Supplementary Material (RQ2)

Abstract

Across the full range of software systems (e.g., in data centers, mobile applications, etc.) energy is considered a critical resource. At the current rate of growth, the total energy required by ICT alone is estimated to be over 10^{20} Joules by 2040; which is approximately a fifth of the global energy production and use in 2013 (5.67 x 10^{20} Joules). The role of the ever increasing use of robotics cannot be discarded in this growth. The **goal** of this technical report is to empirically evaluate architectural tactics for energy-efficient robotics software. The *Robot Operating System* (ROS) ecosystem has been used to extract **four** green tactics from real projects developed in real development contexts. This technical report performs an *empirical evaluation of the identified green tactics*. Specifically, each green tactic is implemented in a real robotic system running a common ROS software stack. Then, an objective assessment and measurement is carried out of the run-time impact of each tactic in terms of the energy consumption of the robot through different missions and physical environments. The experiment is carried out on a ROBOTIS TurtleBot3. The robot is instrumented with a hardware component in order to sample fine-grained data about its power consumption. In total 300 individual runs of the robotics system are performed, for a total of more than 10 hours of sheer run time.

This empirical evaluation concludes its experiments with the analysis of the results and the answering of the five research questions of this study. This study finds that almost all of the researched green architectural tactics show some positive impact, some (significantly) more than another, on energy consumption. This technical report then goes on to conclude if the use of the tactic is encouraged or not and gives the rationale thereof.

Contents

1 Introduction

Across the full range of software systems (*e.g.,* in data centers (1), mobile applications (2), etc.) energy is considered a critical resource. At the current rate of growth, the total energy required by ICT alone is estimated to be over 10^{20} Joules by 2040; which is approximately a fifth of the global energy production and use in 2013 (5.67×10^{20} Joules) [1].

The role of the ever increasing use of robotics cannot be discarded in this growth (3). Industrial robotics stand at the basis of the fourth industrial revolution, also referred to as *Industry 4.0* (4). Industrial firms contribute to 36% of total global energy consumption and 24% of total CO2 emissions (5). Energy consumption in the manufacturing sector has been declining since 1998. For instance, in the U.S., the energy consumption in the manufacturing sector decreased by 17% from 2002 to 2010 (6). Despite these improvements, Fysikopoulos et al. (7) assert that 20% to 40% unnecessary use of energy may still be found in industrial firms. Hence the energy performance of manufacturing systems is a *major area of research* and a concern for many manufacturing companies.

According to the IFR Statistical Department (8), the level of automation in the automobile frame- and body construction process was 90%, which implies a heavy use of industrial robots in related tasks. Also, Engelmann (9) states that about 8% of the total energy consumption in automotive industries belongs to industrial robots.

Robots also exist outside an industrial setting, these are however often in the form of commercially available mobile robots (*e.g.,* vacuum cleaner robots, lawn mower robots, etc.) Some hospitals are using mobile robots to provide quick and safe medicine delivery (10). Batteries are often used to provide power for mobile robots; however, as of writing, they are heavy to carry and have limited energy capacity. A Honda humanoid robot can

[1]https://www.zmescience.com/ecology/climate/how-much-renewable-energy/

walk for only 30 minutes with a battery pack they carry on the back (11); energy is the most important challenge for mobile robots.

Considering the aforementioned, it is logical to understand that the effort to maximize energy efficiency in robotics will have significant impact, being two-fold; (i) It will make a non-negligible impact on the world's energy use and consequently CO2 emissions. (ii) For mobile robots, it will increase their operating time (*i.e.,* battery life) and thus significantly improve the use for such robotic systems.

This technical report is part of a two-part research effort; (i) Identifying architectural tactics for energy-efficient robotics software. (ii) Empirically evaluating said tactics in a real-world experiment.

This technical report, consists of part (ii), based on the energy-efficient tactics as identified in part (i) Supplementary Material (RQ1).

The **goal** of this technical report is to empirically evaluate architectural tactics for energy-efficient robotics software. At the core of the study lies the concept of architectural tactic, *i.e.,* design decisions that influence the achievement of system qualities and can be reused across projects (12).

As part of part (i) of this study, the *Robot Operating System* (ROS) - further detailed in the next section 2.1 - ecosystem has been used to extract **four** green tactics from real projects developed in real development contexts by applying software repository mining techniques. The mined datapoints were rigorously and iteratively analysed and grouped into universal tactics by five different researchers, after which the actual green architectural tactics were defined.

This technical report performs an *empirical evaluation of the identified green tactics.* Specifically, each green tactic is implemented in a real robotic system running a common ROS software stack. Then, an objective assessment and measurement is carried out of the run-time impact of each tactic in terms of the energy consumption of the robot through different missions and physical environments. The experiment is carried out on a ROBOTIS TurtleBot3 (13), a compact and customizable ground robot widely used in research and open-source projects[1]. The robot is instrumented with a hardware component in order to sample fine-grained data about its power consumption. In total 300 individual runs of the robotics system are performed, for a total of more than 20 hours of execution time.

[1]https://github.com/ROBOTIS-GIT/turtlebot3

2

Background

2.1 Robotics Software

The Robot Operating System (ROS) (14), is the de-facto standard and key technological enabler for robotics software. ROS supports more than 140 types of robots and has a vibrant open-source ecosystem with many GitHub repositories containing ROS-based software, 4,152 publicly-available ROS packages, 7,696 ROS Wiki users, and 36,229 ROS Answers users (15)(16).

