SNR: SOFTWARE LIBRARY FOR INTRODUCTORY ROBOTICS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Spencer Shaw

August 2021

© 2021
Spencer Shaw
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	SNR: Software Library for Introductory Robotics
AUTHOR:	Spencer Shaw
DATE SUBMITTED:	August 2021
COMMITTEE CHAIR:	Andrew Danowitz, Ph.D. Professor of Computer Engineering
COMMITTEE MEMBER:	Bridget Benson, Ph.D. Professor of Computer Engineering

ABSTRACT

SNR: Software Library for Introductory Robotics

Spencer Shaw

This thesis introduces "SNR," a Python library for programming robotic systems in the context of introductory robotics courses. Greater demand for roboticists has pressured educational institutions to expand robotics curricula. Students are now more likely to take robotics courses earlier and with less prior programming experience. Students may be attempting to simultaneously learn a systems programming language, a library API, and robotics concepts. SNR is written purely in Python to present familiar semantics, eliminating one of these learning curves. Industry standard robotics libraries such as ROS often require additional build tools and configuration languages. Students in introductory courses frequently lack skills needed for these tools. SNR does not use any additional build tools, so students are faced with fewer compounding learning curves. SNR presents students with concepts important to robotic systems programming such as modular and event driven architectures to bridge the gap between introductory programming courses and industry standard libraries.

ACKNOWLEDGMENTS

Thanks to:

- Dr. Andrew Danowitz, for his guidance on this thesis and throughout my studies at Cal Poly
- Dr. John Seng, for his vision and advice for the Robotics Club
- Dr. Bridget Benson, for advice in her areas of expertise
- Dr. Jane Zhang, for coordination of the Electrical Engineering graduate program
- Dr. Dennis Derickson, for legendary leadership of the Electrical Engineering department and engaging seminars
- Dr. Lynne Slivovski, for coordination of the Computer Engineering Program as an ally to students
- My parents and grandparents for their love, support and feedback in this thesis and my entire academic career

TABLE OF CONTENTS

Page

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF LISTINGS xii

CHAPTER

1 Introduction 1

2 Background 4

2.1 Topics for Robotics Software Libraries 5

2.1.1 Publisher-Subscriber Paradigm 5

2.1.2 Typing Checking in Python 5

2.2 Related Work 7

2.2.1 ROS 8

2.2.2 F' 10

2.2.3 WPILib 10

2.2.4 Survey Summary 11

3 Evaluation Framework 12

3.1 The Arduino Platform 12

3.2 Characteristics 13

3.2.1 Prerequisites 13

3.2.2 Familiarity 14

3.2.3 Discoverability 15

3.2.4 Documentation 15

3.2.5 Error Messages 16

3.2.6 Misuse Resistance 16
3.2.7 Openness 17
3.2.8 Growth Paradigms 17
4 Library Design 19
4.1 Design Goals 19
4.2 Architecture 20
4.2.1 Runners, Configs, and Roles 21
4.2.2 Node Life Cycle 22
4.3 Data Objects 23
4.3.1 Tasks 23
4.3.2 Pages 24
4.4 Endpoints 24
4.4.1 Task Handling 25
4.4.2 Endpoint Life Cycle 26
4.4.3 Loops 26
5 Example Usage and Library Evaluation 28
5.1 Example: Underwater Remote Operated Vehicle 28
5.2 Moving Average Filter Endpoint 31
5.2.1 Moving Average Filter Endpoint 31
5.2.2 Moving Average Filter Endpoint Factory 31
5.2.3 Moving Average Endpoint Tests 32
5.3 Example: Lunar Module Landing Simulation 32
5.4 Example: Benchmark Test 33
5.4.1 Game Controller Over Sockets 35
6 Library Implementation 40

6.1 Code Style Principles . . . 40
6.2 Prelude . . . 41
6.2.1 Page and Task . . . 41
6.2.2 Type Aliases . . . 42
6.2.3 Protocols . . . 43
6.2.4 Interfaces . . . 43
6.2.5 Precursors . . . 43
6.2.6 AbstractNode . . . 44
6.2.7 TaskQueue . . . 46
6.2.8 AbstractEndpoint . . . 47
6.2.9 AbstractLoop . . . 47
6.2.10 AbstractFactory . . . 47
6.3 Core . . . 48
6.3.1 Node . . . 48
6.3.2 Endpoint . . . 48
6.3.3 ThreadLoop . . . 48
6.4 Standard Modules . . . 49
6.4.1 Communications . . . 49
6.4.2 Filters . . . 49
6.4.3 Input/Output . . . 49
6.5 Testing Utilities . . . 50
7 Library Evaluation . . . 52
7.1 Prerequisites . . . 52
7.2 Discoverability . . . 52
7.3 Error Messages . . . 53

7.4 Openness 54
7.5 Familiarity 54
7.6 Documentation 54
7.7 Misuse Resistance 55
7.8 Growth Paradigms 55
8 Future Work 56
8.1 Library Improvements 56
8.1.1 Specification Formalization 57
8.1.2 Publishing 57
8.2 Evaluation Framework 57
8.3 Curriculum Development 57
8.3.1 Introductory Robotics 58
8.3.2 Advanced Programming Topics 58
9 Conclusion 59
BIBLIOGRAPHY 60
APPENDICES
A Code Listings 64
B SNR Library Installation Tutorial 84

LIST OF TABLES

Table Page

2.1 Robotic systems libraries surveyed 11

LIST OF FIGURES

Figure		Page
2.1	Static analysis workflow diagram	6
2.2	ROS communication architecture diagram	8
4.1	SNR node architecture	21
4.2	SNR high-level architecture	22
4.3	SNR Endpoint architecture	25
4.4	SNR Loop architecture	27
5.1	Data flow in example UROV system	28
5.2	Game controller read from using an SNR loop	36
5.3	Raspberry Pi 4 running SNR nodes	37
5.4	Terminal output of controller input changes across sockets connection	39
6.1	SNR library submodule dependency directed acyclic graph	40

LIST OF LISTINGS

5.1 Example configuration of a UROV system 30

5.2 Benchmark shell output 34

5.3 Example configuration using game controller and sockets 38

6.1 Pseudo code interface for node control flow 45

6.2 Pseudo code interface for task execution 45

6.3 Pseudo code interface for task scheduling 45

6.4 Pseudo code interface for datastore access 46

7.1 Example runtime error message 53

A.1 Standard implementation of a moving average filter endpoint 64

A.2 Standard implementation of a factory for MovingAverageFilterEndpoint 65

A.3 Example usage of testing facilities using SNRTestCase and Expector . 66

A.4 Example configuration of lunar module landing simulation 67

A.5 Example implementation of a benchmark test case 71

A.6 Example implementation of a factory for a control input processor endpoint 74

A.7 Example implementation of a control input processor endpoint 75

A.8 Example implementation of a loop for reading controller input 77

A.9 Example implementation of a factory for the controller loop 80

A.10 Example implementation of a factory for the input mapping endpoint 81

A.11 Example implementation of an input mapping endpoint 81

B.1 System information for installation example 84

B.2 Installation of Pip 84

Chapter 1

INTRODUCTION

The use of robots continues to expand from industrial sectors to other spaces, creating more demand for robotics engineers [1], [2]. To meet this demand, educational institutions have expanded robotics curricula and created new programs specializing in robotics. Some universities have taken the opportunity to offer undergraduate and graduate programs concentrating on robotics [3], [4]. As new robotics courses permeate curricula, students are being introduced to the field earlier. Some institutions have found success in introducing students to programming using robotics and other embedded domains [5], [6]. The advancement of robotics concepts in students' academic careers has created a gap between their skills and the tools used for cutting edge robotics.

Without prior systems programming experience, students face obstacles as they learn to program robotic systems. For example, modern robotic system design often benefits from highly modular system implementations in which components of robots, such as sensors, actuators, and controllers, are programmed individually. This allows for isolated development and testing of components and design reuse. Industry standard robotics tools such as ROS and F' encourage modular systems by providing infrastructure to bridge modules. This infrastructure uses paradigms including event-driven architectures and publisher-subscriber communication. Students should learn the skills inherent to these tools as they learn about robotics. Unfortunately, students who have only completed one or two introductory computer science courses are often unprepared to learn these skills while using an industry standard framework like ROS or F'. These frameworks may introduce too many new ideas and skills at

once. Students may then be learning a new programming language, robotics library, and system design skills simultaneously. The learning curves of each of these topics compound, resulting in cognitive overload [7]. A course introducing students to robotics can be more approachable and accessible by introducing fewer new topics at once. This thesis is concerned with providing a library with bindings in a language that students are likely to be familiar with and no dependencies on additional configuration languages or tools. Reducing these requirements lowers academic barriers to entry when introducing students to robotics.

This thesis presents "SNR"(SNR is Not ROS), a software library for programming robots in education. SNR uses concepts important to robotic systems in a Python package friendly to students. SNR does not require the use of an additional build system or configuration language. For example, SNR introduces the publisher/subscriber paradigm while encouraging students to develop discrete modules for different robotic functionality. The library is written purely in Python, providing students with an implementation they can easily inspect and explore. This also makes the library implementation portable from student PC's to single board computers such as the Raspberry Pi. For more advanced systems programming courses, the implementation of SNR serves as a case study for students learning about systems programming concepts, type annotation, and intermediate Python tooling.

A framework for evaluation of software libraries and tools for use in the context of education is also presented. The qualities described in the framework will help to identify friction in student comprehension of software tools and libraries.

Chapter 2 discusses background information on teaching robotics and embedded systems. Then, other robotics software libraries commonly used in industry and education will be introduced. In chapter 3, the framework for evaluating software for students will be introduced. In chapter 4, the goals and guiding principles for the de-

sign of SNR will be covered prior to describing the application programming interface (API). Examples using the library will be presented in chapter 5. The implementation of the library will be covered in more detail in chapter 6. Finally, the library will be analyzed with the aforementioned framework for evaluation in chapter 7.

Chapter 2

BACKGROUND

As undergraduate students' introduction to systems robotics occurs earlier in their academic careers, a gap is formed between students' prior experiences in introductory computing classes and professional robotics libraries. Undergraduate students have difficulty transitioning from introductory computing courses to intermediate systems projects, especially with the introduction of embedded hardware [8]. As students are introduced to robotics systems programming paradigms earlier, they have less knowledge of programming and organizational techniques. Thus, they are not able to fully utilize industry standard robotics libraries like ROS. Instead, these less experienced students should be introduced to robotics libraries in a form they are familiar with, such as a Python library. Thus, new concepts such as publisher-subscriber communication can be taught to students without friction of new programming languages or complex tool-chains. Simultaneously, students can explore concepts such as type systems with more rigor. Libraries can ease students' introductions by offering navigable code that students can learn from. In Python, type annotations (hints) can contribute to student comprehension despite their historical absence.

In this case, it makes sense to introduce embedded robotics systems to students using software language semantics they are already comfortable with. The challenges of robotic systems programming provide a suitable requirement for new skills in modular design, version control, and publisher-subscriber paradigms. Students need these specific skills for working with robotic systems in order to cope with increased system complexity. Learning these skills in a familiar Python environment can prepare students for more advanced projects in the future. In addition, structuring the library

APIs to encourage modularity can smooth the introduction of modular program architectures.

2.1 Topics for Robotics Software Libraries

2.1.1 Publisher-Subscriber Paradigm

Publisher-subscriber systems are a common paradigm for robotics programming. This event-driven pattern, "pub-sub" decouples triggering and handling events [9] [10]. This decoupling is a technique in software engineering that students can also benefit from. While the concept and operation of pub-sub can be grasped easily by students, the motivation (decoupling) and internal implementation can elude students.

Pub-sub works as a message passing paradigm by separating the actions of sending and receiving specific messages. When a message is published, the sender has no knowledge of the component receiving the message. Likewise, the receiver need not be aware of the sender, only the data it gets. In the context of systems programming, this allows decoupling of task creation and execution spatially (between sections of code) and temporally.

2.1.2 Typing Checking in Python

Python provides an excellent learning environment for students learning to program. Many undergraduate students favor it following their initial computing classes. Its dynamic type system can hide important information from students. When requiring students to satisfy an interface, for example, visible types provide concrete specification that can prevent programming errors and misuse. Python 3 supports type hints [11]. While type annotations provide direct benefit to the programmer, the

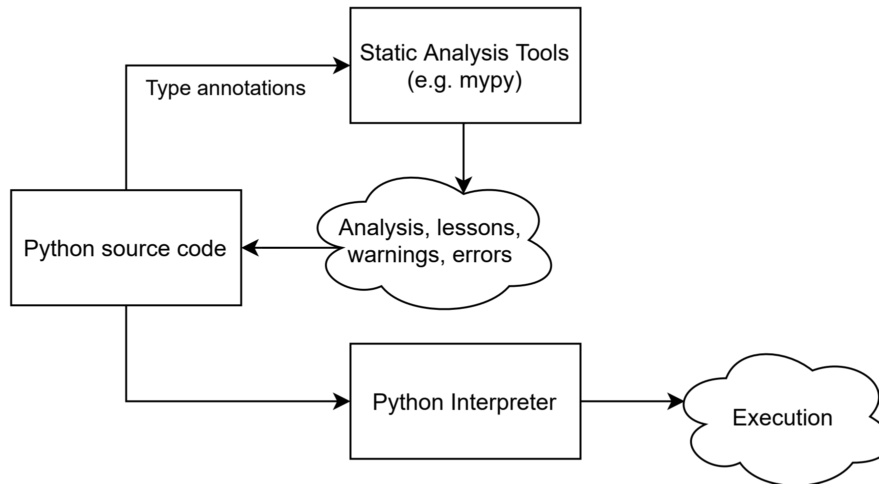

Figure 2.1: Static analysis workflow diagram

Python interpreter ignores them at runtime. More benefit can be had using additional tools designed to make use of the annotations. Figure 2.1 illustrates an abstract Python programming workflow utilizing a static analysis tool. Results from static analysis tools can provide insights into code while it is incomplete.

