, then the following aspects of the file would need to be changed. They are itemized in order of the line number on which they appear.

- The name of the project should match your package name.
- One should call `find_package()` for each build dependency in package.xml.
- The `custom_executable` function definition should list all of these build dependencies. Alternatively, the body of the function could be copied once for each

[16]Packages that do not use `catkin_simple` should thus be much simpler to port as they would already have much of the below.

executable. This would allow one to only make each executable depend on the appropriate packages and nothing more.

Other changes require a better understanding of CMake. They are summarized below.

- Calls to `include_directories()` should be added if the old package uses such a directory. This is perhaps most prominently used for Eigen, which exists entirely in header files. This is also used for including the headers used by one's own package.
- If the package uses its own library then a call to `add_library()` should be added since the executables will not be automatically linked.

4.3.3 Programming changes

4.3.3.1 Message and service files

This subsubsection concerns the text files under `msg/` and `srv/` in the package base directory. They are largely unchanged in ROS2. If they are used by one's package, note that all fields must be named in "snake_case"[17]; as of the second beta of ROS2, the presence of any capital letters anywhere in any field name will crash the build tool.

Projects that include both a library[18] and custom messages will find that the building process will be simplified by moving the messages to their own package. This will have the side-effect of making the messages more re-usable by other packages. All of these changes are back-portable to ROS1 code. However, they have not (as of Lunar) been deprecated by ROS1, which is why this information appears here instead of in the above subsubsection about removing deprecated features.

[17] *i.e.*, with all lowercase letters and words separate with underscores. This is in contrast to "camelCase", which uses initial capital letters to separate one word from the next.

[18] *i.e.*, that produce their own `libpackage_name.so` or `package_name.dll` file.

Packages `baxter/baxter_core_msgs` and `object_manipulation_*` in Part 5 of `learning_ros2` can be seen for more concrete examples.

4.3.3.2 C++ files

This subsubsection will detail various ROS2-related C++ code changes. It will procede in the order that one might encounter the changes in a typical small C++ program of the type that is prevalent in `learning_ros2`.

Formatting: All ROS2 code should be formatted according to the ROS2 Developer Guide[19] One way to automatically comply with these guidelines is to install and configure `clang-format`[20] to use a configuration file. One can then configure one's editor or IDE to automatically run this tool upon saving the file. This feature is variously called "Run on Save" (VS Code), "Save Action" (Eclipse), and "before-save-hook" (Emacs). The config file that was used for porting `learning_ros2` came from a Linux binary installation of ROS2. Visual Studio Code was configured to run it automatically upon saving any C++ files. This practice is recommended to avoid inconsistencies. It is also not inconceivable that this will be required of any official ROS2 packages.

#include: In ROS1, header files for code generated from "msg" and "srv" files resided in the same namespace as the package itself. For example, in the case of the built-in "geometry_msgs" package, the "PoseStamped.msg" file would produce a header file named "geometry_msgs/PoseStamped.h". If the package had a file under "src" that was also named "PoseStamped", then there would be a conflict. By way of contrast, in ROS2 not only is the generated file nested under an intermediate "msg" directory (to avoid the aforementioned namespace conflicts with user-defined

[19]Link: `https://github.com/ros2/ros2/wiki/Developer-Guide`.
[20]Ubuntu users may run `sudo apt install clang-format` to install this tool.

code), the file name is also converted to snake_case, a format that is generally used by client code for file names (at least within the ROS2 codebase)[21]. For example, in the case of the built-in "geometry_msgs" package, the "PoseStamped.msg" file would produce a header file named "geometry_msgs/msg/pose_stamped.hpp" that would contain a class called `geometry_msgs::msg::PoseStamped`. Similarly, for services in ROS2 both the request and the response objects (which are now separate entities, again to allow ROS2 code to control allocation) are included via the same header file, thus maintaining the one-to-one mapping between definition files and resulting header files.

ROS-specific objects: In ROS1, ROS-specific objects (*e.g.*, Nodes, Publishers, and Services) are generally manipulated directly. One calls the constructors oneself, chooses where and how to store the objects (or pointers to them), and generally treats them like any other C++ objects. In contrast, in ROS2, one generally does not instantiate library objects directly, but rather through smart pointers[22]. As of the second beta of ROS2, the predominant form of smart pointer in client code is the *shared pointer*. Shared pointers are implemented with reference counting[23] and are used to help avoid memory leaks and lessen other difficulties sometimes experienced with pointers. Most library objects are returned wrapped in a smart pointer, including Nodes, Publishers, Subscribers, Services, and more.

ROS-specific objects: Publishers, Subscribers, and messages: In ROS1, it was common to instantiate messages directly. For each publication, the message con-

[21] In general, class names in C++ are written in CamelCase like in Java, Python, and other similar languages, and file names match the classes that they contain (but may be written in snake_case). This can be seen in, *e.g.*, the standard ROS2 classes, which follow this pattern.

[22] Specifically, by calling `std::make_shared` and by receiving them from API calls.

[23] If one uses *reference counting* to track the life cycle of a pointer, then a counter inside the struct that contains the pointer is incremented whenever a new reference to the contents is made. This counter is then decremented when each reference is dropped. When it reaches zero, the object is destroyed.

tents would be copied by ROS1 before being available to other nodes. Contrarily, in ROS2, instantiation of messages is handled through smart pointers like other objects in ROS2 are, although (as of the second beta of ROS2) the API still accepts message objects that were allocated on the stack. Despite this, subscriber callbacks must take a smart pointer. For example:

```
$ void sub_cb(const std_msgs::msg::Float32::SharedPtr msg) { ...}
```

(Note that as of Ardent, the initial ROS2 release, this appears to have changed to a UniquePtr, which is another kind of smart pointer.) Using smart pointers automatically allocates memory and may result in less copying than was the case under ROS1. This also allows one to use custom allocators instead of `malloc()`[24], or to avoid allocations at all and thus be more suitable for real-time code.

