Supplementary Material (RQ1)

Abstract

Modern computing devices which are equipped in robots, allow processing of large amounts of data in short periods of time. This led to the fact that, using a small number of sensors and executable devices, robots began to have greater functionality, the implementation of which fell to the software that controls the devices. The Robot Operating System (ROS) is known to be the de-facto standard for robotics software. ROS includes drivers for various electronic components, libraries, visualizers, and also provides a process control system, etc. ROS is open source and is very popular among developers of robotics software systems. Mobile robots typically carry their energy sources such as batteries, so designing an energy-aware and energy-efficient system in the early stage of software development is key for conserving energy. It is imperative to establish a set of green design options known as *architectural tactics* for creating energy-aware and energy-efficient robotics software. This project presents 11 green architectural tactics for energy-aware and energy-efficient robotics systems which can also be applied and already have been applied in other application domains. To achieve this, we **(i)** construct a dataset by mining various ROS open-data sources, **(ii)** identify and extract energy-related data, **(iii)** identify and extract architecturally-relevant data, and **(iv)** synthesize a concrete catalog of 11 green architectural tactics used in the robotics systems as well as in other application domains.

Contents

1 Introduction 1

2 Background 3

2.1 Architectural Tactics 3

2.2 ROS-based Systems 6

3 Study Design 9

3.1 Research Question 9

3.2 Study Design 10

3.2.1 Phase 1: Dataset Construction 11

3.2.1.1 Pre-filtering 12

3.2.1.2 Web Scraping 13

3.2.1.3 Data Storage - `MongoDB` 14

3.2.1.4 Data Extractors 15

3.2.1.5 Dataset Summary 19

3.2.2 Phase 2: Energy-Relevant Data Identification 23

3.2.2.1 Energy Query 23

3.2.2.2 Document Query Engine 24

3.2.2.3 Removing False-Positive Energy Data 25

3.2.2.4 Level Of Agreement Calculation 25

3.2.2.5 Energy Dataset Summary 25

3.2.3 Phase 3: Architecturally-Relevant Data Identification 27

3.2.3.1 Data Point Types 27

3.2.3.2 Inclusion & Exclusion Criteria 28

3.2.3.3 AR Data Point Identification 30

3.2.3.4 Level Of Agreement Calculation 31

3.2.3.5 AR Dataset Summary 32

3.2.4 Phase 4: Green Tactics Extraction . . . 34
3.2.4.1 Stage 1: AT-Relevant Data Extraction . . . 34
3.2.4.2 Stage 2: Green Tactics Categories Identification . . . 35
3.2.4.3 Data Point Classification . . . 36

4 Results **39**
4.1 Tactic Tree . . . 39
4.2 Green Tactics . . . 40
4.2.0.1 EA1: Abort Mission . . . 42
4.2.0.2 EA2: Stop Task & Recharge . . . 45
4.2.0.3 EA3: Dedicated Energy-Level Message . . . 49
4.2.0.4 EA4: Energy-Level Info Within Diagnostics Message . . . 51
4.2.0.5 E5: Aggregated Energy Information . . . 54
4.2.0.6 EA6: Energy-Savings Mode . . . 56
4.2.0.7 EA7: Offline Energy Profiler . . . 60
4.2.0.8 EE1: Limit Task . . . 62
4.2.0.9 EE2: Disable Hardware . . . 67
4.2.0.10 EE3: Energy-Aware Sampling . . . 70
4.2.0.11 EE4: On-Demand Software Components . . . 73

5 Discussion **77**

6 Threats To Validity **79**
6.1 External Validity . . . 79
6.2 Internal Validity . . . 79
6.3 Construct Validity . . . 80
6.4 Conclusion Validity . . . 80

7 Conclusion **81**
7.1 Chosen ROS Projects . . . 83

References **85**

1

Introduction

Creating truly reliable and versatile robotic software is an extremely difficult task. From a robot's point of view, problems that seem trivial to people often require very complex technical solutions. Often, the development of such solutions are beyond the capabilities of one person (1).

The Robot Operating System (ROS) was created to stimulate the joint development of robotics software. Each individual team can work on one specific task and using a single platform allows the entire ROS community to get and use the results of a team for their projects. ROS is a flexible platform (framework) for developing software for robots. It is a set of various tools, libraries and communication middleware, the purpose of which is to simplify the task of developing software for robots (2).

Robotics software is becoming more complex day by day as they often require advanced computational power and depend on various sophisticated sensors and actuators (3). Energy is always a critical factor when dealing with battery-powered robots as robots have a limited allocation of energy. Robots are typically built with distinct hardware and software resources, which often demand a lot of energy, to satisfy high computational demands (4). They cannot operate regularly for long time frames since they have a limited amount of energy available for being battery-powered (5). For example, unmanned aerial vehicles (UAVs) play a critical role in the next generation cellular networks, where they act as flying infrastructure servicing ground users when the ground infrastructure is unavailable or overloaded. UAVs are expected to operate wirelessly which means that they will last only for 7-15 minutes in the air before the battery drops (6).

Energy consumption plays a critical role as a design factor in robotics software and must be considered in the early stage of the development of the system (7). The first step towards designing energy-efficient and energy-aware robotics software is to establish a set

of concrete design options known as *architectural tactics* (ATs) to serve as the foundation to achieve the *quality attributes* energy-efficiency and energy-awareness. A formal definition of ATs is provided by Bachmann, Bass, and Klein; they define architectural tactics as a *means of satisfying quality attribute response measures by manipulating some aspect of a quality attribute model through architectural design decisions* (8). Bass et al. defines a QA as a *measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders* (9).

The **primary goal** of this study is to provide the ROS community with an evidence-based list of energy-efficient and energy-aware architectural tactics for robotics software. A four-phase approach is used to achieve this goal: **(1)** The initial dataset is constructed by mining online data sources dedicated specifically for the ROS community and millions lines of code from open-source ROS projects, **(2)** the dataset is then filtered for energy-specific data, **(3)** further, architecturally-relevant data is extracted, and **(4)** green tactics are identified and formulated.

The **target audience** of this study is the ROS community: developers, researchers, enthusiasts, etc. For ROS researchers, this study is beneficial as they will have a clear understanding on the design decisions behind energy-aware and energy-efficient robotics software and they can use this insight to further make contributions in the form of new algorithms, reference architectures, packages, etc, and/or identify gaps and limitations in this field. The extracted catalogue of green tactics can be used by ROS developers and developers not part of the ROS-community for designing and implementing their own energy-aware and energy-efficient software systems. This knowledge will further guide them in improving already existing energy-aware and energy-efficient robotics software.

The **main contribution** of this study is a concrete list of energy-efficient and energy-aware architectural tactics for robotics software. The architectural tactic template proposed in (10) is used to formally define each AT as well as an architectural template for each tactic is provided.

The rest of this study is organized in the following manner: **(Section 2)** provides the background information on architectural tactics and ROS-bases systems, **(Section 3)** describes the study design process using a multi-phase approach, **(Section 4)** discusses the results (extracted ATs), **(Section 5)** provides relevant discussion points and implications for the ROS community, and **(Section 6)** presents the threats to validity identified during this study. The paper is closed with **(Section 7)** which consists of the conclusion and future work.

2

Background

2.1 Architectural Tactics

An architectural tactic is a design option for an architect. A formal definition of architectural tactics is provided by Bachmann, Bass, and Klein; they define architectural tactics as a *means of satisfying quality attribute response measures by manipulating some aspect of a quality attribute model through architectural design decisions* (8). Architectural tactics are meant to achieve and satisfy a given quality attribute. An example of a quality attribute is *availability* which enables a system to endure system faults such that a service being enabled by the system remains compliant with its specification (11). For example, to ensure the *availability* of a software system, the architect must consider several design options to achieve the QA. Figure 2.1 shows the hierarchy of availability architectural tactics.

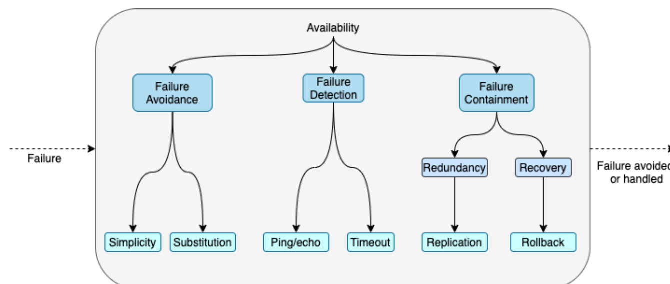


Figure 2.1: A hierarchy of availability tactics for software systems (11)

In a scenario where a failure enters a system, one way to assure the availability of the system, is to integrate a *failure detection* mechanism which can be implemented via a *ping/echo* tactic. This is one of the many options an architect uses to ensure the availability of a system.

Motivation	Availability is an important quality requirement that software systems are constantly trying to promote. The first step towards ensuring the availability of a system and mitigating possible threats is to detect failures in the system. The Ping/Echo tactic is a common tactic used to detect failures in software systems.
Description	The tactic detects a failure by sending ping messages to receivers in a continuous manner. If the sender does not receive an echo message back from a receiver within a certain time-frame,the receiver is considered to have failed. Figure 2.3 provides a concrete example of the Ping/Echo AT and how it works in practice. In this tactic, three main components are involved: the sender, receiver, and a failure monitor which are represented in the figure as the *PingSender*, *PingReceiver*, *FailureMonitor* . The *PingSender* starts sending ping messages every *timeInterval* to the *PingReciever* . In a perfect scenario, the *PingReceiver* sends an echo message back to the *PingSender* component to notify that it is alive. In the case when the *PingReceiver* component fails to send an echo message and the *maxWaitingTime* is exceeded, the *PingSender* component sends a notification to the *FailureMonitor* component which in turn throws a *NotificationException*.
Constraints	The ping/echo AT requires the the three main components - a sender, receiver and a monitor.
Example	Figure 2.4 provides a practical example where the ping/echo tactic is used. Within an access network, there is a host (*PingSender*) and 3 end-users (client - *PingReceivers*). To check the availability and reachability of each client, the host sends a ping message. If the client is reachable, it sends back an echo message. In the case when the client fails to receive a ping message and/or fails to send back an echo message, the host notifies the ping/echo monitor (*FailureMonitor*) which handles the situation
Dependencies	The ping/echo tactic is typically combined with an exception tactic which takes care of notifying a failure/exception to the exception handler.

Table 2.1: `Ping/echo` architectural tactic

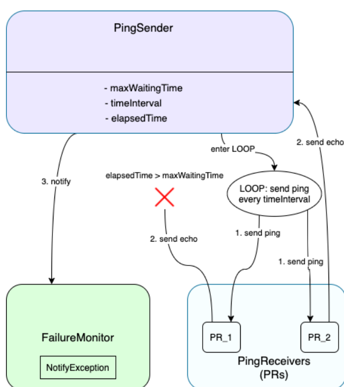


Figure 2.2: The `ping/echo` architectural tactic

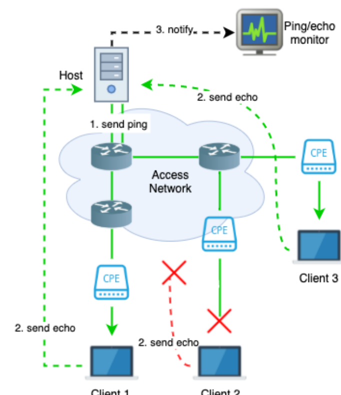


Figure 2.3: The `ping/echo` architectural tactic in an access network

A template (Table 2.1) for describing architectural tactics defined in (10) is used to describe the *ping/echo* AT for *failure detection* in a concrete style. During software design, the designer decides which architectural tactics are appropriate to incorporate in regards to the system's trade-offs and context, and the cost to employ the selected architectural tactics. To achieve this, the software architect typically refers to the seven categories of design decisions: **(i)** allocation of responsibilities - this includes identifying important responsibilities, architectural infrastructure and determining how these responsibilities are allocated to runtime and non-runtime elements, **(ii)** coordination model - identifying elements of the software system that must coordinate and choosing communication methods between components of the system, **(iii)** data model - choosing data abstraction, operations and properties, and organizing the data (e.g., determining what kind of storage to use), **(iv)** management of resources - identifying which resources need to be managed and determining the impact of saturation on the different resources (e.g., trade-offs), **(v)** mapping among architectural elements - mapping of modules and runtime elements, assignment of runtime elements to processors, etc, **(vi)** binding time decisions - establishing the scope and the point in the life cycle, and **(vii)** choice of technology - deciding which technologies and tools are available to realize the design decisions made throughout the previous six categories.

2.2 ROS-based Systems

ROS is focused on maximizing code reuse in development. The main characteristics (12) that make it possible to implement this are **(i)** distributed processes: the ROS framework is designed as minimal units of executable processes (nodes), and each process runs in isolation. The interaction of different nodes occurs only at the messaging level, **(ii)** package management: several processes withing a common task are combined into a package. Package management refers to a set of utilities that allow the developer to automatically download, install, and uninstall packages. The package manager guarantees the health and integrity of installed packages, **(iii)** public repositories and documentation: every available package is published to a public repository. Package documentation is published in a single system that makes it easy to find the packages one needs, **(iv)** unified API: when developing a software system using ROS, one gets a simple and easily embeddable API. In the sample programs, the API usage is not very different from the language (C++ or Python), and **(v)** multiple programming language support: ROS provides client libraries to support various programming languages. The most popular are Python, C++, as well as languages such as Lisp, JAVA, C #, Lua, and Ruby.

A ROS-based system is composed of *Nodes* which are OS processes that perform computations (12). When the *Node* is initiated, it registers information about itself on the *ROS Master* which acts as a server for connecting different *Nodes* to each other (name of the *Node*, types of messages being processed) (13). A registered *Node* can interact with other *Nodes* using a publish/subscribe model based on *Topics* (publish and subscribe to messages), or using a request/response model based on *Services* or *Actions* (request and receive responses) as shown in figure 2.4.

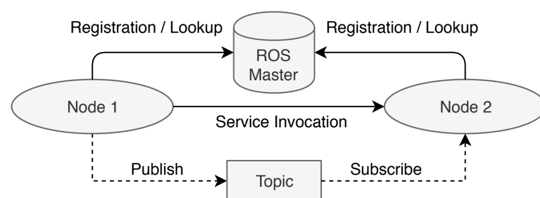


Figure 2.4: ROS Node Communication (14)

It is important to note that the exchange of messages between the *Nodes* works without the participation of the *ROS Master* (the connection between the *Nodes* occurs directly). The *ROS Master* only provides a single namespace for deciding where to connect to a

specific *Node*. The *Node's* launch address is taken from the `ROS_HOSTNAME` environment variable, which must be defined before starting. The port is set to an arbitrary unique value (15).

In ROS, a *Service* is a communication model that operates on the principle of synchronous bidirectional communication between a *Service Client* that requests data and a *Service Server* that responds to requests. A *Service Server* is a communication *Node* (process) that receives a request, processes the data and sends back a response. A *Service Client* is also a communication node that creates a request on the *Service Server* and receives a response after the request is completed. This interaction model represented in Figure 2.5 is used to remotely perform various small operations within different *Nodes*.

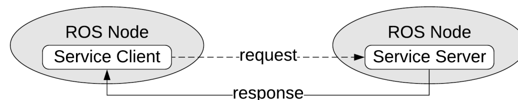


Figure 2.5: ROS Service Model

Another communication model in ROS is the *Action* communication model depicted in Figure 2.6 which is used when the requested task takes a long time to complete (e.g., moving the robot) and feedback from the process is needed. This is very similar to the *Service* communication model: *Service Request* maps to the *Action Goal* and the *Service Response* maps to the *Action Result*. There is also an additional entity, the *Action Feedback*, for transmitting intermediate results to the *Action Client*.

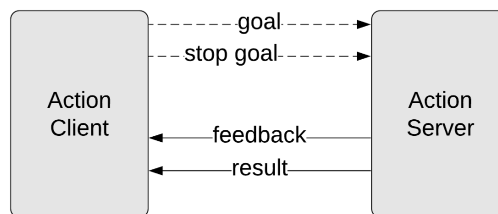


Figure 2.6: ROS Action Model

Figure 2.7 shows the architecture for a fully autonomous robot system for urban search and rescue (16). ROS *Nodes* are represented by blue ovals, and *Topics* by green rectangles.

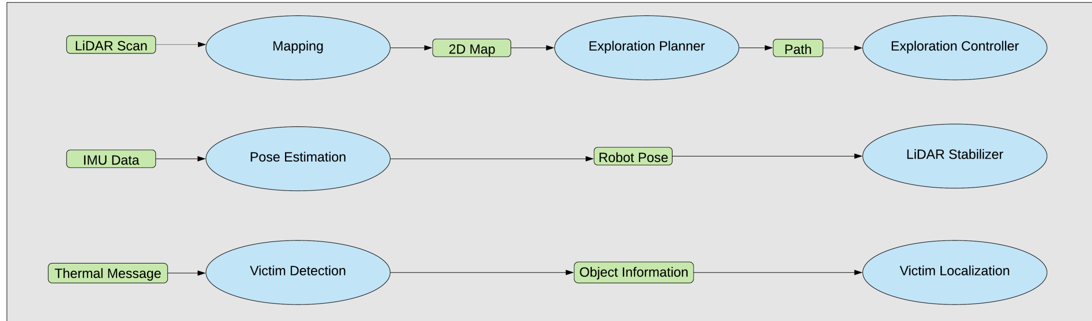


Figure 2.7: Example of a ROS software architecture - a fully autonomous robot system for urban search and rescue

As an example of communication between *Nodes* can be observed between the *Victim Detection Node* and the *Victim Localization Node*. The *Victim Detection Node* is subscribed to a *Thermal Image topic* which is publishes by some other *Node* in the system. The *Victim Detection Node* also publishes messages to the *Object Information topic* which is subscribed to by the *Victim Localization Node*.

3

Study Design

Intuitively, the main **goal** of this project is to identify a set of concrete, repeatable, and quantifiable green architectural tactics for energy-efficient robotics software. Table 3.1 shows a more formal definition of the goal using the Goal-Question-Metric technique (17). To achieve this goal, the research question in Section 3.1 is answered by carrying out the study design elaborated in Section 3.2.

Analyze	ROS data sources
For the purpose of	establishing a set of concrete, repeatable, and quantifiable green architectural tactics
With respect to	energy-efficiency and energy-awareness
From the point of view	of roboticists and researchers
In the context of	ROS-based systems
Combination	
Analyze ROS data sources for the purpose of establishing a set of concrete, repeatable, and quantifiable green architectural tactics with respect to energy-efficiency and energy-awareness from the point of view of roboticists and researchers in the context of ROS-based systems.	

Table 3.1: Goal definition

3.1 Research Question

The goal described above is refined into the following research question:

[RQ1]: *Which green architectural tactics are employed for energy-efficient and energy-aware robotics software?* This research question aims to identify, extract, and establish a concrete set of green architectural tactics used in open-source robotics projects. Answering this research question will help roboticists in designing and developing energy-efficient robotics software via the established green architectural tactics. The extracted green tactics can be used by roboticists as well as other developers not part of the robotics community as a checklist for inspecting if some improvements in the energy-efficiency of software systems can be caught in the early stage of software design. Furthermore, researchers can use the extracted green tactics as a foundation for better supporting the development of greener robots and other software systems (e.g., by inventing new methods or programming models which support a green tactic by design).

3.2 Study Design

Figure 3.1 illustrates the overview of the study design which consists of four sequential phases. In Phase 1, an initial dataset is constructed by crawling open data sources for ROS-specific data. The dataset produced in Phase 1 is queried in Phase 2 to identify data specifically related to energy.

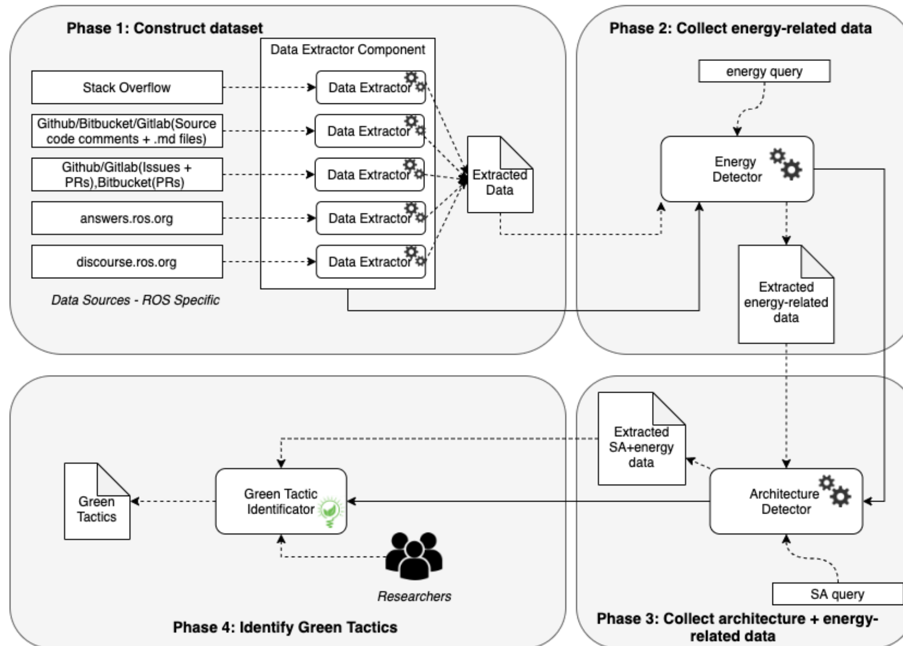


Figure 3.1: Study Design

The energy-related data produced in Phase 2 is queried in Phase 3 to identify data where architecturally-relevant concerns are discussed. Finally, the dataset produced in Phase 3 is used in Phase 4 to identify green architectural tactics for energy-efficient and energy-aware robotics software.

The rest of this section describes the specifics of each phase in detail.

3.2.1 Phase 1: Dataset Construction

It is important to note that the ROS community is extremely active. When a developer encounters a problem, finding a solution and getting help becomes easier, not only from ROS developers(Open Robotics), but also from other enthusiasts and professionals. Everyday developers part of the ROS community are heavily involved in open-source development of publicly available ROS packages. As of 2018, 2711 packages were published on the official ROS website[1] and according to ROS Community Metrics Report, there are about 15000 registered users for *ROS-Answers*[2](Q&A platform) (18). These numbers suggest that publicly available information is a favorable source of data and for this reason, open-data resources are employed in the initial dataset construction.

Data Source	Type	Contents
`https://stackoverflow.com/questions/tagged/ros`	Q&A Platform	Questions related to different aspects of ROS (e.g., error, code problem)
`https://answers.ros.org/questions/`	Q&A Platform	Questions related to different aspects of ROS (e.g., error, code problem)
`https://discourse.ros.org/`	ROS Forum	ROS discussion forum (e.g., announcements of new projects, ROS-industrial related topics, etc).
`http://wiki.ros.org/Documentation`	ROS Wiki	Provides tutorias, ROS packages, libraries, etc.
`https://github.com/`	Repositories for ROS-based systems identified in citemalavolta1.	Source code of ROS-based systems.
`https://bitbucket.org/`	Repositories for ROS-based systems identified in citemalavolta1.	Source code of ROS-based systems.