When students are first introduced to a software library, they are not aware of data types relevant to an interface. Type annotations improve readability for users unfamiliar with an interface. Tools using type annotations can also help users. For example, the user's IDE can take the user to the definition of a type from a type annotation. This reduction in friction can encourage students to gain competency by reading and exploring code. Type annotations can also allow third party tools to report incorrect code through static analysis. Static analysis tools inspect code without directly running it to provide insight on the code's functionality and correctness. This can include warning and error messages for programming mistakes frequently made by students. Key to this benefit, the tool provides error messages from which users and students can learn and take action from. The information provided directly by type annotations and indirectly though tools enhances the learning experience for students.

As previously discussed, needing to learn to use additional tools can be a hindrance to students' direct learning goals. For this reason, it is important that static analysis tools are seamlessly integrated in students' development environments. Integrated development environments (IDEs) such as Microsoft's Visual Studio Code (VSCode) can provide static analysis feedback as students type without additional commands. If the tool provides clear and actionable messages, students can understand mistakes they have made and correct them. This contributes to the learning process. VSCode eases adoption of said tools by prompting users to install and enable them once support for the Python language is installed in VSCode. Beginners benefit from the accessibility of tools configured this way by default. With analysis tooling automated by default, students can focus on learning to use type systems without first learning to use a type checker.

2.2 Related Work

Numerous software frameworks for accelerating development of robots exist. These libraries accelerate robotics development by abstracting away the interconnection of modular components. These frameworks make modularization less complex, allowing for more complex robotic systems. These frameworks are often maintained by large communities that contribute common packages that can be reused. This reuse automates development for common robotic tasks such as path planning and computer vision. Industry and state of the art research primarily use ROS (Robot Operating System). NASA has successfully deployed its new entrant, the F' framework, to the planet Mars on the Ingenuity Mars helicopter project [12]. WPILib, a library developed for use in the FIRST Robotics Competition (FRC), has grown in use, complexity, and scope with the popularity of youth robotics competitions.

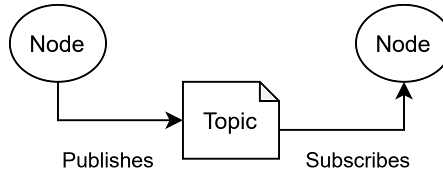

Figure 2.2: ROS communication architecture diagram

2.2.1 ROS

Robot Operating System is an industry standard robotics framework. Despite its name, ROS is not an operating system. Rather, it refers to a core software library, extensible framework, and package ecosystem. Roboticists use standard ROS runtimes to operate a combination of their own packages and those developed by the community and distributed in the package ecosystem. The availability of stable packages in the package ecosystem accelerates development in academia and industry. This attracts a large number of contributors and users across academia and the private sector, serving many niches. A wide community is involved in the development, support, and use of ROS. As a result, ROS is available on many platforms, is well documented, and has a large ecosystem of provided packages [13]. Users with knowledge of ROS's build and packaging systems can program a wide range of hardware platforms and utilize modular packages maintained by the community.

The core library implementations of ROS provide an extensible framework for anyone to develop packages defining the behavior of "nodes". In ROS, a node is the unit of execution available to the programmer [14]. Figure 2.2 shows the modular architecture of ROS. When a ROS-based robot is running, nodes communicate with each other by publishing and subscribing to messages pertaining to specific "topics". A message may contain any type of data as specified by the topic. A central node or manager is used to register the presence of nodes and routing information per topics to subscribing nodes.

Availability across multiple programming languages makes ROS accessible to a wider audience and allows hardware independent design and implementation of system architectures. Unfortunately, ROS's broad applicability adds complexity to the ecosystem as a whole as all ROS packages and tools must be compatible with all supported languages and environments. Standard conventions may conflict between platforms. Users of additional platforms may be confused by non-idiomatic conventions originating from other languages. Students learning to program robot systems for the first time do not need the robustness and ubiquity of ROS and are instead hindered by this added complexity. Development environments for ROS often involve complicated build tool-chains hosted on virtual machines. The size of virtual machine images is a hindrance when inexperienced students are preparing to learn ROS.

Students suffer from the complexity added by build systems. Cross language support tends to beget language independent build systems. This added complexity that impedes beginners because it introduces additional language semantics and constructs for students to learn. In the case of ROS, "Catkin" is the main build system. Catkin leans on the semantics of CMake, a common C++ build system. [15]. Without an additional configuration language, students could be more productive.

When being introduced to language bindings for ROS, students must install ROS and learn how to use Catkin. O'Kane's "A Gentle Introduction to ROS" contains 26 pages on the installation of ROS prior to guiding the user through creating a sample project. While much of this can be attributed to availability on a variety of platforms, installation is still a significant barrier to entry.

2.2.2 F'

F' (F Prime) is a robotics framework developed at NASA Jet Propulsion Laboratory (JPL). [12]. Its most notable use has been on the Ingenuity vehicle on the planet Mars. It has also been used to program cubesats for other missions. Fitting this use case, F Prime provides a Publisher-Subscriber paradigm with a focus on modular, reusable components [16]. The reusability of components accelerates development while benefiting from the robustness and correctness of provided modules that have been verified to be correct.

Like ROS, F Prime uses build tools which complicate its use [17]. F Primes's bindings are available in C++ and additional configuration is written in XML. The build tools generate configuration and compiled results. While additional build tools may slow down learners, this is made up for by reusability of components.

2.2.3 WPILib

WPILib exists as the premier library for programming robots for the FIRST Robotics Competition (FRC) [18]. Using WPILib, high school students develop interdisciplinary team skills to field robots in competitions [19].

WPILib specifically targets high school students who may or may not have experience in AP Computer Science courses. Students are also limited to specific hardware such as the roboRIO, a fully featured micro-controller with numerous peripherals and an integrated field programmable gate array (FPGA) [20]. Despite these academic and technical limitations to scope, the competitive nature of FRC has lead WPILib to be fully featured and well documented.

WPILib supports Java and C++ bindings and can be built with Gradle or CMake [21]. In contrast with Python, these additional build steps add friction to students experiences.

Java bindings give WPILib the advantage of meshing with Java-based AP Computer Science curriculum. C++ and CMake experience can also prepare students for using the same tools when learning to use ROS. Despite this familiarity, a gap is left for undergraduate students who have been introduced to programming in only one Python-based introductory computer science course. These students may have had no experience with compiled programs.

2.2.4 Survey Summary

Table 2.1: Robotic systems libraries surveyed

Name	Usage	Language Bindings
ROS	Industry, academia	C++, Python, Lisp
F Prime	NASA, spaceflight	C++
WPILib	FIRST Robotics Competition	C++, Java

Table 2.1 summarizes the libraries surveyed. Across ROS, F Prime, and WPILib, two trends are visible: modular components and additional build tools. The first is a concept that students can benefit from learning. The second is a complication to software and embedded systems development.

Following this survey of systems robotics libraries and their relationships to students, a method is needed for discussing and evaluating how their design impacts the experiences of students. Chapter 3 presents a framework for qualitative evaluation of software libraries and tools.

Chapter 3

EVALUATION FRAMEWORK

This chapter presents a qualitative framework for evaluating software libraries and tools in the context of undergraduate education. This framework was developed to help guide the course of this thesis. The framework consists of eight characteristics listed in section 3.2. Evaluation of the characteristics can be used to discuss the design and distribution of a software library or tool based on possible experiences of students. These characteristics have been chosen to assess the how design of the tool impacts the user experience of students as they learn to use it to accomplish an external learning objective. The characteristics should not be used to compare tools. They are best used to initiate a discussion surrounding a specific student user experience. Without further refinement, these characteristics should not be used to make a comparative or conclusive assessment. In this chapter the Arduino embedded development platform will be evaluated within the framework as an example. In chapter 7, the SNR library will be evaluated within this framework.

3.1 The Arduino Platform

The Arduino platform was originally developed at the Interaction Design Institute Ivrea in Ivrea, Italy [22] by Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis [22]. The project undercut existing micro-controller development boards in price while championing ease of use. This was achieved by using the Processing integrated development environment (IDE) and developing a user-friendly library [23]. Arduino has enabled learners of diverse backgrounds to create projects

and inventions that would have otherwise required a much greater knowledge of embedded systems [24]. Simultaneously, the platform was introduced many people to the field of embedded systems. The following evaluation of the Arduino platform includes references to Arduino branded development board hardware, the Arduino IDE, the Arduino firmware library, the ecosystem of Arduino-compatible hardware, and the ecosystem of Arduino-compatible libraries.

3.2 Characteristics

The presented framework for evaluation includes the following eight characteristics listed below. Some characteristics were chosen based on friction experienced by students when learning to use a tool or library. Others were selected to reflect desired learning outcomes.

- Prerequisites
- Discoverability
- Error messages
- Openness
- Familiarity
- Documentation
- Misuse resistance
- Growth paradigms

3.2.1 Prerequisites

Prerequisites may refer to either *academic* or *material* barriers to entry for students. Prerequisites are the most direct barriers to entry for students. The direct nature of prerequisites means that they may have quantitative elements.

Academic prerequisites include knowledge students need to make use of a library or tool. This could be an understanding of the semantics of a programming language or mathematical theory of a control system.

Material prerequisites manifest as monetary costs that otherwise prevent students from gaining an education experience. Material prerequisites might include access to a specific hardware platform, software distribution, or software tooling.

The Arduino platform tackled both facets of prerequisites at its creation. It offered development hardware at a lower price, a quantitative reduction in a material prerequisite. The development environment was freely distributed, reducing another material prerequisite. Simple design of the Arduino firmware library decreased academic requirements for users.

3.2.2 Familiarity

Familiarity may refer to the reuse of semantics and patterns already known to students. Familiarity complements prerequisites. While prerequisites tend to exist as hard requirements, familiarity only improves user experiences and is not a requirement. Prior knowledge can greatly improve student comprehension. Accessing knowledge stored in long term memory accelerates comprehension. [25]

Users of the Processing IDE will be familiar with the Arduino IDE as a result of the common ancestry. The C++ language used in the Arduino platform will also be familiar to students who have learned the C or Java programming languages. As a result, they will spend less time learning to use the development environment, programming language syntax, or object-oriented paradigm respectively.

3.2.3 Discoverability

Discoverability may refer both to discovering the existence of the tool and to the ease with which students can explore the interfaces of the library and find information needed to solve their problem. For the introduction of a tool, a teacher may introduce it to students or students may discover its existence on their own. More generally, discoverability can refer to exploration of the surface of the tool during its use.

The popularity of the Arduino platform means many students and educators are aware of it and may recommend it to uninformed students. The interface surface of the Arduino library is documented in a website (bundled with the IDE) linked from the "Help" menu of the Arduino IDE [26]. As a result of bundling reference materials with the development environment, the documentation is available without an internet connection.

3.2.4 Documentation

Documentation refers to the accessibility, coverage, and quality of reference materials specifying usage of the library. While discoverability pertains to the experience of users, documentation concerns the material information available for a tool, in the best case scenario that it is discovered. Good documentation includes both concrete definitions and examples of usage.

In the Arduino library reference for each provided function, the following fields are included: a description, the syntax, the parameters, a complete example program, additional notes and warnings, and a link to further documentation on related concepts. Some translations to languages other than English are available on the same page.

3.2.5 Error Messages

Error messages may refer to the ease of interpretability of tooling responses to invalid or anomalous input. If a user makes a mistake or typo, error messages should warn them in the context of their mistake and not in the context of the underlying system. The program should gracefully exit after providing the context of the issue and possible courses of action. Bad error messages might only describe an irrelevant symptom of the mistake and signify the occurrence of a catastrophe.

The Arduino IDE sometimes exposes users to C++ compiler errors or irrelevant warnings. These may confuse the user. The syntactic construction of C++ sometimes prevents the compiler from being able to identify specific typos or mistakes. The Arduino IDE sometimes produces Java-based error messages of its own, unrelated to C++ code being compiled. Students using C++ for the first time may be confused by error messages unrelated to the language or library.

3.2.6 Misuse Resistance

Misuse resistance may refer to preventative measures that reduce the occurrence of errors or mistakes. Specifically these preventative measures may be implemented during design of the tool with the expectation that users are likely to make mistakes. This characteristic might be assessed with the question, “How does the library or tool minimize the negative consequences of a mistake?”. Tools that excel in misuse resistance may even approach mistakes differently, seeking to maximize the potential for learning from mistakes.

The Arduino library hides complexity from users and protects them from complexity well. This provides misuse resistance within the programming environment. However,

difficulties with the hardware, relating to signal integrity (unplugging cables), can cause errors during program flashing of micro-controllers. This can temporary or permanently damage the firmware bootloader or development board. Development boards themselves are fragile and vulnerable to electrical dangers ranging from static discharge to incorrect pin connections and defective peripheral circuits. This example shows that misuse resistance can vary between different components and aspects of a tool.