ROS comes in two major versions: *ROS1*[1] and *ROS2*[2]. Each major version has multiple distributions; a new one being developed approximately each year. As of writing, multiple distributions for both major versions are currently LTS (*Long Term Supported*) and regularly updated. The multiple distributions are available and LTS as to not break any existing software which might not be compatible with a new distribution. Distributions across major versions are by definition not compatible, as their underlying communication protocol differs significantly. It should be noted however, that some distributions within major versions are compatible with one another, such as ROS1 Melodic[3] (2018) and ROS1 Kinetic[4] (2016). ROS Allows for the development of hardware components and robotic systems without having to be aware of the development environment that they are possibly deployed in (*i.e.,* Operating Systems (OS), Programming Languages, System Hardware, etc.). As long as the development environment implements a mutually supported and compatible ROS version and distribution, the two components, be it only a hardware component added to a robotic system or a complete robotic system added to a cell of robotic

[1]http://wiki.ros.org/Distributions
[2]https://index.ros.org/doc/ros2/Releases/
[3]http://wiki.ros.org/melodic
[4]http://wiki.ros.org/kinetic

systems, will be able to communicate with one another.

This is made possible by the ROS communication methods available to ROS components. Each component can register itself as a ROS Node, each node can register the following communication methods with other nodes:

Topics Allow for many-to-many writing or reading asynchronous, continuous data streams by *Publishing* or *Subscribing* to a topic respectively[1].

Actions Allow for, one-to-one, synchronous and asynchronous client-to-server calls for specifically meant for use with long lasting tasks[2].

Services Allow for, one-to-one, client-to-server synchronous RPC (*Remote Procedure Calls*)[3].

These methods are the only communication methods available to ROS nodes. This significantly simplifies communication across nodes and it allows one node to make assumptions about another node's access points.

One could see the similarities between ROS and an API, *e.g.,* a RESTful API, which regardless of OS, programming language or hardware, allows for the communication of data between systems / nodes / components.

2.2 Architectural Tactics

At the core of the study lies the concept of architectural tactic, *i.e.,* design decisions that influence the achievement of system qualities and can be reused across projects (12).

Architectural tactics are a part of a software architect's design arsenal in order to achieve some desired system quality. 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* (17).

For this study, the quality attribute desired is *Energy Efficiency* and the green architectural tactics mentioned before are identified as *response measures* specifically for this *quality attribute* as they are observed to be *manipulating some aspect of a quality attribute model through architectural design decisions.*

To empiricially observe and quantify their impact on *Energy Efficiency*, is the single most important contribution of this technical report.

[1]http://wiki.ros.org/Topics
[2]http://wiki.ros.org/actionlib
[3]http://wiki.ros.org/Services

3

Study Design

In this section the design of this technical report is explained. In subsection 3.1 the tactics, as identified by part (i) of this study - as explained in section 1 - are given and detailed. In subsection 3.2 the GQM (*Goal Question Metrics*) method is applied and the therefrom arising experiment definition is further explained in subsection 3.3. The robot missions embodying the experiment are explained in subsection 3.4. The hardware additions to the ROBOTIS TurtleBot3, needed for instrumenting and facilitating the experiment, are explained in subsection 3.5. Anything involving the data, its manipulation and its analysis is explained in section 3.6, concluding the study design.

3.1 Energy-Efficient Architectural Tactics Defined

The set of **four** green architectural tactics have been identified using four sequential phases. These will be briefly discussed, as they were part of part (i) of this study - as explained in section 1. However describing the process, albeit more brief than as described in the technical report; Supplementary Material (RQ1), is paramount for understanding the validity of the four identified green architectural tactics, and thus consequently the validity of the results of this technical report's empirical evaluation.

3.1.1 Data Identification Phases

3.1.1.1 Phase 1: Dataset Construction

The goal of this phase was to build a dataset containing as much ROS-related data as possible. The following data sources were mined:

- *Open-source repositories*: Publicly-available dataset of 335 GitHub/GitBucket repositories containing real open-source ROS-based systems (18).

  - Cloned all the repositories and extracted 17,165 source code comments and Markdown files
  - Crawl[1] all pull requests/issues (including their discussions) and all commit messages for a total of 271,625 distinct data points.

- *Stack Overflow*: Crawl all questions with the ROS tag, their answers, comments, and related metadata, for a total of 1,180 data points.
- *ROS Answers*: This is the Q&A platform for ROS developers. The same information as for Stack Overflow has been crawled for *all* its posts, answers, comments, and related metadata.
- *ROS Discourse*: This is the platform for announcements and discussions of the ROS community. All its posts, discussions, and related metadata has been crawled.
- *ROS Wiki*: It hosts all the documentation, guidelines, and tutorials about ROS and official ROS packages. We crawl all its pages and related metadata.

This phase resulted in a total set of **339,563** datapoints.

3.1.1.2 Phase 2: Energy-Relevant Data Identification

In this phase, the data points mentioning energy-related topics, such as battery/power/energy consumption, sustainability etc. were identified. The keywords for identification have been identified by considering, analyzing and combining mining strategies in previous empirical studies on software energy efficiency (19)(20)(21)(22).

After the filtration, step one of this phase was completed and resulted in a filtered set of **3,354** datapoints. The high discard rate is not surprising, as it is in accordance with with existing research; confirming that developers have limited knowledge of energy efficiency (23).

The second step of this phase involved the manual validation of all 3,354 datapoints by three separate researchers. The validation involved removing datapoints that were clearly outside the scope of the study, *e.g.,* datapoints not automatically filtered out because of sentences like *"Problems with powering on a robot"*.

When step two was finished, this phase was finished too and the final set of datapoints consisted of **562** datapoints.

[1]We use the term "crawl" to refer to the systematic navigation of all target pages of a website, while extracting relevant data in an automated manner.

3.1.1.3 Phase 3: Architecturally-Relevant Data Identification

In this phase an in-depth assessment of the 562 datapoints is performed to consider only those discussing architecturally-relevant concerns. Borrowing the definition of a system concern from (24), where it is defined as the interest in a system relevant to one or more of its stakeholders (*e.g.,* presence of integrator nodes, system layers, interfaces to other systems). Inspired by the systematic literature review methodology (25), each data point is *manually analyzed* and selected it according to a set of well-defined inclusion and exclusion criteria. Two examples of representative inclusion criteria are: (i) data points concerning a ROS architectural entity (*e.g.,* ROS nodes, topics, services), (ii) data points mentioning architecturally-relevant design decisions and rationale.

The result of this phase was a set of **97** datapoints.