Table 3.2: Data sources used for this study

The data sources listed in Table 3.2 are a good representation of ROS-related topics and are heterogeneous enough to achieve the goal of this study as these sources are known to be used by practitioners as well as ROS-developers to post, answer, and collaborate

[1]https://www.ros.org/
[2]https://answers.ros.org/questions/

on ROS-based topics. The ROS open-source repository dataset[1] used in this project is constructed and refined in an earlier Study (14).

3.2.1.1 Pre-filtering

To ensure that the initial dataset is of high quality, it is essential to perform specific filtering procedures for pull requests from `GitHub` repositories. `GitHub`[2] has several *bots* which automate dependency requirements, perform code reviews, automate deployments, etc. Figure 3.2 shows a pull request generated by `dependabot` - a `GitHub` bot that automates dependency updates.

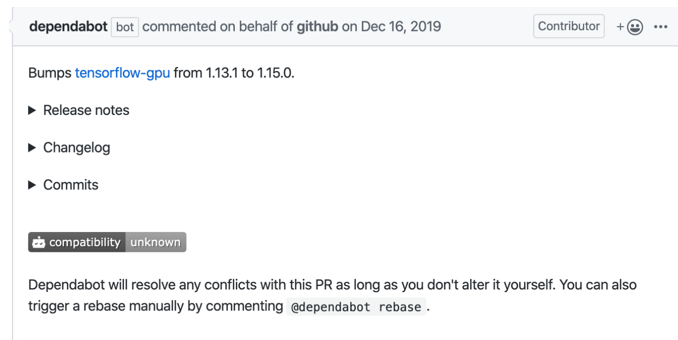


Figure 3.2: `Dependabot` - a `GitHub` bot

Intuitively, pull requests generated by the bots provide no relevant information for this study, and therefore, can be discarded.

Figure 3.3 shows the steps for filtering the `GitHub` repositories before extracting their pull request data.

[1] https://github.com/S2-group/icse-seip-2020-replication-package/blob/master/ICSE_SEIP_2020.pdf
[2] https://github.com/marketplace

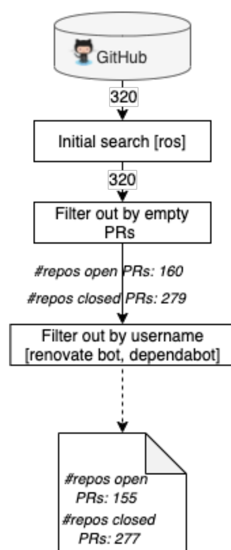


Figure 3.3: `GitHub` Repository Filtering (pull requests)

First, repositories with no open or closed pull requests are filtered out, and then, repositories with pull requests initiated *only* by `GitHub` bots are removed from the set. After scraping a list of `GitHub` bots from `GitHub Marketplace` and a list of `GitHub` users which created the pull requests, it was evident that only two kinds of *dependency* bots were present in the `GitHub` pull requests - `renovate bot`[1] and `dependabot`[2], by comparing the two lists (`GitHub` bot list and `GitHub` pull request users list). This resulted in 155 `GitHub` repositories with open pull requests and 277 `GitHub` repositories with closed pulled requests.

Similarly to `GitHub`, `BitBucket` also has bots in charge of automating pull requests, however, there are only 15 `BitBucket` repositories in the initial ROS-repository dataset and no bots were identified in the `BitBucket` pull requests.

3.2.1.2 Web Scraping

In order to collect information from each data source, a dedicated software module - a *Data Extractor* is implemented for each data source, as each one requires a different methodology to collect the data. More specifically, for `StackOverflow`, `ROS-Answers`, `ROS-Discourse`, `ROS Wiki`, and `GitHub/Bitbucket` issues and pull requests data scraping techniques are used to extract questions, answers and other metadata.

[1]https://github.com/renovatebot/renovate
[2]https://dependabot.com/

Web scraping has become one of the most effective data scraping techniques for extracting data from the web; two most commonly used tools are used accomplish this task - Scrapy[1] and BeautifulSoup. For this study, Scrapy is chosen over BeautifulSoup as the speed and load time of each webpage is very fast via Scrapy, big jobs (bulk of URLs are crawled in less than a minute) are done easily with Scrapy whereas BeautifulSoup is good for simple scraping jobs, and Scrapy offers custom configurations such as download speed, middleware functionality, etc. Scrapy is a powerful, open-source, Python-based web crawling framework used to harvest and process the data from websites.

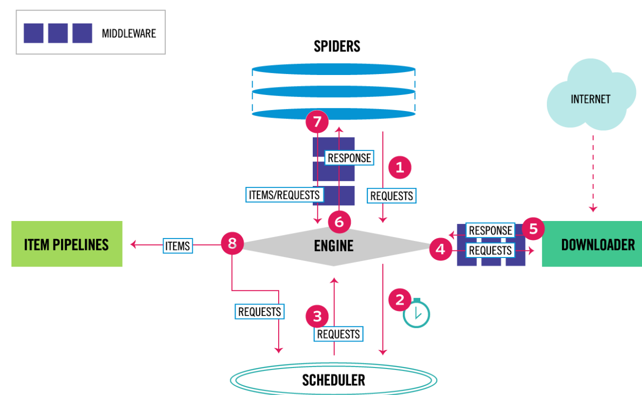


Figure 3.4: Scrapy Architecture

Figure 3.4 illustrates the Scrapy architecture; it is an integrated system that consists of a centralized engine which controls the data flow between the *scheduler* (receives web requests), the *downloader* (fetches web pages), the *spider* (custom class for parsing HTML responses), and the *item pipeline* (process parsed data by the *spider*).

3.2.1.3 Data Storage - MongoDB

Due to the heavy amount of data mined for this project, it is imperative to store and host the dataset in a remote database. MongoDB is a cross-platform document-oriented database under the hood of NoSQL databases and is chosen as the data-storage for this project. MongoDB employs the key-value pair format, similar to a Python dictionary structure, making it easily integrable with a Python project. MongoDB supports *documents*

[1] https://scrapy.org/
[1] https://docs.scrapy.org/en/latest/topics/architecture.html

in JSON format - a human-readable format, making it user-friendly. On top of this, MongoDB is *schema less* meaning that there are not restrictions on schema design. MongoDB stores the documents in *collections* - grouping of MongoDB documents. Figure 3.5 is an example of a StackOverflow post stored as a *document* in a MongoDB collection. The collections are schema less - thus, documents in the same collection can have different fields type, as shown in Figure 3.5. The NOSQL database is also known for its flexibility, power, speed and ease of use (19).


```
_id: ObjectId("5e6e9758090caa2899605395")
title: "Using a ROS custom message type across machines/hosts"
time: "2019-09-17 08:34:35Z"
post_content: Array
    0: "I am unable to use a custom ROS message type on machine B that I have ..."
    1: "When I do these things locally, means all on machine A, everything wor..."
    2: " I can even successfully subscribe to it with getting all the messages..."
    3: "So, what am I missing in order to use my custom message from machine A..."
    4: "Please, I really need help on that!"
answer: Array
    0: "The solution is to simply have the custom message definition on all ma..."
    1: "This might be most comfortable if the custom message is realized as a ..."
question_code: Array
    0: "rostopic pub /test_topic my_custom_msg test_value"
    1: "rostopic list"
    2: "runmsg list"
    3: "rostopic echo"
url: "https://stackoverflow.com/questions/57970393/using-a-ros-custom-messag..."
```


Figure 3.5: MongoDB document example

The dataset is stored and hosted in a MongoDB database in the MongoDB Atlas[1] - a global cloud database service. The database is also part of the public replication package for this project.

3.2.1.4 Data Extractors

Before crawling and extracting the data it is fundamental to inspect the data sources and the kind of metadata to extract. Figure 3.6 shows the StackOverflow website and its webpages containing posts and the post details (question and answers).

StackOverflow Posts   StackOverflow Question   StackOverflow Answer

Figure 3.6: StackOverflow Posts, Question, Answer

[1]https://www.mongodb.com/cloud/atlas

The blue box around the text in the question and answer is an example of what is extracted and stored in `MongoDB`. The goal of the crawler is to crawl through all of the pages containing posts and extract the post details. In order to understand which parts of the webpage to crawl and extract, it is important to inspect the elements on the webpage. Figure 3.7 shows the element details of a post on StackOverflow using the developer tools in Google Chrome.

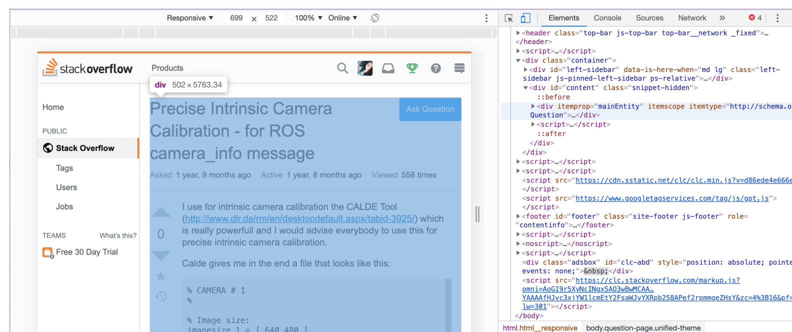


Figure 3.7: StackOverflow Post Element Details

By navigating through the different `HTML` tags, it is clearly visible which tags contain the post details. After inspecting the webpages from each data source and understanding which elements on the webpage to scrape, the *Data Extractor* module is ready to be implemented.

The custom *Data Extractor* module consists of a custom *spider* class implemented via `Scrapy` for each data source due to the unique `HTML` structure of each website. Figure 3.8 shows the source code of a *spider* class for one of the data sources - `StackOverflow`.

```
1  class SOSpider(CrawlSpider):
2      name = "stackoverflow"
3      start_urls = ['https://stackoverflow.com/questions/tagged/ros']
4
5      rules = (
6          Rule(LinkExtractor(allow=(), restrict_css=('.pager',)),
7               callback="parse_item",
8               follow=True),)
9      def parse_item(self, response):
10         item_links =
           ↪  response.css('.question-hyperlink::attr(href)').extract()
11         for a in item_links:
12             yield scrapy.Request(response.urljoin(a),
               ↪  callback=self.parse_detail_page)
13     def parse_detail_page(self, response):
14         item = SOItem()
15
16         title =
           ↪  response.css('.question-hyperlink::text').extract()[0].strip()
17         item['title'] = title
18         time = response.css('.user-action-time
           ↪  span::attr(title)').extract()[0].strip()
19         item['time'] = time
20         post_content = response.css('.question p::text').extract()
21         item['post_content'] = post_content
22         answer = response.css('.answer p::text').extract()
23         item['answer'] = answer
24         if response.css('blockquote > p::text').extract():
25             quote = response.css('blockquote > p::text').extract()
26             item['quote'] = quote
27         if response.css('.question code::text').extract():
28             question_code = response.css('.question code::text').extract()
29             item['question_code'] = question_code
30         if response.css('.answer code::text').extract():
31             answer_code = response.css('.answer code::text').extract()
32             item['answer_code'] = answer_code
33
34         item['url'] = response.url
35         yield item
```

Figure 3.8: StackOverflow Spider Class

Line 1 defines the spider class - `SOSpider` which inherits from the Scrapy `CrawlSpider` class. This class provides powerful mechanisms for crawling websites and serves as the default Scrapy spider class. ***Line 2*** declares the name of the crawler - `stackoverflow`. This is used to later start the crawler from the command line. ***Line 3*** defines the *start_ urls* to be crawled. Several URLs can be passed into this variable, and all of them will be crawled one-by-one. ***Lines 5-8*** declare a `Rule` object: This rule uses a `LinkExtractor` object to extract links from specified `HTML` elements. In this example, after inspecting the source code of `StackOverflow`, it is evident that the pagination is bounded by the `.pager` element. The extracted link is passed to the `parse_item` method.

Lines 9-12 describe the `parse_item` method. This method navigates to the webpage where the post is located by accessing the extracted link by the `LinkExtractor` rule. Finally, ***Lines 13-35*** describe the `parse_detail_page` method called by the `parse_item` method. This method extracts the details of a post; i.e.: title, URL, answer, etc.

After crawling each data source and extracting relevant information such as the post title, post contents, date of the post, url, etc, the extracted data is inserted into a designated *collection* in a `MongoDB` database. Figure 3.9 provides an overview of the *Data Extractor*

module for crawling the data sources.

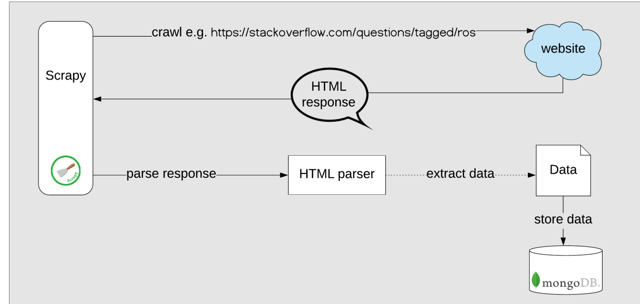


Figure 3.9: Data Extractor - Web Crawler

To extract information from the `GitHub/Bitbucket` ROS repositories such as source code comments and contents of `.md` files, a separate *Data Extractor* module is implemented in Python, as Python libraries provide powerful file and data manipulation functionalities. Figure 3.10 illustrates the *Data Extractor* for the source code comments and `.md` file contents of the ROS repositories.

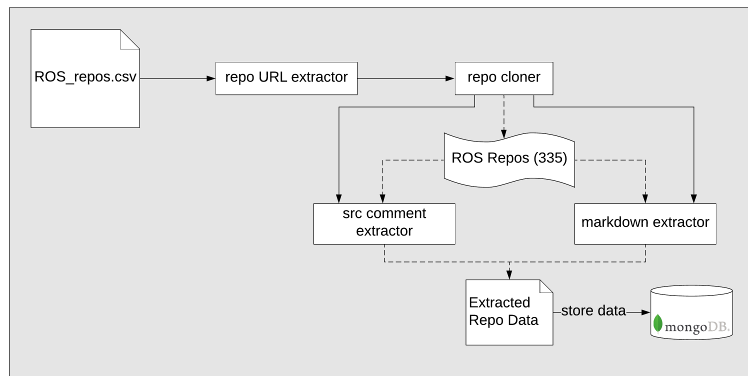


Figure 3.10: Data Extractor - Source code comments & `.md` contents

The 335 identified ROS open-source repositories in (14) are cloned and inspected for source code files and `.md` files. `C++` and `Python` are identified as the two main programming languages used for the 335 ROS-based systems (14). A *source code comment extractor* and *`markdown` extractor* are implemented in order to iterate through the cloned repositories, parse files, and extract source code comments and `.md` file contents.

The `markdown` extractor module simply extracts the entire contents of an `.md` file and stores them in a `MongoDB` collection. The `src comment extractor` module extracts com-

ments from .c/.cpp and .py files as these were the identified languages in a previous study used to implement the 335 ROS projects (14). For example, to detect Python comments, lines starting with or strings following a # are searched in the .py files and extracted. Figure 3.11 shows a snippet of code from the src comment extractor for Python files.

```
1 # iterate through .py files only
2 comments_file_location = [s for s in comments_file_location if ".py" in s]
3 for line in contents:
4     comment_search_p = re.search('#(.*)\n', line, re.IGNORECASE)
5         if (comment_search_p):
6             comment = comment_search_p.group(1)
```

Figure 3.11: Python comments src extractor

Line 2 ensures that only files with the .py extension are opened and iterated through in order to improve the algorithm efficiency. ***Line 4*** specifies the search pattern via the Python built-in re package. Lines of code that start with a # or strings following a # are considered as Python comments and are successfully extracted if found. A similar approach is used to implement the src comment extractor for .c/.cpp files with the exception of specifying different search patterns. After iterating over the repositories and extracting comments, the collected data points are stored in a separate collection in the MongoDB database.

3.2.1.5 Dataset Summary

Table 3.3 recaps the timeline for extracting data from each data source.

Data Source	Extraction Date
Stack Overflow	February 17, 2020
ROS-Answers	February 17, 2020
ROS-Discourse	February 21-22, 2020
ROS Wiki	April 12-14, 2020
GitHub src comments and .md files	February 24, 2020
GitHub/Bitbucket issues and PRs	February 25 - March 1, 2020

Table 3.3: Data Extraction Timeline

Data Source	#Repos	#Repos(PRs)	#Repos(Issues)	#PRs	#Issues
GitHub	320	280	289	30,045	23,214
Bitbucket	15	7	0	70	0

Table 3.4: Data Sources(1) Overview

Data Source	#Posts
Stack Overflow	1,880
ROS-Answers	43,672
ROS-Discourse	2,604
ROS Wiki	2,547

Table 3.5: Data Sources(2) Overview

Programming Language	#Repositories
C++	59
Python	44
C++/Python	232

Table 3.6: Overview of # of repositories per programming language

Tables 3.4 and 3.5 present the summaries of the number of data points collected from the repositories and websites. A big portion of the data originates from `GitHub` issues and pull requests and `ROS-Answers` which indicates the popularity of the sources in the ROS community. Table 3.6 shows the number of repositories for `C++`, `Python` and `C++/Python` ROS projects.

Table 3.7 displays the demographics of the social discussion data sources, Table 3.8 shows the demographics of the 232 `C++/Python` ROS repositories, Table 3.9 shows the demographics for 59 `C++` ROS repositories, and Table 3.10 shows the demographics for 44 `Python` ROS repositories. As shown in the tables, the dataset is quite heterogeneous - for the social discussion data sources in terms of the post age and the number of posts per distinct user, and for the ROS repositories in terms of commits, PRs, issues, contributors, and `.md` which proves that the data sources considered for this study are of competent quality and accurately representative of real-world use cases.

Post age (in # of days)						
Data Source	**Min**	**Max**	**Median**	**Mean**	**SD**	**CV**
`StackOverflow`	0	2903	758	917.151	689.051	0.751
`ROS-Answers`	0	3289	1662	1635.377	926.820	0.566
`ROS-Wiki`	4	3834	1731	1715.986	943.656	0.549
`ROS-Discourse`	1	1460	502	557.002	365.672	0.656
#Questions per distinct user						
StackOverflow	1	27	1	1.346491	1.405331	1.043698
`ROS-Answers`	1	486	1	2.956587	7.121926	2.408834
`ROS-Wiki`	1	270	1	5.213992	15.67964	3.007223
`ROS-Discourse`	1	217	1	3.062874	10.21929	3.336502

Table 3.7: Descriptive Statistics - Social Data Sources

Repository demographics						
Details	**Min**	**Max**	**Median**	**Mean**	**SD**	**CV**
Commits	101	7700	398.5	821.689	1231.91	1.499
Pull requests	0	2222	25.5	104.765	235.184	2.244
Issues	0	1070	25	81.509	141.58	1.736
Contributors	1	278	15.5	27.144	35.157	1.295
`.md` files	0	231	3	9.693	26.285	2.711

Table 3.8: Descriptive Statistics - `C++/Python` ROS Repositories

Repository demographics						
Details	**Min**	**Max**	**Median**	**Mean**	**SD**	**CV**
Commits	102	677	221	237.203	127.576	0.537
Pull requests	0	352	20	36.203	59.433	1.641
Issues	0	195	11	34.033	45.843	1.347
Contributors	2	43	8	11.457	9.156	0.799
`.md` files	0	49	1	2.983	6.587	2.208

Table 3.9: Descriptive Statistics - `C++` ROS Repositories

Repository demographics						
Details	**Min**	**Max**	**Median**	**Mean**	**SD**	**CV**
Commits	103	4175	208	452.045	696.664	1.541
Pull requests	0	746	7	61.909	149.865	2.420
Issues	0	848	8.5	42.5	130.595	3.072
Contributors	1	126	10	15.477	20.125	1.301
`.md` files	0	14	1	1	1	1.863

Table 3.10: Descriptive Statistics - `Python` ROS Repositories

It is also interesting to note that even though there are 59 `C++` ROS projects and 44 `Python` ROS projects, the number of commits, pull requests, issues, and contributors is higher in `Python`-based projects which might suggest that the `Python`-based ROS projects are in more active development. There are more `.md` file present in the `C++`-based ROS projects, which might also hint that the `C++`-based projects are more meticulously documented.

3.2.2 Phase 2: Energy-Relevant Data Identification

After successfully collecting, constructing and loading the initial dataset into a database in `MongoDB`, an *Energy Detector* is implemented and used to identify data specifically related to energy-efficiency. Figure 3.12 illustrates the architecture of our *Energy Detector*.

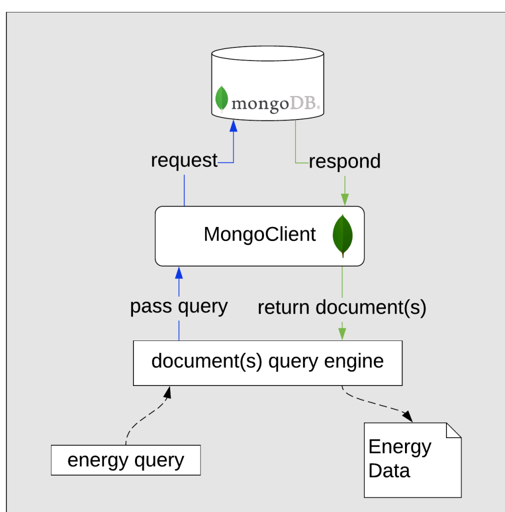


Figure 3.12: Energy Detector

To implement the *Energy Detector*, `PyMongo` - a python library is used to interact with the `MongoDB` database. The *Energy Detector* consists of a *document query engine* software component which is in charge of processing documents from a `MongoDB` collection. The *MongoClient* (a `PyMongo` object) serves as a bridge between the `MongoDB` database and the *document engine query* component by establishing a connection to the server where the `MongoDB` database is hosted. The *MongoClient* sends a query to the server and receives a document(s), and then forwards the document(s) to the *document query engine* to process. The *document query engine* then exports each processed document to a `JSON` file.

3.2.2.1 Energy Query

To mine data related to energy-efficiency and energy-awareness, a combination of different energy-related keywords are used to construct the *energy query*. The inspiration for constructing the *energy query* was taken from an earlier study on mining energy-aware commits from open-source repositories (20). It was important for us to include as many keywords as possible in order to capture all energy-related data. We agreed to include the

keywords *battery, energy, power, green* and *sustainability* in our query as all of these keywords typically concern energy-efficiency. We were debating whether or not to include the keywords *consumption* and *efficiency* in our query; we ruled out these two keywords out by agreeing that *consumption* and *efficiency* are typically paired with *energy* and *battery* keywords which are already present in our query. The following is the final constructed *energy query*:

(`*batter*` OR `*power*` OR `*energy*` OR `*green*` OR `*sustainab*`)

The `*` character acts as a wildcard: documents where the fields (e.g., post title, post content/answer, source code comment) contain at least one of the keywords will be selected regardless of the beginning or the end of the fields contents. The *energy query* is fed to the *document query engine* which uses it to retrieve energy-related data.