3.2.7 Openness

Openness may refer to the ease of extension or inspection. A open system should allow students to investigate its inner workings and then replicate or extend the implementation.

The Arduino hardware is primarily focused on enabling users to extend it with peripheral devices. The Arduino IDE and C++ language encourage students to build abstractions off their code. However, the tool-chain infrastructure is not readily exposed to users and the inner workings of Arduino micro-controllers are not generally available. Students are not directly encouraged to develop a deeper understanding our how the compiler is used. This comes as a trade-off to preventing misuse of the compiler tool-chain.

3.2.8 Growth Paradigms

Growth paradigms may refer to the new semantics and patterns that prepare students for other libraries and tools. In this way, the tool acts as a prerequisite for something else. Growth paradigms might also refer to the level of encouragement or lack of discouragement that the tool provides students. Tools promoting growth paradigms

encourage students to have a growth mindset. Growth mindsets have been shown to improve math and science achievement for students [27]. Openness can also lead to orthogonal growth paradigms in which students begin to learn about the systems that underlie the tool.

The Arduino platform is a gateway to electronics and embedded development. The numerous features of the platform encourage students to explore all the capabilities of hardware while exercising their own creativity.

Chapter 4

LIBRARY DESIGN

This chapter discusses the abstract design of the SNR software library, including the design of the application programming interface (API), communication paradigm, and concurrency primitives.

4.1 Design Goals

The main design goal of the SNR library is to use a familiar programming language to introduce students to programming robotic systems. This is achieved by providing an API for small modules to communicate discrete pieces of data to each other. This approach gives students experience with the organizational problems resulting from complex robotic systems that lend themselves to smaller, reusable modules. This can help students learn paradigms such as publisher-subscriber communication earlier in their undergraduate education. Understanding concepts like this prepares students of more complex architectures in robotic systems and other career paths. The programming model of SNR, introduced in 4.2, is intended to avoid cognitive overload by providing a simple Python library that students can inspect and extend. This architecture encourages modular components, which can be programmed incrementally, tested, and reused. As a pure python library, SNR avoids the requirement of additional build tools. Only the Python interpreter and SNR library are needed to run a program that uses SNR.

4.2 Architecture

SNR provides the end user with three levels of abstraction (shown in figure 4.2). The highest of these levels is a complete robotic system that may include multiple computing devices. A system is defined by the user in one configuration. The next level, nodes, corresponds to each computing device within a system. The user defines which components exist inside each node. These components are the final level of abstraction. The user implements component modules.

The communication paradigm in SNR consists of discrete components passing messages to each other. User defined components, called “endpoints”, do not communicate with each other directly. Rather than use network based direct communication, events called “tasks” are always passed to a task queue consumed by the main event loop. Endpoints are modules of user code that provide functions that are run in response to specific tasks. Endpoints are described in section 4.4. SNR does not provide an internal way for nodes to communicate with each other. Instead, cross node communication is provided as a standard component that users can include with a node. This design choice reduces the complexity of nodes while pushing students towards understanding how data flows between components.

Figure 4.1 shows the internal components of a running “node.” Each node contains a task queue serviced by the main event loop, a key-value database (“datastore”), and a collection of endpoints. One node is intended to be run per device. To execute a task, endpoints are queried for matching “task handler” functions. Task handlers are registered within a dictionary for each endpoint.

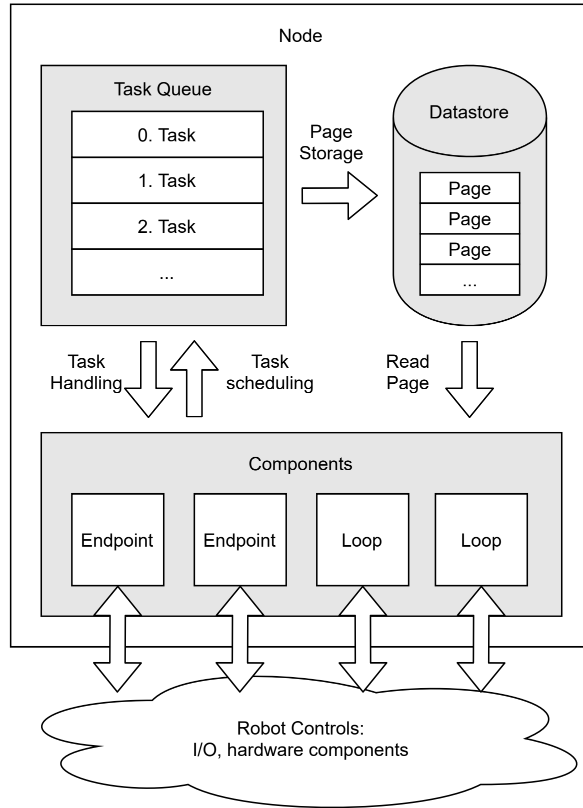

Figure 4.1: SNR node architecture

4.2.1 Runners, Configs, and Roles

The life cycle of an SNR based program begins with a "runner" that wraps the execution of a node based on a provided configuration (abbreviated to "config"). A config is the highest level description of a system in SNR. A config describes a blueprint for all nodes used to operate a robotic system. Each node is differentiated by a "role" identifier and may have a unique set of endpoints. Figure 4.2 depicts this relationship.

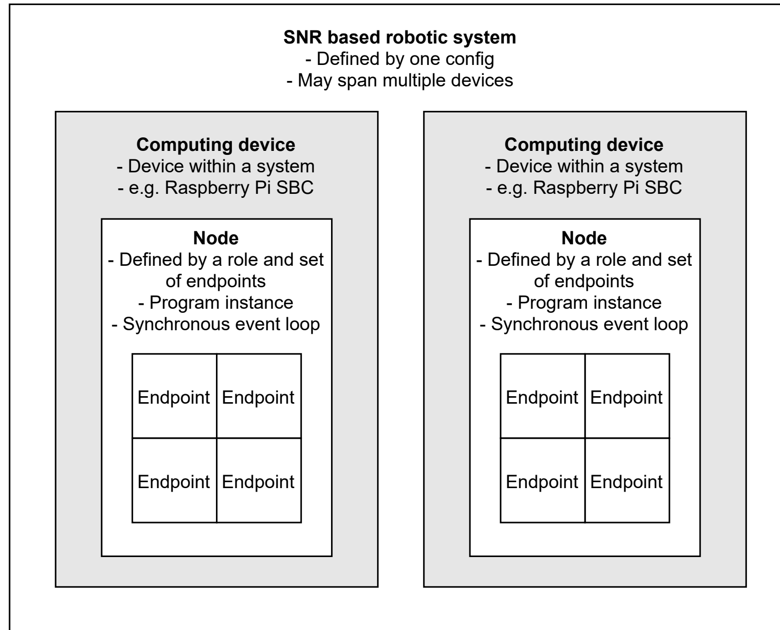

Figure 4.2: SNR high-level architecture

A config defines a different set of endpoints to be initialized within a node based on the "role" of the node with the larger system. For example, a node running on a computer embedded in a robot would have different endpoints than a node running on a computer monitoring the robot remotely. Specifying the configuration of both nodes at once allows code reuse and shared parameters such as communication channels. At runtime, the runner differentiates the role of the node and provides it the specific configuration. Then, a node is constructed.

4.2.2 Node Life Cycle

A node is constructed by a runner with a given role and endpoint collection. The per-role configuration contains endpoint factory objects that construct endpoints for the node. Each factory can provide the node with one or more endpoints. After being constructed, the node can enter its main event loop . Prior to beginning the loop, each endpoint has a post-initialization method run. Once in the event loop, the node

pops a task off the task queue and queries each of its endpoints for a matching task handler. For all endpoints with matching task handlers, the task handlers are run on the main event loop thread. Task handlers may result in zero or more additional tasks. These subsequent tasks are scheduled after all task handlers have executed the current task. The task queue feeds the main event loop with tasks sent by endpoints.

Pages stored in the datastore persist during program execution. They are not implicitly saved between separate program executions (`Recorder`s and `Replayer`s can achieve this as discussed in section 6.4).

4.3 Data Objects

SNR stores and transmits data as two types of objects, Tasks, and Pages. Task represents ephemeral units of work to be processed. Pages represent stored information that can be utilized by components.

4.3.1 Tasks

Tasks represent transitory events. Tasks are ephemeral in that they are available only when handled by an endpoint task handler. A task is discarded once it has been handled by all registered task handlers.

A task is identified by two pieces of information, its `TaskType` and name (represented as a string). These two properties can be mapped to a task ID that matches registered task handlers in multiple ways. A task ID can be either a tuple of a task type and string or a singular task type. Section 4.4.1 discusses the task matching process. Tasks enable information passing between endpoints. Tasks can also carry arbitrary data stored in a "value list" accessible to task handlers. This data is not stored

outside the task while it is queued and is destroyed with the task after execution unless otherwise stored by a task handler.

4.3.2 Pages

Pages provide data storage that outlives individual tasks. Pages consist of a key-value pair. Endpoints may read and write pages based on keys. Keys are represented by strings. The data representation of values contained in pages is not specified. Pages also store the name of the node that created them (the origin) and the time they were created. When a page is created, a task is scheduled to store the page within the node's datastore. By creating a task to store the page, endpoints can be notified of the page and react accordingly. This is useful for transparently communicating pages from one node to another using endpoints and loops. As a task, writing a page to a node's datastore happens synchronously. In line with this, the page storage task is handled by a core endpoint that has access to the datastore data structure. Pages also contain a "process" field that causes a second task to be triggered once it has been stored in the datastore. This enables an additional task execution that is guaranteed to be after the first.

4.4 Endpoints

Endpoints are modules containing user defined behavior and surfacing it to a node in the form of registered task handlers. Endpoints are associated with a specific node and identified by a name. Endpoints have no knowledge of each other. Endpoints within a node pass messages to each other by queuing tasks and storing data pages. Figure 4.3 shows the internals of an endpoint. An endpoint's registration of task handlers consists of a mapping of task IDs to method references. Endpoints defined

by the user may have additional private state accessible only to its task handling methods.

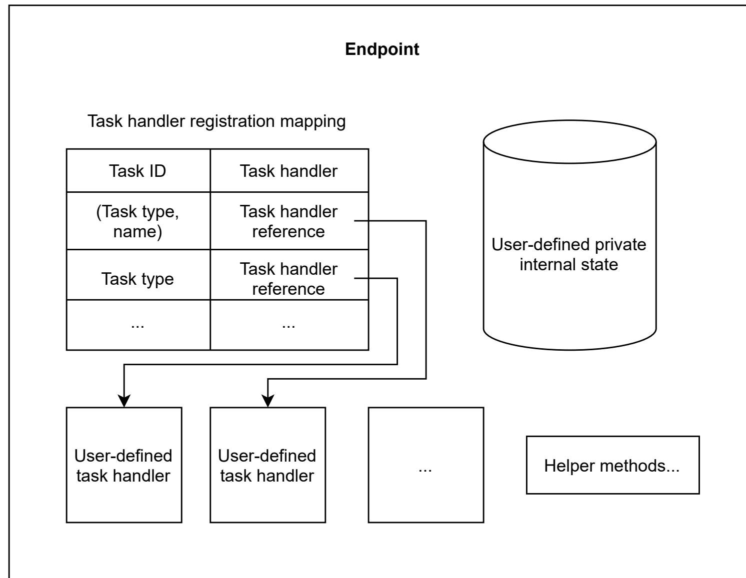

Figure 4.3: SNR Endpoint architecture

4.4.1 Task Handling

When the main event loop of a node processes a task, it first gathers all matching task handlers from all of its endpoints. Two sequential attempts are made to match a task to a registered task handler. First, a task ID consisting of both the task type and name is used. If no handler is found, a task ID consisting of just the task type is used. This query is repeated for each endpoint (The runtime complexity of this lookup algorithm scales linearly with the number of endpoints but allows endpoints to modify their mapping of task handlers during execution without communication with the node). Once all task handlers matching a task have been gathered, they are executed sequentially. All endpoints receive the same task but may have match a handler to it with a different task ID. Each task handler may result in zero or more new tasks, which are collected and queued after the execution of all remaining

handlers. A task handler may result in communication with other endpoints through tasks and pages or through side-effects such as physical motion of a robot using hardware peripherals.

4.4.2 Endpoint Life Cycle

During creation of a node, all of the node's endpoints are initialized. The `get()` method is called for each `EndpointFactory` object provided by the configuration, returning constructed endpoint objects. Once the node begins execution of the main event loop, the `begin()` method is called on each endpoint, running any additional initialization code written by the end user.

When execution of a node ends, an endpoint object's `halt()` and `terminate()` methods are called successively. Halt notifies an endpoint that it will be destructed. This warning enables the endpoint to store any required state within its original parent factory. This preservation of state allows a second instance of the factory to reload the Python source code of the endpoint and construct a new instance with some preserved state. Terminate is used to notify an endpoint that it will not be reloaded.