ROS-specific objects: Services: In ROS1 Services, the request and the response were two parts of the same data structure. In ROS2, they are not. Instead, each is a separate class nested under the general namespace that comes from the service definition file. Furthermore, callback methods no longer return anything; instead, they fill out the provided `ServiceName::Response` object with the data to be sent back to the client. For example, here is what that callback would look like for a service that used Triggers from the standard services package:

```
    void serviceCallback(
const std::shared_ptr<rmw_request_id_t> request_header,
const std::shared_ptr<std_srvs::srv::Trigger::Request> request,
std::shared_ptr<std_srvs::srv::Trigger::Response> response) { ...}
```

The first argument can usually be ignored. The second argument points to the Request issued by the client. The ROS2 library will handle instantiating the object

[24]For typical C programs, `malloc()` is the function that is used to allocate memory. It is implemented by standard C libraries. On a Linux system, the specific implementation that is used is probably the GNU C Library.

and ensuring that the correct data have been loaded into it (if applicable; Trigger messages have no request component so there would not be any fields in this case). The third argument points to an empty Response object that the body of the function will set.

ROS-specific objects: Action servers: In ROS1, action servers are a special kind of server that can track a goal and report when this goal has been reached. Unfortunately, as of Ardent, this feature is not currently available in ROS2. As such, code relying on action servers is currently unportable, although there are plans for action server availability in the future.

4.3.4 Other changes

4.3.4.1 Documentation

A newly-created ROS1 package starts out with a largely-empty file in its base directory named "README.md". Over the life of the package, this file likely received multiple updates, some of which may pertain to implementation details. If this is the case, then this file will need to be updated to reflect any changes made as a result of switching to ROS2. At the very least, one should mention that the package has been ported so that systems that depend on it will know that they can now be ported, too.

4.4 Packages ported from `learning_ros2`

For completeness, here is the full list of `learning_ros2` packages that were ported for this chapter:

- Part_1/creating_a_ros_library
- Part_1/custom_msgs

- Part_1/example_parameter_server
- Part_1/example_ros_class
- Part_1/example_ros_msg
- Part_1/example_ros_service
- Part_1/minimal_nodes
- Part_1/minimal_nodes_py
- Part_1/using_a_ros_library
- Part_2/example_eigen
- Part_2/example_rviz_marker
- Part_2/example_tf_listener
- Part_2/lidar_alarm
- Part_2/sine_commander
- Part_2/stdr_control
- Part_2/xform_utils
- Part_4/localization_w_gps
- Part_4/mobot_drifty_odom
- Part_4/mobot_nl_steering
- Part_4/mobot_pub_des_state
- Part_4/odom_tf
- Part_4/traj_builder

- Part_5/arm7dof/arm7dof_fk_ik
- Part_5/arm7dof/arm7dof_nac_controller
- Part_5/arm7dof/nested_loop_control
- Part_5/baxter/baxter_core_msgs
- Part_5/baxter/baxter_fk_ik
- Part_5/baxter/baxter_head_pan
- Part_5/baxter/simple_baxter_gripper_interface
- Part_5/example_controllers
- Part_5/generic_gripper_services
- Part_5/joint_space_planner
- Part_5/object_manipulation_msgs
- Part_5/object_manipulation_properties

Chapter 5

Miscellaneous addenda

5.1 Introduction

This chapter groups together sections that are closely associated with a particular chapter in Part I. However, since the content of Part I largely pertains to experiments and their results, the more tutorial-like nature of these sections would be out of place if they were to be interleaved with their corresponding chapters. As such, like the other "miscellaneous" chapter in Part I, the sections below are in no particular order and can be thought about independently of each other.

5.2 How to debug ROS2 programs with Visual Studio

5.2.1 Introduction

Print statements are not necessarily the best debugging tool for every situation. Sometimes, one wants to be able to insert breakpoints and pause the program to inspect the state of the stack at the time that they are reached. While one could use a command-line debugger such as `gdb` (GNU Debugger), some IDEs offer a more inte-

grated approach. In the case of Microsoft Visual Studio debugging a ROS2 program, one does not simply open a `.cpp` file and click the relevant line number to set a breakpoint. Rather, there are additional steps that one must follow in order for Visual Studio to build the project and be able to understand the breakpoints. The rest of this section details those steps and their results. These were tested with the first beta release of ROS2, but should work equally well for future releases.

5.2.2 Step-by-step instructions

See figure 5.1 for the final result.

1. Run as administrator a VS native tools x64 prompt. In it, enter the following to run a debug build with ament. Note that this might take a long time to finish since it has to rebuild every package in the workspace. This can be alleviated with either overlay workspaces (which function as under ROS1) or with the `--only package_name` argument to ament. The last line opens the development environment, Visual Studio (`devenv.exe` is the name of the VS executable). This should be run from the prompt in order to inherit all of the environment variables.

```
admin> cd ...\ros2_ws
admin> call install\setup.bat
admin> ament build --cmake-args -DCMAKE_BUILD_TYPE=Debug
admin> devenv
```

2. In the resulting Visual Studio window, open an existing solution[1] from your package's build directory[2]. This solution file is generated by ament from calling

[1] Shortcut: Ctrl+Shift+O

[2] Location: `...\ros2_ws\build\your_package\your_package.sln` where "your_package" is the name of your package

CMake, which it turn reads from CMakeLists.txt. The Solution Explorer should have a list of your executables as defined by your CMakeLists.txt file.

3. Ensure that the configuration dropdowns at top of the Visual Studio window read "Debug" and "x64" (assuming that the host computer runs 64-bit Windows).

4. Optionally, edit the source files as needed. Note that, despite the solution file being located in the build directory, edits will be made to the original source files in the "src" directory; `ros2_ws\build` does not have copies of the source, only pointers.

5. Set breakpoints[3] where desired. This even works inside lambda expressions, as can be seen in figure 5.1.

6. In Solution Explorer, right-click on the executable that you would like to debug, and in the resulting context menu click "Debug" → "Start new instance". Note that pressing the usual Run button [4] will (by default) run the first entry in this list, which is probably "ALL_BUILD"; while this can be configued, if one switches executables often and rarely restarts the program then the configuration may not be worth the time. Note also that restarting[5] while the program is running will do the same thing.

7. If a popup asks to rebuild anything because something is out of date, say yes. This most commonly happens when code has changed in the Visual Studio editor.

8. The program should now run in Visual Studio debug mode. You may wish to run other programs to trigger callbacks, parse publications, or otherwise

[3]Shortcut: F9. Alternatively, click the line number in the margin.
[4]Shortcut: F5
[5]Shortcut: Ctrl+Shift+F5

interact with the process that is being debugged. For example, figure 5.1 uses "minimal_publisher" to help debug "minimal_subscriber_lambda".

(Reference: [46].)