3.2.2.2 Document Query Engine

The *document query engine* is a software module written in Python. The sole responsibility of this module is to query a collection in the `MongoDB` database using the *energy query* and process the matched documents. Figure 3.13 shows an example of a *document query engine* module for the `StackOverflow` data source.

```
1 collection_key = ['title', 'post_content', 'answer', 'quote',
  ↪  'question_code', 'answer_code']
2
3 regex_keyword = ['.*battery.*', '.*energy.*', '.*power.*', '.*sustainab.*',
  ↪  '.*green.*']
4
5 for ck in collection_key:
6     energy_query = collection.find({"$or":[ {ck: {'$regex':
      ↪  regex_keyword[0]}},{ck: {'$regex': regex_keyword[1]}}, {ck:
      ↪  {'$regex': regex_keyword[2]}},{ck: {'$regex': regex_keyword[3]}},
      ↪  {ck: {'$regex': regex_keyword[4]}}]})
7         for document in energy_query:
8                 original_energy_list.append(document)
```

Figure 3.13: Document Engine Query

Line 1 defines the list of field names (keys) in the `StackOverflow` collection. ***Line 3*** defines the `regex` keywords of the *energy query*. In ***Lines 5-6***, the fields in the `StackOverflow` collection are queried with the `regex` keywords from the *energy query*. Finally, in ***Lines 7-8*** the returned documents by the *energy query* are appended to a list.

The same principle is used to implement the *document query engine* for the rest of the data sources.

3.2.2.3 Removing False-Positive Energy Data

After querying all of the collections in the `MongoDB` database for energy-related data, a total of 3,354 data points are identified to be related to energy. However, it is necessary to perform a manual selection before to proceeding to the next phase.

The purpose of the manual selection process of each data point is to ensure that the final energy-related dataset does not contain any false-positives. "Problems with powering on a robot", "Kinect green light keeps freshing", "What is the next ROS distribution?...I think most of OSRF's energy is focused on ROS2" are all examples of a false-positives that should not be present in the final energy-related dataset.

3.2.2.4 Level Of Agreement Calculation

The manual selection for deciding if a data point is indeed talking about energy is divided into two rounds; for each round, 50 data points from each collection are randomly selected and analyzed by two reviewers (each reviewer analyzes the same exact set of the randomly chosen data points in each round) in order to avoid bias. After completing each round, the level of agreement is assessed using Cohen's kappa. If the level of agreement is not high enough(Cohen's kappa is below .80), an arbiter is called to perform the manual selection a third time. Table 3.11 provides the summary of the Cohen's kappa results for each round.

	% of agreement	Cohen's kappa
Round 1	96.012%	0.904
Round 2	92.409%	0.804

Table 3.11: Cohen's Kappa Results Overview

The level of agreement is nearly perfect for both of the rounds and even though Cohen's kappa is less in Round 2 than in Round 1, it is still high enough to proceed without an arbiter.

3.2.2.5 Energy Dataset Summary

After completing both rounds of randomly selected data points, one reviewer is left to finish off the manual selection of the rest of the data points. A total of 562 data points are identified to be strictly related to energy.

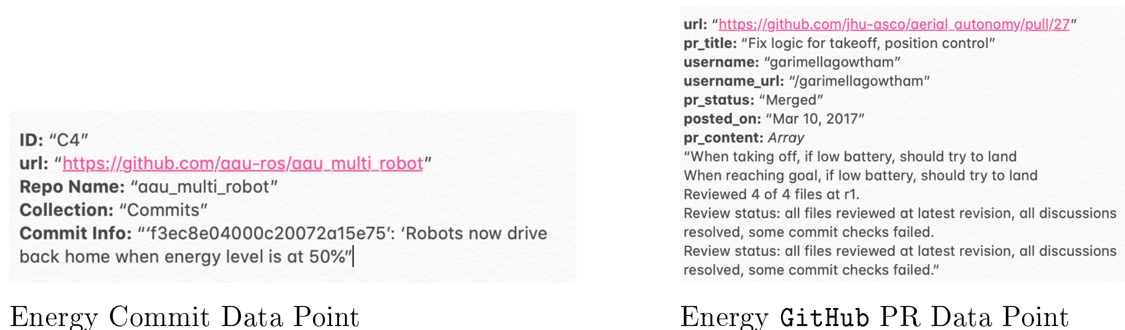
Data Source	# Data Points
BitBucket (PRs)	1
BitBucket SRC Comments	4
BitBucket Commits	3
GitHub PRs	62
GitHub Issues	6
GitHub SRC Comments	72
GitHub Commits	206

Table 3.12: Data Sources (1) Overview

Data Source	# Data Points
StackOverflow	3
ROS-Answers	170
ROS-Wiki	23
ROS-Discourse	12

Table 3.13: Data Sources (2) Overview

Tables 3.12 and 3.13 show the number of energy data points per data source. `StackOverflow` and `BitBucket` have significantly less energy data points in comparison to the other data sources indicating that majority of the content in these two data sources is not related to energy. Figure 3.14 shows examples of extracted energy-related data points.



Energy Commit Data Point

Energy `GitHub` PR Data Point

Figure 3.14: Examples of energy data points

3.2.3 Phase 3: Architecturally-Relevant Data Identification

In this phase, the 562 data points identified to be related to energy in *Phase 2* are filtered once more to identify data points where architecturally-relevant (AR) concerns are conferred (e.g., presence of integrator nodes, system layers). In order to figure out which data points talk about AR concerns, a manual approach is used to perform the identification instead of an automated method similar to the one used in *Phase 2*. Prior to settling down with the manual approach, we came up with three options on how to approach this phase of the project. Table 3.14 depicts the three different strategies for approaching this phase along with their pros and cons.

Option	Pro	Con
1. Manual approach: identify architecturally relevant data manually.	Maximum accuracy.	Time consuming.
2. Manual approach + architecture recovery tool: apply manual approach on StackOverflow, ROS-Answers, ROS-Wiki, ROS-Discourse. Apply an architecture recovery tool (e.g., Haros (10)) on repository comments, commits and data points with complete source code of the system(e.g, pull requests, issues).	More precise for data points with source code.	More time consuming in comparison to Option 1. Manual check will still be needed.
3. ML pipeline similar to (10): identify if a data point is an ARP (architecture-relevant post/data point). Apply preprocessing on the data points and classification feature. A manual check is still needed to ensure the final dataset is accurate.	Mostly an automated process.	Time consuming to develop. Manual check will still be needed.

Table 3.14: Proposed strategies for identifying architecturally-relevant data points

It is obvious from the above table that the first option - the manual approach is the winner, as the other two options still require a manual approach at the end for ensuring that the resulting dataset is accurate. We also have to deal with only 562 data points in this phase, so we agreed that the manual approach would not be very time-consuming and decided to proceed with it, as it promises maximum accuracy. The 562 data points are manually analyzed and classified using a set of constructed inclusion and exclusion criteria. The result is a dataset with strictly AR data points.

3.2.3.1 Data Point Types

In the energy dataset there are two kinds of data points: a *code* data point and a *social discussion* data point. A *code* data point is a data point where code is the main source

of information and likewise, a *social discussion* data point is a data point where textual discussions are the primary source of information.

Table 3.15 lists the data sources of *social discussion* and *code* data points. A `GitHub` issue can be classified either as a *social discussion* data point or a *code* data point reason being that sometimes the issue might be solely a conversation post with no linked code, thereby classified as a *social discussion* data point. Due to this heterogeneous nature of the dataset, 3 lists of inclusion/exclusion criteria are compiled to accommodate the filtering of the data points.

Data Point	Type
`StackOverflow`	Social discussion
`ROS-Answers`	Social discussion
`ROS-Wiki`	Social discussion
`ROS-Discourse`	Social discussion
`GitHub/BitBucket` commit	Code
`GitHub/BitBucket` pull request	Code
`GitHub/BitBucket` src comments	Code
`GitHub` issue	Code, Social Discussion

Table 3.15: Data Point Types

3.2.3.2 Inclusion & Exclusion Criteria

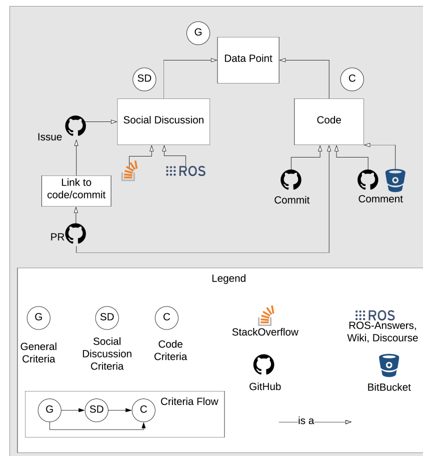


Figure 3.15: Inclusion & Exclusion Criteria Flow

Figure 3.15 illustrates the process of filtering the data points via the 3 sets of inclusion/exclusion criteria. A *General* inclusion/exclusion list is built to serve as the starting point for classifying the energy dataset. This list is composed of basic definitions for an AR data point such as data points concerning ROS entities (see Table 3.16) or data points concentrating on energy consumption. In the case where the *General* list is insufficient to judge whether or not a data point is architecturally-relevant, a *Social Discussion* or a *Code* criteria inclusion/exclusion list are used to assess the data point depending on the nature of the data point's source. These lists are constructed in a more meticulous fashion compared to the *General Criteria* list in order to further accurately asses the data points.

ROS Entities
1. Node/nodelet
2. Topic
3. Publisher
4. Subscription
6. Service call
7. Action client
8. Action server

Table 3.16: ROS Entities

Inclusion Criteria	**Exclusion Criteria**
I1. Data points concerning an architectural entity (*see ROS entity table*)	E1. Data points only about visualizing energy data (e.g., using green leds for battery levels)
I2. Data points focussing on architectural solutions (e.g., tactics, patterns, styles, views, models, reference architectures).	E2. Data point reporting only technical specifications or fact sheets of a robot (e.g., fact sheet of a commercial robot)
I3. Data points focussing on energy consumption.	E3. Data points related to low-level aspects whose scope is only on the inner details of ROS nodes.

Table 3.17: General Criteria List

Inclusion Criteria	Exclusion Criteria
I1. Data points mentioning architecturally-relevant design decisions and rationales	E1. Data points without any relevant discussion (e.g., a question in `StackOverflow` without any answers).
	E2. Data points asking about a (system) energy warning/error (i.e. laptop battery state) but with no relevance to SA.
	E3. Tutorial-like data points (e.g., a `StackOverflow` question asking how to install a specific ROS package).

Table 3.18: Social Discussion Criteria List

Inclusion Criteria	Exclusion Criteria
I1. Data points containing architecturally -relevant source code(referring to at least one architectural element of ROS - see ROS entity table).	E1. Data points reverting or redoing the same source code changes performed in an already-considered data point.

Table 3.19: Code Criteria List

Tables 3.17, 3.18, 3.19 present the inclusion and exclusion criteria for the *General*, *Social Discussion* and *Code* lists. A data point is *not* classified architecturally-relevant *if and only if* at least one of the exclusion criteria is satisfied.

3.2.3.3 AR Data Point Identification

The following is an example of the classification process of a sample data point coming from the actual energy dataset which is depicted in Figure 3.16.

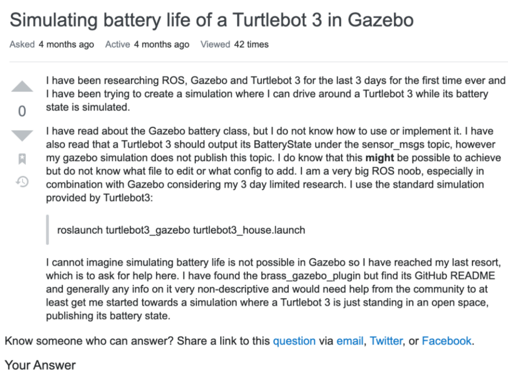


Figure 3.16: `StackOverflow` sample data point

In summary, this `StackOverflow` post asks how to navigate a `Turtlebot 3` while simulating its battery state. The user is looking for input from experienced ROS developers.

First, the *General* criteria list is used to identify whether or not this data point is architecturally relevant. It is clear that the *General* criteria list is insufficient for this data point as there is no evidence of architectural ROS entities or solutions being discussed in the post. Secondly, being a `StackOverflow` data point, the *Social Discussion* criteria list is used to classify the post. The data point satisfies exclusion criteria **E1** and **E3** since `(i)` there are no answers, comments or discussions attached to this post and `(ii)` the question asks how to perform, configure a ROS package, implying that this is a *tutorial-like* post. With this type of reasoning, it is evident that this data point is *not* architecturally relevant. The rest of the data points in the energy dataset are classified using the same identification method and logical reasoning.

3.2.3.4 Level Of Agreement Calculation

Prior to filtering the entire energy dataset, the initial classification is divided into two rounds similar to the one in *Phase 2*; for each round, 25 data points from each data source are randomly selected and analyzed by two researchers. Each researcher analyzes the same exact set of the randomly chosen data points in each round in order to mitigate the possibility of bias. After completing each round, the level agreement is once more assessed using Cohen's kappa. Table 3.20 shows the summary of the Cohen's kappa results for each round.

	% of agreement	Cohen's kappa
Round 1	100%	1
Round 2	97.11538461538461%	.9259259259259259

Table 3.20: Cohen's Kappa Results Overview

As shown in the table, the level of agreement is perfect for Round 1 and nearly perfect for Round 2, indicating that no bias is present in the classification process. The rest of the dataset is classified by one of the reviewers. A total of 97 data points are identified to be architecturally relevant.

3.2.3.5 AR Dataset Summary

Data Source	# Data Points
`BitBucket` SRC Comments	2
`GitHub` Issues	1
`GitHub` PRs	24
`GitHub` SRC Comments	16
`GitHub` Commits	38
`ROS-Answers`	7
`ROS-Wiki`	7
`ROS-Discourse`	2

Table 3.21: Data Source Overview

Table 3.21 shows the overview of the final AR dataset. `StackOverflow` is not present in this dataset indicating that there are no AR-relevant data points present in this data source. `GitHub` PRs and `GitHub` commits seem to carry most of the AR data points, which emphasizes that `GitHub` is a heterogeneous data source.

title: “Have two nodes publish to same topic”
time: “2019-03-18 12:01:25 -0500”
post_content: *Array*
“So I have two nodes publishing to the topic /mavros/setpoint_position/local and I want one to have priority over the other. Right now it just switches between the two. Is there a way to have them both publish then assign priority to one?
More info on my particular problem. I have a exploration node sending to the topic, then have another node which sends only when battery is too low. I don't want to change anything in the exploration alghoritm, just have the battery node publish to the topic / mavros/setpoint_position/local and overwrite what the exploration node publishes.”
post_answer: *Array*
“It sounds like multiplexing may help.
mux is a ROS node that subscribes to a set of incoming topics and republishes incoming data from one of them to another topic, i.e., it's a multiplexer that switches an output among 1 of N inputs. Services are offered to switch among input topics, and to add and delete input topics. At startup, the first input topic on the command line is selected.”
url: “https://answers.ros.org/question/318738/have-two-nodes-publish-to-same-topic/”

AR ROS-Answers Data Point

url: “https://github.com/ipa320/cob_command_tools/pull/129”
pr_title: “use aggregated power state message”
username: “fmessmer”
username_url: “/fmessmer”
pr_status: “Merged”
posted_on: “Feb 4, 2016”
pr_contents: *Array*
“Will this require any change in the diagnostic_aggregator.yaml's or diagnostic launch files for the robots?
PRs have the same name, but they are not cross-referenced:
ipa320/cob_driver#259
ipa320/cob_common#182
why has this Subscriber been removed?
accesspoint monitoring is not working since years and was never used. I now only use the features which are working and supported on all robots.
FR: ok
Already tested on a robot?
as discussed...and travis is happy...merging”

AR GitHub PR Data Point from cob_command_tools

Figure 3.17: Examples of AR data points

```
-       # Circuit Breaker
-       rospy.Subscriber("power_board/state", PowerBoardState, self.powerBoardCB)
-       self.last_power_board_state = 0
-       # Battery
-       rospy.Subscriber("power_state", PowerState, self.powerCB)
-       self.last_power_state = 0
-       # Wireless
-       rospy.Subscriber("ddwrt/accesspoint", AccessPoint, self.accessPointCB)
-       self.last_access_point = 0
+       # Diagnostics
+       rospy.Subscriber("diagnostics_toplevel_state", DiagnosticStatus, self.DiagnosticStatusCB)
+       # Power state
+       rospy.Subscriber("power_state", PowerState, self.PowerStateCB)
+       # Emergency stop state
+       rospy.Subscriber("emergency_stop_state", EmergencyStopState, self.EmergencyStopStateCB)
```

Figure 3.18: Sample of GitHub PR code from cob_command_tools

Figure 3.17 illustrates examples of AR data points from ROS-Answers and GitHub (pull request). Figure 3.18 shows a sample piece of code for the GitHub pull request in Figure 3.17. Here we can see that depicted piece of changed code contains an architectural change; the *dashboard_aggregator node* is subscribed to different *topics* which are published by different *nodes*.

3.2.4 Phase 4: Green Tactics Extraction

The 97 architecturally relevant data points identified in *Phase 3* are carefully examined in order to identify and extract green architectural tactics. The identification and extraction of green tactics is conducted via a manual approach divided into 5 stages; **(i)** a list of parameters is identified for extracting AT-relevant data from the 97 AR data points (some data points are discarded if it is not possible to extract green tactics from them), **(ii)** three researchers compose separate lists describing potential tactic categories and merge the lists together to establish the final tactic categories, **(iii)** the final tactic category list is then used to classify the final set of data points , **(iv)** a tactic tree depicting the identified green tactics and their corresponding families is compiled, and **(v)** each identified tactic is described via the tactic template established in study (10).

3.2.4.1 Stage 1: AT-Relevant Data Extraction

In this stage, AT-relevant data is extracted from each AR data point. In order to ensure meticulous and sound results, several parameters are identified for the data extraction process. These parameters indicate what kind of data is extracted from the data points. The following is a list of parameters identified for the extraction of AT-relevant data from each data point:

- `code` - *list[a,b,...]*: this parameter is used to identify if the data point has any relevant source code.
- `targeted QA` - *list[a,b,...]*: this parameter is used to identify the QAs targeted by the data points.
- `tactic-relevant contents` - *String*: this parameter is used to extract the relevant tactic descriptions from each data point.
- `response` - *String*: this parameter is used to extract the relevant-tactic response from each data point (e.g., saves energy, warns user).
- `ROS entity` - *list[a,b,...]*: this parameter is used to extract the relevant ROS-entities featured in the data points.
- `external data sources` - *String*: this parameter is used to identify other data sources mentioned in the data points.

- `is tactic` - *Boolean*: this parameter is used to identify whether or not an architectural tactic can be extracted from the data point based on the previously identified parameters.

After characterizing each data point using the parameters described above, 67 data points are identified to contain green tactics, and *energy-efficiency* and *energy-awareness* are identified as the two QAs targeted by the 67 data points. Figure 3.19 shows an example of a characterized `GitHub` pull request data point which is identified as a source for a green tactic.

Source: "GitHubPR"
Title: "CAMERA: Add stereo launch file"
Code: "YES"
Targeted QA: "Energy Efficiency"
Tactic-Relevant Contents:
"If camera is not being used, do not have the ROS
nodes using its data running and consuming energy"
Response: "save energy by brigning up/shutting down nodes"
ROS entity: "nodelets"
External data sources: "https://github.com/uf-mil/SubjuGator/pull/76/files"
Is Tactic: "YES"

Figure 3.19: Data point identified as a source for a green tactic

3.2.4.2 Stage 2: Green Tactics Categories Identification

To avoid bias and to ensure that the final list of green tactics is sound, three researchers compose a separate list containing the description of the tactics targeting energy-efficiency and energy-awareness based on the AT-relevant data extracted in the previous stage. During this process, the source code of each data point is closely inspected to understand how the data point is implemented. After the compilation of the three tactic category lists, the following observations were made:

- Researcher #1: provided general descriptions of the tactic (e.g., tactic families).
- Researcher #2: provided general descriptions of the tactic (e.g., tactic families).
- Researcher #3: provided specific scenarios based on the contents of the data points (e.g., technical descriptions).

Based on these observations, we came to an agreement to merge the three tactic category lists. Even though we mainly focus on ROS-based systems, we ensure that the identified tactics are not limited to only ROS-based systems, but can be applied in other robotics systems as well. Tables 3.22 and 3.23 depict the identified energy-awareness and energy-efficiency tactics. The tables provide the tactic ID, the name of the tactic, the family the tactic belongs to, and a short tactic description.

ID	Name	Family	Description
EA1	Abort Mission	Task Interruption	Aborts a task when the energy level of the robot is too low.
EA2	Stop Task & Recharge	Task Interruption	Stop a task when the battery level of the robot is too low, recharge the battery and resume the task.
EA3	Dedicated Energy-Level Message	Energy-Level Provider	Monitors the energy level of the robot and provides it in a dedicated message.
EA4	Energy-Level Info Within Diagnostics Message	Energy-Level Provider	Includes the energy level of a hardware component within a generic diagnostics message.
EA5	Aggregated Energy Information	Energy-Level Provider	Aggregates energy-level information of different components into a single interface.
EA6	Energy-Savings Mode	Energy-Level Provider	Dictates to the components in a system whether or not to enter into a state in which energy must be saved.
EA7	Offline Energy Profiler	Other	Builds an energy profiler from previous executions which is then used in a current execution for providing energy level state diagnostics.

Table 3.22: Energy-Awareness Tactics

ID	Name	Family	Description
EE1	Limit Task	Energy-Efficiency	Configures a task based on any set criteria for tasks under the energy-savings mode.
EE2	Disable HW	Energy-Efficiency	Disables hardware components when they are not strictly needed.
EE3	Energy-Aware Sampling	Energy-Efficiency	Adjusts the rates for sampling a sensor based on the energy-level of the robot.
EE4	On-Demand Components	Energy-Efficiency	Brings up new components only when their functionalities are needed.

Table 3.23: Energy-Efficiency Tactics

3.2.4.3 Data Point Classification

After establishing the green tactic categories in the previous stage, the 67 data points are classified using Tables 3.22 and 3.23. *During the classification of the data points, it was evident that some of the data points employ more than one tactic.* Figure 3.20 provides an overviews of the classification of the 67 data points by assigning a tactic(s) to each data point.

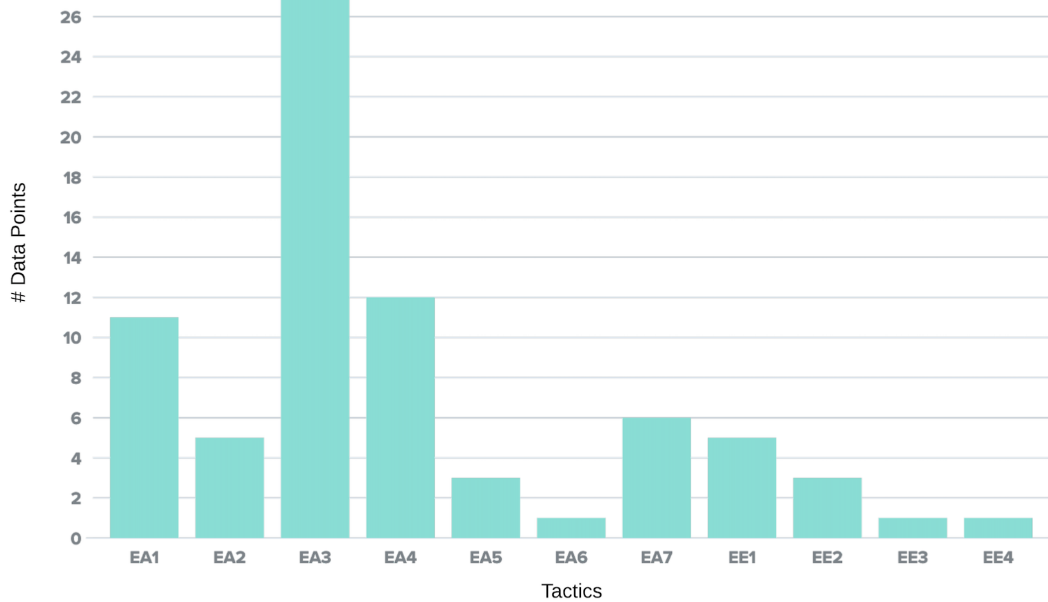


Figure 3.20: Data Points Classification Summary

After classifying and associating each data point with a tactic(s), it is easy to notice that energy-awareness tactics are prominent. A large portion of the data points relate to energy-level warnings, indicating that ROS developers are constantly trying to promote energy-awareness in systems they develop, which implies that energy is always a concern in robotics-software systems. This observation leads to the fact that even though the presence of energy-efficient tactics in the final dataset is slightly less than the presence of energy-awareness tactics, the energy-efficient tactics identified target different areas of the robotics-software systems which shows their diversity and importance.