4.4.3 Loops

Loops are an additional class of user defined module that allows code to run outside the main event loop. While the behavior of endpoints is triggered by handling a task, loops are run continuously in an execution loop internal to the component. This is typically implemented as a child thread, but the exact backing mechanism is not specified. A safely synchronized API to the parent node's thread is required. Loops are useful for reacting to external events such as communication from a separate node. A loop in its own thread can block on a communications socket and schedule tasks

when data is received. Figure 4.4 shows the interface available for communicating to and from a loop. While the loop is used for its asynchronous context, endpoint functionality such as task handling are still available. However, the user becomes responsible for synchronizing the synchronous and asynchronous internal state of a loop. The Python standard library provides the multiprocessing module to facilitate this manner of synchronization. Loops are constructed in the same fashion as endpoints and begin executing their routine when their `begin()` method is called. Loops also use the `halt()` method to shutdown their execution.

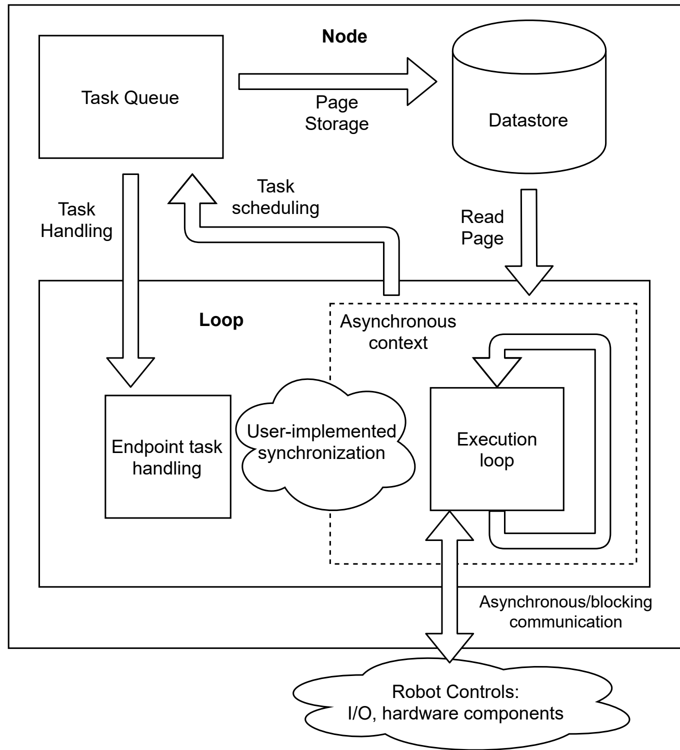

Figure 4.4: SNR Loop architecture

SNR does not provide an internal mechanism for communication between nodes. To serve this propose, the user can use standard communications loops, implement their own, extend the base communication loop. These components communicate between nodes through an external channel such as a pipe or network socket. The relevant implementations are discussed in section 6.4.

Chapter 5

EXAMPLE USAGE AND LIBRARY EVALUATION

This chapter provides examples of the usage of the SNR library for programming robots. Afterwords, the SNR library is examined using the evaluation framework presented in chapter 3. Instructions for installing SNR are shown in appendix B.

5.1 Example: Underwater Remote Operated Vehicle

This example presents a high level definition for the configuration of an underwater remote operated vehicle (UROV). By definition, UROVs impose the requirement of remote operation. This problem originally motivated the development SNR.

Figure 5.1 shows the flow of data in the UROV system. This flow of data directly translates to nodes and components within an SNR configuration.

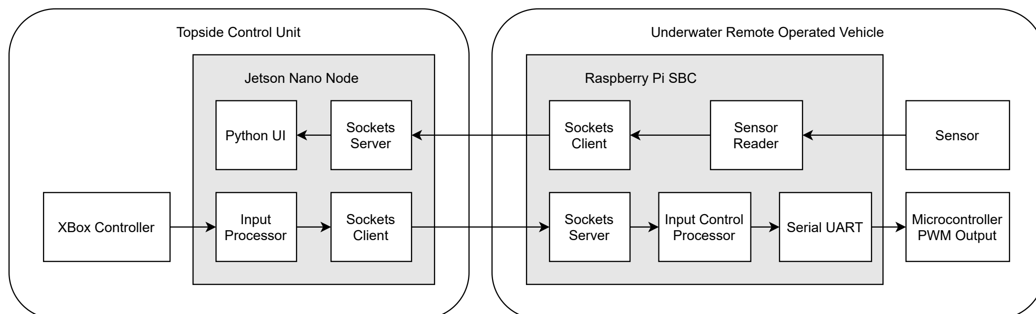

Figure 5.1: Data flow in example UROV system

In this partial implementation, two SNR nodes will be configured. Implementation of the factories and components is left as an exercise for the reader.

One node takes the role of the underwater vehicle and the other acts as a control unit from which the robot is operated. In practice each node would be run on a separate single board computer (SBC) such as a Raspberry Pi. The devices would be connected via an ethernet cable incorporated into a tether that supports the vehicle. The vehicle may have multiple pulse width modulation-controlled thrusters and other sensors. The control unit has a game controller for user input and a graphical user interface to report telemetry. In the first file, included in listing 5.1, the configuration for both nodes is defined. Factories for each component are described for each role. Each factory defined mirrors as component shown in figure 5.1. The order in which the components are defined does not affect the program but it helps to visualize the flow of data. Following the configuration, the `main()` method calls a runner that takes in the config as a parameter. When the program is run, the role of the computing device running the program is given as a command-line argument and the corresponding set of components are constructed.

Listing 5.1: Example configuration of a UROV system

```python
import sys
import snr
from urov import *

controller_input_sockets = snr.SocketsPair(
    server_tuple=("192.168.1.10", 9000))
telemetry_sockets = snr.SocketsPair(
    server_tuple=("192.168.1.11", 9001))
components: snr.ComponentsByRole = {
    "topside": [
        # Topside control pipeline
        XBoxControllerFactory("raw_controller_input"),
        InputProcessorFactory("raw_controller_input",
                              "controller_input"),
        controller_input_sockets.client("controller_input"),
        # Topside sensor pipeline
        telemetry_sockets.server(),
        SensorDisplayEndpointFactory("sensor_data")],
    "urov": [
        # Robot control pipeline
        controller_input_sockets.server(),
        ControlsProcessorFactory("controller_input",
                                 "motor_speed"),
        SerialConnectionFactory("motor_speed"),
        # On robot sensor pipeline
        SensorReaderFactory("sensor_data"),
        telemetry_sockets.client("sensor_data")],
}

if __name__ == "__main__":
    snr.CliRunner(components, sys.argv).run()
```

5.2 Moving Average Filter Endpoint

The next example presents an implementation of a moving average filter as an SNR endpoint as found in the `std_mod` submodule. A factory implementation is also provided in listing A.2. Listing A.3 demonstrates the testing utilities available in SNR.

5.2.1 Moving Average Filter Endpoint

Listing A.1 defines an endpoint class as it would be instantiated by an node. The `__init__()` method defined on line 6 calls the `Endpoint` super-class constructor. Then, the endpoint registers its task handlers in the `self.task_handlers` dictionary. Finally, the constructor stores the filter's state.

Following the constructor, the task handler `update_filter()` is defined on line 22. First, the type of the value from the task is checked. This ensures the rest of the task handler can be correctly type checked by static analysis tools. The state of the filter is updated and the task handler returns a task to store the filter's output.

The `begin()`, `halt()`, and `terminate()` methods are defined to satisfy the AbstractNode interface. This endpoint does not use the control flow provided by these methods. They are more useful when the endpoint must manage an external resource such as a socket for file descriptor.

5.2.2 Moving Average Filter Endpoint Factory

Listing A.2 defines the `MovingAvgFilterFactory` class. Factories primarily consist of constructors and `get()` methods. In this case, the constructor calls the `EndpointFactory` super constructor and initializes the backing state of the filter.

The super constructor takes a list of dependant modules that should be reloaded in order to regenerate Python bytecode for the endpoint. This enables the endpoint to incorporate changes to its source code without restarting the node. The backing state of the endpoint is maintained by the factory and provided to the new instance of the endpoint. The `get()` method instantiates the endpoint and returns it to the node.

5.2.3 Moving Average Endpoint Tests

Listing A.3 shows an `SNRTestCase` class that tests the `MovingAvgFilterEndoint`. The test runs a test node on line 17. The `with` statement on line 16 ensures that the expectations of the expector are satisfied regardless of how the node terminates. The Node is configured with three components. The list replayer sequentially stores the values from the list passed to it with the data key `input_data`. The filter endpoint under test will receive the data storage task from the list replayer and run its task handler, producing data with the `output_data` key. The expector endpoint wraps the expector and expects a task for each page of data stored. The node is terminated once the expector is satisfied.

5.3 Example: Lunar Module Landing Simulation

This example tackles the classic problem of landing the lunar module on the Moon. In this algorithmic challenge, students must select how much fuel to use at each stage of the descent. In this hypothetical assignment, students are instructed to develop a Q-learning model for controlling the lunar module. The implementation and training of the Q function are left as an exercise. Listing A.4 shows the implementation including an endpoint, loop, associated factories, and configuration.

5.4 Example: Benchmark Test

This section describes a synthetic benchmark designed to show the task throughput of an SNR node running on a device. This example also shows the testing APIs provided by the SNR library. Listing A.5 contains the implementation of the benchmark as component definitions and test cases. First, an endpoint and an associated factory are defined. Then, an `SNRTestCase` class is defined. The test class contains two test methods. The first, `test_stress_endpoint_init()` ensures that the previously defined endpoint operates as intended. The second, `test_stress_endpoint()`, allows the node to run to a predetermined period of time to measure the throughput of the node. A node is constructed without a runner so its profiling information is available after the test. This configuration includes the stressing endpoint as defined at the beginning of listing A.5. The standard modules `TimeoutLoop` and `StopwatchEndpoint` are also included. The `TimeoutLoop` schedules the terminate task after the benchmarking period. The `StopwatchEndpoint` records the time at which the terminate task is handled. The stressing endpoint schedules a “stress” task, and then later handles the task by scheduling a task to create another endpoint using its factory. By exponentially growing the number of endpoints, the rate at which tasks can be handled and endpoints created is measured. This example differs from normal usage of the library in two ways: the factory contains more logic than it would normally and the endpoint creates additional endpoints.

Listing 5.2 shows the output of running the test cases shown in listing A.5. The output includes the results of the benchmark, profiling information from the node, and the runtimes of the test cases. Over the two second duration of the benchmark, hundreds of endpoints were created. The profiling data from the node is printed using the `node.profiler.dump()` method. The information printed shows that the most

time was spent handling the stress task. The `add_component` and terminate tasks occupied a minority of the time.

Listing 5.2: Benchmark shell output

```
$ python3 tests/bench/bench_stress_endpoint_fac.py

Stressed with:
        547 stressor endpoints
        553 stressor factory calls
        Terminate expected at 2000 ms
        2008.915 ms terminate handled
        2010.324 ms total time

Times called,          Avg runtime, Task/Loop type,
      149878 x  2.317 us -> 17.281%: TaskType.event(stress):__main__
         552 x 30.942 us ->  0.850%: TaskType.event(add_component):
   snr.core.node_core.node_core_endpoint
           1 x 39.577 us ->  0.002%: TaskType.terminate(Timeout):snr
   .core.node_core.node_core_endpoint
           1 x  2.623 us ->  0.000%: TaskType.terminate(Timeout):snr
   .std_mods.utils.stopwatch_endpoint

2010.66 ms: __main__.BenchStressEndpointFac.test_stress_endpoint
. 19.66 ms: __main__.BenchStressEndpointFac.
   test_stress_endpoint_init
.
----------------------------------------------------------------------

Ran 2 tests in 2.031s

OK
```

This benchmark can also be used to compare Python interpreter implementations. Notably, PyPy, an implementation featuring just-in-time compilation and optimizations, performs much faster than the defacto standard CPython.