5.2.3 Explanation of Visual Studio debugging screenshot

Figure 5.1 shows a simple ROS2 program being debugged. It has been littered with print statements to help clarify what has been executed so far. Current execution is paused on an invocation of a Subscriber callback; earlier breakpoints have already been caught and resumed from. The output of this program appears in the separate window on the right. This window was moved to this location for screenshot purposes only; normally, one would want it to be somewhere else so as to avoid losing sight of it when clicking back in to Visual Studio. Each of the red annotations can be explained as follows:

- The two connected regions at the top are to show that this is all the same program, "minimal_subscriber_lambda".
- The center-height line on the left shows the active breakpoint on which the program is paused.
- The big arrow in the middle shows that the line on which the active breakpoint is stopped has not been executed yet (since the associated text is missing from the output).
- The box in the lower-left shows the current local variables. Note that only "msg" (the argument of the callback) is visible since "node" and "sub" were not captured by the lambda. If one wanted to reference "node", then one would have to enclose the name in the square brackets that start the lambda expression.

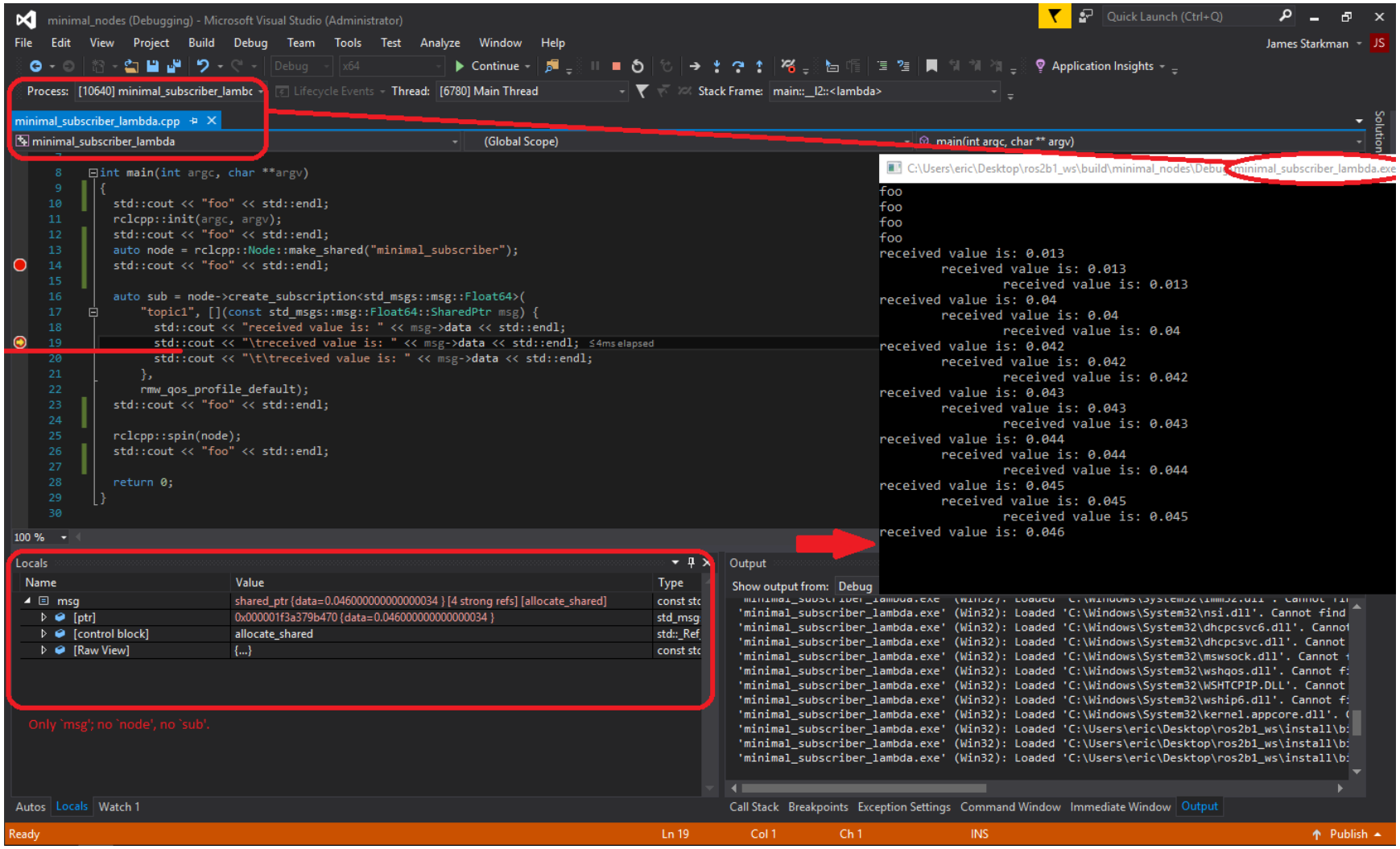

Figure 5.1: Debugging the minimal subscriber. See subsection 5.2.3 for an explanation of what the red annotations mean.

5.3 How to run ROS1 or ROS2 on C.H.I.P.

Steps that should run even when you log out (*e.g.*, build commands) should be run with the following command:

```
nohup long_running_command &
```

This will prevent the shell from killing the long-running command. It is especially useful for steps that will likely need to run overnight.

1. Decide on whether to use ROS1 or ROS2.

2. Decide on whether to use Debian Jessie or Stretch. Note that a pre-built ROS2 workspace tarball is only available for Stretch.

3. Decide on whether to build your own workspaces or to use the workspace tarballs mentioned in subsubsection 1.2.1.2.

4. Acquire as many C.H.I.P. computers as you wish to use. The following steps will need to be repeated for each one, although each C.H.I.P. can be done in parallel to the others. Note that the experiments and analyses presented earlier only used a single C.H.I.P..

5. Go to `http://flash.getchip.com/` from a Chromium-family browser and flash the 4.4 headless image by following the instructions on the page. Confirm that the flash worked by logging in to C.H.I.P..

6. If the usage of Debian Stretch is desired, perform a distribution upgrade by following the instructions at this source: [47]. Do not disable the Jessie repository that contains the packages listed at this source: [48]. This will likely need to run overnight. This would be a good place to use `nohup`.

7. If you decided to install ROSn from a pre-built tarball, go to `https://github.com/jstarkman/jasadc/releases` and follow the instructions on that page.

This will download the file (`wget`) and extract it (`tar`) into the appropriate directory (*e.g.*, `~/ros_ws/`).