3.1.1.4 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 by applying the *thematic analysis* methodology (26). Thematic analysis was chosen because architectural information can be strongly dependent on project- and system-specific characteristics and thematic analysis copes well with context-dependent data (26)(18)(27). Four researchers are involved in this phase, whose activities can be decomposed into four main sequential steps: (i) for each data point two researchers independently collect the list of mentioned architectural entities (*e.g.,* ROS nodes, topics, services) and a extract a brief summary of the main design decisions employed for achieving energy efficiency; (ii) three researchers independently analyze each data point in its context (*e.g.,* by looking at the specific code changes associated to a pull request, checking the system documentation) and categorize them into common themes (*e.g.,* threshold-based mechanisms, usage of low-power mode); (iii) all researchers collaboratively organize the themes into a coherent set of distinguishable tactics via several iterations so to generalize, refine, and name each tactic; and (iv) each identified tactic is carefully reported via an extended version of the tactics template established in (28). The tactic template includes fields such as motivation, description, a concrete example coming from the dataset, constraints, dependencies, and variations.

The result of this phase is the set of **four** architectural tactics for energy-efficient robotics software which are defined in the next subsection 3.1.2 and which form the backbone of this technical report.

3.1.2 The Energy-Efficient Architectural Tactics

In this subsection, the four identified architectural tactics for energy-efficient robotics software are defined. In the previously described phases, two families of architectural tactics were eventually identified; *Energy-Aware* and *Energy-Efficient.* As *Energy-Aware* tactics do not necessarily improve energy efficiency in any concretely measureable way, this technical report's experiment explicitly focuses on the **four** *Energy-Efficient* architectural tactics for robotics software. Each tactic is denoted by the prefix *EE*, representing *Energy-Efficient.*

The definitions of the tactics were to be very precisely followed by the actual implementation in the robot missions, these are explained in section 3.4. The validity of the results of this empirical evaluation is decided by the adherence of the developed implementation in the robot missions compared to the provided formal defintion of the identified tactics.

We therefore give a **brief** (*i.e.,* the definition, motivation and technical information required for the implementation of the tactics) formal definition, as they are constructed in the technical report; Supplementary Material (RQ2).

3.1.2.1 EE1 - Limit Task:

Motivation There are robotic activities, such as data sampling or large amounts of data transfer, that consume a significant amount of energy. When the robot enters the *Energy-Savings Mode* - based on some battery threshold - limiting various robotic activities is pivotal in order to meet the required reduction of energy usage during *Energy-Savings Mode.*

Description

1. The Task Requester sends a task to the Arbiter.
2. After receiving the task, the Arbiter checks the battery level of the robot (provided by another component in the system).
3. If battery level is below the established threshold, the Arbiter obtains the energy-savings mode task configuration from the Energy-Savings Mode Manager.
4. The Arbiter forwards the task to the Task Executor for execution in either default mode or energy savings mode.
5. The Arbiter continues checking the battery level during the execution of the task.

6. If the battery level falls below the threshold, the Arbiter obtains the energy-savings mode task configuration from the Energy-Savings Mode Manager and instructs the Task Executor to continue execution of the task in its energy-savings mode.
7. Similarly, if the battery level rises above the threshold, the Arbiter instructs the Task Executor to continue execution of the task in its default mode.
8. Once the task is completed the Task Requester is notified.

Documentation The sequence diagram which the robot mission implementing EE1 - defined in section 3.4 - had to precisely adhere to, is given in figure 3.1.