5.4.1 Game Controller Over Sockets

The next example defines two nodes. One node reads user input from a connected game controller and sends the result to the other over a TCP sockets connection. In this example, both nodes are run on the same device, however this is not a limitation of the system. The nodes could be run on separate computing devices connected on and IP network. This example uses a Raspberry Pi 4 as configured in appendix B. The PyGame library is also required. A "Steam Controller" game controller is used in this example, but PyGame is compatible with other models of controller. For other controllers, the input mapping used in this example will likely be incorrect. PyGame can be installed via `atp` like so:

```
sudo apt install python3-pygame
```

Listing 5.3 defines the configuration of the system. The game controller and Raspberry Pi 4 used in this example are shown in figures 5.3 and 5.2 respectively. Matching the configuration in listing 5.3, the nodes can be run with the terminal commands `python3 main.py receiver` and `python3 main.py input_device`. The implementations of ControlProcessorFactory, ControlProcessorEndpoint, ControllerLoopFactory, ControllerLoopFactory, InputMappingFactory, and InputMappingEndpoint are shown in listings A.6 through A.11. The `ControllerLoopFactory` produces a `ControllerLoop` that reads raw input from the controller. The `InputMappingEndpoint` produced by the `InputMappingFactory` maps the raw input to human-readable button and axis states. This data is then sent over the sockets connection via the sockets client. The

Figure 5.2: Game controller read from using an SNR loop

`TimeoutLoopFactory` terminates the demonstration after twenty seconds for convenience. The `receiver` node receives the controller input and checks which buttons have changed state since data was last received. The `PrinterEndpoint` prints when a specific task is handled. The output of the system is shown in figure 5.4

Figure 5.3: Raspberry Pi 4 running SNR nodes

Listing 5.3: Example configuration using game controller and sockets

```python
import sys
import snr

from control_processor_factory import ControlProcessorFactory
from controller_loop_factory import ControllerLoopFactory
from input_mapping_factory import InputMappingFactory

sockets = snr.SocketsPair(server_tuple=("localhost", 9000))
components: snr.ComponentsByRole = {
    "input_device": [
        ControllerLoopFactory("raw_controller_data"),
        InputMappingFactory("raw_controller_data",
                            "controller_data"),
        sockets.client(["controller_data"]),
        snr.TimeoutLoopFactory(seconds=20),
    ],
    "receiver": [
        sockets.server(),
        ControlProcessorFactory("controller_data"),
        snr.PrinterEndpointFactory([
            (snr.TaskType.event, "button_pressed"),
            (snr.TaskType.event, "button_released"),
        ])
    ]
}

if __name__ == '__main__':
    snr.CliRunner(components, sys.argv).run()
```

```
pi@raspi4:~/SNR/examples/controller $ python3 main.py receiver
(event, 'button_pressed'): ['d_pad_tap']
(event, 'button_released'): ['right_stick_press']
(event, 'button_pressed'): ['right_trigger']
(event, 'button_released'): ['right_trigger']
(event, 'button_released'): ['d_pad_tap']
(event, 'button_pressed'): ['left_trigger']
(event, 'button_released'): ['left_trigger']
(event, 'button_pressed'): ['right_stick_press']
(event, 'button_released'): ['right_stick_press']
(event, 'button_pressed'): ['right_stick_press']
(event, 'button_released'): ['right_stick_press']
(event, 'button_pressed'): ['A']
(event, 'button_released'): ['A']
(event, 'button_pressed'): ['B']
(event, 'button_released'): ['B']
(event, 'button_pressed'): ['X']
(event, 'button_released'): ['X']
(event, 'button_pressed'): ['Y']
(event, 'button_released'): ['Y']
^CInterrupted by user, exiting
pi@raspi4:~/SNR/examples/controller $
```

```
pi@raspi4:~/SNR/examples/controller $ sudo python3 main.py input_device
pygame 1.9.4.post1
Hello from the pygame community. https://www.pygame.org/contribute.html
Found joysticks: [<Joystick object at 0x7f8112ad98>]
Using joystick: Steam Controller with 4 axes and 21 buttons and 2 d pads
Loop halted
Loop terminated
pi@raspi4:~/SNR/examples/controller $
```

Figure 5.4: Terminal output of controller input changes across sockets connection

Chapter 6

LIBRARY IMPLEMENTATION

This chapter discusses implementation specific details of the SNR library. The SNR library Python package contains 5 submodules: `prelude`, `core`, `std_mods`, `utils` and `tests`. Figure 6.1 shows dependencies between the library's submodules using a directed acyclic graph. The prelude defines type aliases, abstract class definitions, and interfaces (Protocol) definitions later used in the `core` and `std_mods` submodules. The `core` submodule contains the logic that implements important SNR features including the node and `Endpoint` and `ThreadLoop` classes inherited by end users. The `std_mods` submodule contains standard endpoints and associated classes that can be reused in end user SNR programs. The `utils` submodule contains testing utilities and relies on the prelude and core. The `tests` submodule contains unit tests that check for the correct behavior of the preceding modules.

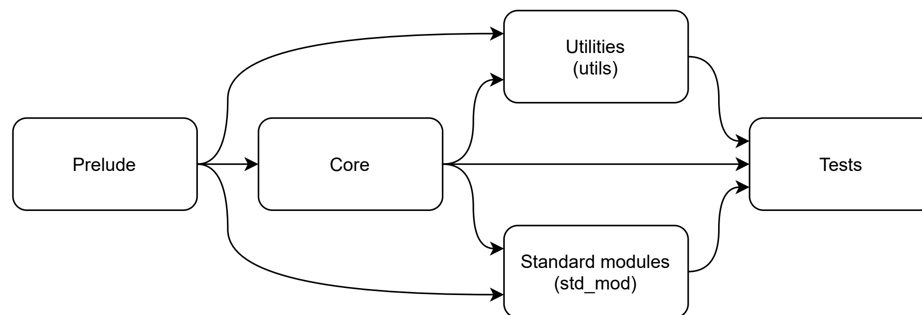

Figure 6.1: SNR library submodule dependency directed acyclic graph

6.1 Code Style Principles

SNR's code style furthers the goal of accessibility for students by using type annotations throughout the library. This enables students to know the type of an object

with certainty. This also formalizes the API that users interact with. Typed interfaces enable static analyzers such as mypy to check for correctness within the type system prior to runtime [28]. Mypy is used to check for type system-level correctness of the SNR library implementation. Pytype is another type checking analyzer for Python developed by Google [29]. Pytype focuses on accepting the advantages of duck typing in Python by inferring types later in its analysis.

SNR encourages testing. Utility classes for testing are included. The SNR library implementation includes tests for many of its components. These tests also serve as example usages of the components they test.

6.2 Prelude

The prelude contains definitions for interfaces and data types commonly used throughout the SNR library. This includes Abstract Base Classes (ABCs) that force implementation of an interface. In the context of ABCs in Python, an interface requires concrete implementations of abstractly specified method signatures by inheriting subclasses. If the end user does not correctly implement the requirements of an ABC, an error message will be shown. This behavior is complemented by static type checking. The most important data types defined in the prelude are the `Page` and `Task` classes.

6.2.1 Page and Task

The `Page` structure is implemented as a Python data class. Data classes automatically implement some methods [30]. The definition of a dataclass provides type information to static type checkers, which can be taken advantage of to show correctness. On top of dataclasses, a library, `dataclasses_json`, is used to automatically generate

serialization and deserialization methods for a dataclass. This decouples the implementation of these methods from the definition of the dataclasses. Similar to `Page`, `Task` is implemented as a Python dataclass and uses `dataclasses_json`.

6.2.2 Type Aliases

Numerous type aliases are defined for working with task objects. The type aliases `TaskName` and `TaskType` are defined to represent identifying fields of a Task object. The `TaskId` type alias as described in section 4.4.1 is defined as the union of a `TaskType` and a tuple of a `TaskType` and `TaskName`.

The type aliases `SomeTasks`, `TaskHandler`, `TaskHandlerMap`, and `TaskScheduler` are defined to represent interfaces that handle tasks. `SomeTasks` is an alias for the union of a `Task` object, a list of `Task` objects, and None. `TaskHandler` is an alias for a callable function that takes in a `Task` object and the `TaskID` that it was matched with. It returns `SomeTasks`. `TaskHandlerMap` is the alias of the object an endpoint must provide to register tasks. A `TaskScheduler` is method that takes `SomeTasks` as a parameter and enqueues them in the task queue of an endpoint's parent node.

Additional type aliases are defined to ease user comprehension. `DataDict` is a type alias for a dictionary that stores Pages by string keys. By specifying the datastore for nodes as a simple Python dictionary, the burden of synchronizing read and write operations is placed on the Endpoint-Node API (the built-in dictionary does offer some synchronization safety). This safely requirement restricts access to the data structure to operations in the main event loop thread.

6.2.3 Protocols

Protocols are dynamic typing feature of Python that define a functional interface. A protocol acts as a type that includes objects that implement the required methods without referencing the protocol explicitly . Protocols are a form of structural typing where classes that implement the required methods of a protocol are considered a subtype of the protocol at runtime [31]. This works without references to the protocol itself in the definition of the implementing class.

Serialization methods generated by the `dataclasses_json` library are wrapped in the methods required for the `Serializable` protocol. The `Page` and `Task` classes are subtypes of the `Serializable` protocol.

6.2.4 Interfaces

The prelude defines ABCs and Protocols that define the application programming interface that end users interact with and extend. Most importantly, the `AbstractNode`, `AbstractEndpoint`, and `AbstractLoop` classes are defined.

6.2.5 Precursors

The `AbstractConfig` and `AbstractRunner` define abstract base classes for classes that always precede the instantiation of a node. The runner selects components from the config and wraps execution of the node. An interface for an additional variant of a runner is defined by `AbstractMultiRunner`, which intends to wrap the execution of multiple nodes within one runner. This can be useful for testing multi-node systems on one device. `AbstractConfig` requires the `get(role: Role) -> List[AbstractFactory]` and `get_profiler() -> Optional[AbstractProfiler]` meth-

ods. The first retrieves the factory objects needed to construct the components of a node specified by the provided role. The latter permits the config to provide or omit a diagnostic profiler object.

The `AbstractComponent` ABC precedes Endpoints and Loops, mandating that they implement the component interface, generally referred to as the endpoint interface:

- `begin() -> None`
- `join(self) -> None`
- `halt(self) -> None`
- `terminate(self) -> None`
- `store_page(self, page: Page) -> None`

6.2.6 AbstractNode

The `AbstractNode` interface is important because it defines the interface for user implemented components. By exposing on the abstract class to end user programs, implementation details of the `Node` class can be hidden or the implementation can be replaced. Likewise, the `AbstractEndpoint` class provides an interface to endpoints for classes implementing a node. While the included node implementation does not treat loops differently than other endpoints, the `AbstractLoop` class requires implementing methods needed to operate a loop. Loop backing implementations should inherit from this abstract class. This is demonstrated in chapter 5.

The `AbstractNode` base class specifies that a node implementation must have the following properties:

- role: Role
- config: AbstractConfig
- mode: Mode
- components: Dict[ComponentName, AbstractComponent]

The following methods are used by the runner to control the node's execution:

Listing 6.1: Pseudo code interface for node control flow

```
1 loop() -> None
2 set_terminate_flag(reason: str) -> None:
3 terminate() -> None:
4 is_terminated() -> bool:
```

Task execution is specified by the following methods:

Listing 6.2: Pseudo code interface for task execution

```
1 get_task_handlers(task: Task) -> List[Tuple[TaskHandler, TaskId]]
2 handle_task(handler: TaskHandler, task: Task, key: TaskId
3             ) -> Optional[List[Task]]
4 execute_task(t: Task) -> None:
```

As described in section 4.4.1, the node executes tasks in two phases. First, it queries all its endpoints using `get_task_handlers()`. Then, each found task handler is wrapped `handle_task()` to collect runtime diagnostics.

The following methods provide an interface for endpoints create and schedule tasks:

Listing 6.3: Pseudo code interface for task scheduling

```
1 schedule(t: SomeTasks) -> None
2 task_store_page(page: Page) -> Task
3 task_store_data(key: DataKey, data: Any, process: bool) -> Task
```

Finally, the following methods provide an interface for endpoints create, write, and read datastore pages:

Listing 6.4: Pseudo code interface for datastore access

```
1 page(key: DataKey, data: Any, process: bool) -> Page
2 store_data(key: DataKey, data: Any, process: bool) -> None
3 get_page(key: DataKey) -> Optional[Page]
4 get_data(key: DataKey) -> Optional[Any]
```

Each of these methods wraps or unwraps arbitrary data with a page. The process parameter denotes whether the page requires an additional processing task to be scheduled after the page has been stored in the datastore. This provides a second point in time for endpoints to receive data corresponding to a page. First, the store page task includes the page in the value list of the task. Second, the process data task occurs once the page is available from the datastore. Thus two task handlers can be run sequentially as the result of one page.

These methods are implemented by the node because the node provides the creation timestamp for the page and the name of the node is recorded as the origin of the page. This origin field is useful when transmitting pages between nodes.

6.2.7 TaskQueue

The task queue is defined separately from the node to force an interface where synchronization can occur. This is necessary because loops belonging to a node may have access to the node from outside the context of the main thread. An asynchronous API for scheduling tasks is the main form of communication from asynchronous loops to the main thread event loop context.

The task queue is implemented as a a standard Python collections deque. Synchronization is provided by the standard library implementation.

6.2.8 AbstractEndpoint

`abstract_endpoint` defines an abstract class that all endpoints must fully implement. If the end user fails to implement a member required by the abstract definition, an error is displayed at startup and runtime is aborted. End users indirectly implement this class by inheriting from `Endpoint`.

6.2.9 AbstractLoop

`AbstractLoop` defines an abstract class that all loops must fully implement. This includes `setup()`, `loop()`, `set_terminate_flag()`, and `is_terminated()`. This `abstract_connection`

6.2.10 AbstractFactory

Factories provide a degree of separation between configs and components because endpoints and loops should not be run until the node enter the main event loop. To prevent this, configs pass factories to the node and the node delegates the construction of the component to the factory. Keeping information about the node being construct also permits "hot-reloading" of components. This can be accomplished because factories keep a reference to the Python module containing the component they construct. This module reference is passed to `importlib.reload()` and the source code is re-read and re-compiled. Then, the component can be reconstructed for the node.

6.3 Core

This module contains the core implementation of the library. This includes concrete implementations of abstract classes defined in the library prelude.

6.3.1 Node

The `Node` class implements the main logic that interconnects user components. It houses the datastore and task queue. All components belonging to a node are maintained in a dictionary keyed by the component's name.

6.3.2 Endpoint

The `Endpoint` class provides the core synchronous task handler interface that characterizes the SNR model of execution. End users should have their endpoints inherit from `Endpoint`.

6.3.3 ThreadLoop

`ThreadLoop` implements the `AbstractLoop` abstract base class using the Python standard library's `threading` module. End users should have their endpoints inherit from the `ThreadLoop` class.

6.4 Standard Modules

The `std_mod` module contains implemented components that can be utilized in an existing usage of the library. The module is broken down into four submodules: `comms`, `filters`, `io`, and `utils`.