8. If you decided to install ROSn by building it from source, follow either `http://wiki.ros.org/kinetic/Installation/Debian` for ROS1 or `https://github.com/ros2/ros2/wiki/Linux-Development-Setup` for ROS2.

    - ROS1 users are recommended to install the "bare bones" package set due to C.H.I.P.'s limited storage capacity and the lack of display-related software in the headless image. If one were to flash a non-headless image instead, then the C.H.I.P. would still have poor graphics, which here means a 640×480 analog signal sent through a TRRS (Tip-Ring-Ring-Sleeve) connector. One can also purchase a "D.I.P." to extend the video capabilities of C.H.I.P., but if one intends to use C.H.I.P. as a simple signal sampling system (as explored in this thesis) then such a purchase would not provide any benefit.
    - ROS2 users are recommended to modify the list of packages to be installed to change `libpocofoundation9v5` to `libpocofoundation9` in the relevant call to `apt install`. Alternatively, wait for the "v5" package to become available for Stretch, or port it yourself. Instructions for doing that are beyond the scope of this thesis.
    - Note that each call to `catkin` or `ament` can take a while to complete. For the major build steps of each ROSn, this will likely need to run overnight. This would be a good place to use `nohup`.

9. Reboot C.H.I.P. and fix the WiFi if needed and desired.

10. Run a test program for your chosen version of ROS.

Concluding thoughts for tutorials

Porting an existing ROS1 package to ROS2 can involve varying levels of difficulty. Fortunately, the most difficult aspects are either due to a simple lack of functionality in ROS2 (in which case a port should not be attempted) or due to reliance on deprecated features that were dropped in the switch to ROS2 (in which case the deprecated dependencies can be eliminated in ROS1 before attempting the port). Assuming that neither of these is an issue, there still remains the issue that every line of code that touches the ROS1 API will need to be modified, though usually in predictable ways. While the official guide is a good outline, it can be rather short on details; as such, this chapter and the associated repository (`learning_ros2`) may be helpful. Assuming that these issues are accepted, the set of changes that one would need to make can broadly be broken down into two main categories: architectural and programmatical.

Architectural changes impact launchers, parameters, and (for C++ code) the usage of pointers and "CMakeLists.txt". Specifically, as of the second beta of ROS2 one can no longer specify the programs that one wants to run in an XML file; rather, one must write a Python file that spawns the requested processes. While in ROS1 one could store parameters in `roscore`, with ROS2 one must manage one's own parameter server. For C++ code, no longer does one instantiate ROS-specific objects oneself; instead, one creates them behind smart pointers, a new feature in C++11. One can also specify a custom memory allocator, which may be a more valuable feature for embedded and real-time applications. Also for C++ code, the structure of the

"CMakeLists.txt" file is drastically different from what one might have used with `catkin_simple`, and might require one to gain familiarity with the CMake language.

Programming changes include how one uses messages and service objects, as well as specific conventions associated with C++ files and the ROS API accesses inside them. One should also update the documentation associated with the package to reflect its new internals and interfaces. While the files that define messages and services may not need much modification[6], the way that one accesses them from C++ code has changed and now does not necessarily involve allocating them oneself. Most parts of the ROS2 API reflect this, using smart pointers (new in C++11) to help hide the details of memory allocation and object lifecycle management. Overall, the net line count of a given package may not change significantly.

In summary, porting an existing ROS1 package to ROS2 is not likely to be a terribly difficult endeavor for a competent programmer, although some of the work will be tedious. The porting process will be simplified by dropping dependencies on deprecated ROS1 features like the "tf" library and the original format of the "package.xml" file. Aside from removing these unsupported features, most of the work involved in the port will not require major re-engineering of the original application.

[6]Any changes needed to these files can also be made under ROS1, thus happening before the port and being less of an issue within the port.

Conclusion

Since first being made available, usage of ROS has grown considerably, and with it the number of potential applications and use cases. Unfortunately, early decisions continue to apply and over time begin to show their age. These decisions include high processing power requirements, limited official platform support, and no security features. Rather than addressing these limitations within the existing project, OSRF decided to break backwards-compatibility and start a new project, ROS2, which aims to address some of these limitations. However, the project is missing important functionality and sometimes performs worse than ROS1; as such, it cannot quite replace its predecessor yet.

One of the claims of ROS2 is that it works better on smaller target computers than the "workstation-class" machines that ROS1 uses. This was evaluated against a "tiny target" (specifically, the US$9 C.H.I.P. computer) and found not to be the case for a full installation of ROS2. Not only did ROS1 have two to eight times the publication rate of ROS2 in the experiments, ROS1 performance was also substantially more predictable. While it may remain theoretically possible that ROS2 could operate on sensor-level hardware (*e.g.*, devices costing less than US$1), there has not been much exploration on that front. Finally, given how well ROS1 runs on cheap modern hardware, the benefits for ROS2 for low-volume production may be marginal.

Another argued benefit of ROS2 is its support for Windows. This is true, as ROS2 works about as well on Windows as on Linux. However, since Microsoft released the Windows Subsystem for Linux (WSL) in 2016, ROS1 is quite usable on Windows as long as one does not need hardware-accelerated graphics for, *e.g.*, Gazebo. If one does need Gazebo, then for neither ROSn will Windows suffice, since WSL cannot access system GPUs and ROS2 does not yet support Gazebo on any platform.

The second part of this thesis explored what kind of effort porting a package from ROS1 to ROS2 would entail, backed empirically by this author porting the repository associated with [26]. Missing functionality and dependencies on deprecated ROS1

functionality (which can be eliminated without switching to ROS2) will both have significant impact for many applications. However, after those are overcome, the remaining effort will be somewhat tedious but not particularly challenging.

Overall, ROS2 has not yet achieved significant benefit over ROS1. Furthermore, functionality is incomplete, thus limiting the scope of ROS2 applications. Performance is no better — and in the tiny target that was studied, worse. Porting is a significant effort. The promised benefit of running on sensor-level hardware has yet to be demonstrated, and with the cost of hardware that can run ROS1 falling significantly, the case for ROS2 seems very limited at this time.

Part III

Appendices

Appendix A

Abbreviations

Most of the domain-specific proper nouns here should be defined when they are first used. However, for both reference and for the sake of those who skip around when reading, all abbreviations are also collected here. This list also includes other computer-related abbreviations with which the reader may be unfamiliar. Metric units (such as "MHz" for megahertz) are not included.