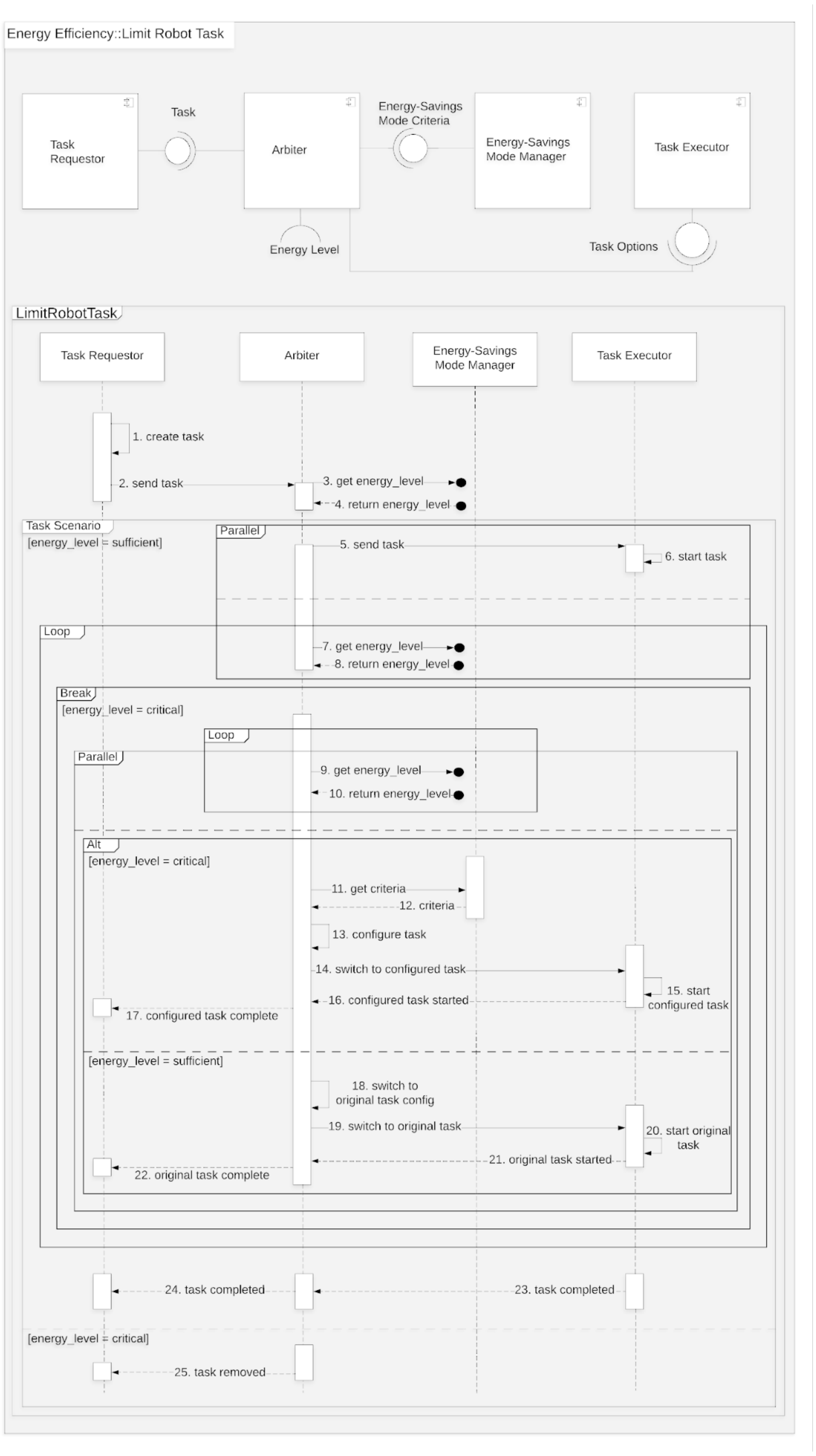

Energy Efficiency::Limit Robot Task
Task Requestor
Task
Arbiter
Energy-Savings Mode Criteria
Energy-Savings Mode Manager
Task Executor
Energy Level
Task Options
LimitRobotTask
1. create task
2. send task
3. get energy_level
4. return energy_level
Task Scenario
[energy_level = sufficient]
Parallel
5. send task
6. start task
Loop
7. get energy_level
8. return energy_level
Break
[energy_level = critical]
9. get energy_level
10. return energy_level
Alt
[energy_level = critical]
11. get criteria
12. criteria
13. configure task
14. switch to configured task
15. start configured task
16. configured task started
17. configured task complete
[energy_level = sufficient]
18. switch to original task config
19. switch to original task
20. start original task
21. original task started
22. original task complete
23. task completed
24. task completed
[energy_level = critical]
25. task removed

Figure 3.1: EE1 Limit Task Sequence Diagram

3.1.2.2 EE2 - Disable Hardware:

Motivation Hardware components (*e.g.,* sensors, drivetrains) of a robot often consume a significant amount of energy; for example, sensors consume energy to be able to function and measure whatever it is that they are designed to measure. If this sensor data is then not used, the energy is wasted. It is essential that unnecessary utilization of hardware resources is prevented as much as possible, in order to be as energy efficient as possible.

Description Hardware is needed:

1. The HW Requestor requests to use the HW from the HW State Controller.
2. After receiving the request, the HW State Controller switches the HW Controller to *ENABLED*.
3. Being enabled triggers the HW Controller to enable the HW device.
4. The message of the HW being successfully enabled is now sent back the the HW Requestor.

Hardware is no longer needed:

1. The HW Requestor requests to disable the HW from the HW State Controller.
2. After receiving the request, the HW State Controller switches the HW Controller to *DISABLED*.
3. Being disabled triggers the HW Controller to disable the HW device.
4. The message of the HW being successfully disabled is now sent back to the HW Requestor.

Documentation The sequence diagram which the robot mission implementing EE2 - defined in section 3.4 - had to precisely adhere to, is given in figure 3.2.