Stages 4 and 5 - that compiled tactic tree and detailed descriptions of each tactic are presented in the *Results* section of this paper.

4

Results

In this chapter, the tactic tree and the extracted green tactics are presented.

4.1 Tactic Tree

Figure 4.1 illustrates the tactic tree that is constructed using Tables 3.22 and 3.23 presented in *Phase 4, Stage 2*. The energy tactics are divided into two categories: *energy-awareness* and *energy-efficiency tactics*.

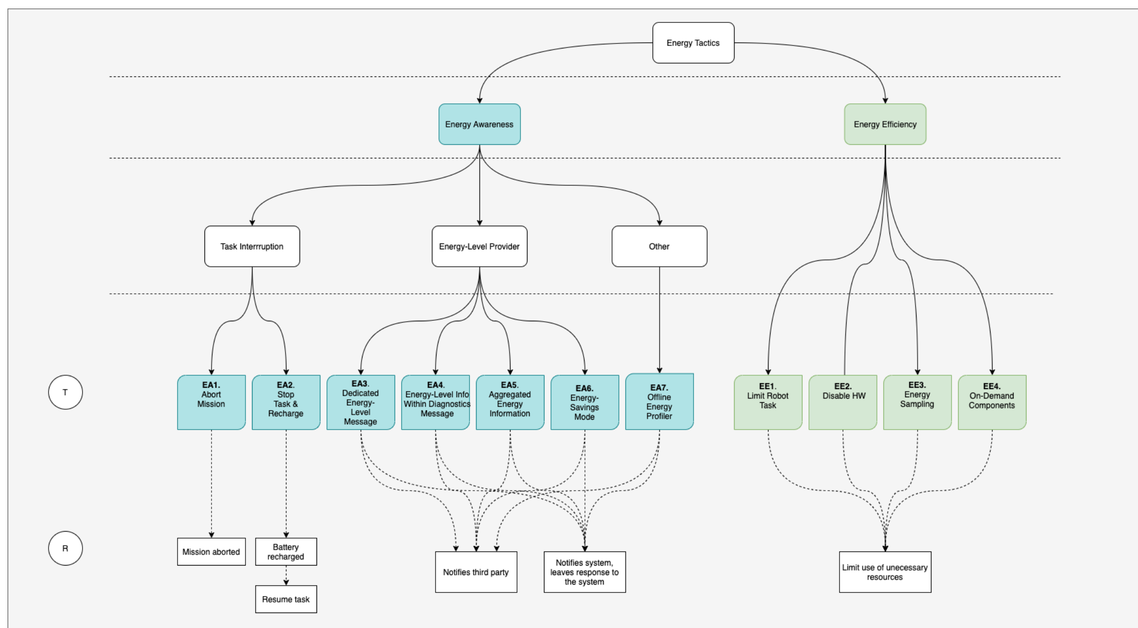


Figure 4.1: Tactic Tree

The second level of the tree contain the families identified for some of the tactics. The energy-awareness tactics are grouped under three families: tactics EA1 and EA2 belong to the Task Interruption family as their essential goal is to interrupt a task when the energy-levels are critical, tactics EA3-EA6 belong to the Energy-Level Provider family as these tactics provide some means of delivering the energy-level to the user or a third party, and tactic EA7 belongs to the *Other* family as it does not fit the criteria for a *Task Interruption* or *Energy-Level Provider* tactic. Tactics EE1-EE4 are direct children of the *Energy-Efficiency* family, as the main goal of these tactics is to save energy. The third level of the tree marked with T represents the actual tactics denoted as the leaves of the tree. The final level of the tree marked with R shows the responses of the green tactics.

4.2 Green Tactics

In this section, the 11 green tactics are presented using the tactic template introduced in study (10). The template is slightly adjusted by introducing new fields. Table 4.1 shows the contents of the adjusted tactic template and how each tactic is described.

Tactic Name	The name of the tactic.
Targeted QA	The QA targeted by the tactic.
Family	The family the tactic belongs to.
Motivation	The rationale behind the tactic.
Description	A component interface + sequence diagrams followed by a detailed explanation of the diagrams.
ROS Example	A diagram depicting the tactic in a ROS-based system followed by a detailed explanation of the diagram. The example is taken from an actual data point in the dataset. Section 7.1 in the Appendix provides the list of ROS projects used to formulate the ROS example.
Other Examples	Examples of other application domains where the tactic is used.
Constraints	Assumed conditions in order to apply the tactic in an existing robotic system.
Dependencies	(Optional) Other tactics required by the tactic.
Variations	(Optional) Different ways of implementing the tactic.

Table 4.1: Tactic Template

Since the scope of our project is limited to ROS-based systems, we provide specific ROS examples for each tactic illustrating a specific scenario presented in some of the data points in the dataset. However, even though our dataset mainly focuses on ROS-based systems, we believe that our tactics can be applied in other robotic systems, therefore, we present the extracted tactics in a general fashion by not limiting the tactic descriptions to ROS-specific cases. We also ensure that we do not impose a specific implementation of the tactics: for this reason, we use component diagrams to depict the components involved in the system and the interfaces provided and required by the components. A sequence diagram is also included to illustrate the order of interactions between the components and

how operations are carried out. It is up to the developer(s) to decide how to carry out the tactic implementations and which tools to use. The extracted green tactics will help roboticists extend their design reasoning and development towards energy-efficient robotics software.

Table 4.2 contains the motivations for different ROS and non-ROS communication methods depicted in the ROS examples of the tactics:

Topic	Typically, a *ROS-topic* is used to communicate the energy-level state information to a ROS node as topics are used for one-way continuous data streams (21).
Action	Actions are typically used for any type of behaviour that involves moving the robot, or tasks that run for a longer time. Actions can also be preempted and they satisfy real-time constraints and provide feedback during the execution of a task (22).
Service	Service calls are blocking so typically, they are used for remote procedure calls that terminate quickly (e.g., query state of a node, calculations) (23).
Bag	A bag is used to represent the logged energy data as bags are the primary mechanism in ROS for data logging (24).
Non-ROS communication	Information sent directly from a physical device employs non-ROS communication methods as a non-ROS component represents the hardware component and the communication does not use ROS primitives.

Table 4.2: ROS Communication Method Motivations

Energy awareness is a crucial tool in robotics software systems as robots are often required to manage their energy wisely. Tactics EA1-EA7 help in facilitating robot energy management by providing different methods for promoting energy-awareness in the robotics software systems.

4.2.0.1 EA1: Abort Mission

- **Tactic Name**: Abort Mission
- **Targeted QA**: Energy Awareness
- **Motivation**: This tactic offers system-wide energy-awareness by monitoring the state of the robot's energy level and ensuring that tasks execute to completion only if the energy levels are enough for the robot to execute them safely.
- **Description**: This tactic aborts a task when the energy-level of the robot reaches a critical point. Figure 4.3 presents the component interface and sequence diagram of the tactic.The *Task Requestor* is responsible for requesting to execute a certain task, the *Arbiter* is responsible for deciding whether or not to abort or execute the task, and the *Task Executor* is responsible for either aborting or executing the task. First, the *Task Requestor* creates a task and sends it to the *Arbiter* (labels 1,2). After receiving an initial task, the *Arbiter* gets the energy-level of the robot (labels 3, 4); this information is provided to the *Arbiter* by another component in the system. If the energy-level is critical, the *Arbiter* immediately removes the task (label 15, the task is not forwarded to the *Task Executor*). In the case when the energy-level is sufficient, in a separate thread, the *Arbiter* sends the task to the *Task Executor* (label 5) and the *Task Executor* starts executing the task (label 6). In another parallel thread, the *Arbiter* starts checking the energy-level within a loop (labels 7,8). If the energy-level remains to be sufficient throughout the entire execution of the task, the *Task Requestor* is notified that the task has been completed (labels 13, 14). If during the execution of the task the energy-level becomes critical, a break statement is issued and the loop is exited (break fragment). The *Arbiter* then instructs the *Task Executor* to gracefully abort the task (labels 9, 10, 11, 12). *The order of the calls in the sequence diagram does not necessarily take place in a chronological order: for example, after label 4, label 15 may take place instead of label 5, depending on the scenario.*
- **ROS Example**: Data point #41: Figure 4.2 below shows how the *Abort Mission tactic* is implemented in an autonomous multi-robot ROS-based system. There are two nodes involved - the *explorer node* and the *exploration_planner node.* The *explorer node* represents the *Task Executor* and the *Arbiter*; it creates a task and is also subscribed to a *battery_state topic* which advertises information regarding the

robot's battery such as the battery percentage. The *exploration_planner* node advertises an *explore action* (starts executing a task) and an *exploration_finished action* (aborts a task). The *explorer node* sends a task to the *exploration_planner node* and instructs the *exploration_planner node* to either start exploring (start task), or finish exploring (abort task), depending on the battery_level.

- **Other Examples**: (i) A quadrotor (UAV) - dictate to land when the energy levels are critical. (ii) An autonomous marine surface vehicle - instruct to power off vehicle when energy levels are critical.
- **Constraints**: The tactic as described assumes that the communication between the physical device and the *Arbiter* is already implemented.
- **Dependencies**: This tactic can be combined with tactic EA3 (e.g., provide the energy level in dedicated energy-level component) or EA4 (e.g., include the energy level in some generic component along with other diagnostics information).
- **Variations**: (i) Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime. (ii) The *Arbiter* may already check the energy-level in a loop prior to receiving a task.

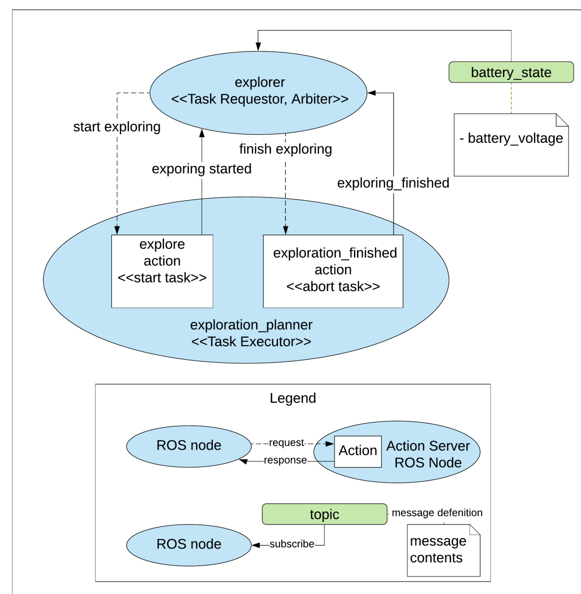


Figure 4.2: EA1 ROS Example

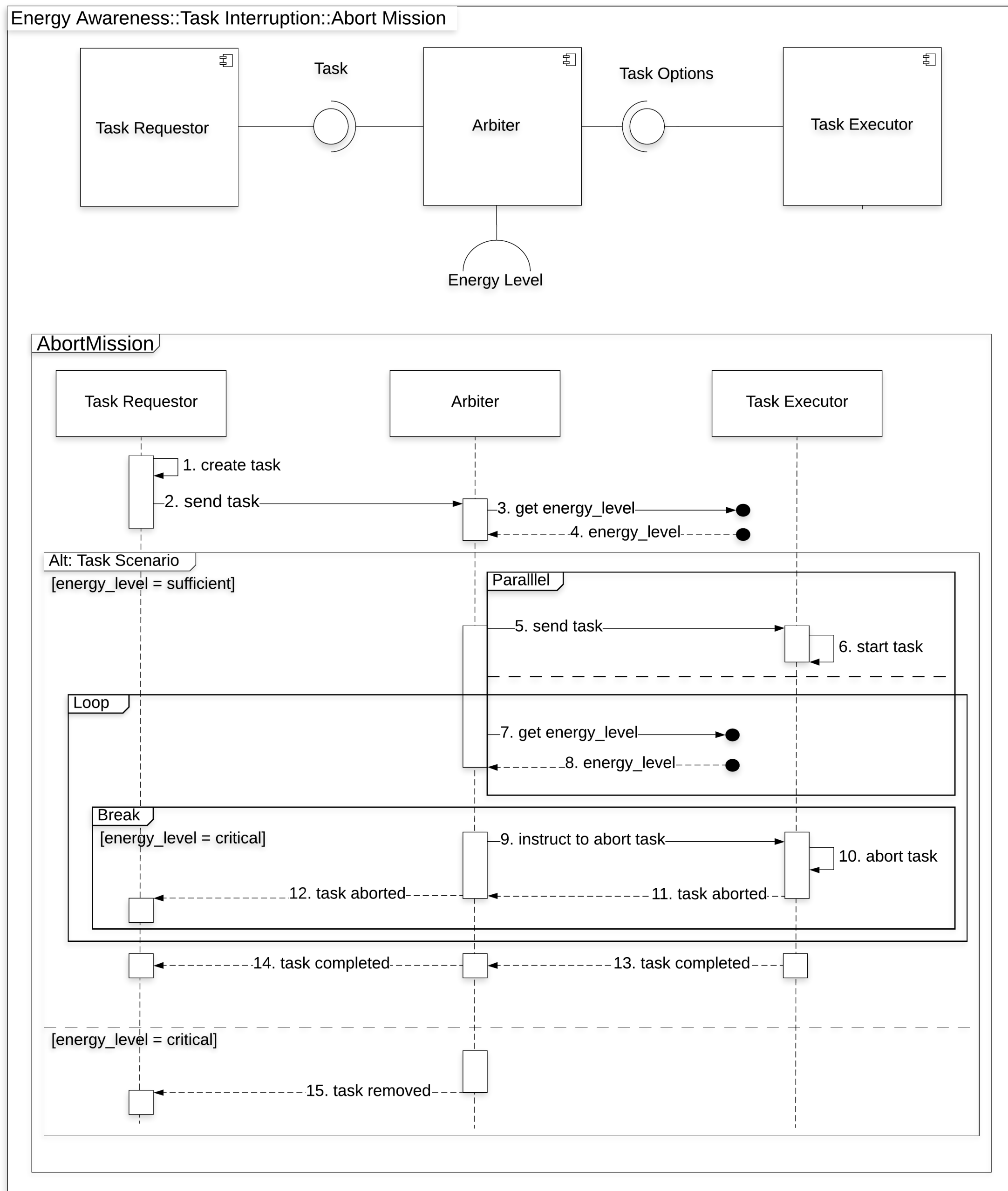


Figure 4.3: EA1 Component Interface & Sequence Diagram

4.2.0.2 EA2: Stop Task & Recharge

- **Tactic Name**: Stop Task & Recharge
- **Targeted QA**: Energy Awareness
- **Family**: Task Interruption
- **Motivation**: Ensuring that robots are able to continuously execute a task is an important part of continuous, repetitive, and autonomous robotic systems. Human intervention is not always available when the battery levels are critical, therefore, it is important that the robot is energy-aware, able to replenish its battery power when needed, and is able to safely complete its current mission. This tactic is useful when it is essential that the task is completed.
- **Description**: This tactic gracefully interrupts a task to prevent the robot from fully discharging its battery by instructing it to recharge when the energy level reaches a critical point. The task is resumed when the battery is sufficiently charged. Figure 4.5 shows the component interface and sequence diagrams for this tactic. The *Task Requestor* is responsible for requesting to execute a certain task, the *Arbiter* is responsible for deciding whether or not to stop a task and recharge the battery or execute the task, and the *Task Executor* is responsible for either stopping or executing the task. After creating a task, the *Task Provider* sends the task (label 2) to the *Arbiter* which then checks the energy-level of the robot's battery. If the energy-level is critical, the *Arbiter* immediately removes the task (label 23, the task is not forwarded to the *Task Executor*). In the case when the energy-level is sufficient, in a separate thread, the *Arbiter* sends the task to the *Task Executor*(label 5) and the *Task Executor* starts executing the task (label 6). In another parallel thread, the *Arbiter* starts checking the energy-level within a loop (labels 7,8). If throughout the entire execution of the task the energy-level stays sufficient, the *Task Requestor* is notified about the completion of the task (labels 21, 22). If during the execution of the task the energy-level becomes critical, a break statement is issued and the loop is exited (break fragment). A new loop is issued with two parallel threads running inside: one thread where the *Arbiter* checks for the energy-level (labels 9, 10), and another thread where the *Arbiter* decides what to do based on the energy-level. If the energy-level is critical, the *Arbiter* will instruct the *Task Executor* to stop the task (label 11) and request another component in the system to recharge the battery

(labels 15, 16). Once the battery is recharged (the energy-level is sufficient), the *Arbiter* instructs the *Task Executor* to resume the task (label 17). If the energy-level drops to a critical point, the previously described steps will take place (labels 11-16). *The order of the calls in the sequence diagram does not necessarily take place in a chronological order: for example, after label 4, label 23 may take place instead of label 5,depending on the scenario.*

- **ROS Example**: Data point #24: Figure 4.5 shows how the *Stop Task & Recharge tactic* is implemented in an autonomous multi-robot ROS-based system. There are two nodes involved - the *explorer node* and the *exploration_planner node.* The *explorer node* represents the *Task Executor* and the *Arbiter*; it creates a task and is also subscribed to a *battery_state topic* which advertises information regarding the robot's battery such as the battery percentage. The *exploration_planner node* advertises an *explore action* (starts executing a task) and an *exploration_finished action* (stops a task). The *explorer node* sends a task to the *exploration_planner node* and instructs the *exploration_planner node* to either start exploring (start task), or finish exploring (stop task), depending on the battery_level. If the battery level is critical, the *explorer node* sends a request to the *exploration_finished action* to stop the task. It then saves the current progress of the task by calling an internal method. It then instructs another ROS node in the system to recharge the battery. Once the battery is sufficiently charged, the *explorer node* sends a request to the *explore action* to resume the interrupted task by providing the saved task information.

- **Other Examples**: (i) Autonomous vehicle - instruct vehicle to stop driving and recharge when energy levels are critical. (ii) Robot executing tasks via SMACH (state machine) - store state of containers, and when the battery level is critical, navigate robot to a charging station, and resume current state when the battery is sufficiently charged.

- **Constraints**: The tactic as described assumes that the communication between the physical device and the *Arbiter* is already implemented.

- **Dependencies**: This tactic can be combined with tactic EA3 (e.g., provide the energy level in dedicated energy-level component) or EA4 (e.g., include the energy level in a generic component along with other diagnostics information).

- **Variations**: (i) Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime. (ii) The *Arbiter* may already check the energy-level in a loop prior to receiving a task.