A.1 Table of abbreviations

Note that "Abbrev." means "Abbreviation" and "PN" means "proper noun". The entry "N/A" under the *Proper Noun* column means that a given abbreviation is never expanded and that the words in the *Meaning* column are of historical interest only. This is the case for, *e.g.*, file extensions, directory names, words that are not really abbreviations (but are stylized as such), and words whose origins are abbreviations but who are not usually thought of as abbreviations, such as "laser", "scuba", and "Emacs" (which is still a proper noun).

The table is sorted in case-insensitive ASCII order, meaning that file extensions come first, followed by directory names, followed by more typical initialisms and acronyms.

Abbrev.	Meaning	PN?
&c.	*et cetera*; "and the rest"	N/A
.bag	ROS1 bag file	N/A
.ipynb	IPython Notebook file	N/A
/bin	Binary files (Unix)	N/A
/dev	Device files (Unix)	N/A
/etc	*et cetera*; (modern) Edit To Configure	N/A
/lib	Library files (Unix)	N/A
/tmp	Temporary files (Unix)	N/A
ADC	Analogue-to-Digital Converter	No
AND	"AND" logic gate	Yes
ANSI	American National Standards Institute	Yes
API	Application Programming Interface	No
ARM	Advanced RISC Machines	Yes
ASCII	American Std. Code for Info. Interchange	Yes
AVR	Nothing; processor from Atmel	N/A
BDFL	Benevolent Dictator For Life	Yes
C++	Nothing; programming language	N/A
CCC	Credit-Card Computer (*e.g.*, C.H.I.P.)	No
CHIP	Nothing; a CCC from NTC	N/A
chroot	Change root	N/A
Co.	Company	No
CPU	Central Processing Unit	Yes
CVS	Concurrent Versions System	Yes
CWRU	Case Western Reserve University	Yes
DAG	Directed Acyclic Graph	No

Abbrev.	Meaning	PN?
DC	Direct Current	No
DDS	Data Distribution Service	Yes
e.g.	*exempli gratia*; "example given"	N/A
ECMA	European Computer Manufacturer's Association	Yes
EECS	Electrical Engineering and Computer Science	Yes
Emacs	Editor MACroS	N/A
EOL	End-Of-Life	No
EP	Enhancement Proposal	No
ES	ECMAScript (similar to JS)	Yes
ext4	Fourth Extended FS (for Linux)	N/A
FFT	Fast discrete Fourier Transform	Yes
FS	File System	No
GCC	GNU C Compiler (also written "gcc")	Yes
gdb	GNU DeBugger	N/A
glibc	GNU C Library	Yes
GNU	GNU's Not Unix	Yes
GPU	Graphics Processing Unit	No
GUI	Graphical User Interface	No
HKCU	HKEY_CURRENT_USER (registry hive in Windows)	N/A
HP	Hewlett-Packard	Yes
i.e.	*id est*; "that is"	N/A
I2C	Inter-Integrated Circuit (type of comm. bus)	N/A
IDE	Integrated Development Environment	No
IEC	Int'l Electrotechnical Commission	Yes
Inc.	Incorporated	No

Abbrev.	Meaning	PN?
Info.	Information	No
Int'l	International	No
IP	Internet Protocol	Yes
ISO	Int'l Organization for Standardization	Yes
JAS	James Starkman (me)	Yes
JS	JavaScript	Yes
lib	Library	No
LTS	Long-Term Support	No
MAC	Media Access Control (Address)	No
MATLAB	MATrix LABoratory ®	Yes
MCU	MicroController Unit	No
MS	MicroSoft	Yes
MSI	MS Installer (file format)	No
NAND	Not AND logic gate	No
NTC	Next Thing Co.	Yes
NTP	Network Time Protocol	Yes
OMG	Object Management Group	Yes
OS	Operating System	No
OSRF	Open-Source Robotics Foundation	Yes
PCL	Point-Cloud Library	Yes
PCRE	Perl-Compatible Regular Expressions	Yes
PEP	Python EP	Yes
PID	Process IDentifier	No
Qt	Nothing; graphical framework	N/A
RAM	Random Access Memory	No

Abbrev.	Meaning	PN?
rc	Run Commands; config. file (as suffix)	Yes
RMW	ROS MiddleWare	Yes
ROS	Robot OS (not really an OS)	Yes
ROS1	Robot OS, first major version	Yes
ROS2	Robot OS, second major version	Yes
ROSn	Robot OS, either/both major version(s)	Yes
ROSCON	ROS Convention	Yes
rqt	ROS Qt	Yes
REP	ROS EP	Yes
RTI	Real-Time Innovations (company name)	Yes
RTOS	Real-Time OS	No
RTPS	Real-Time Publish-Subscribe	Yes
RViz	ROS Visualization tool	Yes
SBCL	Steel Bank Common Lisp	Yes
SoC	System-on-a-Chip	No
SROS	Secure ROS	Yes
SSD	Solid-State Drive	No
SSH	Secure SHell	Yes
SSL	Secure Sockets Layer	Yes
Std.	Standard	No
TCP	Transmission Control Protocol	Yes
TCP/IP	TCP over IP	Yes
TRRS	Tip-Ring-Ring-Sleeve	No
UBIFS	Unsorted Block Image FS	Yes
UDP	User Datagram Protocol	Yes

Abbrev.	Meaning	PN?
URI	Uniform Resource Identifier	Yes
URL	Uniform Resource Locator	Yes
UTF	Unicode Transformation Format	Yes
VM	Virtual Machine	No
VS	(MS) Visual Studio (IDE)	Yes
WiFi	Nothing; wireless networking system	N/A
Win32	Traditional Windows API	No
WSL	Windows Subsystem for Linux	Yes
XML	eXtensible Markup Language	Yes
YA	Yet Another	No
YAML	YA Markup Language	Yes

Appendix B

How to run ROS1 and ROS2 on C.H.I.P.

The work in the Tiny targets chapter was dependent on running both versions of ROSn on C.H.I.P.. This appendix documents how that was accomplished.

Both versions of ROSn (specifically, ROS1 Kinetic and ROS2 beta three) were built from source on C.H.I.P. without any major difficulties beyond multi-hour build times (see below for details and minor difficulties). As such, if one wishes to replicate these builds, one may wish to power C.H.I.P. via a wall socket or a desktop computer instead of via a laptop, as the laptop might cut off power to external devices when it goes to sleep[1]. To keep the build running when one disconnects from the shell session that was used to run the build command, one can use `nohup`[2] along with an ampersand (&) to run the process in the background[3]:

```
$ nohup long_running_command &

$ echo $! > meaningful-name.pid
```

[1] This lesson was learned empirically.