6.4.1 Communications

The `comms` module provides implementations of loops that communicate pages between nodes. Each of included classes implements a `CommsLoop`. The comms loop is provided an abstract connection when it is constructed. The comms loops included with the library communicate over Unix-style pipes and TCP sockets. A connection using pipes is useful for test environments where multiple nodes are run by the same runner. A connection using TCP is useful for devices connected by an IP network. When transmitting pages over a connection, the page is serialized into JSON text.

6.4.2 Filters

An implementation of moving average filter is provided as an endpoint. This implementation is demonstrated in listing A.1

6.4.3 Input/Output

The `io` submodule provides components for human interfaces and disk operations. A system for communicating from a shell terminal is provided in the `CommandProcessorEndpoint`, `CommandReceiverFactory`, and `RemoteConsole` classes. The `CommandReceiverFactory` constructs both the CommandProcessorEndpoint and a communications loop that

receives commands over a TCP connection. The `RemoteConsole` is a standalone application that connects to the node via the communications loop. The console sends arbitrary commands entered by the user and the `CommandProcessorEndoint` on the node responds. The endpoint interprets the following commands:

- `exit`: Terminate the node and console
- `task`: Schedule a task
- `reload`: Reload 'all' or a specific endpoint
- `list`: List all 'commands' or 'endpoints'
- `dump`: Dump data from 'profiler' or 'datastore'
- `help`: List all commands

`RecorderEndpoints` and `ReplayerLoops` write and read pages and data from disk respectively. These standard components are useful for simulating scenarios during testing.

6.5 Testing Utilities

The implementation of SNR includes a test suite that tests for correctness of the implementation and demonstrates testing practices. A number of classes are provided to ease writing tests. First, the `SNRTestCase` class inherits from the standard unittest module's `TestCase`. `SNRTestCase` provides quick access to "expectors", a test runner, temporary file allocation, and SNR specific assertion statements.

Expectors are way of specifying expected behavior. Expectors are constructed with a list of expectations that may or may not be ordered. `ExpectorEndpoints` are

implemented so expected tasks can be checked for during tests. Tests can assert that an expector's expectations have been satisfied. Coupled with `TestRunner`s, `ExpectorEndpoint`s allow user implemented components to be tested in the same way a node is run.

Listing A.3 shows the usage of the `SNRTestCase` class, a `ListReplayerLoop`, and expectors.

Chapter 7

LIBRARY EVALUATION

7.1 Prerequisites

SNR is designed to avoid prerequisites of other robotics libraries such as ROS. By existing as an open-source pure Python library, it is available in a compact package. The entire SNR library source code occupies less than 500 KB of disk space (depending on the file system). This makes it easy to develop and use on inexpensive computers such as the Raspberry Pi. SNR requires a Python interpreter version of at least 3.6.

SNR requires the Pip Python package manager to install itself. This process also needs packages from the Python Package Index (PyPI) including `setuptools` and `wheel`. The SNR library's immediate dependencies include `dataclasses` and `dataclasses_json`. Python versions 3.7 and 3.6 require the `typing_extensions` package to handle typing features such as protocols.

The SNR library source also uses the following packages to improve development: `pdoc>=6.1`, `nox`, `pytest`, `pytest-timeout`, `mypy>=0.800`, and `flake8>=3.8`. They are useful for developing the library but are not needed for use.

7.2 Discoverability

Discoverability represents a shortcoming of SNR. Limited web presence and documentation make it hard for teachers and students to find. In addition to this, prebuilt

binaries for SNR are not yet available, making the source code repository the sole source.

7.3 Error Messages

SNR uses Python ABCs to require implementation of some important interfaces. Listing 7.1 shows the message shown to the user when they attempt to run a program that uses SNR without correctly implementing a used ABC.

Listing 7.1: Example runtime error message

```
1 Traceback (most recent call last):
2   File "/home/student/SNR/tests/test_loop.py", line 121, in
3   test_invalid_construction_fails
4     construct(InvalidLoopNoHalt)
5   File "/home/sfshaw/git/SNR/tests/test_loop.py", line 107, in
6     construct
7     invalid_constructor(fac, node, None)
8 TypeError: Can't instantiate abstract class InvalidLoopNoHalt with
9     abstract methods halt
```

The benefit of error messages like that shown in listing 7.1 is the error, offending class, and reason are stated on the final line. The correct location of the call to the broken class is also given. Despite this brevity, the message presents other issues. Inexperienced students may be intimidated by the mention of an abstract class when they have written a concrete class definition. While this inconsistency is correct in the interpreter's perspective, students with limited experiences with object-oriented programming in Python may be confused. This could be remedied by checking of types at runtime coupled with friendlier error messages.

7.4 Openness

The availability of SNR's source code for inspection, extension, and modification make the library very open. While prebuilt binaries are not yet available on package repositories, all users already have the source code from installation.

7.5 Familiarity

SNR should feel familiar to students that have recently taken an object oriented programming (OOP) course and have some experience with Python3. SNR's implementation and API draw on concepts core to OOP classes. Students should feel like they are a brushing up on OOP concepts.

7.6 Documentation

SNR's documentation is incomplete. Tooling for automatically generating hypertext based documentation is experimentally supported using the `pdoc` tool, an optional development dependency [32]. `pdoc` generates web pages based on "doc strings" embedded in Python source code. `pdoc` can be configured to run as part of continuous integration (CI). With such a CI setup, documentation can be automatically published. IDEs such as Visual Studio Code can also source information from doc strings. An additional benefit of doc strings is their consolidation with source code. Coupled with CI, online documentation can be kept in sync with library releases.

7.7 Misuse Resistance

As shown in section 7.3, SNR can raise exceptions to prevent misuse. Despite this, no additional guards to prevent the user from misusing task scheduling and handling. If students do not understand the serial nature of the node's event loop, they may find that tasks are not delivered when they expect.

7.8 Growth Paradigms

For students with one course of experience in Python, SNR should promote intermediate topic in Python such as type checking, and packaging. By learning about type annotations, students can gain a more complete understanding of the behavior of the Python interpreter. The use of type hints in the implementation of the SNR library extends the reasoning students can preform about the correctness of a program. This is a valuable skill. A rudimentary understanding of type checking can prepare students for the study of discrete structures and more strictly typed programming languages.

SNR prepares students for robotics libraries used in industry by introducing modular patterns. Students grasping event driven architectures will have an easier time understanding patterns such as publisher-subscriber in ROS.

Chapter 8

FUTURE WORK

The work presented in this thesis is open for extension in two areas: improvements to the SNR library and development of curriculum utilizing the SNR library.

8.1 Library Improvements

The SNR software library could be improved. In its current state, single-threaded operation is well supported. While threaded loops are supported, parallelism is limited at the level of the Python interpreter. The global interpreter lock (GIL) prevents more than one thread running at once [33]. The multiprocessing module permits parallelism through additional interpreter processes. Additional specification of SNR is needed to safely synchronize data between threads.

Alternatively, additional loop implementations could provide synchronous backing to the loop interface. This could be accomplished by scheduling loop iterations as tasks so they are executed synchronously on the main thread.

Another possible library improvement could cache component task handlers for more efficient look ups when a node executes a task. This would require a consolidated task handler data structure and synchronization when components modify their task handlers.

8.1.1 Specification Formalization

The specification of the software framework could be further formalized. This would enable additional implementations SNR to be compatible. For example, communication between SNR nodes of different implementations must used a specified communication format.

8.1.2 Publishing

To ease installation for students, SNR should be made available on the PyPi package repository. This would mark release readiness for the library and enable students to start working with SNR quickly. Until then, the library source must be downloaded and installed, requiring two terminal commands instead of one.

8.2 Evaluation Framework

The framework for evaluation educational tools presented in chapter 3 could be improved by developing quantitative methods for assessing software. Then, the framework could be used to make evaluations on a larger scale.

8.3 Curriculum Development

Educational curriculums utilizing SNR could be developed. These course curriculums should outline learning goals and include assignments that follow a incremental approach in building components for SNR.

8.3.1 Introductory Robotics

Pursuant to its purpose, SNR could be used in an undergraduate course introducing students to programming robots. Robot components can be mapped to SNR component modules. This supports focused lessons and assignments. For example, students can develop different control systems on a weekly cadence. Given a background in Python, students can begin prototyping components without needing to set up additional tools. Students can start with a simple proportional controller. Later, students can build a perception based controller as a drop-in replacement. This modular approach shows students the advantages of modular software architectures within the framework of the course.

8.3.2 Advanced Programming Topics

The implementation of SNR can serve as a case study for systems programming topics. Modular architectures in robotics software frameworks are interesting systems programming problems. Disparate components need to communicate with each other with coupling of components. For implementation, students can explore multiprocessing and synchronization to distribute computation across components. Following the introduction of synchronization primitives, higher level constructs, such as multi-producer queues, can be demonstrated.

The modularity of SNR's internal architecture enables assignment-based implementations of internal components such as the datastore or node core endpoint. The available source code is provided for introspection by students.

Chapter 9

CONCLUSION

This thesis presents SNR, a software library for undergraduate robotics education that addresses core concepts in robotics systems, such as modular components. This work was motivated by the gap between students' introductory experiences in computer science and the skills needed to make use of industry standard robotics libraries. The SNR library introduces students to robotics as a familiar Python package without additional build systems. The complete source code of the library is freely available [34].

While having reached an initial level of feature completeness, the SNR library falls short in documentation and user experience as evaluated in chapter 7. These shortcomings can be improved upon in future work that improves the library implementation and develops curriculum.

With an introduction to paradigms native to industry standard robotics tools, students can accelerate their course of study in the field of robotics.

BIBLIOGRAPHY

[1] M. Hagele, “Double-digit growth highlights a boom in robotics [industrial activities],” *IEEE Robotics Automation Magazine*, vol. 24, no. 1, pp. 12–14, 2017.

[2] M. Bowen, “Beware hospitality industry: the robots are coming,” *Worldwide Hospitality and Tourism Themes*, vol. 10, no. 6, pp. 726–733, 2018.

[3] M. A. Gennert and G. Tryggvason, “Robotics engineering: A discipline whose time has come,” *IEEE Robotics Automation Magazine*, vol. 16, no. 2, pp. 18–20, 2009.

[4] M. Gennert, W. Michalson, and M. Demetriou, “A robotics engineering ms degree,” in *2010 Annual Conference & Exposition*, 2010, pp. 15–85.

[5] Z. J. Wood, J. Clements, Z. Peterson, D. Janzen, H. Smith, M. Haungs, J. Workman, J. Bellardo, and B. DeBruhl, “Mixed approaches to cs0: Exploring topic and pedagogy variance after six years of cs0,” in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 2018, pp. 20–25.

[6] G. Cooper, “Incorporating a raspberry pi into a computer information systems initial course,” in *2017 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2017, pp. 1–5.

[7] S. Krishnamurthi and K. Fisler, “Programming paradigms and beyond,” *The Cambridge Handbook of Computing Education Research*, vol. 37, 2019.

[8] A. H. El-Mousa and A. Al-Suyyagh, “Embedded systems education for multiple disciplines,” *Journal of computer science*, vol. 6, no. 2, pp. 186–193, 2010.

[9] B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services.* " O'Reilly Media, Inc.", 2018.

[10] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[11] M. Shannon, "Pep 484 – type hints," https://www.python.org/dev/peps/pep-0484/.

[12] C. I. o. T. NASA jet Propulsion Laboratory, "Meet the open-source software powering nasa's ingenuity mars helicopter," https://www.jpl.nasa.gov/news/meet-the-open-source-software-powering-nasas-ingenuity-mars-helicopter, 2021.

[13] I. Open Source Robotics Foundation, "Open robotics," https://www.openrobotics.org/company.

[14] K. Conley, "Ros/introduction–ros wiki," *ROS Wiki*, vol. 3, 2012.

[15] J. M. O'Kane, *A gentle introduction to ROS.* Jason M. O'Kane, 2014.

[16] C. I. of Technology, "F'software framework," https://nasa.github.io/fprime/Architecture/FPrimeArchitectureShort.pdf, 2018, accessed: 2021–08-02.

[17] C. I. o. T. NASA jet Propulsion Laboratory, "F´ flight software & embedded systems framework," https://nasa.github.io/fprime/.

[18] F. Inspiration, R. of Science, and T. (FIRST), "Wpilib github," https://wpilib.org/.

[19] ——, "What is first robotics competition?" https://www.firstinspires.org/robotics/frc/what-is-first-robotics-competition.

[20] ——, “roborio introduction,” https://docs.wpilib.org/en/stable/docs/software/roborio-info/roborio-introduction.html.

[21] ——, “First robotics competition control system,” https://wpilib.org/.

[22] D. Kushner, “The making of arduino,” *IEEE spectrum*, vol. 26, 2011.

[23] B. Fry and C. Reas, “Overview. a short introduction to the processing software and projects from the community.” https://processing.org/overview/.

[24] “Arduino education,” https://www.arduino.cc/education/about.

[25] D. Willingham, *Why Don’t Students Like School?: A Cognitive Scientist Answers Questions About How the Mind Works and What It Means for the Classroom*. 111 River St, Hoboken NJ, 07030: Wiley, 2021. [Online]. Available: https://books.google.com/books?id=zCQqEAAAQBAJ

[26] “Language reference,” https://www.arduino.cc/reference/en/.

[27] C. S. Dweck, “Mindsets and math/science achievement,” 2014.

[28] the mypy project, “mypy - optional static typing for python,” http://mypy-lang.org/.