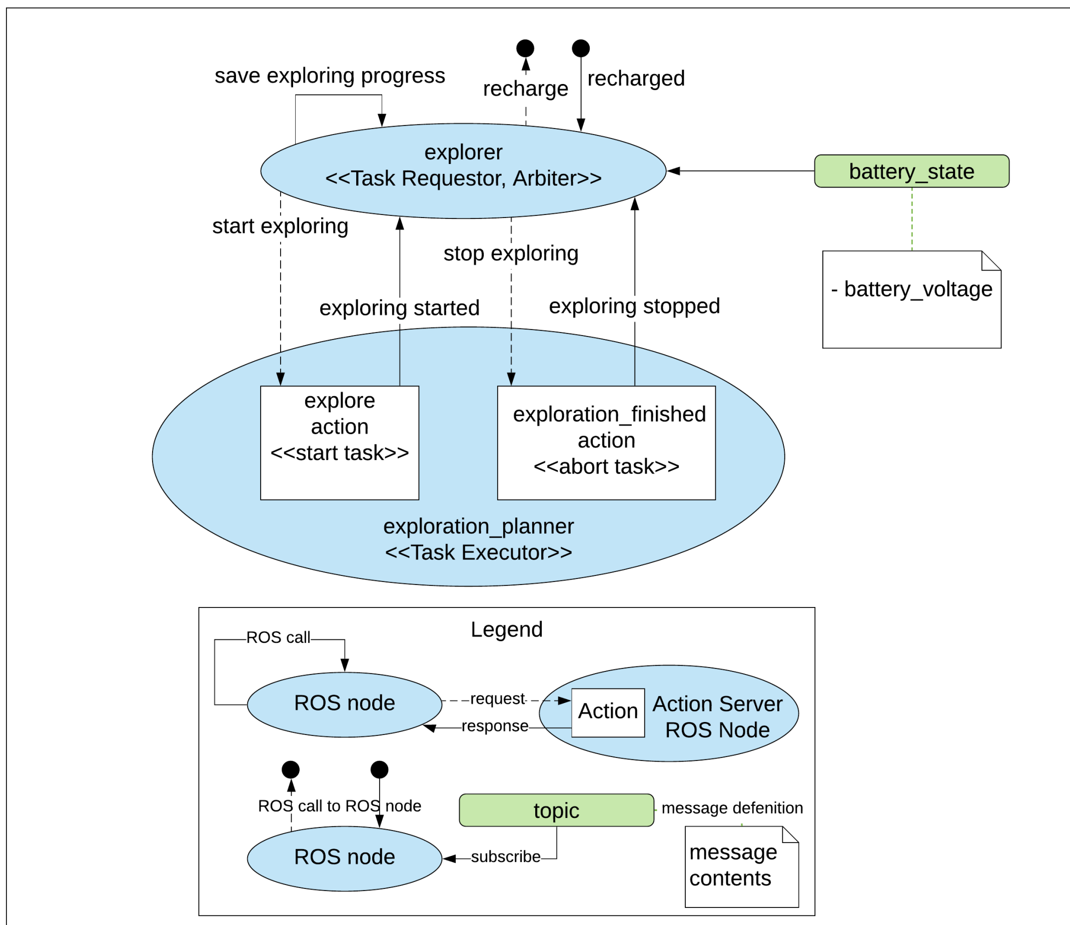


Figure 4.4: EA2 ROS Example

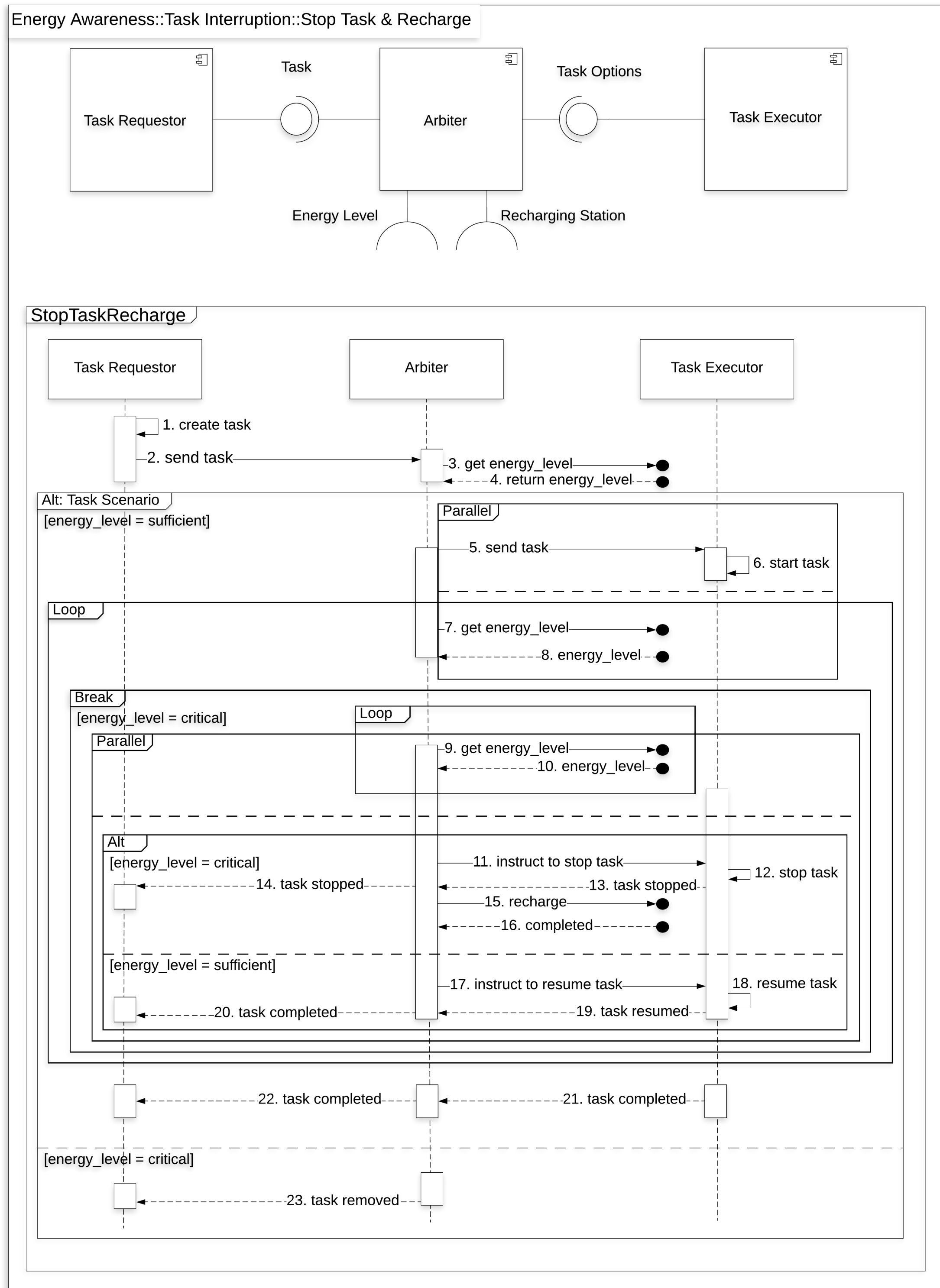


Figure 4.5: EA2 Component Interface & Sequence Diagram

4.2.0.3 EA3: Dedicated Energy-Level Message

- **Tactic Name**: Dedicated Energy-Level Message
- **Targeted QA**: Energy Awareness
- **Family**: Energy-Level Provider
- **Motivation**: The *Dedicated Energy-Level Message tactic* comes in handy when robots require continuous energy-level management as the message communicates information regarding the robot's energy information. This information can be used in several scenarios to promote energy awareness, such as: i) a component that has access to the dedicated energy-level message and uses the information to control the energy-level of a component (e.g., physical battery), ii) without the need for human intervention, a robot can also use the dedicated energy-level message to take certain actions to manage the energy state on its own, and iii) a component which has access to the dedicated energy-level message and uses the energy-level information to notify the user directly. This tactic is ideal when a dedicated single access point for energy-level information is needed.
- **Description**: This tactic monitors the energy level of the robot and provides it in a dedicated message. Figure 4.7 shows the component interface and sequence diagram for this tactic. The *Energy Level Manager* gets the energy level information (label 1) directly from the *Energy Level Provider* (e.g., physical battery) and provides this information in a dedicated *Energy Level State Interface* which can be accessed by other components in the system.
- **ROS Example**: Data Point #22: Figure 4.6 illustrates an example of the Dedicated Energy-Level Message tactic and how it is employed in a maritime ROS-based system. In this tactic, there is one ROS node and one non-ROS component: the *battery_monitor node* which represents the *Energy Level Manager* (a ROS node which monitors the state of the battery) and the *battery component* - the *Energy Level Provider* (a non-ROS component which represents the physical battery of the robot). The *battery component* is responsible for sending the information regarding the robot's battery via a non-ROS communication method to the *battery_monitor node*. The *battery_monitor node* publishes the received battery information in a *battery_state topic* which can then be subscribed to by any other ROS nodes in the system.

- **Other Examples**: (i) Energy management in mobile devices: Cinder operating system allows users and applications to control and manage limited device resources such as energy by collecting information about the device's battery (25). (ii) Energy management for cloud computing: collecting sensor data from multiple locations and managing physical sensor devices (26).
- **Constraints**: The tactic as described assumes that the communication between the physical device and the *Energy Level Manager* is already implemented.
- **Dependencies**: -
- **Variations**: (i) The energy level information can be provided by a software component (which is talking directly to the physical battery). (ii) Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point at runtime.

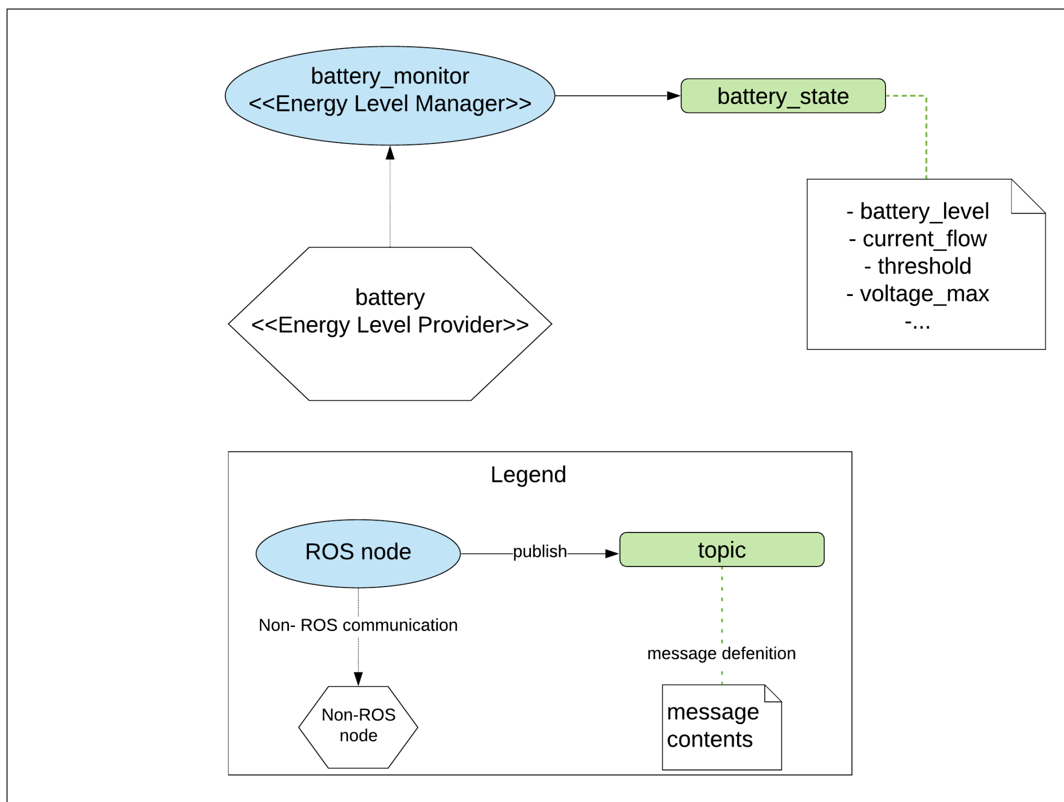


Figure 4.6: EA3 ROS Example

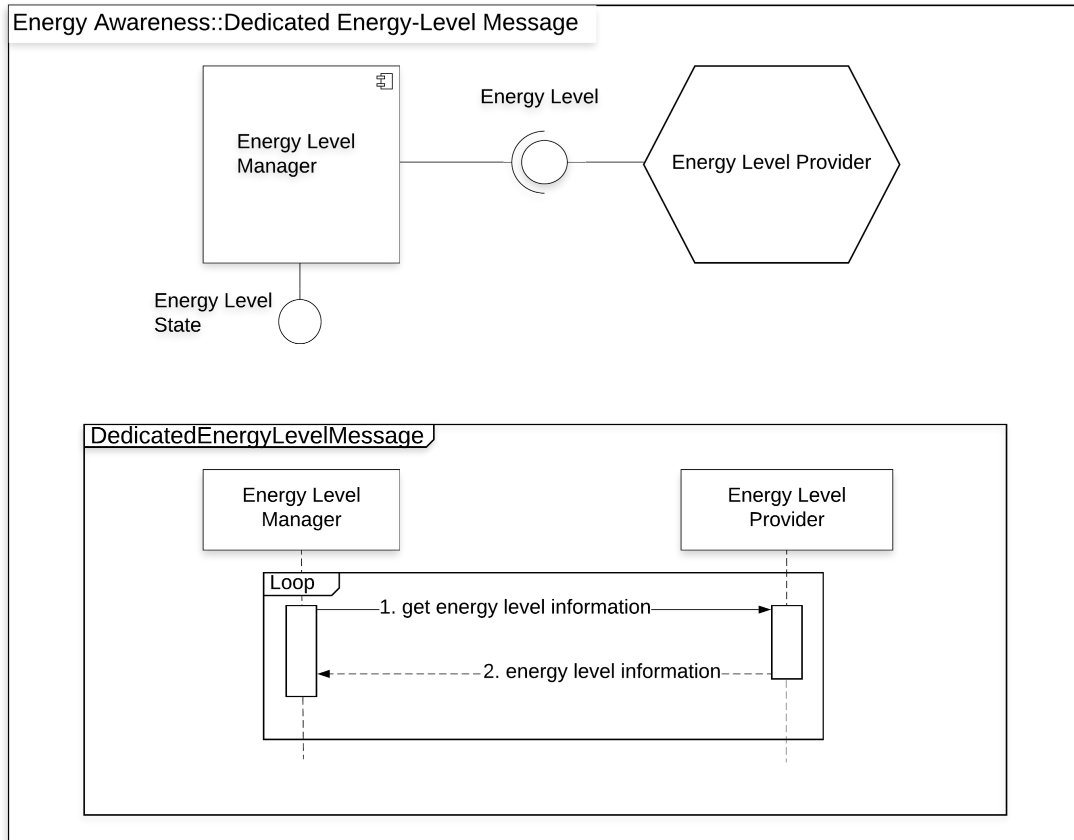


Figure 4.7: EA3 Component Interface & Sequence Diagram

4.2.0.4 EA4: Energy-Level Info Within Diagnostics Message

- **Tactic Name**: Energy-Level Info Within Diagnostics Message
- **Targeted QA**: Energy Awareness
- **Family**: Energy-Level Provider
- **Motivation**: It is important for the system or user to be aware of the robot's state and ensure that all parts of the robot are running correctly (in addition to the energy level of the robot, information such as frequency, individual hardware component state is important to provide). This tactic is employed when other information (other than the robot's energy-level) regarding the robot's state is important to provide.

- **Description**: This tactic includes the energy level state of a hardware component within a generic diagnostics message. Figure 4.9 provides the component interface and sequence diagram of the tactic. The *Diagnostics Provider* is a generic component which monitors the state of a *Hardware Component* via some monitoring mechanism (labels 1,2) and provides the *Hardware Component* diagnostics information (e.g., connection, power, temperature, warnings) in the *Diagnostics Interface*. The status of the *Hardware Component's* energy level is also included in the *Diagnostics Interface* which is requested by the *Energy Level State Requestor* for further analysis (labels 3,4).

- **ROS Example**: Data Point #16: Figure 4.9 is an example of the *Energy-Level Info Within Diagnostics Message tactic* and how it is employed in a humanoid robot via a ROS-based system. In this tactic there are two ROS nodes - the *battery_warning node* (maps to the *Diagnostics Provider*) and the *battery_visualization node* (maps to the *Energy Level State Requestor*). There is also a *battery component* which represents a physical battery. The *battery_warning node* monitors the *battery component*, which provides its diagnostics information. The *battery_visualization requestor node* is subscribed to the *diagnostics_agg* topic which is published by the *battery_warning node* for the battery_level state information, and publishes the received message in a *battery_state topic* which can then be subscribed to by other ROS nodes in the system for visualizing the information.

- **Other Examples**: (i) Diagnostics in mobile phones - built in diagnostics tools are used for running tests to check things like the device's touch screen, audio, video, camera, microphone, and other components of the phone - the diagnostics information of the mobile device (e.g, frequency, battery state, sensor state) is used to promote energy-awareness. (ii) On-board diagnostics (OBD) - self-diagnostic and reporting capability of a vehicle. Provides access to the diagnostics information of different vehicle subsystems (e.g., airbag system, radio system, sensors) to the user (27).

- **Constraints**: The tactic as described assumes that the communication between the *Hardware Component* and the *Diagnostics Provider* is already implemented.

- **Dependencies**: -

- **Variations**: (i) Event-based logic can be used for monitoring the Hardware Component: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime.

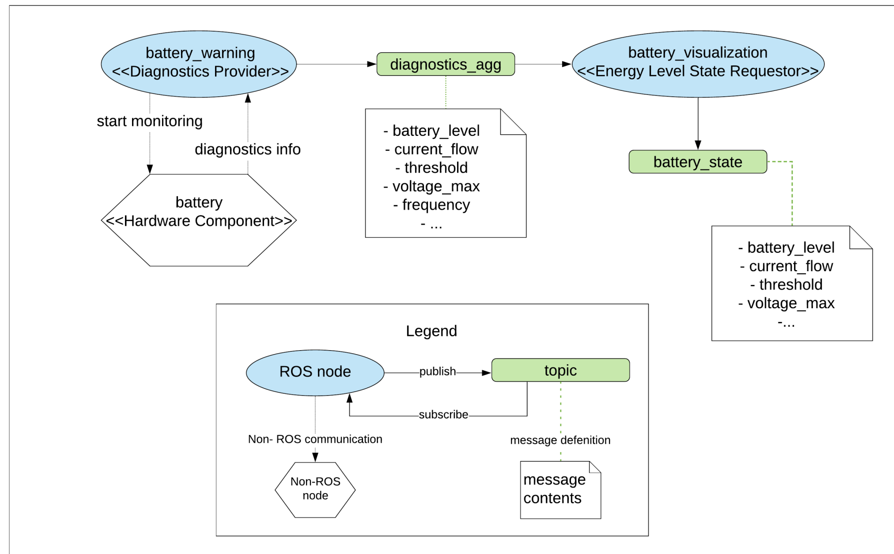


Figure 4.8: EA4 ROS Example

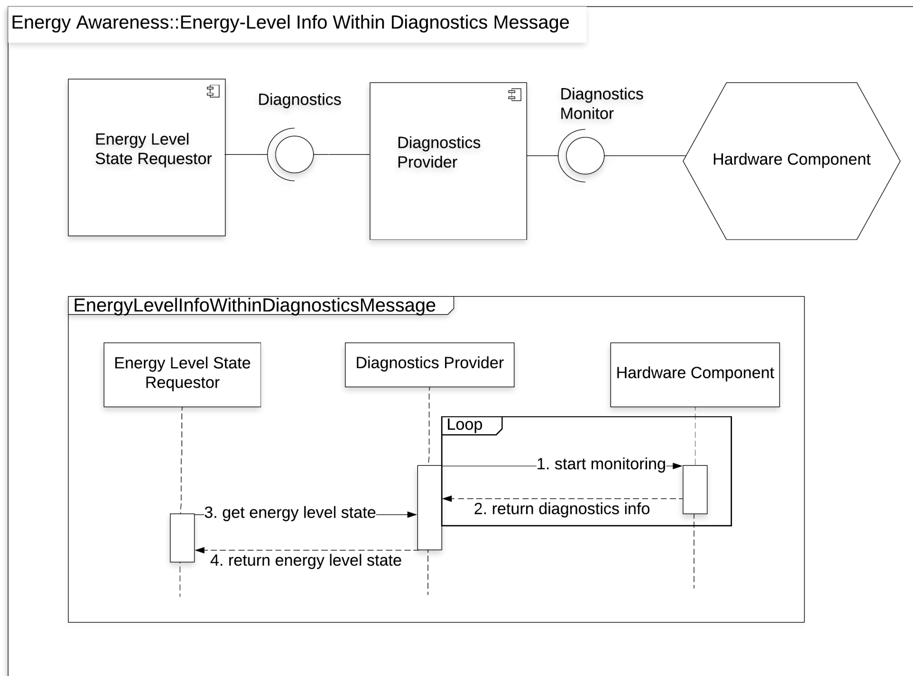


Figure 4.9: EA4 Component Interface & Sequence Diagram

4.2.0.5 E5: Aggregated Energy Information

- **Tactic Name**: Aggregated Energy Information
- **Targeted QA**: Energy Awareness
- **Family**: Energy-Level Provider
- **Motivation**: It is important for the robot to be aware of the energy state of its different components. Gathering the energy state of different components and aggregating it makes it easier for the robot or user to be aware of the energy state of the different components, identify patterns and trends that were not visible before for data analysis, and provide a single access point for other software components in the system to access the components' energy state data.
- **Description**: This tactic aggregates energy-level information (e.g., average charge, battery capacity, voltage) of different components (e.g., batteries) into a single interface. Figure 4.11 provides a component interface and sequence diagram for the tactic. The *Energy Level Providers* represent different components of a robot which provide their energy state. It is not limited to only two *Energy Level Providers*; there an be more *Energy Level Providers* in the system. First, an initial request to the *Energy Level Aggregator* is made by another component in the system to get the aggregated energy information (label 1). Then, the *Energy Level Aggregator* gets the energy state of the different *Energy Level Providers* in two parallel separate threads (labels 2-5) and aggregates the data (label 6). The aggregated data is then returned to the requestor (label 7).
- **ROS Example**: Data Point #40: Figure 4.11 illustrates an example of the Aggregated Energy Information tactic and how it is used in ROS drivers for controlling Segway-based robots in a ROS-based system. In this tactic there are three ROS nodes - the *arduino_ driver node* (maps to the *Energy Level Aggregator*) and the *battery_ diagnostics nodes* (maps to the *Energy Level Providers*, there can be more than two *battery_ diagnostics nodes* in the system). The *battery_ diagnostics nodes* (e.g., batteries) publish energy-related information in the separate *battery_ state topics*. The *arduino_ driver node* is subscribed to these topics and aggregates the energy-related messages into a single message; it then publishes the message to a *diagnostics topic*. This topic can be later subscribed to by other nodes in the system.

- **Other Examples**: (i) Microgrid community: peer-to-peer energy-sharing method; aggregated control of many small-scale batteries to manage energy requirements of the entire community (28).

- **Constraints**: It is assumed that the *Energy State Providers* have an established connection with the physical components in the system.

- **Dependencies**: -

- **Variations**: (i) Event-based logic can be used for getting the energy-level information: for example, a global value (e.g., ROS topic) can be accessed at any execution point at runtime.

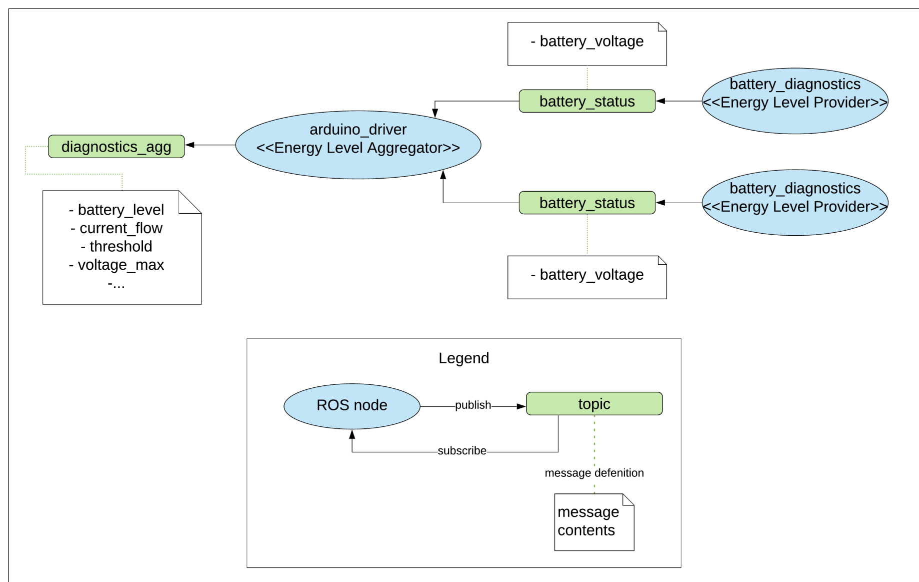


Figure 4.10: EA5 ROS Example

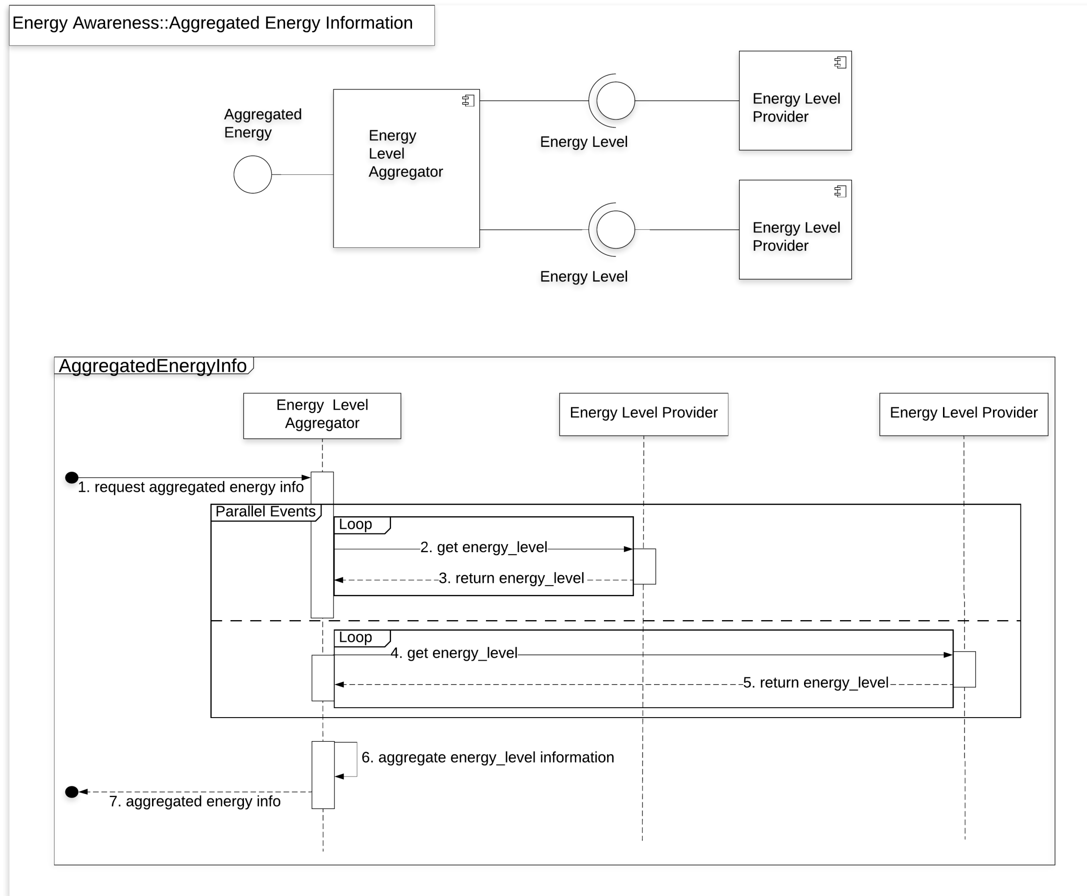


Figure 4.11: EA5 Component Interface & Sequence Diagram

4.2.0.6 EA6: Energy-Savings Mode

This tactic is implemented by tactics EE1-EE4 which introduce different methods on how to implement the energy-savings mode.