[2] Abbreviation for "do not hang up", *i.e.*, do not kill the process when the shell disconnects.

[3] "Running a command in the background" is analogous to clicking out of a window, thus freeing the shell to allow other commands to be issued instead of being stuck waiting for the command that was run in the background to finish.

The second command above will write the PID[4] of the long-running command to a file for future reference. Alternatively, one can watch for when the process has completed by using a process monitor such as `htop`.

B.1 ROS1

C.H.I.P. runs Debian, so one should follow the build instructions for Debian on the ROS wiki [49]. The "ROS-Comm (Bare Bones)" installation was used for these experiments. The CPU on C.H.I.P. only has a single core, so when building ROS one should use `-j1` instead of the default eight. This CPU uses the armv7lhf architecture, although that is not relevant unless one cross-compiles. Cross-compilation is beyond the scope of this thesis.

B.2 ROS2

B.2.1 Debian Jessie (8) versus Debian Stretch (9)

As of this writing, the latest published pre-built headless[5] image for C.H.I.P. that is available from the manufacturer[6] is based on Debian 8 (Jessie), which uses version 4.4 of the Linux kernel. Since Jessie is rather old and does not have the latest version of glibc[7], before ROS2 was installed the C.H.I.P. that was used for the experiments below was upgraded to Debian 9 (Stretch) via the usual Debian `dist-upgrade` mechanic [47]. After this finished and C.H.I.P. was rebooted, WiFi

[4] A *process identifier* (PID) is a not-inherently-meaningful (usually sequential) number used by other processes on the operating system to uniquely identify a process.

[5] A *headless* system has no graphical components and is commonly used for servers.

[6] `http://flash.getchip.com/` The image is called "Headless 4.4". Their image flashing tool requires a Chromium-family browser (such as Chromium, Google Chrome, Opera version 14 or later, or Vivaldi) to use, although one can also download the image directly and flash it via the command line (requires Ubuntu; an Ubuntu VM should suffice for those using other operating systems).

[7] The GNU C Library.

was found not to work. WiFi was restored by appending the following two lines to `/etc/NetworkManager/NetworkManager.conf`:

```
[device]
wifi.scan-rand-mac-address=no
```

and then restarting NetworkManager by running:

```
$ sudo service NetworkManager restart
```

After the service came back up (replace "restart" with "status" in the above command to check), Internet access on C.H.I.P. worked as well as it did before the upgrade. Accordingly, all of the experiments in this thesis that involve ROS2 on C.H.I.P. use Stretch.

B.2.2 Dependencies

C.H.I.P. runs Debian. ROS2 beta three is not supported on Debian. However, ROS2 beta three is supported on Ubuntu 16.04, and Ubuntu is derived from Debian, meaning that there is a chance of the build not being overly complicated. As it turns out, ROS2 beta three depends on a package called `libpocofoundation9v5`, as well as a related one called `libpocofoundation9v5-dbg`. This package can be found in the Ubuntu Universe repository [50], but not in the Debian default repositories [48]. However, a slightly older version of this package (`libpocofoundation9`) *is* available for Debian, and as it turns out that package suffices for the build to proceed successfully.

B.2.3 Cross-compiling

If one does not want to wait for the build to finish on C.H.I.P., one might wish to consider cross-compiling for C.H.I.P. on a more powerful machine. One of the developers of ROS and ROS2 has published a build system that uses a Docker[8] instance to

[8] *Docker* is a containerization tool, which means that it can run processes as though they were running under particular environments (operating system, installed packages, environment variables,

host ament, the ROS2 build tool [51]. The developer uses this to build ROS2 for the Raspberry Pi. However, when the exact directions on the cited page were followed, working ROS2 binaries were not created. Instead, the Docker engine indicated that the file system produced by the script on the cited page was unacceptable. Various attempts at tweaking the cited source failed to change matters.

file system contents, &c.), similarly to what a virtual machine would do. However, since the processes use the same kernel as the host system, a container is much more lightweight that a virtual machine, at the cost of less isolation. A Docker "container" is a running instance. An "image" is a static blob on the hard drive. The "engine" is the program that manages containers.