[29] Google, “pytype - a static type analyzer for python code,” https://google.github.io/pytype/.

[30] E. V. Smith, “Pep 557 – data classes,” https://www.python.org/dev/peps/pep-0557/, 2017.

[31] I. Levkivskyi, J. Lehtosalo, and Łukasz Langa, “Pep 544 – protocols: Structural subtyping (static duck typing),” https://www.python.org/dev/peps/pep-0544/.

[32] M. Hils, “pdoc,” https://github.com/mitmproxy/pdoc.

[33] D. Beazley, “Understanding the python gil,” in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[34] S. Shaw, “Snr,” https://github.com/sfshaw-calpoly/SNR.

APPENDICES

Appendix A

CODE LISTINGS

Listing A.1: Standard implementation of a moving average filter endpoint

```
from snr.core import *
from snr.prelude import *


class MovingAvgEndpoint(Endpoint):
    def __init__(self,
                 factory: EndpointFactory,
                 parent: AbstractNode,
                 name: str,
                 input: DataKey,
                 output: DataKey,
                 filter: MovingAvgFilter,
                 ) -> None:
        super().__init__(factory, parent, name)
        self.input = input
        self.output = output
        self.task_handlers = {
            (TaskType.store_page, self.input): self.update_filter,
        }
        self.filter = filter

    def update_filter(self, task: Task, key: TaskId) -> SomeTasks:
        if not (task.name == self.input and
                isinstance(task.val_list[0], Page) and
```

```
25                 (isinstance(task.val_list[0].data, float) or
26                  isinstance(task.val_list[0].data, int))):
27             self.warn("Invalid data from %s", task)
28             return None
29         self.filter.update(task.val_list[0].data)
30         return tasks.store_page(self.page(self.output, self.filter.
    avg()))
31
32     def begin(self) -> None:
33         pass
34
35     def halt(self) -> None:
36         pass
37
38     def terminate(self) -> None:
39         pass
```

Listing A.2: Standard implementation of a factory for MovingAverage-FilterEndpoint

```
1 import collections
2 from typing import Deque
3
4 from snr.core import *
5 from snr.core.core_utils import moving_avg_filter
6 from snr.prelude import *
7
8 from . import moving_avg_endpoint
9
10 DEFAULT_FILTER_LENGTH: int = 3
11
12
13 class MovingAvgFilterFactory(EndpointFactory):
14     def __init__(self,
```

```
15                  input: DataKey,
16                  output: DataKey,
17                  length: int,
18                  ) -> None:
19         super().__init__([moving_avg_endpoint, moving_avg_filter])
20         self.input = input
21         self.output = output
22         self.backing_deque: Deque[float] = collections.deque(maxlen=
    length)
23 
24     def get(self,
25             parent: AbstractNode,
26             ) -> moving_avg_endpoint.MovingAvgEndpoint:
27         filter = moving_avg_filter.MovingAvgFilter(self.
    backing_deque)
28         return moving_avg_endpoint.MovingAvgEndpoint(
29             self,
30             parent,
31             "moving_avg_filter",
32             self.input,
33             self.output,
34             filter)
```

Listing A.3: Example usage of testing facilities using SNRTestCase and Expector

```
1 from snr import *
2 
3 
4 class TestMovingAvgEndpoint(SNRTestCase):
5 
6     def test_moving_avg_endpoint(self):
7         input_data = "input_data"
8         output_data = "filtered_Data"
```

```
9
10        expectations = {
11            (TaskType.store_page, input_data): 4,
12            (TaskType.process_data, input_data): 4,
13            (TaskType.store_page, output_data): 4,
14            (TaskType.process_data, output_data): 4,
15        }
16        with self.expector(expectations) as expector:
17            self.run_test_node([
18                ListReplayerFactory([0, 1, 2, 4], input_data),
19                MovingAvgFilterFactory(input_data, output_data, 2),
20                ExpectorEndpointFactory(expector,
21                                        exit_when_satisfied=True),
22            ])
```

Listing A.4: Example configuration of lunar module landing simulation

```
1 from typing import Any, Callable
2
3 import snr
4 from snr import (AbstractNode, Endpoint, EndpointFactory,
    LoopFactory,
5                  ThreadLoop, tasks)
6 from snr.prelude import *
7
8
9 class RadarSensorLoop(ThreadLoop):
10
11    fuel: int
12    altitude: float
13    velocity: float
14
15    def __init__(self,
```

```
16                  factory: LoopFactory,
17                  parent: AbstractNode,
18                  fuel: int,
19                  ) -> None:
20          super().__init__(factory, parent,
21                           "radar_sensor_endpoint",
22                           max_tick_rate_hz=0.5)
23          self.fuel = fuel
24          self.altitude = 50.0
25          self.velocity = 0.0
26 
27      def setup(self) -> None:
28          self.store_data("thruster_data", 0)
29 
30      def loop(self) -> None:
31          page = self.get_page("thruster_data")
32          assert isinstance(page, Page) and isinstance(page.data, int)
33          thruster_value: int = page.data
34 
35          self.velocity += -9.8 + thruster_value * 0.1
36          self.altitude += self.velocity
37 
38          if self.altitude <= 0:
39              # *Attempted Landing*
40              if self.velocity < -1.0:
41                  raise Exception("High velocity impact")
42              else:
43                  self.schedule(tasks.terminate("Landed Safely"))
44                  self.set_terminate_flag()
45 
46          self.store_data("radar_data", (self.altitude, self.velocity)
    )
```

```
47
48      def halt(self) -> None:
49          pass
50
51      def terminate(self) -> None:
52          pass
53
54
55  class RadarSensorLoopFactory(LoopFactory):
56
57      initial_fuel: int
58
59      def __init__(self, initial_fuel: int) -> None:
60          super().__init__()
61          self.initial_fuel = initial_fuel
62
63      def get(self, parent: AbstractNode) -> RadarSensorLoop:
64          return RadarSensorLoop(self, parent, self.initial_fuel)
65
66
67  class QLearningControllerEndpoint(Endpoint):
68
69      def __init__(self,
70                   factory: EndpointFactory,
71                   parent: AbstractNode,
72                   q_function: Callable[[Any, int], float],
73                   ) -> None:
74          super().__init__(factory, parent, "q_learning_controler")
75          self.task_handlers = {
76              (TaskType.store_data, "radar_data"): self.set_throttle,
77          }
78          self.q_function = q_function
```

```
79          self.actions = [0, 1, 2, 3, ]
80
81      def set_throttle(self, t: Task, key: TaskId) -> SomeTasks:
82          assert (len(t.val_list) > 0 and
83                  len(t.val_list[0]) == 2 and
84                  isinstance(t.val_list[0][0], int) and
85                  isinstance(t.val_list[0][0], float) and
86                  isinstance(t.val_list[0][0], float))
87          radar_data = t.val_list[0]
88          value = self.policy(radar_data)
89          return tasks.store_data("thruster_data", value)
90
91      def policy(self, data: Any) -> int:
92          return max(self.actions, key=lambda a: self.q_function(data,
    a))
93
94
95  class QLearningControllerEndpointFactory(EndpointFactory):
96      def __init__(self) -> None:
97          super().__init__()
98
99      def get(self, parent: AbstractNode) ->
    QLearningControllerEndpoint:
100         return QLearningControllerEndpoint(self, parent)
101
102
103 config = snr.Config(factories={
104     "lunar_module": [
105         RadarSensorLoopFactory(initial_fuel=100),
106         QLearningControllerEndpointFactory(),
107     ],
108 })
```

```
109
110
111 def main():
112     snr.SynchronousRunner("lunar_module", config).run()
113
114
115 if __name__ == "__main__":
116     main()
```

Listing A.5: Example implementation of a benchmark test case

```
1 import logging
2 import unittest
3 from typing import List, Optional
4
5 from snr import *
6
7 STRESS_TASK_NAME: TaskName = "stress"
8
9
10 class StressorEndpoint(Endpoint):
11     def __init__(self,
12                  factory: EndpointFactory,
13                  parent: AbstractNode,
14                  name: ComponentName,
15                  ) -> None:
16         super().__init__(factory, parent, name)
17         self.task_handlers = {
18             (TaskType.event, STRESS_TASK_NAME): self.replicate,
19         }
20
21     def replicate(self, task: Task, key: TaskId) -> SomeTasks:
22         return tasks.add_component(self.factory)
```

```
23
24     def begin(self) -> None:
25         self.schedule(tasks.event(STRESS_TASK_NAME))
26
27     def halt(self) -> None:
28         pass
29
30     def terminate(self) -> None:
31         pass
32
33
34 class StressorEndpointFactory(EndpointFactory):
35     def __init__(self,
36                  max_endpoints: int = 1000,
37                  time_limit_s: float = 0.500,
38                  ) -> None:
39         super().__init__()
40         self.max_endpoints = max_endpoints
41         self.time_limit_s = time_limit_s
42         self.calls = 0
43         self.num_children = 0
44
45     def get(self, parent: AbstractNode) -> Optional[Endpoint]:
46         self.calls += 1
47         if self.is_limited(parent):
48             return None
49         self.num_children += 1
50         return StressorEndpoint(self,
51                                 parent,
52                                 f"stressor_endpoint_{self.
    num_children}")
53
```

```
54      def is_limited(self, parent: AbstractNode) -> bool:
55          return (self.num_children >= self.max_endpoints or
56                  (self.time_limit_s != 0 and
57                   parent.get_time_s() >= self.time_limit_s))
58
59
60  class BenchStressEndpointFac(SNRTestCase):
61
62      def test_stress_endpoint_init(self):
63          stressor_fac = StressorEndpointFactory(max_endpoints=1,
64                                                 time_limit_s=0.050)
65          times: List[float] = []
66          node: AbstractNode = Node(
67              "test",
68              self.get_config([
69                  stressor_fac,
70                  TimeoutLoopFactory(seconds=0.010),
71                  StopwatchEndpointFactory(times,
72                                           [TaskType.terminate]),
73              ], Mode.DEBUG))
74          node.log.setLevel(logging.WARNING)
75          node.loop()
76          self.assertEqual(stressor_fac.num_children, 1)
77          self.assertEqual(len(times), 1)
78
79      def test_stress_endpoint(self):
80          time_target_s: float = 2.000
81          stressor_fac = StressorEndpointFactory(max_endpoints=10000,
82                                                 time_limit_s=
    time_target_s)
83          times: List[float] = []
84          timer = Timer()
```

```
        node: AbstractNode = Node(
            "test",
            self.get_config([
                stressor_fac,
                TimeoutLoopFactory(
                    seconds=time_target_s),
                StopwatchEndpointFactory(times,
                                         [TaskType.terminate]),
            ], Mode.DEBUG))
        node.log.setLevel(logging.WARNING)
        node.loop()
        t_s = timer.current_s()
        print("\nStressed with:\n",
              f"\t{stressor_fac.num_children} stressor endpoints\n",
              f"\t{stressor_fac.calls} stressor factory calls\n",
              f"\tTerminate expected at {time_target_s * 1000:.0f}
   ms\n",
              f"\t{times[0] * 1000:.3f} ms terminate handled\n",
              f"\t{t_s * 1000:.3f} ms total time\n",
              )
        if node.profiler:
            print(node.profiler.dump())


if __name__ == '__main__':
    unittest.main()
```

Listing A.6: Example implementation of a factory for a control input processor endpoint

```
from snr import *

import control_processor_endpoint

```

```
5
6 class ControlProcessorFactory(EndpointFactory):
7
8     input_data: DataKey
9
10    def __init__(self,
11                 input_data: DataKey = "controller_input",
12                 ) -> None:
13        super().__init__(control_processor_endpoint)
14        self.input_data = input_data
15
16    def get(self, parent: AbstractNode) -> Endpoint:
17        return control_processor_endpoint.ControlProcessorEndpoint(
18            self,
19            parent,
20            self.input_data)
```

Listing A.7: Example implementation of a control input processor endpoint

```
1 from typing import Any, List, Dict
2
3 from snr import *
4
5
6 class ControlProcessorEndpoint(Endpoint):
7
8     input_data: DataKey
9     watch_buttons: List[str] = [
10        "d_pad_tap",
11        "right_stick_press",
12        "A",
13        "B",
14        "X",
```

```
15          "Y",
16          "left_shoulder",
17          "right_shoulder",
18          "left_trigger",
19          "right_trigger",
20          "left_stick_press",
21          "left_paddle",
22          "right_paddle",
23      ]
24
25      def __init__(self,
26                   factory: EndpointFactory,
27                   parent: AbstractNode,
28                   input_data: DataKey,
29                   ) -> None:
30          super().__init__(factory, parent, "input_processor_endpoint"
    )
31          self.input_data = input_data
32          self.task_handlers = {
33              (TaskType.store_page, input_data): self.process_input
34          }
35
36      def process_input(self, task: Task, id: TaskId) -> SomeTasks:
37          page = task.val_list[0]
38          assert isinstance(page, Page)
39          prev_data: Dict[str, Any] = self.get_data(  # type: ignore
40              self.input_data+"_prev")
41          if prev_data is None:
42              self.info("No previous input data")
43              return None
44          data: Dict[str, Any] = page.data
```

```
45          assert isinstance(prev_data, dict) and isinstance(data, dict
    )
46          events: List[Task] = []
47          for key, value in data.items():
48              prev = prev_data[key]
49              if value == 0 and prev == 1:
50                  events.append(tasks.event("button_released", [key]))
51              if value == 1 and prev == 0:
52                  events.append(tasks.event("button_pressed", [key]))
53          return events
54
55      def begin(self) -> None:
56          pass
57
58      def halt(self) -> None:
59          pass
60
61      def terminate(self) -> None:
62          pass
```