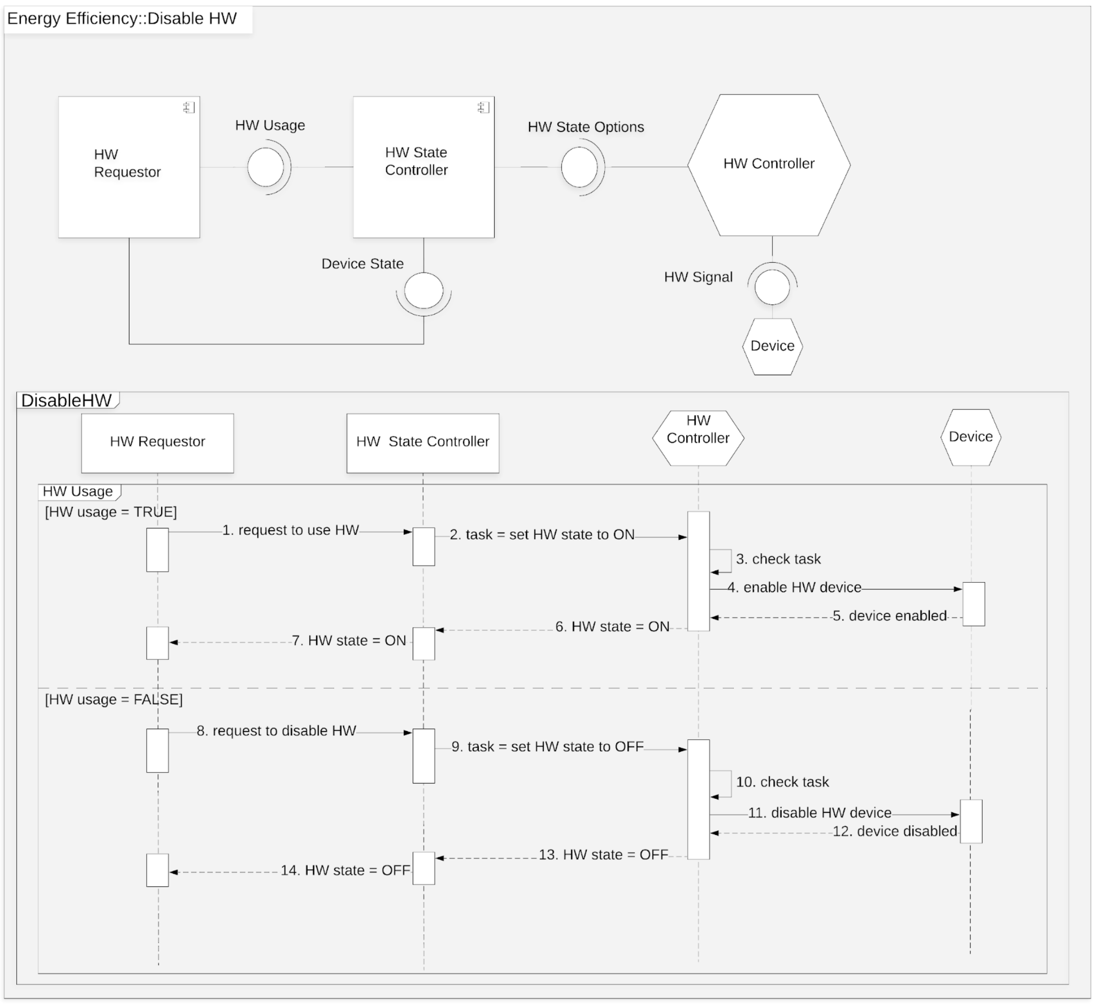


Figure 3.2: EE2 Disable Hardware Sequence Diagram

3.1.2.3 EE3 - Energy-Aware Sampling:

Motivation In robotics, many sensors are designed to provide a continuous stream of data (e.g., accelerometers, lidars). Sampling data from sensors is, varying based on the type of the sensor and the energy consumption typically associated with such a type of sensor, still an energy consuming task. 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 consumption is reduced to avoid to drain the - now critical - battery at the same pace.

Description

1. The Sensor Requestor requests to sample the sensor from the Sampling Rate Controller.
2. The Sampling Rate Controller requests the sampling rate from the Sensor Controller
3. The Sampling Rate Controller adjusts the sampling rate of the Sensor Controller continuously based on the *energy level* being *sufficient* or *critical.*
4. The Sensor Requestor samples the Sensor Controller through the Sampling Rate Controller.

Documentation The sequence diagram which the robot mission implementing EE3 - defined in section 3.4 - had to precisely adhere to, is given in figure 3.4.

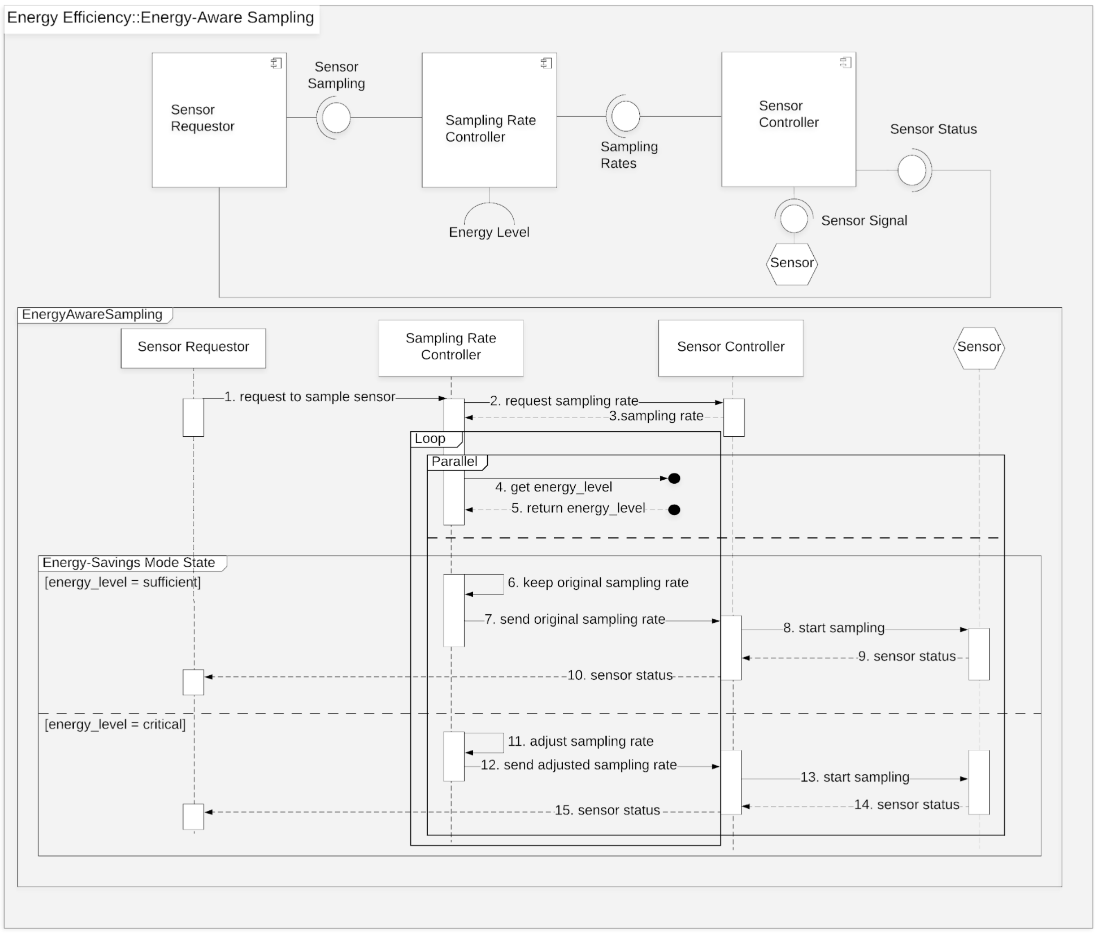


Figure 3.3: EE3 Energy-Aware Sampling Sequence Diagram

3.1.2.4 EE4 - On-Demand Components:

Motivation Continuously running a component requires the spawning of a process which is an energy-consuming task in terms of CPU usage (*e.g.,* executing a CPU-intensive loop) and other resources (e.g., sensors, motors, fans for cooling). For this reason, it is necessary to ensure that the processes that are not being utilized do not consume energy unnecessarily by running on the system.