- **Tactic Name**: Energy-Savings Mode
- **Targeted QA**: Energy Awareness
- **Family**: Energy-Level Provider
- **Motivation**: To ensure that all components are energy-aware, know when they need to start saving energy, and adjust their behaviour accordingly, a shared space for storing the information about the robot's state (e.g., blackboard) is needed.

- **Description**: This tactic dictates to the components in a system whether or not to enter into a state in which energy must be saved (enable or disable the energy-savings mode). Figure 4.13 provides a component interface and sequence diagram for the tactic. The *Energy Savings Mode Controller* acts as a blackboard (decentralized) that shares the current energy-savings mode state (on/off) with the rest of the components in the system. The blackboard requests the current energy-savings mode state from another component in the system, and based on the response, it dictates to the rest of the components to either disable or enable the energy-savings mode and change their state accordingly. Every component which is represented as the *Observer* receives the current energy-savings mode state, switches to it, and updates the blackboard by sending an acknowledgement message. With this approach, all of the *Observers* are aware of the current energy-savings mode state of the system, and the blackboard is aware of whether or not each component switched to the instructed energy-savings mode state.

- **ROS Example**: Data Point #15: Figure 4.13 illustrates an example of the *Energy-Savings Mode tactic* and how it is employed in an ROV via a ROS-based system. There are two nodes involved in this tactic - the *stepper node* (maps to the *Observer*) and the *manipulator node* (maps to the *Energy Savings Mode Controller*). The *stepper node* represents nodes in the system which must adhere to a certain behaviour depending on the current manipulator command (e.g.c current energy-savings mode). The *manipulator node* publishes the current command to a *manipulator_ command topic* which is subscribed to by the *stepper node*. After receiving the current manipulator command and adhering to it, the *stepper node* publishes its new state to the *stepper_ state topic*, which is in turn subscribed to by the manipulator node. With this kind of structure, the *manipulator node* can keep track of whether or not all nodes in the system behave according to the current issued command.

- **Other Examples**: (i) Mobile computing - e.g., power-saving mode (PSM) for mobile computing in WiFi-hotspots - IoT - e.g., solar energy for IoT devices that are exposed to sunlight. (ii) Appliances such as ovens, microwaves - e.g., turn off the display when in energy-savings mode. (iii) Hybrid vehicles - increase fuel efficiency by adding a battery-powered electric motor. When the hybrid auto is in “idle”mode (e.g., stops at a stoplight), the engine is turned off, and the battery-powered electric motor still provides accessories such as audio, air conditioning, etc. (iv) IG-L - regulates CO2 emission by adjusting the digital speed signs on the highways to a lower rate (29).

(v) Industrial emissions within the EU - industrial companies are able to lower their carbon footprint at moments where it will be increased significantly (30).

- **Constraints**: It is assumed that the criteria for enabling or disabling the energy-savings mode is already included in the *Energy Savings Mode Controller*. It is also assumed that the communication between the *Energy Savings Mode Controller* and the rest of the component in the system already exists and that the components know how to behave when the energy-savings mode is on/off.

- **Dependencies**: -

- **Variations**: (i) Pull-based approach: based on the energy-level, the *blackboard* maintains the state of the energy-savings mode, and the components in the system request the mode.

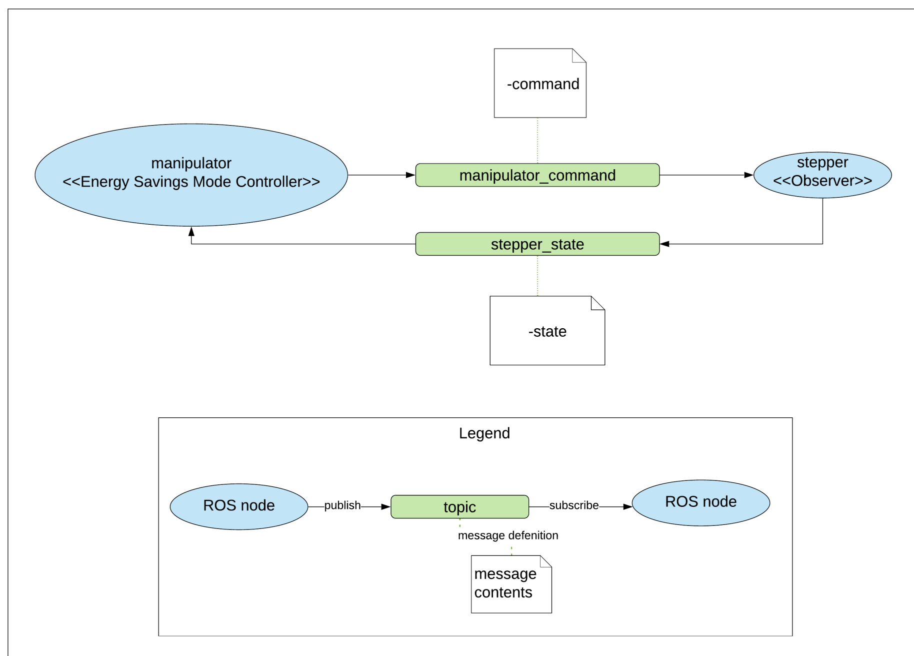


Figure 4.12: EA6 ROS Example

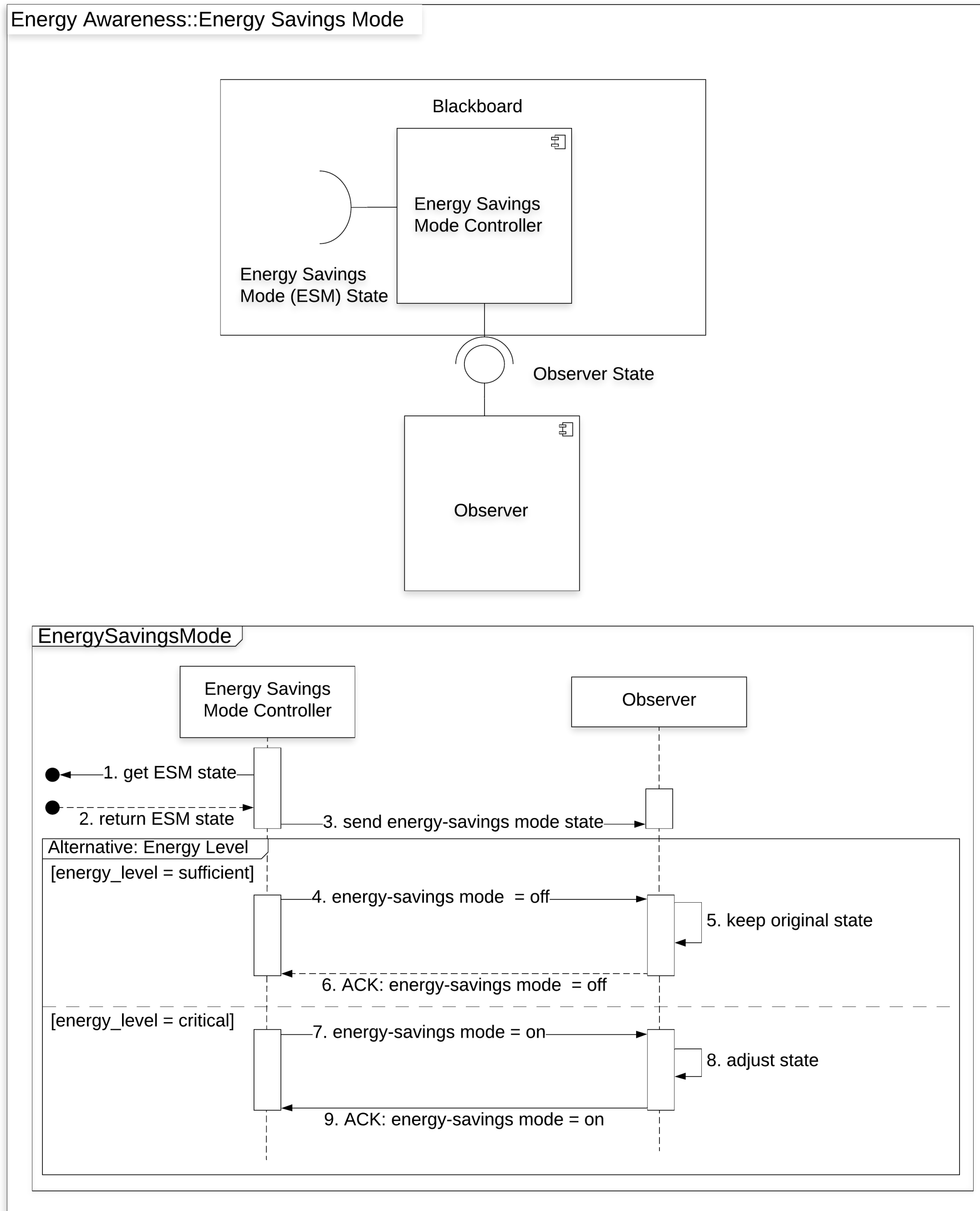


Figure 4.13: EA6 Component Interface & Sequence Diagram

4.2.0.7 EA7: Offline Energy Profiler

- **Tactic Name**: Offline Energy Profiler
- **Targeted QA**: Energy Awareness
- **Family**: Other
- **Motivation**: Offline profiling provides multiple benefits for robotics developers and researchers such as recording profiled datasets, visualizing and labeling them, and storing them for future use. Besides the previously listed benefits of offline profiling, offline energy profiling also promotes energy-awareness by providing energy diagnostics information from the previous executions to the current execution. For example, this information can be used for estimating the remaining operating time of a battery. For robotics developers and researchers, the recorded energy datasets can be used to analyze trends, patterns, and provide insights on which robotic activities consume the most energy.
- **Description**: This tactic builds an energy profiler from previous executions which is then used in a current execution for providing energy level state diagnostics. Figure 4.15 provides a component interface and sequence diagram for the tactic. The *Energy Profiler* builds an energy profiler by extracting logged energy-related data (labels 1,2) (e.g., energy-related messages logged in ROS bag files) and providing its estimations to other components in the system. The *Energy Level State Requestor* requests energy-level diagnostics information from the *Diagnostics Provider* (label 3). After receiving the request, the *Diagnostics Provider* opens an existing energy-profiler and gets the requested energy-level diagnostics information (label 4). This information is then sent back to the *Energy Level State Requestor* (labels 5,6).
- **ROS Example**: Data Point #13: Figure 4.15 illustrates an example of the *Offline Energy Profiler tactic* and how it is employed in ROS drivers for controlling Segway-based robots in a ROS-based system. There are two nodes involved in this tactic: a *battery_profiler node* (represents the *Diagnostics Provider* and *Energy Level State Requestor*), and a *battery_diagnostic node* (represents the *Energy Profiler*). The *battery_profiler node* is in charge of building a battery-profiler based on logged battery data from previous executions (e.g., bag files). The *battery_diagnostics node* requests battery diagnostics information (e.g., battery life estimation) by opening

an existing battery profiler. It then performs some computation and dispatches the result to a local SMTP client.

- **Other Examples**: (i) Silicon Labs - multi-node energy profiler: several nodes monitor different devices, build energy profilers and use them in offline mode for visualizations, computations (e.g., battery estimation), etc (31). (ii) ALEA: A Fine-Grained Energy Profiling Tool: fine-grained energy profiling tool based on probabilistic analysis for fine-grained energy accounting. An online module transfers the results of profiling to an offline module that derives energy estimates (32).

- **Constraints**: This tactic as described assumes that the energy profiler exists.

- **Dependencies**:

- **Variations**: In the case when the energy profiler does not exist, error checking is applied when opening the energy profiler.

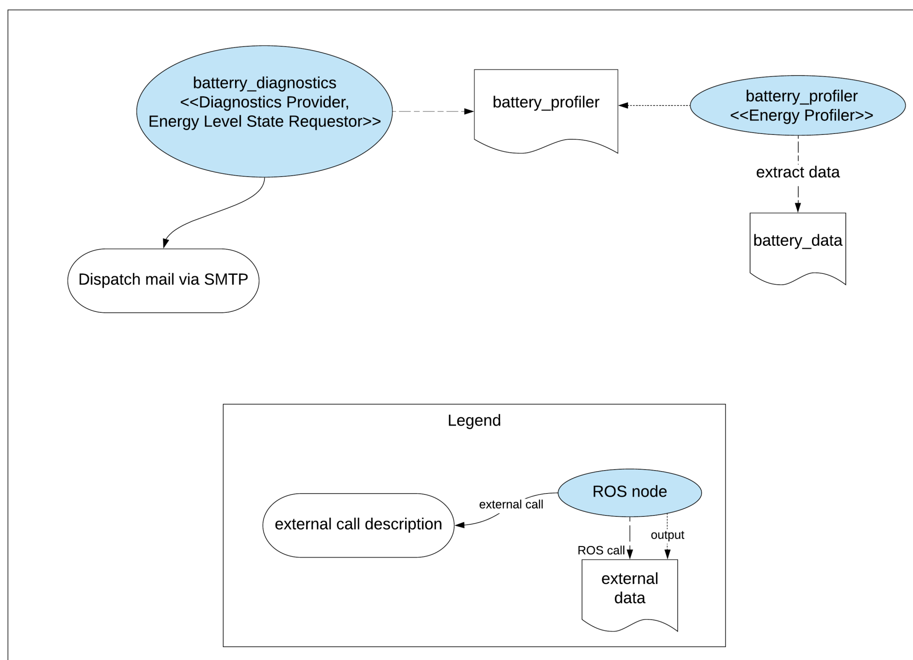


Figure 4.14: EA7 ROS Example

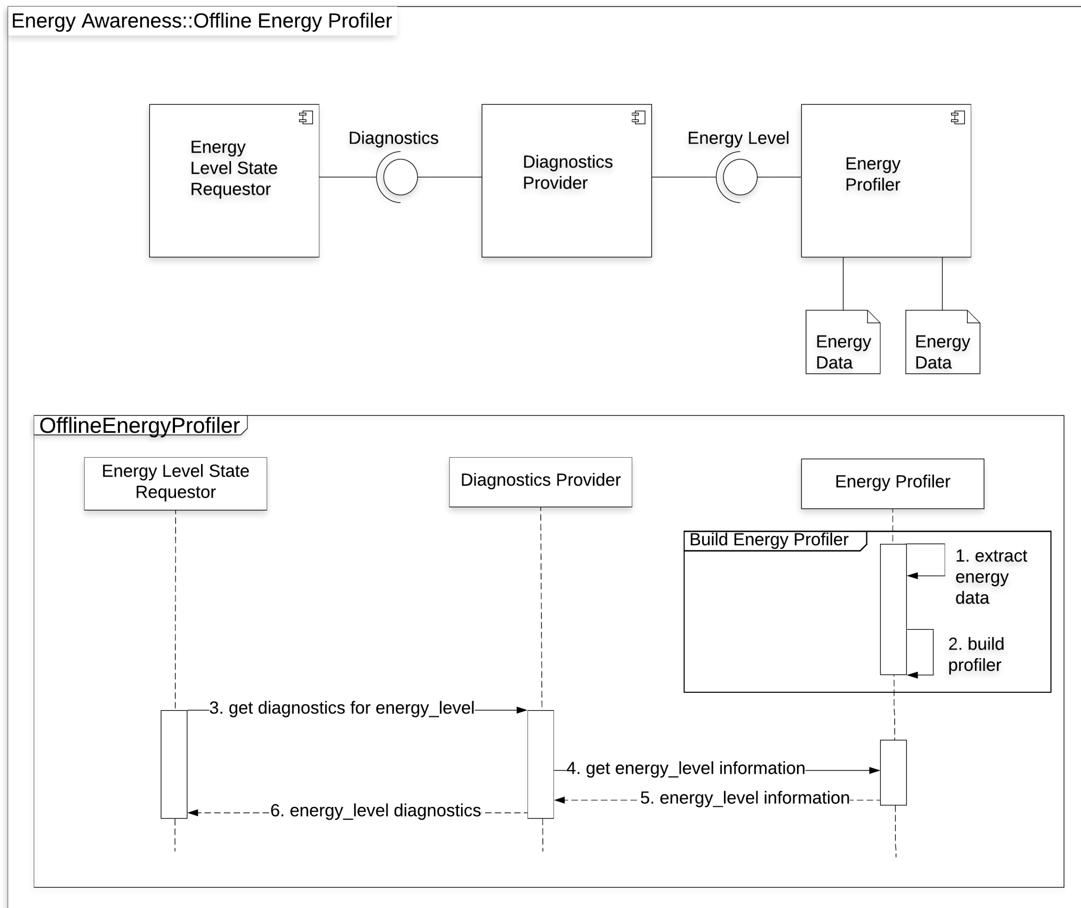


Figure 4.15: EA7 Component Interface & Sequence Diagram

As of now, batteries (e.g., Lithium Ion, Lithium Polymer) are heavy to carry and have a limited energy budget; therefore, preserving the robot's battery life is a critical task in robotics software. Tactics EE1-EE4 are different ways of how components in the system behave when the energy-savings mode is enabled. The following tactics implement tactic EA6 (the behaviour of the components under the energy-savings mode).

4.2.0.8 EE1: Limit Task

- **Tactic Name**:
- **Targeted QA**: Energy Efficiency
- **Family**: Energy Efficiency

- **Motivation**: Robotic activities such as data sampling or large amounts of data transfer consume a significant amount of energy. When the robot enters the energy-savings mode, limiting different activities (see Description for list of possible robotic activities) is pivotal in order to meet the energy-savings mode requirements.

- **Description**: This tactic configures a robotic task based on any set criteria for tasks under the energy-savings mode. Figure 4.17 provides a component interface and sequence diagram for the tactic. This tactic configures a robotic task based on any set criteria for tasks under the energy-savings mode. The *Task Requestor* is responsible for requesting to execute a certain task, the *Arbiter* is responsible for deciding whether or not to configure the original task, the *Energy-Savings Mode Manager* is responsible for providing the criteria for any task under the energy-savings mode (energy-level is critical), and the *Task Executor* is responsible for executing the original or configured task. First, the *Task Requestor* sends an initial task to the *Arbiter* (label 2). After receiving the task, the *Arbiter* checks the energy-level of the robot (labels 3,4) provided by another component in the system. If the energy-level is critical, the *Arbiter* immediately removes the task (label 25), thus, not forwarding it to the *Task Executor*. In the case when the energy level is sufficient, in a separate thread the *Arbiter* forwards the task to the *Task Executor* (label 5) and the *Task Executor* starts executing the task (label 6). In a parallel thread within a loop, the *Arbiter* also starts checking the energy-level (labels 7,8). If during the execution of the original task the energy-level becomes critical, the *Arbiter* breaks out of the loop. A new loop is started, and in a separate thread, the *Arbiter* starts checking the energy-level of the robot (labels 9,10). If the energy-level is critical, in a parallel thread, the *Arbiter* gets the energy-savings mode criteria for the task provided by the *Energy-Savings Mode Manager* (labels 11, 12), configures the task (label 13), and instructs the *Task Executor* to start executing the configured task instead of the original task (label 14). If the energy-level becomes sufficient during the execution of the configured task (label 17), the *Arbiter* re-configures the configured task back to the original one (label 18) and instructs the *Task Executor* to start executing the original task instead of the configured one (label 19). If the energy-level drops again to a critical point, the previously described steps will take place (labels 11-15). Some examples of task configurations: (i) Initial task: move in any direction at a set max power rate. Configured task: adjust power rate to a lower rate specified in the set criteria. (ii) Initial task: publish large amounts of data (e.g., publish PCL point

clouds, 3D map). Configured task: stop publishing data. (iii) Initial task: sample data at a set configurable rate. Configured task: alter the sampling rate to a lower rate specified in the set criteria.

- **ROS Example**: Data point #36: Figure 4.17 is an example of how the *Limit Task tactic* can be employed in a ROS-based system with haptic devices. Haptic teleoperation allows a user to perform manipulation tasks in distant, scaled, hazardous, or inaccessible environments. *Haptic teleoperation provides telepresence by allowing a user to remotely control a secondary robot through a main device while haptically perceiving the remote environment* (33). In this example, the *Main* represents a ROS-based system which consists of a *haptic_ device_ controller node* (maps to the *Task Requestor*) which communicates directly with the physical haptic device. The *Secondary* ROS-based system consists of an *arm_ controller_ node* (represents the *Arbiter*, *Task Executor* and the *Energy-Savings Mode Manager*) which communicates directly with the *robot arm*. The *arm_ controller_ node* is subscribed to a *battery_ state topic* which is published by some other node. This topic publishes information regarding the robot arm battery such as the battery percentage. The *Secondary* communicates the data relevant to the *robot arm* to the *Main* and vice versa via the network. In the *Main*, the *haptic_ device_ controller node* is subscribed to an *arm_feedback topic* which publishes messages regarding the state of the *robot arm* such as the battery percentage. This information is directly communicated by the *haptic_ device_ controller node* to the user monitor which is used by an actual user controlling the haptic device. If the *arm_ controller_ node* (*Secondary*) receives some task from the *Main*, and the *robot arm* battery level is critical, the *arm_ controller_ node* adjusts the task by not providing feedback from the *robot_ arm* to the *Main*. Simultaneously, the *haptic_ device_ controller node* in the *Main* is subscribed to the arm_feedback topic and communicates the battery feedback to the user monitor. In this way, the user is aware of the robot arm's battery state.

- **Other Examples**: (i) Cloud integrated sensor network: when energy needs to be saved, data-transmission techniques are replaced with energy-efficient data-transmission techniques such as a customizable sensor information system model which used to modify data transmission and frequency of data collection to make it energy efficient; this approach also reduces CO2 emissions (34). (ii) Plug-in hybrid electric vehicles: disable certain drive modes for a better power management.

- **Constraints**: The tactic as described assumes that the communication between the component providing the energy level information and the *Arbiter* already exists, and that the criteria for any task under the energy-savings mode is already set.

- **Dependencies**: This tactic may involve tactics - EA1 for configuring a task to abort when the energy-savings mode is on - EA2 for configuring a task to stop and recharge the battery when the energy-savings mode is on - EA3-EA6 to get the energy level state.

- **Variations**: Event-based logic can be used for getting the energy-level information: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime by the *Arbiter*.