Listing A.8: Example implementation of a loop for reading controller input

```
1 from typing import Dict, List, Optional
2
3 import pygame
4 from snr import *
5
6
7 class ControllerLoop(ThreadLoop):
8
9     _JOYSTICK_EVENTS: List[int] = [
10        pygame.JOYAXISMOTION,
11        pygame.JOYBALLMOTION,
12        pygame.JOYHATMOTION,
```

```
13            pygame.JOYBUTTONUP,
14            pygame.JOYBUTTONDOWN,
15        ]
16
17        output_data: DataKey
18        joystick: Optional[pygame.joystick.Joystick]
19        num_axes: int
20        num_buttons: int
21        num_hats: int
22
23        def __init__(self,
24                     factory: LoopFactory,
25                     parent: AbstractNode,
26                     output_data: DataKey,
27                     ) -> None:
28            super().__init__(factory, parent,
29                             "controller_loop",
30                             max_tick_rate_hz=20)
31            self.output_data = output_data
32            pygame.init()
33            pygame.display.init()
34            pygame.joystick.init()
35            joysticks = [pygame.joystick.Joystick(x)
36                         for x in range(pygame.joystick.get_count())]
37            if len(joysticks) < 1:
38                self.fatal("Controller not found")
39                raise Exception("Controller not found")
40            print(f"Found joysticks: {joysticks}")
41            self.joystick = joysticks[0]
42            self.num_axes = 0
43            self.num_buttons = 0
44            self.num_hats = 0
```

```
45            self.joystick.init()
46
47        def setup(self) -> None:
48            assert self.joystick is not None
49            self.num_axes = self.joystick.get_numaxes()
50            self.num_buttons = self.joystick.get_numbuttons()
51            self.num_hats = self.joystick.get_numhats()
52            print(f"Using joystick: {self.joystick.get_name()}",
53                  f"with {self.num_axes} axes",
54                  f"and {self.num_buttons} buttons",
55                  f"and {self.num_hats} d pads")
56
57        def loop(self) -> None:
58            assert self.joystick is not None
59            events: List[pygame.event.Event] = pygame.event.get()
60            if pygame.QUIT in [e.type for e in events]:
61                self.set_terminate_flag()
62                return
63            if any(event.type in ControllerLoop._JOYSTICK_EVENTS
64                   for event in events):
65                raw_input: Dict[str, float] = {}
66                for axis_id in range(self.num_axes):
67                    raw_input[f"axis_{axis_id}"] = \
68                        self.joystick.get_axis(axis_id)
69                for button_id in range(self.num_buttons):
70                    raw_input[f"button_{button_id}"] = \
71                        self.joystick.get_button(button_id)
72                for hat_id in range(self.num_hats):
73                    raw_input[f"dhat_{hat_id}"] = \
74                        self.joystick.get_hat(hat_id)
75                # print(raw_input)
76                self.store_data(self.output_data, raw_input)
```

```
77          else:
78              pass
79
80      def halt(self) -> None:
81          pygame.joystick.quit()
82          pygame.quit()
83          print("Loop halted")
84
85      def terminate(self) -> None:
86          print("Loop terminated")
```

Listing A.9: Example implementation of a factory for the controller loop

```
1 from snr import *
2
3 import controller_loop
4
5
6 class ControllerLoopFactory(LoopFactory):
7
8     output_data: DataKey
9
10    def __init__(self, output_data: DataKey = "raw_controller_input"
      ) -> None:
11        super().__init__([controller_loop])
12        self.output_data = output_data
13
14    def get(self,
15            parent: AbstractNode,
16            ) -> controller_loop.ControllerLoop:
17        return controller_loop.ControllerLoop(self,
18                                              parent,
19                                              self.output_data)
```

Listing A.10: Example implementation of a factory for the input mapping endpoint

```
1 from snr import *
2
3 import input_mapping_endpoint
4
5
6 class InputMappingFactory(EndpointFactory):
7
8     input_data: DataKey
9     output_data: DataKey
10
11     def __init__(self,
12                  input_data: DataKey = "raw_controller_input",
13                  output_data: DataKey = "controller_input",
14                  ) -> None:
15         super().__init__(input_mapping_endpoint)
16         self.input_data = input_data
17         self.output_data = output_data
18
19     def get(self, parent: AbstractNode) -> Endpoint:
20         return input_mapping_endpoint.InputMappingEndpoint(self,
21                                                             parent,
22                                                             self.
    input_data,
23                                                             self.
    output_data)
```

Listing A.11: Example implementation of an input mapping endpoint

```
1 from typing import Any, Dict
2
3 from snr import *
4
```

```
5
6 class InputMappingEndpoint(Endpoint):
7
8     output_data: DataKey
9     mapping: Dict[str, str] = {
10         "button_0": "d_pad_tap",
11         "button_1": "right_stick_press",
12         "button_2": "A",
13         "button_3": "B",
14         "button_4": "X",
15         "button_5": "Y",
16         "button_6": "left_shoulder",
17         "button_7": "right_shoulder",
18         "button_8": "left_trigger",
19         "button_9": "right_trigger",
20         "button_13": "left_stick_press",
21         "button_15": "left_paddle",
22         "button_16": "right_paddle",
23         "axis_0": "left_stick_X",
24         "axis_1": "left_stick_Y",
25         "axis_2": "right_stick_X",
26         "axis_3": "right_stick_Y",
27     }
28
29     def __init__(self,
30                  factory: EndpointFactory,
31                  parent: AbstractNode,
32                  input_data: DataKey,
33                  output_data: DataKey,
34                  ) -> None:
35         super().__init__(factory, parent, "
    controller_input_map_endpoint")
```

```
36        self.output_data = output_data
37        self.task_handlers = {
38            (TaskType.store_page, input_data): self.map_raw_input
39        }
40
41    def map_raw_input(self, task: Task, id: TaskId) -> SomeTasks:
42        page = task.val_list[0]
43        assert isinstance(page, Page)
44        raw_data: Dict[str, Any] = page.data
45        assert isinstance(raw_data, dict)
46        data: Dict[str, Any] = {}
47        for key, value in raw_data.items():
48            new_key = InputMappingEndpoint.mapping.get(key)
49            if new_key is not None:
50                data[new_key] = value
51        return self.store_data_task(self.output_data, data)
52
53    def begin(self) -> None:
54        pass
55
56    def halt(self) -> None:
57        pass
58
59    def terminate(self) -> None:
60        pass
```

Appendix B

SNR LIBRARY INSTALLATION TUTORIAL

Given a Raspberry Pi running an update installation of Raspberry Pi OS, this chapter instructs the installation of SNR for use of the library. The library test suite will be run to verify correct operation. Listing B.1 shows information about the system used in this example. Specifically, a Raspberry Pi 4 running Raspberry Pi OS Lite released on May 7th, 2021 is used. Python version 3.7 installed by default. SNR also supports versions 3.6 through 3.10. Pip, the Python package manager is also needed. Git will be used to clone the SNR library source code repository. Pip and Git can be installed using `apt` as shown in listing B.2.

```
pi@raspberrypi:~ $ uname -a
Linux raspberrypi 5.10.17-v7l+ #1414 SMP Fri Apr 30 13:20:47 BST
   2021 armv7lpi@raspberrypi:~ $ python3 --version
Python 3.7.3 GNU/Linux
pi@raspberrypi:~ $ python3 --version
Python 3.7.3
```

Listing B.1: System information for installation example

```
pi@raspberrypi:~ $ sudo apt update
[output omitted]
pi@raspberrypi:~ $ sudo apt install python3-pip git
[output omitted]
```

Listing B.2: Installation of Pip

Now the source code repository can be cloned and entered:

```
pi@raspberrypi:~ $ git clone https://github.com/sfshaw-calpoly/SNR
Cloning into 'SNR'...
```

```
remote: Enumerating objects: 5317, done.
remote: Counting objects: 100% (2685/2685), done.
remote: Compressing objects: 100% (1319/1319), done.
remote: Total 5317 (delta 1857), reused 2128 (delta 1313), pack-
    reused 2632
Receiving objects: 100% (5317/5317), 702.23 KiB | 4.20 MiB/s, done.
Resolving deltas: 100% (3663/3663), done.
pi@raspberrypi:~ $ cd SNR
pi@raspberrypi:~/SNR $
```

Pip can then be used to install the SNR library package. During this step, additional Python dependencies will be installed from PyPI, the Python Package Index.

```
pi@raspberrypi:~/SNR $ python3 -m pip install --user --upgrade .
[output omitted]
```

SNR can now be used on the system:

```
pi@raspberrypi:~ $ python3
Python 3.7.3 (default, Jan 22 2021, 20:04:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
    information.
>>> import snr
>>>
```

Two additional dependencies are required to test the full functionality of SNR. They can be installed as follows:

```
pi@raspberrypi:~/SNR $ python3 -m pip install --user --upgrade
    pytest pytest
-timeout
[output omitted]
```

Now the library test suite can be run using the built-in `unittest` module. As the test run completes in under one second, it can be used to quickly validate modifications to the library, even on an inexpensive embedded computer such as the Raspberry Pi.

```
pi@raspberrypi:~/SNR $ python3 -m unittest
\  0.12 ms: tests.test_config.TestConfig.test_empty
.  0.12 ms: tests.test_config.TestConfig.test_one_fac
.  0.07 ms: tests.test_config.TestConfig.test_two_facs
.  5.64 ms: tests.test_console.TestConsole.
   test_console_fails_to_connect
.  3.15 ms: tests.test_consumer.TestConsumer.test_consumer_put
.  1.50 ms: tests.test_consumer.TestConsumer.
   test_consumer_start_join
.  0.07 ms: tests.test_consumer.TestConsumer.test_increment
.  0.13 ms: tests.test_context.TestContext.
   test_context_handler_no_profiler
.  5.72 ms: tests.test_datastore_ping.TestDatastorePing.
   test_dds_ping
.  6.10 ms: tests.test_endpoint.TestEndpoint.test_endpoint_methods
.  0.74 ms: tests.test_endpoint.TestEndpoint.
   test_invalid_construction_fails.  0.16 ms: tests.test_endpoint.
   TestEndpoint.test_valid_construction
...... 16.35 ms: tests.test_expector_endpoint.TestExpectorEndpoint.
   test_expector_endpoint_empty
. 15.71 ms: tests.test_expector_endpoint.TestExpectorEndpoint.
   test_expector_endpoint_terminate
.  1.91 ms: tests.test_factory_reload.TestFactoryReload.
   test_factory_swap
.  0.49 ms: tests.test_loop.TestLoop.test_invalid_construction_fails
.  5.85 ms: tests.test_loop.TestLoop.test_loop_methods
.  1.46 ms: tests.test_loop.TestLoop.test_valid_construction
.  6.98 ms: tests.test_moving_avg_endpoint.TestMovingAvgEndpoint.
   test_moving_avg_endpoint
```

```
.   0.12 ms: tests.test_moving_avg_filter.TestMovingAvgFilter.
   test_filter
. 39.09 ms: tests.test_multi_proc_runner.TestMultiProcRunner.
   test_proc_runner
.   5.21 ms: tests.test_node.TestNode.test_get_task_handlers
.   0.16 ms: tests.test_node.TestNode.test_lookup_proof_of_concept
. 15.97 ms: tests.test_pipe_loop.TestPipeLoop.test_one_pipe
.   6.11 ms: tests.test_pipe_loop.TestPipeLoop.test_pipe_loop_noop
.105.76 ms: tests.test_pipe_loop.TestPipeLoop.test_two_pipe_loops
....   1.14 ms: tests.test_recorder.TestRecorder.test_invalid_task
. 35.80 ms: tests.test_recorder.TestRecorder.test_recorder_encoding
.111.69 ms: tests.test_replayer.TestReplayer.test_replayer
.   0.11 ms: tests.test_sockets_header.TestSocketsUtils.
   test_sockets_utils
. 81.36 ms: tests.test_sockets_listener.TestSocketsLsitener.
   test_sockets_listener_recv
. 83.03 ms: tests.test_sockets_listener.TestSocketsLsitener.
   test_sockets_listener_send
. 39.46 ms: tests.test_sockets_loop.TestSocketsLoop.
   test_sockets_loop
.   2.28 ms: tests.test_sockets_wrapper.TestSocketsLoop.
   test_sockets_wrapper
.   1.16 ms: tests.test_text_reader.TestTextReader.test_raw_reader
. 43.28 ms: tests.test_text_replayer.TestTextReplayer.
   test_raw_data_replayer_none
. 25.77 ms: tests.test_text_replayer.TestTextReplayer.
   test_raw_data_replayer_one
. 41.19 ms: tests.test_text_replayer.TestTextReplayer.
   test_raw_data_replayer_two
.   0.79 ms: tests.test_text_replayer.TestTextReplayer.
   test_raw_reader
```

```
. 11.67 ms: tests.test_timeout_loop.TestTimeoutLoop.
   test_timeout_loop_ms
. 11.52 ms: tests.test_timeout_loop.TestTimeoutLoop.
   test_timeout_loop_s
.
----------------------------------------------------------------------

Ran 49 tests in 0.810s

OK
```