Description

1. The Requestor orders the Component Manager to (de)spawn the On-Demand Component
2. The Component Manager (de)spawns the On-Demand Component.
3. The On-Demand Component lets the Requestor know, through the Component Manager, that it has been (de)spawned.
4. If *SPAWNED*; The Requestor can now request services form the On-Demand Component.

Documentation The sequence diagram which the robot mission implementing EE3 - defined in section 3.4 - had to precisely adhere to, is given in figure 3.4.

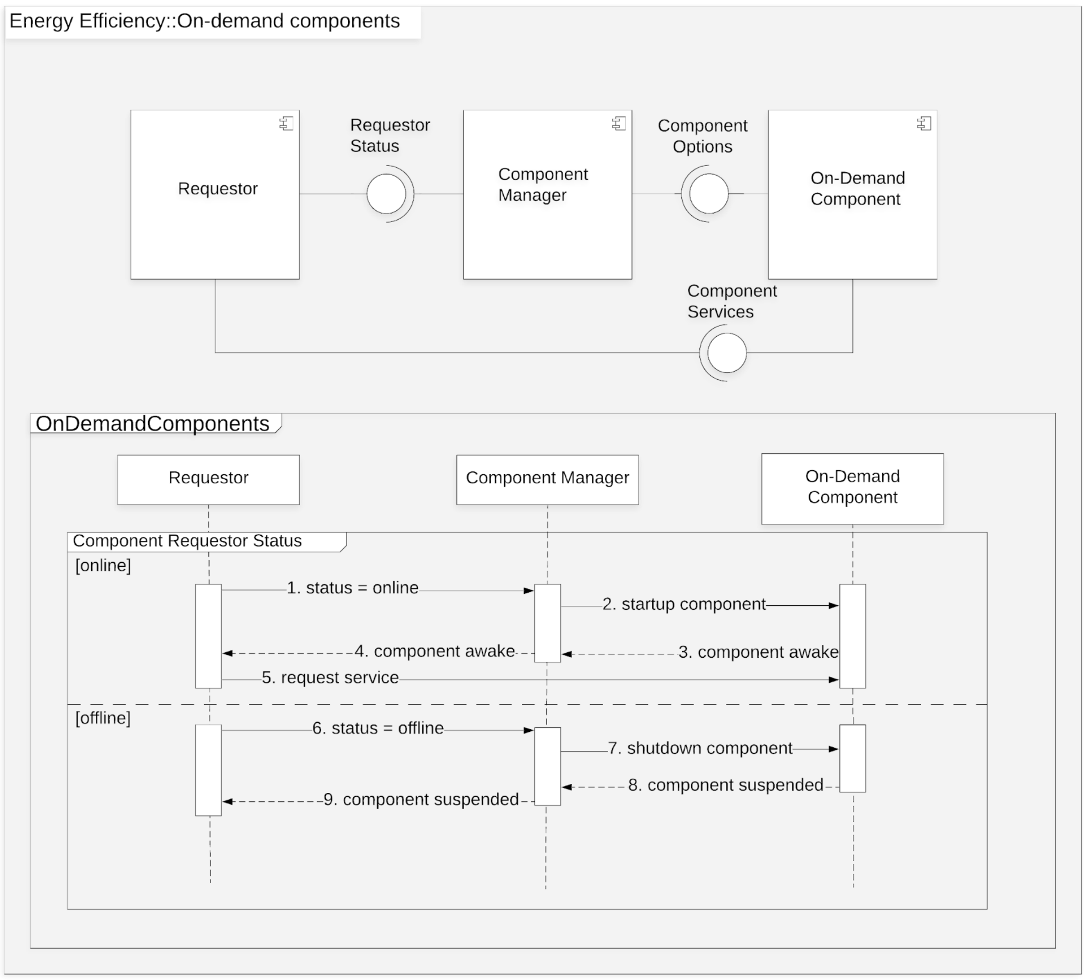


Figure 3.4: EE4 On-Demand Components Sequence Diagram

3.2 Goal and Research Question

Following the Goal-Question-Metric approach (25), the **goal** of this study is to analyse *the ROS ecosystem* for the purpose of *identifying and evaluating* a set of *architectural tactics* with respect to their *energy-efficiency* from the point of view of *roboticists and researchers* in the context of *open-source ROS-based systems*. The goal drives the design of the full study and leads us to the following research questions.

RQ1 – Which architectural tactics are applied in the development of energy-efficient robotics software? This research question is answered *qualitatively*; we identify, extract, and establish a concrete set of green tactics used in real-world robotics projects. The tactics are synthesized via a multi-stage experimental study targeting several open-source robotics projects and their related artifacts. Answering this research question helps both (i) roboticists in designing and developing energy-efficient robotics software via the established tactics and (ii) researchers by providing an initial foundation for new scientific contributions, such as techniques to automatically improve the energy efficiency of robotics software.

RQ2 – To what extent does the application of green tactics impact robotics software' energy efficiency? This research question is answered *quantitatively*; for each green tactic we carry out an empirical assessment of its run-time impact in terms of energy consumption of a real robot. By answering this RQ, roboticists are offered objective data on how different green tactics can make their robotics systems more efficient.

3.3 Experiment Definition

In this section the definition of the experiment is given, consisting of:

1. The factors of the experiment.
2. The treatments of those factors.
3. The dependent and in-dependent variables involved.
4. The mission definition for the robotic platform in natural language.
5. The run schedule.
6. The amount of runs and time involved.

Type	Name	Value	Category
Independent (main factors)	Tactics	baseline	nominal
		EE1	nominal
		EE2	nominal
		EE3	nominal
		EE4	nominal
		combined	nominal
	Movement	No Movement	nominal
		Fixed Movement	nominal
		Autonomous Movement	nominal
	Environment	Empty	nominal
		Obstacles	nominal
Fixed Dependent (co-factors)	Robotic Platform	ROBOTIS TurtleBot3	nominal
	ROS Version	ROS1 Melodic	nominal
	Network Connection	Wi-Fi	nominal
	Device OS	Raspbian	nominal
	Remote PC OS	Ubuntu 18.04	nominal
	Run Duration	120 Seconds	ratio
Dependent	Energy Consumption	(J) Joules	ratio
	CPU Usage	(%) Percentage	ratio
	RAM Usage	(%) Percentage	ratio

Table 3.1: Factors, treatments and other experiment related variables.