Appendix C

Patch for `jasadc` from ROS1 to ROS2

```
From 8b85326b3851e5a73e2c321b49777beed3fbb0dc Mon Sep 17 00:00:00 2001
From: James Starkman <-1@case.edu>
Date: Mon, 30 Oct 2017 09:40:48 -0400
Subject: [PATCH] Updated to ROS2.

---
 CMakeLists.txt  | 50 ++++++++++++++++-------------------------------
 package.xml     | 20 ++++++++++++++-------
 src/main.cpp    | 30 ++++++++++++++++++++++++++++
 3 files changed, 38 insertions(+), 62 deletions(-)

diff --git a/CMakeLists.txt b/CMakeLists.txt
index e5f765c..0a560ab 100644
--- a/CMakeLists.txt
+++ b/CMakeLists.txt
@@ -1,42 +1,20 @@
-cmake_minimum_required(VERSION 2.8.3)
+cmake_minimum_required(VERSION 3.5)
```

```
 project(jasadc)

-find_package(catkin REQUIRED COMPONENTS
-  roscpp
-  std_msgs
-)
+find_package(ament_cmake REQUIRED)
+find_package(rclcpp REQUIRED)
+find_package(rmw REQUIRED)
+find_package(std_msgs REQUIRED)

-include_directories(${catkin_INCLUDE_DIRS})
+set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14 -Wall -Wextra -Wpedantic")

-catkin_package(
-#  INCLUDE_DIRS include
-#  LIBRARIES jasadc
-#  CATKIN_DEPENDS roscpp std_msgs
```

```
-#  DEPENDS system_lib
-)
+set(target main)

-add_executable(${PROJECT_NAME}_node src/main.cpp)
+add_executable(${target} src/${target}.cpp)
+ament_target_dependencies(${target}
+  "rclcpp"
+  "std_msgs")
+install(TARGETS ${target}
+  DESTINATION lib/${PROJECT_NAME})

-set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME main PREFIX "")
-
-target_link_libraries(${PROJECT_NAME}_node ${catkin_LIBRARIES})
-
-install(TARGETS ${PROJECT_NAME}_node
-  ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
```

```
-  LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
-  RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
-)
-
-## Mark cpp header files for installation
-# install(DIRECTORY include/${PROJECT_NAME}/
-#    DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
-#    FILES_MATCHING PATTERN "*.h"
-#    PATTERN ".svn" EXCLUDE
-# )
-
-## Mark other files for installation (e.g. launch and bag files, etc.)
-# install(FILES
-#    # myfile1
-#    # myfile2
-#    DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
-# )
+ament_package()
```

```
diff --git a/package.xml b/package.xml
index 5cb3261..f19452b 100644
--- a/package.xml
+++ b/package.xml
@@ -1,25 +1,21 @@
 <?xml version="1.0"?>
-<package>
+<?xml-model href="http://download.ros.org/schema/package_format2.xsd" schematypens="http://www.w3.org/20
+<package format="2">
   <name>jasadc</name>
-  <version>0.0.1</version>
+  <version>1.0.0</version>
   <description>The jasadc package</description>
-
   <maintainer email="-1@case.edu">JAS</maintainer>
-
   <license>GPLv3</license>
-
```

```
    <url type="website">https://github.com/jstarkman/jasadc</url>
-
    <author email="-1@case.edu">JAS</author>

-   <buildtool_depend>catkin</buildtool_depend>
-
-   <build_depend>roscpp</build_depend>
-   <build_depend>std_msgs</build_depend>
+   <buildtool_depend>ament_cmake</buildtool_depend>

-   <run_depend>roscpp</run_depend>
-   <run_depend>std_msgs</run_depend>
+   <depend>rclcpp</depend>
+   <depend>rmw_implementation</depend>
+   <depend>std_msgs</depend>

    <export>
+     <build_type>ament_cmake</build_type>
```

```
    </export>
  </package>
diff --git a/src/main.cpp b/src/main.cpp
index 3f9f399..11c6b7e 100644
--- a/src/main.cpp
+++ b/src/main.cpp
@@ -10,8 +10,8 @@
  #include <linux/i2c-dev.h>


-#include <ros/ros.h>
-#include <std_msgs/Int32.h>
+#include "rclcpp/rclcpp.hpp"
+#include "std_msgs/msg/int32.hpp"


  #define OFFSET 0 /* use 1 for 0.7V--2.7V) */
@@ -116,14 +116,16 @@ int main(int argc, char** argv) {
```

```
        strcat(buf, "chip_adc_");
        strcat(buf, mac);

-       ros::init(argc, argv, buf);
-       ros::NodeHandle n("~");
-       ros::Publisher pub_muv = n.advertise<std_msgs::Int32>("microvolts", 1);
-       ros::Publisher pub_adc = n.advertise<std_msgs::Int32>("adc", 1);
-       std_msgs::Int32 muv_output;
-       std_msgs::Int32 adc_output;
-       muv_output.data = 0;
-       adc_output.data = 0;
+       rclcpp::init(argc, argv);
+       auto node = rclcpp::node::Node::make_shared(buf);
+       auto pub_muv = node->create_publisher<std_msgs::msg::Int32>("microvolts", rmw_qos_profile_default
+       auto pub_adc = node->create_publisher<std_msgs::msg::Int32>("adc", rmw_qos_profile_default);
+
+
+       auto muv_output = std::make_shared<std_msgs::msg::Int32>();
```

```
+        auto adc_output = std::make_shared<std_msgs::msg::Int32>();
+        muv_output->data = 0;
+        adc_output->data = 0;

         open_adc(0x34);
         enable_adc();
@@ -131,10 +133,10 @@ int main(int argc, char** argv) {
                 adc_raw = read_adc();
                 microvolts = convert_adc_to_microvolts(adc_raw);

-                muv_output.data = microvolts;
-                pub_muv.publish(muv_output);
-                adc_output.data = adc_raw;
-                pub_adc.publish(adc_output);
+                muv_output->data = microvolts;
+                pub_muv->publish(muv_output);
+                adc_output->data = adc_raw;
+                pub_adc->publish(adc_output);
```

```
        nanosleep(&sample_period, NULL);
    }
```

References

[1] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe. 2009, p. 5.

[2] Brian Gerkey, Morgan Quigley, and Ken Conley. *Robot Operating System repository*. 2007. URL: `https://sourceforge.net/p/ros/code/10/log/?path=`.

[3] Keenan Wyrobek. *The Origin Story of ROS, the Linux of Robotics*. 2017. URL: `https://spectrum.ieee.org/automaton/robotics/robotics-software/the-origin-story-of-ros-the-linux-of-robotics`.

[4] Brian Gerkey. *Why ROS 2.0?* 2015–2017. URL: `http://design.ros2.org/articles/why_ros2.html`.

[5] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. "Exploring the Performance of ROS2". In: *Proceedings of the 13th International Conference on Embedded Software*. ACM. 2016, p. 5.

[6] Loïc Dauphin et al. "NDN-based IoT robotics". In: *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM. 2017, pp. 212–213.

[7] Victor Mayoral and nuttx. *GitHub - ros2/ros2_embedded_nuttx*. 2014. URL: `https://github.com/ros2/ros2_embedded_nuttx`.

[8] Morgan Quigley. *ROS2 on "small" embedded systems*. 2015. URL: `https://roscon.ros.org/2015/presentations/ros2_on_small_embedded_systems.pdf`.

[9] Open Source Robotics Foundation. *TurtleBot2*. 2017. URL: http://www.turtlebot.com/turtlebot2/.

[10] Next Thing Co. *Get C.H.I.P. and C.H.I.P. Pro — The Smarter Way to Build Smart Things*. 2017. URL: https://getchip.com/pages/chip.

[11] A13. *A13 — linux-sunxi.org*. 2017. URL: http://linux-sunxi.org/index.php?title=A13&oldid=19534.

[12] Artem Bityutskiy. *UBIFS — new flash file system*. 2008. URL: https://lwn.net/Articles/275706/.

[13] Gerardo Pardo-Castellote. "OMG data-distribution service: Architectural overview". In: *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. IEEE. 2003, pp. 200–206.

[14] eProsima. *News*. 2017. URL: http://www.eprosima.com/index.php/company-all/news.

[15] Keysight Technologies. *33120A Function / Arbitrary Waveform Generator*. 2014. URL: http://literature.cdn.keysight.com/litweb/pdf/5968-0125EN.pdf?id=1000032746:epsg:dow.

[16] Jaime Martin Losa. *ROS2 Fine Tuning*. 2017. URL: https://roscon.ros.org/2017/presentations/ROSCon%202017%20ROS2%20Fine%20Tuning.pdf.