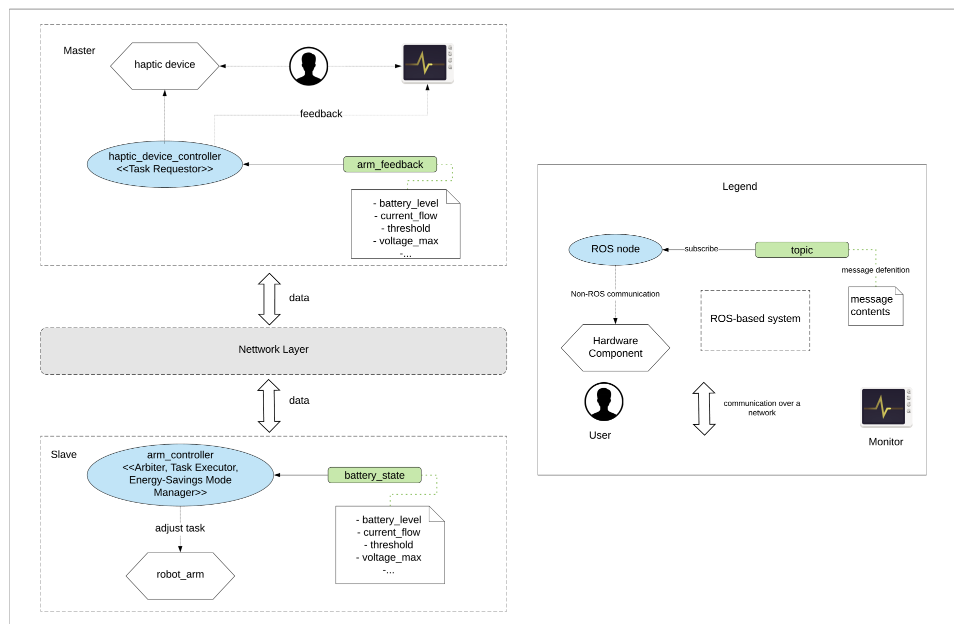


Figure 4.16: EE1 ROS Example

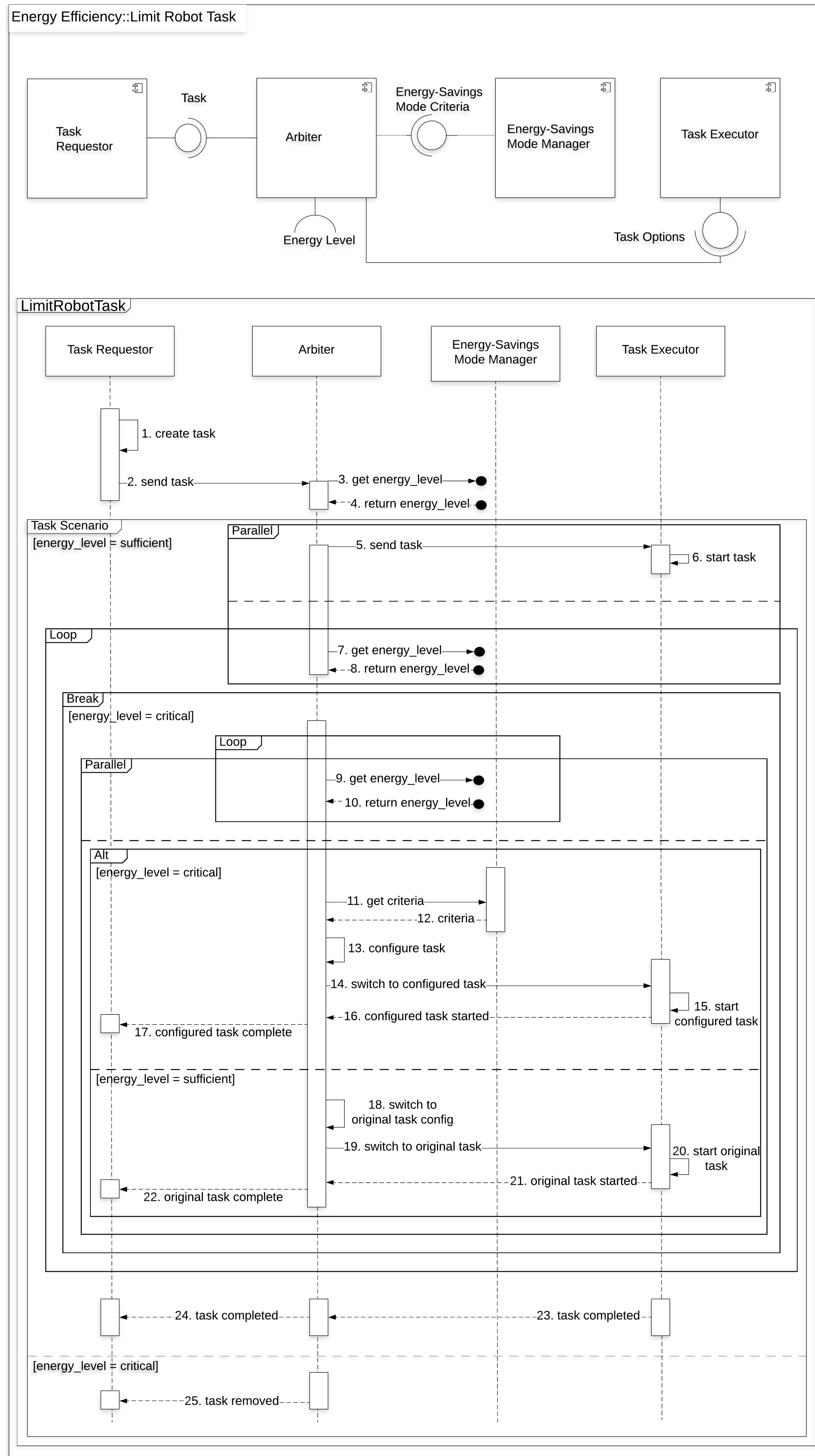


Figure 4.17: EE1 Component Interface & Sequence Diagram

4.2.0.9 EE2: Disable Hardware

- **Tactic Name**: Disable Hardware
- **Targeted QA**: Energy Efficiency
- **Family**: Energy Efficiency
- **Motivation**: Hardware components (e.g., sensors, drivetrains) of a robot often consume a significant amount of energy; for example, sensors consume energy in terms of CPU usage (i.e., sensors provide feedback based on the robot's surroundings and send it to the CPU which then uses the feedback to make further decisions). It is crucial to prevent unnecessary utilization of hardware resources in order to extend the maximum operational time of the robot's battery.
- **Description**: This tactic disables hardware components when they are not strictly needed, which results in the robot consuming less energy and in a more efficient power management during its tasks. Figure 4.19 illustrates the component interface and sequence diagrams of the tactic. The tactic is implemented to control the physical communication between the *HW Controller* and the actual hardware device. The *HW Requestor* notifies the *HW State Controller* whether or not the hardware device is needed for a certain task (labels 1, 8). Then, the *HW State Controller* instructs the *HW Controller* to disable or enable the hardware device (labels 2, 9). Before enabling or disabling the hardware device, the *HW Controller* checks the task to determine if it is safe to change the state of the hardware device (labels 3, 10) (e.g., toggle hardware pin, set boolean variable for instruction to TRUE/FALSE). In this tactic, we assume that it is always safe to either enable or disable the HW device, so the *HW Controller* enables or disables the HW device (labels 4, 11), based on the input from the *HW State Controller*.
- **ROS Example**: Data Point #23: Figure 4.19 illustrates an example of the *Disable Hardware tactic* and how it is employed in *ros_control* package which includes ROS-based controller managers and controller and hardware interfaces. There are two nodes: a *controller_requestor node* (maps to the *HW Requestor*) and a *controller_manager node* (maps to the *HW State Controller* and *HW Controller*). There is also one non-ROS component - the *robot_hw* (maps to the *Device*). The *controller_manager node* advertises two services: a *load_controller service* (enable HW) and an *unload_controller service* (disable HW). If the *controller_requestor node*

wishes to enable the *robot_hw*, it sends a request to the *load_ controller service* to enable the robot_hw. If it wishes to disable the *robot_hw*, it sends a request to the *unload_ controller service* to disable the *robot_ hw.* After receiving a request, the *controller_ manager node* performs the request by either enabling or disabling the *robot_ hw.* The *controller_ requestor node* is notified with the result; this ensures that the *controller_ requestor node* is aware of the *robot_ hw* status.

- **Other Examples**: (i) Windows power management: when the computer enters a sleeping state, Windows puts the network interface card to sleep as well. (ii) Mobile activity recognition system (ARS) for detecting human activities: disable GPS hardware while the user is indoors (35). (iii) LG air conditioners: power consumption is reduced by automatically turning off the circulation fan as well as the exhaust fan when the compressor is off.

- **Constraints**: The tactic as described assumes that the communication between the actual hardware device and the *HW Controller* already exists and that it is safe to change the state of a hardware device. It is also assumed that the HW device has the capability of being turned off and that other components in the system already notified the *HW State Controller* regarding the demand for a hardware device.

- **Dependencies**: -

- **Variations**: (i) In the case when switching the state of the hardware device is not safe, the *HW Controller* will check the task and suspend it. (ii) The *HW State Controller* can check whether or not the actual hardware device is being currently utilized.

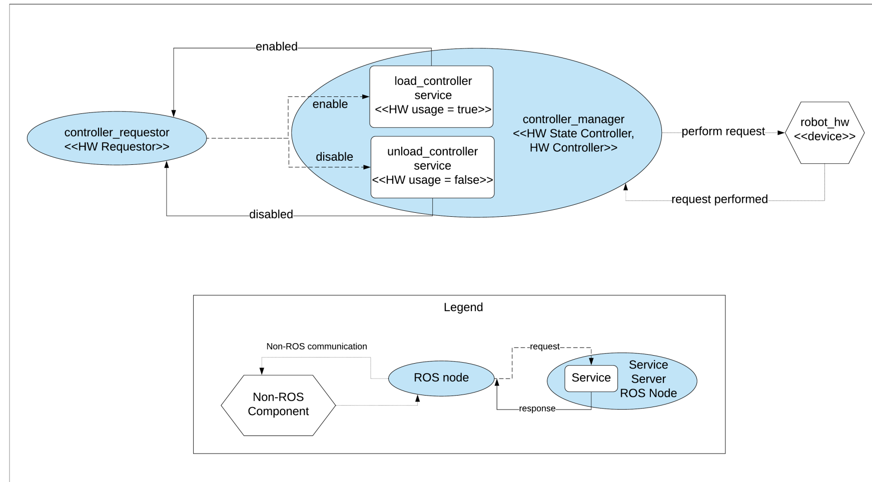


Figure 4.18: EE2 ROS Example

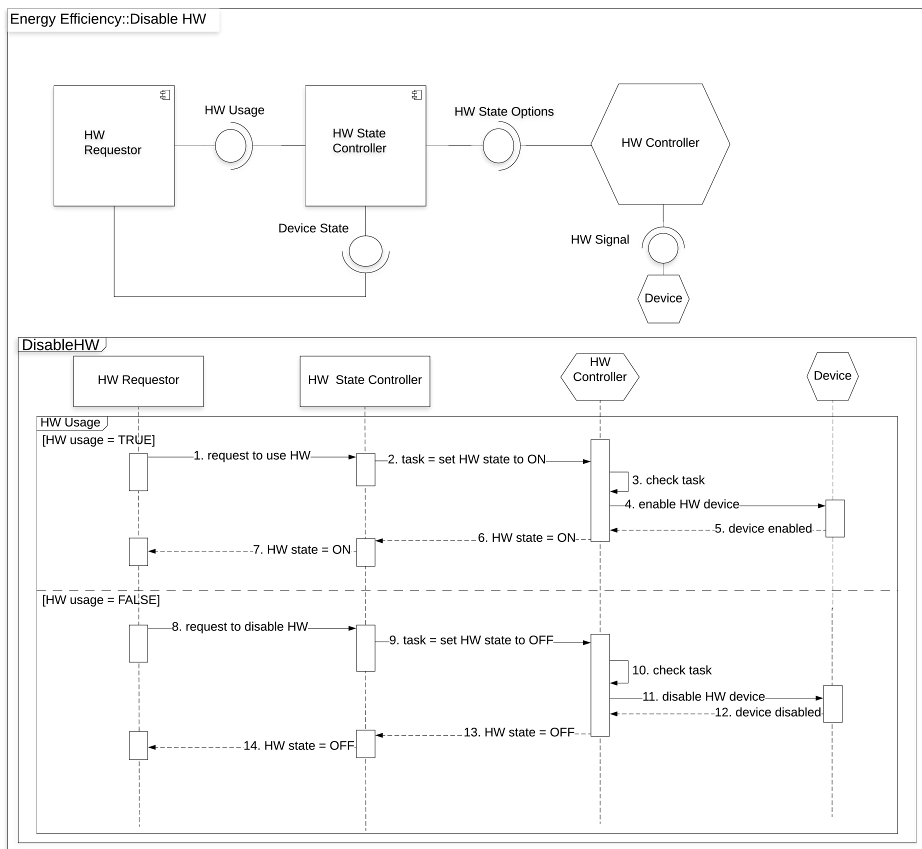


Figure 4.19: EE2 Component Interface & Sequence Diagram

4.2.0.10 EE3: Energy-Aware Sampling

- **Tactic Name**: Energy-Aware Sampling
- **Targeted QA**: Energy Efficiency
- **Family**: Energy Efficiency
- **Motivation**: In robotics, many sensors are designed to provide a continuous stream of data (e.g., accelerometers, LiDARs) (35). Sampling data from sensors is an energy consuming task as the transmission of data which occurs at regular intervals, leads to a larger energy waste (36). When the robot's battery reaches a critical point, the component in charge of sampling the sensor should be able to continue sampling, and at the same time, enter into a state in which energy must be saved to avoid further drain of the battery.
- **Description**: This tactic adjusts the rates for sampling a sensor based on the energy-level of the robot. Figure 4.21 illustrates the component interface and sequence diagrams of the tactic. The *Sensor Controller* provides configurable sampling rates for the sensor. First, the *Sensor Requestor* requests to start sampling the sensor and sends the request to the *Sampling Rate Controller* (label 1). After receiving the request, the *Sampling Rate Controller* gets the initial sampling rate from the *Sensor Controller* (labels 2,3). Then, in an infinite loop and in a separate thread the *Sampling Rate Controller* gets the energy-level which is provided by another component in the system (labels 4,5); if the energy-level is critical (e.g., the battery level is below a set threshold), the initial sampling rate is lowered (within the same loop and in a parallel thread) (label 11). The adjusted sampling rate is then sent back to the *Sensor Controller* (label 12) which uses the adjusted sampling rate to sample the sensor (label 13). The sampled sensor state information is then provided by the *Sensor Controller* which communicates this information back to the *Sensor Requestor* (labels 14, 15). Under normal operation (e.g, energy-level is sufficient to operate, battery level above a set threshold), the initial sampling rate is not altered (label 6-10).
- **ROS Example**: Data Point #51: Figure 4.21 illustrates an example of the *Energy-Aware Sampling tactic* and how it is employed in a ROS-based driver for InvenSense's 3-axis gyroscope. There are two nodes and one non-ROS component involved:

the *inv_mpu_controller node* (maps to *Sensor Requestor* and *Sampling Rate Controller*), the *inv_mpu node* (maps to *Sensor Controller*), and the *sensor component.* The *inv_mpu_controller node* is subscribed to a *battery_state topic* to check the battery level. It is also subscribed to a *sampling_rates topic* in which the sampling rates are published by the *inv_mpu node.* After getting a sampling rate, the *inv_mpu_controller node* checks the battery level and either adjusts or keeps the original sampling rate based on the battery level. If the battery level is sufficient, the *inv_mpu_controller node* keeps the original sampling rate and sends a request to the *sampling action* advertised by the *inv_mpu node* to start sampling the sensor. If the battery_level is critical, the *inv_mpu_controller node* first adjusts the sampling rate, and then makes a request to the *sampling action* to start sampling the sensor. The *inv_mpu node* samples the sensor via non-ROS communication methods and publishes the sensor status to a *sensor_status topic* which is subscribed to by the *in_mpu_controller node.*

- **Other Examples**: (i) Sensor network for automated water quality monitoring: if the monitored parameters hardly fluctuate, lower sampling frequency is used; less energy will be needed for data sampling, data processing and transmission (37). (ii) Mobile activity recognition system (ARS) for detecting human activities: energy-efficient ARS - low sampling rates, can achieve high recognition accuracy and low energy consumption (38).

- **Constraints**: This tactic assumes that the communication between the physical sensor and the *Sensor Controller* is already established and that the initial sampling rates are already configured. It also assumed that the sensor has configurable rates and that the communication between the component providing the energy level information and the *Sampling Rate Controller* already exists.

- **Dependencies**: This tactic can employ tactics EA3 - EA6 to get the state of the energy level.

- **Variations**: (i) Event-based logic can be used for checking the energy-level: for example, a global value (e.g., ROS topic) can be accessed at any execution point during runtime.

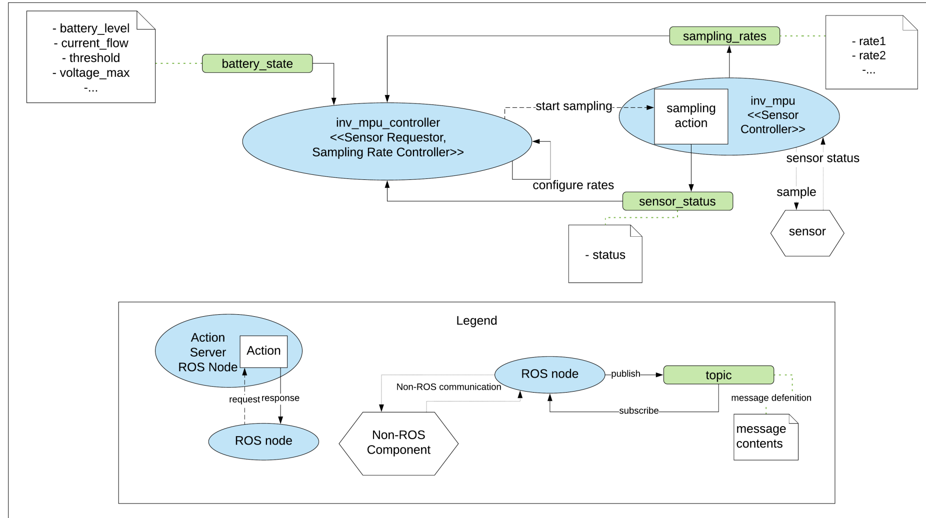


Figure 4.20: EE3 ROS Example

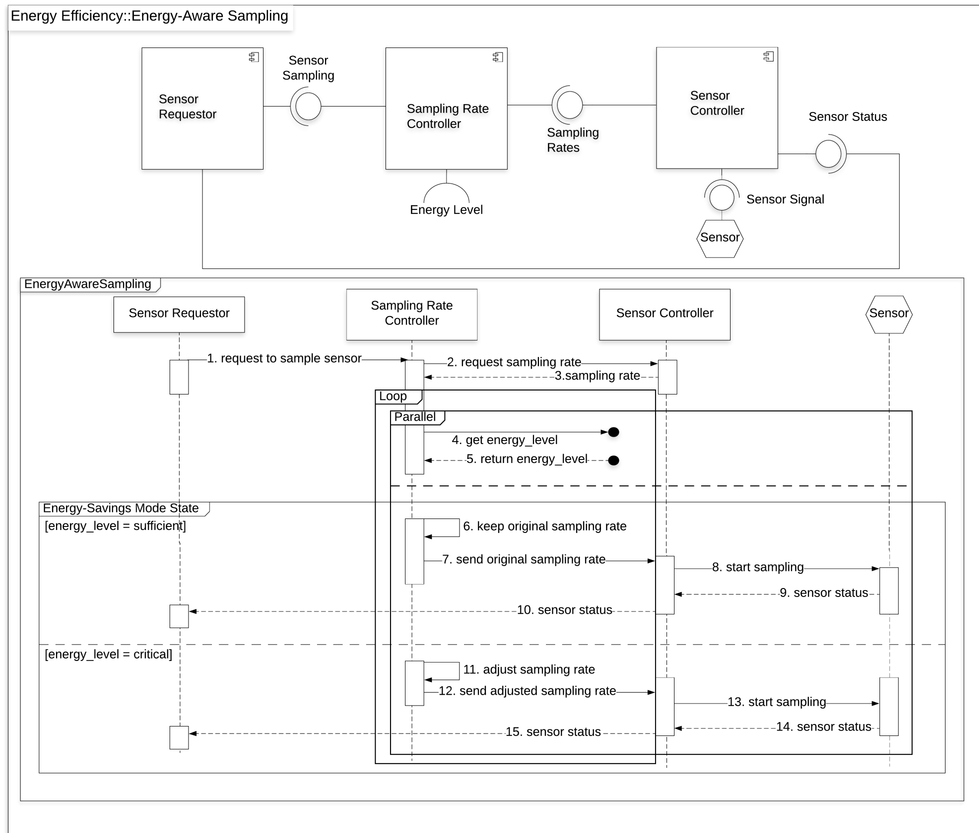


Figure 4.21: EE3 Component Interface & Sequence Diagram

4.2.0.11 EE4: On-Demand Software Components

- **Tactic Name**: On-Demand Software Components
- **Targeted QA**: Energy Efficiency
- **Family**: Energy Efficiency
- **Motivation**: Continuously running a component requires the spawning of an OS process which is an energy-consuming task in terms of CPU usage (i.e., executing a CPU-intensive loop) and other resources (e.g., sensors, motors, fans for cooling). For this reason, it is necessary to ensure that OS processes that are not being utilized do not consume energy unnecessarily by running in the system.
- **Description**: This tactic brings up new components only when their functionalities are needed. Figure 4.23 illustrates the component interface and sequence diagrams of the tactic. The *Component Manager* acts as a controller which either starts up or shuts down a component based on its demand. The *Requestor* represents a component which requires the *On-Demand Component* based on its status: online (startup a component) or offline (shutdown a component). First, the *Requestor* sends its status to the *Component Manager* (labels 1,6) which either brings up or shuts down the *On-Demand Component* (labels 2,7). The status of the *On-Demand Component* is communicated from the *Component Manager* to the *Requestor* (labels 3,4 & 8,9) so that it is aware whether or not the *On-Demand Component* is up and running; if it is up and running, the *Requestor* can start communicating with the *On-Demand Component* (label 5).
- **ROS Example**: Data Point #14: Figure 4.23 is an example of how the On-Demand ROS Component tactic can be employed in a practical scenario when dealing with cameras. In this scenario, ROS nodelets are used instead of ROS nodes. The reason behind this is that when ROS nodes are dealing with messages that contain large amounts of data (i.e. camera images, point clouds), sending and unpacking the messages takes a longer time over TCP. With nodelets, messages are communicated within the same process via a booster shared pointer. This helps to reduce overhead of data transfer (39). In this example, there is a *nodelet_manager* (maps to *Component Manager*) which has a pool of threads shared among the nodelets spawned within the same *nodelet_manager*. The *camera nodelet* (maps to *Requestor*) is a nodelet

that is one of the threads running within the *nodelet_manager*. In order for the *camera nodelet* to operate, it requires the *camera_driver nodelet* (maps to *On-Demand Component*) to be up and running. The *camera nodelet* publishes its status (online, offline) to a *camera_status topic* which is subscribed to by the *nodelet_manager*. Based on the *camera nodelet's* status, the *nodelet_manager* either starts up or shuts down the *camera_driver nodelet*. The *camera_driver nodelet's* status is communicated back to the *camera nodelet*. If the *camera_driver nodelet* is up and running, it advertises a *Service* which can be used by the *camera nodelet*.