3.3.1 Factors and Treatments

In table 3.1, the factors, their treatments and other experiment related variables are given. The experiment definition is rigorously defined and iteratively implementing feedback and new ideas into the design.

For the *Tactic* factor, one can see the treatment *baseline*, which has not yet been discussed. The *baseline* treatment is just that, it is performing the mission, as defined in the next subsection, without any green architectural tactic implemented. This, so that any of the results of the *tactics (EE1 - EE4)* or the *combined* treatment can be compared to the baseline, say; *the control group.*

3.3.2 Experiment Mission

The experiment as performed by the robotic platform; the ROBOTIS TurtleBot3, has to satisfy three main factors: *The Tactic Considered*, *The Movement of the TurtleBot3* and *The Environment it moves in.*

The various treatments to these factors are given in table 3.1. Based on these treatments, it is logical to understand that the missions as performed by the TurtleBot3 will differ significantly.

3.3.2.1 Differentiating Aspects

Firstly, the various tactics - as defined in section 2.2 - differ in what they change about the robotic system significantly. So significantly, that the addition of hardware was needed to be able to construct a single mission format in which all tactics could be tested. More about this in section 3.5. For example, tactic EE3 controls the sample rate of a sensor, however the TurtleBot3 does not have a sensor that allows for custom sample rates out-of-the-box.

Secondly, the code for performing the mission should be the bare minimum required to make the robot succeed at the mission, so that the share of code which embodies the tactic is as high as possible. Therefore, code that would deal with traversing an object, as needed for the *Obstacles* treatment of the *Environment* factor, will not be present in the mission which is designed for the *Empty* treatment of the same factor. The same goes for the *No Movement* treatment of the *Movement* factor, compared to the *Fixed or Autonomous movement* treatments. It would be illogical to have code present for moving the robot in the mission defined for the *No Movement* treatment.

3.3.2.2 The Arena

The robotic system will perform all of its missions inside a predetermined area, called the *arena*. The arena measures $4.5m \cdot 3.5m$ totalling $15.75m^2$. The arena can be seen empty and with obstacles in figures 3.5 and 3.6 respectively.

Obstacles: For the obstacles, a standard issue cone - used for automotive driving skill training - has been used.

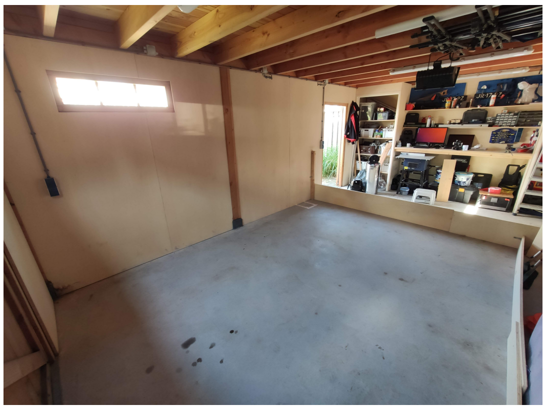


Figure 3.5: The arena set up as an empty environment

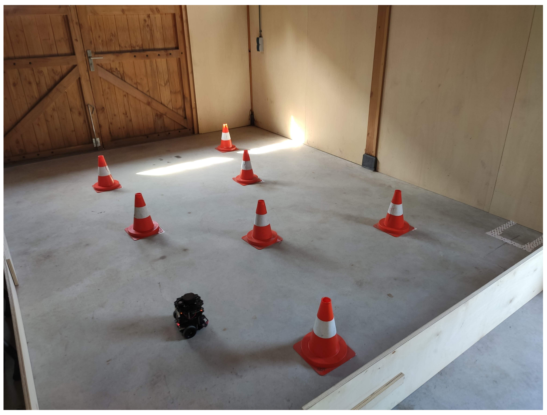


Figure 3.6: The arena set up as an obstacled environment

3.3.2.3 The Missions

It should be noted here, that the description of the missions require the knowledge of the hardware additions as described in the coming section 3.5. The hardware addition in question, is the Raspberry Pi Camera Module that has been added to the TurtleBot3. Any further details are given in section 3.5. For this section, it is enough to know that the TurtleBot3 has a camera module that it can control through code.

The missions as performed by the TurtleBot3, are significantly influenced by the treatments of the main factors; *Tactics, Movement.* It is therefore, that the missions are defined based on the the treatments of these factors.

The descriptions of each of the *Movement* treatments are given below, it should be noted that for each mission the robot will rotate a full 360° every 20 seconds (even for the *No Movement* treatment). This interval is chosen as it ensures the robot driving around between rotations for 20 seconds, it is during these 20 seconds that the the six treatments of the *Tactics* factor influence the robot in different ways; hopefully resulting in different energy consumption. A 20 second interval will cause the robot to drive ≈ 50% and rotate ≈ 50% of the mission duration.

No The robot stands still, it will always be in the same position.

Fixed The robot is performing a 'sweeping' like motion - see figure 3.7 - across the arena.