[17] eProsima. *Publisher-Subscriber Layer*. 2017. URL: http://docs.eprosima.com/en/latest/pubsub.html.

[18] Cygwin. *Cygwin*. 2017. URL: https://cygwin.com/.

[19] Robert Collins. *cygwin.com Git - newlib-cygwin.git/blob - winsup/cygwin/sched.cc*. 2016. URL: https://cygwin.com/git/gitweb.cgi?p=newlib-cygwin.git;a=blob;f=winsup/cygwin/sched.cc;hb=08d77e5154b58d0d153e99d270f7d1907f7e160d#l407.

[20] Jack Hammons. *WSL System Calls*. 2016. URL: https://blogs.msdn.microsoft.com/wsl/2016/06/08/wsl-system-calls/.

[21] Jan Bernlöhr. *Running ROS on Windows 10*. 2017. URL: https://janbernloehr.de/2017/06/10/ros-windows.

[22] Sarah Cooley and Aleksandar Nikolić. *Windows 10 Installation Guide*. 2017. URL: https://docs.microsoft.com/en-us/windows/wsl/install-win10.

[23] Mikael Arguedas and D. Hood. *lunar/Installation/Ubuntu*. 2017. URL: http://wiki.ros.org/action/recall/lunar/Installation/Ubuntu?action=recall&rev=6.

[24] Chocolatey Software, Inc. *Chocolatey — The package manager for Windows*. 2017. URL: https://chocolatey.org/.

[25] Rob Reynolds et al. *GettingStarted — chocolatey/choco Wiki — GitHub*. 2017. URL: https://github.com/chocolatey/choco/wiki/GettingStarted/19885416c5913ecffd430c2a69b36ddeff928dfc.

[26] Wyatt Newman. *A Systematic Approach to Learning Robot Programming with ROS*. CRC Press, 2017.

[27] Tully Foote and Ken Conley. *Target Platforms*. 2017. URL: http://www.ros.org/reps/rep-0003.html.

[28] Client Libraries. *Client Libraries*. 2016. URL: http://wiki.ros.org/action/recall/Client%20Libraries?action=recall&rev=48.

[29] Russell Toris and Brandon Alexander. *roslibjs*. 2015. URL: http://wiki.ros.org/action/recall/roslibjs?action=recall&rev=14.

[30] Adnan Ademovic. *GitHub — adnanademovic/rosrust: Pure Rust implementation of a ROS client library*. 2017. URL: https://github.com/adnanademovic/rosrust.

[31] Pauli Virtanen and Charles R. Harris. *Developer notes on the transition to Python 3*. 2010. URL: https://github.com/numpy/numpy/blob/master/doc/Py3K.rst.txt.

[32] Nathaniel Smith. *PEP 465 – A dedicated infix operator for matrix multiplication*. 2014–2016. URL: https://raw.githubusercontent.com/python/peps/04a6af2ab1b19a56db74d5ae85a96656cc04bfa6/pep-0465.txt.

[33] Benjamin Peterson. *PEP 373 — Python 2.7 Release Schedule*. 2008–2017. URL: https://raw.githubusercontent.com/python/peps/647e1bbe39f5e056bbcd4535992fa8a
pep-0373.txt.

[34] Dirk Thomas. *Changes between ROS 1 and ROS 2*. 2015–2017. URL: http://design.ros2.org/articles/changes.html.

[35] Jules Kouatchou. *Basic Comparison of Python, Julia, R, Matlab and IDL*. 2017. URL: https://modelingguru.nasa.gov/docs/DOC-2625/diff?secondVersionNumber=19.

[36] Python Software Foundation. *Python 3.0 Release*. 2008. URL: https://www.python.org/download/releases/3.0/.

[37] Python2orPython3. *Python2orPython3 - Python Wiki*. 2017. URL: https://wiki.python.org/moin/Python2orPython3?action=recall&rev=92.

[38] Scipy community. *Release Notes — NumPy v1.10 Manual*. 2015. URL: https://docs.scipy.org/doc/numpy-1.10.1/release.html.

[39] Sebastian Raschka. *The key differences between Python 2.7.x and Python 3.x with examples*. 2014. URL: http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html.

[40] Bjarne Stroustrup. *C++11 FAQ*. 2016. URL: http://www.stroustrup.com/C++11FAQ.html.

[41] Brian Gerkey, Morgan Quigley, and Ruffin White. *SROS — ROS Wiki*. 2016. URL: http://wiki.ros.org/action/recall/SROS?action=recall&rev=33.

[42] Mikael Arguedas. *GitHub - ros2/sros2: tools to generate and distribute keys for SROS 2*. 2017. URL: https://github.com/ros2/sros2.

[43] Open Source Robotics Foundation. *Migrating from format 1 to format 2*. 2014. URL: http://docs.ros.org/groovy/api/catkin/html/howto/format2/migrating_from_format_1.html.

[44] Dirk Thomas and Jack O'Quin. *Package Manifest Format Two Specification*. 2013–2017. URL: http://www.ros.org/reps/rep-0140.html.

[45] Brian Gerkey et al. *Migration Guide*. 2016–2018. URL: https://github.com/ros2/ros2/wiki/Migration-Guide.

[46] Jackie Kay. *Debugging ROS2 in Visual Studio*. 2016. URL: https://groups.google.com/d/msg/ros-sig-ng-ros/bhAMgFCIOnE/GNMM105iCgAJ.

[47] Joonas Tuomi. *Is NTC ready for the move from Debian Jessie to Stretch?* 2017. URL: https://bbs.nextthing.co/t/is-ntc-ready-for-the-move-from-debian-jessie-to-stretch/14706/9.

[48] Debian Webmaster. *Debian — Package Search Results — libpocofoundation*. 2017. URL: https://packages.debian.org/search?suite=default&section=all&arch=any&searchon=names&keywords=libpocofoundation.

[49] Jackie Kay, Tully Foote, and D. Hood. *kinetic/Installation/Debian*. 2016. URL: http://wiki.ros.org/action/recall/kinetic/Installation/Debian?action=recall&rev=12.

[50] Gerfried Fuchs. *Ubuntu — Package Search Results — libpocofoundation*. 2017. URL: https://packages.ubuntu.com/search?keywords=libpocofoundation&searchon=names.

[51] Esteve Fernandez. *Tools for crosscompiling ROS2 for the Raspberry Pi*. 2017. URL: `https://github.com/esteve/ros2_raspbian_tools`.