- **Other Examples**: (i) Cloud computing: delivery of on-demand computing resources, ability to scale computing resources. (ii) iOS: on-demand resources (ODR) API to reduce original download size of the application. Instead, the ODR API is used to deliver the application contents after it has been downloaded. This is especially useful for gaming applications (40).

- **Constraints**: It is assumed that *On-Demand Component* is already implemented and is waiting to be brought up or shut down.

- **Dependencies**:

- **Variations**: Instead of having the *Component Manager*, the *Requestor* can spawn the *On-Demand Component* directly.

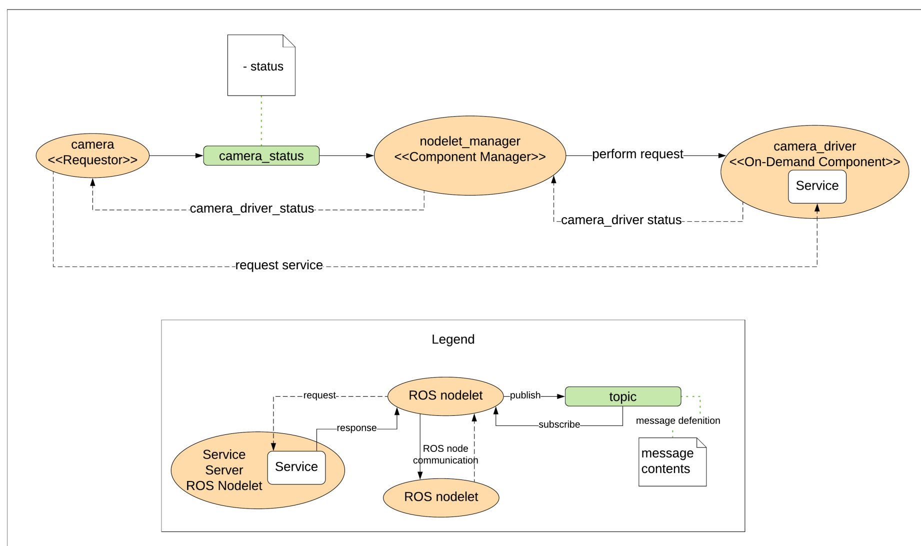


Figure 4.22: EE4 ROS Example

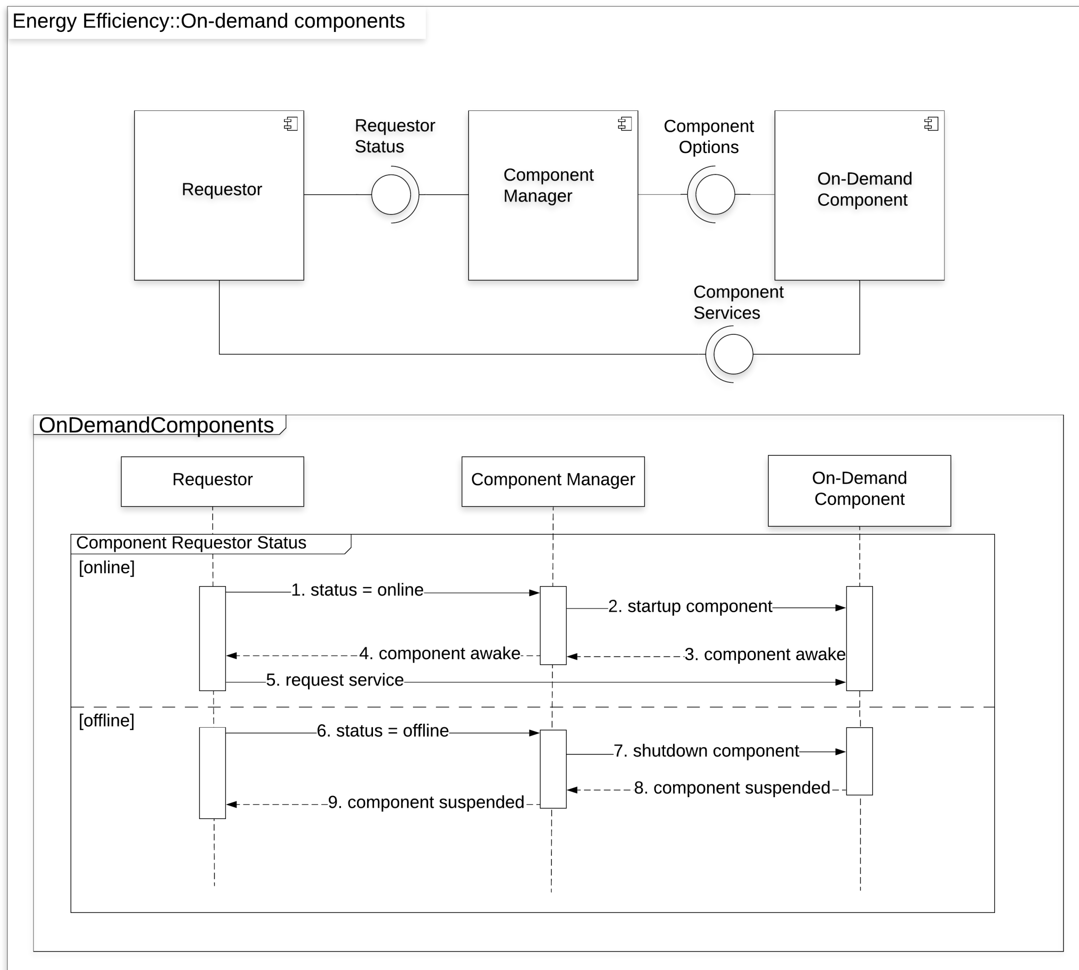


Figure 4.23: EE4 Component Interface & Sequence Diagram

5

Discussion

In this project we mined ROS data-sources for identifying and extracting green tactics for energy-efficient robotics software. We discovered that energy-awareness tactics are prominent amongst the extracted green tactics which might indicate that there are already many methods available for roboticists to design and implement energy-aware robotics software. The four extracted energy-efficiency tactics target both the software and the hardware which also emphasizes the fact that roboticists are concerned with energy-efficiency in both levels of abstraction. This is further supported by the observation that for example, tactic EE1 (software-level) is found in five data points and EE2 (hardware-level) in three data points where each data point is associated with a unique ROS project.

It is also interesting to note that majority of the tactics are associated with battery-operated robots even though we did not intend to only focus on such robots. This might highlight the fact that batteries like Lithium Ion and Lithium Polymer have a limited energy budget and therefore, battery-operated robots are constantly in need of a wise energy-management scheme. Even though it happened that most of the tactics are extracted from data points concerning battery-operated robots, we can firmly say that most of the tactics are applicable in robots that are powered directly from a cable and also in other application domains – we support this claim by providing other examples where these tactics have been already applied.It is essential to consider methods and techniques for the minimization of energy consumption in batteryless robots. For example, the non-battery operated robot arm is one of the most prominent and used robots in various industries such as the automotive and medical industry. The robot arm has several capabilities such as capturing and transporting a product or piece in a production line, perform welding, cutting, milling, assembling, and performing or assisting surgery in medicine (41). Robot arms are often operated dynamically to maximize production outputs, which results in both

high energy loss at high velocities as well as energy excess when decelerating. Furthermore, several tasks are followed by idle times associated with a loss of productivity. (42)

During *Phase 2* where we identified energy-relevant data, we noticed that a significant portion of the data points concerned *safety* in robots, an area outside the scope of this project, which happens to be a prominent QA in robotics software. For example, the robot arm used in medical industries to perform surgeries comes into direct contact with a human, hence, the most important design consideration at premium is safety. Robot safety depends on several factors, ranging from software dependability, to possible mechanical failures, to human error, etc. Therefore, it is also important to consider design options (ATs) for safety during the early stage of software design. This observation is interesting as we suspected energy to be the major concern in robotics software, as robots are autonomous and have restricted energy supplies. Interestingly enough, the `StackOverflow` and `ROS-Discourse` data sources were discarded during *Phase 3*, indicating that `GitHub`, `BitBucket`, `ROS-Answers`, and `ROS-Wiki` are good sources for green tactics. We also observed that some of the energy-related data points identified in *Phase 2* did not concern energy-efficiency but focused on either visualizing energy-related data or focusing on technical specifications of fact sheets of a robot. It would be interesting to use a thematic analysis approach (43) to examine, identify and record patterns (or *themes*) within the dataset in *Phase 2*.

We believe that these green tactics are generally applicable, however, they should be employed within the proper context. For example, it makes sense to apply tactic EE2 in a battery-powered environment rather than a batteryless environment, as tactic EE2 incorporates a *recharging* activity. That being said, tactic EE1 can be applied in both battery-operated and batteryless environments; an example of a batteryless environment is the soft, autonomous robot known as the octobot. The octobot runs on liquid hydrogen peroxide fuel which gives off oxygen gas to controll the ocotobot's eight arms (44). EE1 can be applied in the octobot to monitor the level of the fuel and adjust a task when there is not enough fuel to operate continuously.

6

Threats To Validity

The following chapter reports on the different kinds of threats to validity which have been analyzed for the purpose of evaluating to what extent the results are sound and applicable to the real world. In the following sections, the kinds of threats to validity are discussed in detail.

6.1 External Validity

External validity refers to how well the outcome of a study can be expected to apply to other settings. While rigorous research methods can ensure internal validity, external validity, on the other hand, may be limited by these methods (45). This threat deals with the fact that the data sources and the 335 ROS-based open-source projects hosted on `GitHub` and `Bitbucket` may not be representative of the robotics community. The data sources `StackOverflow`, `ROS-Answers`, `ROS-Wiki` and `ROS-Discourse` are heterogeneous in terms of the age of the posts, and number of questions per distinct user and the 335 ROS repositories in terms of number of contributors, number of commits, etc. This potential threat is avoided as the primary motivation for using ROS is that the ROS community is very active in terms of the number of packages, questions posted in the ROS forums, and open-source ROS projects. These reasons make us confident in the long term future of ROS.

6.2 Internal Validity

Internal validity is the extent to which a study establishes a trustworthy cause-and-effect relationship between a treatment and an outcome. Internal validity depends largely on the

procedures of a study and how rigorously it is performed (45). The study design of this project was identified and rigorously defined prior to the dataset construction, energy-relevant data collection, architecturally-relevant data collection, and the green tactics extraction. The replicability of this project and verification of the results are documented in a publicly available research protocol.

6.3 Construct Validity

Construct validity is about ensuring that the method of measurement matches the construct wanted to be measured (46). To mitigate this threat, a well-defined goal and research question have been identified to cover the scope of this project. Each phase of the study design was carefully designed and carried out; the energy-relevant and architecturally-relevant data was rigorously identified and selected via an established inclusion and exclusion criteria and verified by a second researcher. The green tactics were identified and extracted using a pre-defined protocol and verified by three researchers.

6.4 Conclusion Validity

Conclusion validity is the degree to which conclusions reached about relationships in some data are reasonable (47). To mitigate this threat and reduce potential biases, two researchers were involved in identifying the energy-relevant and architecturally-relevant data by using Cohen's Kappa to measure the level of agreement to ensure that an arbiter was not needed. Furthermore, three researchers were involved in the extraction process of the green tactics and the final results were verified by two other researchers.

7

Conclusion

In this project we mined green tactics for energy-efficient robotics software by using ROS-based systems as the main focus in our dataset as the ROS community is proven to be very active and an accurate representation of the robotics environment. We were interested in studying and answering one research question: *Which green architectural tactics are employed for energy efficient-robotics software?* (RQ1). We answered this question by designing and carrying out a multi-phase study which resulted in 7 *energy-awareness* tactics and 4 *energy-efficiency* tactics. We discovered that all 11 tactics have already been applied in other application domains, hence, we aimed to describe the tactics in a general *implementation-free* manner to benefit developers not part of the ROS-community. The extracted green tactics can serve as an extensive guidance for roboticists as well as other developers interested in architecting and implementing cutting-edge energy-efficient software. Furthermore, the extracted green tactics can serve as a basis for ROS developers on understanding the gaps and limitations in green robotics software.

For future work, a thematic analysis approach can be applied to gather the patterns across the dataset and extract recurring themes present in the data points. Moreover, this would bring light to some of the most common energy-related problems faced by ROS-developers. We noted previously that the *safety* QA concerned a significant amount of data points - exploring *safety* tactics for green-robotics software could be another future work for roboticists and researchers interested in such direction. Due to the time-limit and scope of this project, we are not able to state with certainty which of the extracted green tactics are most commonly used by the ROS-developers; to achieve this, we will need to survey roboticists actively involved in the ROS projects used for extracting the green tactics.

Due to the scope and allocated time for this project, we did not provide evidence regarding the effectiveness of the green extracted tactics. In fact, in a currently on-going project, the extracted energy-efficient tactics are empirically evaluated by implementing each energy-efficient tactic into a real ROS-based system. Each tactic is assessed by evaluating the impact of each tactic in terms of energy consumption of the robot through different missions and physical environments.

Appendix

7.1 Chosen ROS Projects

.

Repository Name	Description	Data Point #	Tactic Example
aau_multi_robot	Autonomous multi-robot system.	41	EA1
		24	EA2
robotx_core	Robotx_core packages are ROS packages for Jetson TX2 or other embeded boards which is on our wam-v.	22	EA3
jsk_robot	Fundamental functions and systems necessary for future intelligent robots that will live and work in the daily life field and human society.	16	EA4
kobuki	Low-cost mobile research base designed for education and research on state of art robotics.	40	EA5
UBC-Thunderbots /Software	Software for autonomous soccer-playing robots.	15	EA6
SubjuGator	An autonomous underwater vehicle project.	13	EA7
		14	EE4
turtlebot	Low-cost, personal robot kit with open-source software.	36	EE1
ros_control	Set of packages that include controller interfaces, controller managers, transmissions and hardware_interfaces.	23	EE2
segbot	ROS drivers for controlling Segway-based robots.	51	EE3

Table 7.1: ROS projects used in the ROS examples for the green tactics

References

[1] KHAN SAAD BIN HASAN. **What, Why and How of ROS**. *Medium: Towards Data Science*, 2019. 1

[2] MORGAN QUIGLEY, BRIAN GERKEY, KEN CONLEY, JOSH FAUST, TULLY FOOTE, JEREMY LEIBS, ERIC BERGER, ROB WHEELER, AND ANDREW NG. **ROS: an open-source Robot Operating System**. *ICRA Workshop on Open Source Software*, 2009. 1

[3] TAKESHI OHKAWA, KAZUSHI YAMASHINA, TAKUYA MATSUMOTO, KANEMITSU OOTSU, AND TAKASHI YOKOTA. **Architecture exploration of intelligent robot system using ros-compliant FPGA component**. *2016 International Symposium on Rapid System Prototyping (RSP)*, 2016. 1

[4] DAYANG N. A. JAWAWI, ROSBI MAMAT, AND SAFAAI DERIS. **A Component-Oriented Programming for Embedded Mobile Robot Software**. *Int. Journal of Advanced Robotic Systems*, **4**:40, 2007. 1

[5] ARIEL PODLUBNE AND DIANA GOHRINGER. **FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs**. *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2019. 1

[6] R. IREM BOR-YALINIZ, AMR EL-KEYI, AND HALIM YANIKOMEROGLU. **Efficient 3-D placement of an aerial base station in next generation cellular networks**. *IEEE International Conference on Communications*, 2016. 1

[7] GANG HOU, ZHOU KUANJIU, TIE QIU, XUN CAO, MINGCHU LI, AND JIE WANG. **A novel green software evaluation model for cloud robotics**. *Computers and Electrical Engineering*, **63**:139–156, 2017. 1

[8] Felix Bachmann, Len Bass, and Mark Klein. **Deriving Architectural Tactics: A Step Toward Methodical Architectural Design**. *Carnegie Mellon University, Software Engineering Institute*, 2003. 2, 3

[9] Len Bass, Paul Clements, and Rick Kazman. **Software Architecture in Practice**. *Addison-Wesley Professional*, 2012. 2

[10] Grace Lewis. **Software Architecture Strategies for Cyber-Foraging Systems**. *PhD Dissertation: Vrije Universiteit Amsterdam*, 2016. 2, 5, 34, 40

[11] James Scott and Rick Kazman. **Realizing and Refining Architectural Tactics: Availability**. *Carnegie Mellon University*, 2009. 3

[12] Morgan Quigley, Brian Gerkey, and William D. Smart. **Programming Robots with ROS: A Practical Introduction to the Robot Operating System**. *O'Reilly Media*, 2015. 6

[13] Nicholas DeMarinis, Stefanie Tellex, Vasileios P. Kemerlis, George Konidaris, and Rodrigo Fonseca. **Scanning the Internet for ROS: A View of Security in Robotics Research**. *2019 International Conference on Robotics and Automation*, 2019. 6

[14] Ivano Malavolta, Grace A. Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. **How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems**. *Software Engineering in Practice*, 2020. 6, 12, 18, 19

[15] Sergi Hernández Juan and Fernando Herrero Cotarelo. **Multi-master ROS systems**. 2015. 7

[16] Yi Liu, Yuhua Zhong, Xieyuanli Chen, Pan Wang, Huimin Lu, Junhao Xiao, and Hui Zhang. **Fully Autonomous Robot System for Urban Search and Rescue**. *Proceedings of the IEEE International Conference on Information and Automation*, 2016. 7

[17] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. **The Goal Question Metric Approach**. *Encyclopedia of Software Engineering*, 1994. 9

[18] Pablo Estefoa, Jocelyn Simmonds, Romain Robbes, and Johan Fabryc. **The Robot Operating System: Package reuse and community dynamics**. *Journal of Systems and Software*, **151**:226–242, 2 2019. 11

[19] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. **MongoDB vs Oracle - database comparison**. *Emerging Intelligent Data and Web Technologies*, 9 2012. 15

[20] Irineu Moura, Gustavo Pinto, Felipe Ebert, and Fernando Castor. **Mining Energy-Aware Commits**. *ROS Wiki*, 2015. 23

[21] ROS Wiki. **ROS Topic**. *ROS Wiki*. 41

[22] ROS Wiki. **ROS Actionlib**. *ROS Wiki*. 41

[23] ROS Wiki. **ROS Service**. *ROS Wiki*. 41

[24] ROS Wiki. **ROS Bag**. *ROS Wiki*. 41

[25] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazieres, and Nickolai Zeldovich. **Energy Management in Mobile Devices with the Cinder Operating System**. *European Conference on Computer Systems*, 2011. 50

[26] Fumiko Satoh, Hiroki Yanagisawa, Hitomi Takahashi, and Takayuki Kushida. **Total Energy Management System for Cloud Computing**. *Cloud Engineering (IC2E)*, 2013. 50

[27] Thorsten Ochs, Henrik Schittenhelm, Andreas Genssle, and Bernhard Kamp. **Particulate Matter Sensor for On Board Diagnostics (OBD) of Diesel Particulate Filters (DPF)**. *SAE International Journal of Fuels and Lubricants*, 2010. 52

[28] Chao Long, Jianzhong Wu, Yue Zhou, and Nick Jenkins. **Aggregated battery control for peer-to-peer energy sharing in a community Microgrid with PV battery systems**. *Energy Procedia*, 2018. 55

[29] Thomas Greiner. **Control of Pollutant Emissions by ITS on the Austrian High Level**. *16th ITS World Congress and Exhibition on Intelligent Transport Systems and Services*, 2009. 57

[30] Danae Diakoulaki and Maria Mandaraka. **Decomposition analysis for assessing the progress in decoupling industrial growth from CO2 emissions in the EU manufacturing sector**. *Energy Economics*, 2007. 58

[31] Audun Bugge, Daymon Rogers, Jørn Norhei, Øivind Loe, Raman Sharma, Timotej Ecimovic, and Tom Zudock. **Silicon Labs: Multi-Node Energy Profiler**. *Silicon Laboratories*, 2017. 61

[32] Lev Mukhanov, Pavlos Petoumenos, Zheng Wang, Nikos Parasyris, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Hugh Leather. **ALEA: A Fine-Grained Energy Profiling Tool**. *ACM Transactions on Architecture and Code Optimization*, 2017. 61

[33] Paulo A.F. Rezeck, Bruna Frade, Jessica Soares, Luan Pinto, Felipe Cadar Chamone, Héctor Azpúrua, Douglas Guimarães Macharet, Luiz Chaimowicz, Gustavo Freitas, and Mario Fernando Montenegro Campos. **Framework for Haptic Teleoperation of a Remote Robotic Arm Device**. *Workshop on Robotics in Education*, 2018. 64

[34] Kalyan Das, Satyabrata Das, Rabi Kumar Darji, and Ananya Mishra. **Survey of Energy-Efficient Techniques for the Cloud-Integrated Sensor Network**. *Journal Of Sensors*, 2018. 64

[35] Dawud Gordon, Jürgen Czerny, and Michael Beigl. **Activity recognition for creatures of habit: Energy-efficient embedded classification using prediction**. *Personal and Ubiquitous Computing*, 2013. 68, 70

[36] Eduardo Souto, Reinaldo Gomes, Djamel Sadok, and Judith Kelner. **Sampling Energy Consumption in Wireless Sensor Networks**. *International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, 2006. 70

[37] Tongxin Shu, Min Xia, Jiahong Chen, and Clarence de Silva. **An Energy Efficient Adaptive Sampling Algorithm in a Sensor Network for Automated Water Quality Monitoring**. *Sensors*, 2017. 71

[38] Lingxiang Zheng, Dihong Wu, Xiaoyang Ruan, Shaolin Weng, Ao Peng, Biyu Tang, Hai Lu, Haibin Shi, and Huiru Zheng. **A Novel Energy-Efficient Approach for Human Activity Recognition**. *Sensors*, 2017. 71

[39] Lentin Joseph and Jonathan Cacace. **Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System**. *Packt Publishing*, 2018. 73

[40] Stephen Richard Lewallen, David Makower, Jonathan Joseph Hess, Patrick Heynenand Terry J. SANTAMARIA, William M. Bumgarner, David Pickford, Christopher L. OKLOTA, and Anthony S. PARKER. **Apple: On-demand resources**. *Apple*, 2016. 74

[41] Mohammad Vahid Samet Siar and Ahmad Fakharian. **Energy Efficiency in the Robot Arm using Genetic Algorithm**. *Artificial Intelligence and Robotics*, 2018. 77

[42] Giovanni Carabin, Erich Wehrle, and Renato Vidoni. **A Review on Energy-Saving Optimization Methods for Robotic and Automatic Systems**. *MDPI*, 2017. 78

[43] Jennifer Fereday and Eimear Caitlin Muir-Cochrane. **Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development**. *The International Journal of Qualitative Methods*, 2006. 78

[44] Michael Wehner, Ryan L. Truby, Daniel J. Fitzgerald, Bobak Mosadegh, George M. Whitesides, Jennifer A. Lewis, and Robert J. Wood. **An integrated design and fabrication strategy for entirely soft, autonomous robots**. *Nature*, 2016. 78

[45] Arlin Cunic. **Understanding Internal and External Validity How These Concepts Are Applied in Research**. *Social Research Methods*, 2020. 79, 80

[46] Fiona Middleton. **The four types of validity**. *Social Research Methods*, 2019. 80

[47] William M.K. Trochim. **Social Research Methods - Conclusion Validity**. *Social Research Methods*, 2020. 80