Autonomous The robot performs autonomous movement - see figure 3.8 - driving around the arena, deciding the next direction to drive in on each detection of an obstruction of the motion path. The next direction the turtlebot will drive in is autonomously

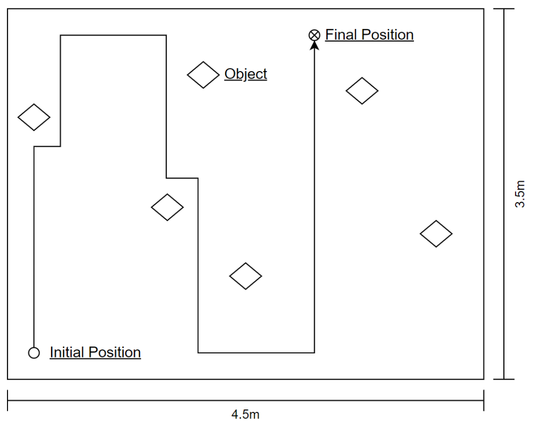


Figure 3.7: The *sweeping* trajectory as performed for *Fixed Movement*

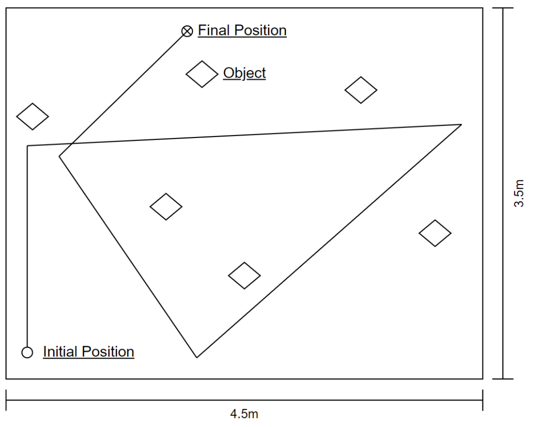


Figure 3.8: The *non-deterministic* trajectory as performed for *Autonomous Movement*

decided on runtime by the turtlebot. The next direction is chosen as the direciton in which the longest distance can be traveled before being obstructed again.

Now that a definition for the missions is given based on the *Movement* factor, the next defining factor can be considered; *Tactics*. However, before these definitions could be constructed, one would first need to know what the *actual* implementation of the *tactics* would entail for the robotic system (*i.e.,* the TurtleBot3 with its hardware additions). The implementation details, and their rationale, for the tactics are given below:

baseline Just perform the *Movement* treatment, recording the entire mission at 60 FPS.

(a) 60 FPS was chosen as it is the highest supported sampling rate for video for the Raspberry Pi Camera Module.

EE1 Limit task: *Movement* by **waiting 5 seconds before each rotation on interval**[1]**.**

(a) Five seconds has been chosen to investigate the impact of limiting the movement on energy efficiency.

EE2 Disable HW: **Disable the camera** while moving from a location to another.

(a) The camera was chosen as the 'to-be-disabled hardware' as no other hardware present on the TurtleBot3 platform would be able to be disabled without compromising the mission or the TurtleBot3 itself.

[1]Specifically interval, as the robot also performs rotations to navigate the arena. However, only rotations on the set 20s interval are meant here.

i. Disabling the motors would make the robot not able to move.
ii. Disabling the LiDar sensor would make the robot blind, driving into obstacles or the walls of the arena.
iii. By default, there are not any other sensors or hardware components programmatically controllable by the TurtleBot3. This was the main reason a hardware addition, in the form of the Raspberry Pi Camera Module, was needed.

EE3 Energy-Aware sampling: sample the **camera** at **30 FPS**.

(a) The camera was chosen as it is the only sensor now present on the TurtleBot3 that allows for a customisable sample rate.
(b) 30 FPS (50%) was chosen as a large enough value to be significantly different from 60 FPS, and also still be a standard, de-facto sample rate for any 'normal' camera system.

EE4 On-demand Components: On-demand **camera component**, despwaned while moving from a location to another.

(a) The camera component was chosen for the same reason as for EE2; no other component would be able to be on-demand without being compromising.
(b) This tactic is different from EE2, although it encapsulates it, considering it does not only disable the hardware, but also completely removes the hardware controller (the on-demand component) from any CPU scheduling.

combined Perform the mission, implementing EE1 to EE4.

(a) It should be noted for the *Combined* treatment of the *Tactics* factor; that *EE4 - (de)spawning on-demand components* by definition encapsulates *EE2*, as to be able to *spawn or despawn* a component, the hardware used by that component needs to be *enabled or disabled.* This *Combined* treatment therefore still adheres to combining all four tactics (*EE1 - EE4*).

Now that the treatments of the *tactics* factor are defined in their implementation, the implementation of the actual missions can be described:

baseline The robot will perform the *Movement* treatment, while recording the entire mission with the camera module at 60 FPS.

EE1 The robot will perform the *Movement* treatment, **waiting 5 seconds before each rotation** to limit the amount of movement per run, while recording the entire mission with the camera module at 60 FPS.

EE2 The robot will perform the *Movement* treatment, while recording only the full 360° rotation at 60 FPS, **disabling the HW** while moving from a location to another.

EE3 The robot will perform the *Movement* treatment, while recording the entire mission with the camera module **at 30 FPS**.

EE4 The robot will perform the *Movement* treatment, while recording only the full 360° rotation at 60 FPS, **despawning the on-demand camera component** while moving from a location to another.

combined The robot will perform the *Movement* treatment **waiting 5 seconds before each rotation**, while recording only the full 360° rotation **at 30 FPS**, **despawning the on-demand camera component** while moving from a location to another.

3.3.3 Run Schedule

Now that the missions and the various treatments of the main factors are defined, a run schedule can be created. The run schedule depicts the combinations of treatments, in table 3.2 all the combinations and the amount of runs are shown, depicting a total of **300** experiment runs as performed for this empirical evaluation.

Note: Considering the three main factors *Tactics, Movement and Environment* have *6, 3 and 2* treatments respectively, the total amount of variations can easily be calculated:

$$6 \cdot 3 \cdot 2 = 36$$
$$Tactics \cdot Movement \cdot Environment = Variations$$

However, we only consider 30 variations; considering the combination of the *No Movement* treatment for the *Movement* factor with the *Obstacles* treatment of the *Environment* factor would make no sense as the robot is standing still the entire mission and thus will never cross an obstacle.

Therefore these 6 variations, *the 6 runs of the tactics for this combination*, are omitted from the run schedule.

Considering the 30 variations; each variation is run 10 times, the total amount of runs for this experiment will thus total **300 runs**. Considering table 3.1, and the *Run Duration* co-factor set to a agreed upon 2 minutes, or 120 seconds; the **300** runs will thus take **600 minutes** or **10 hours**.

This is **10 hours** of sheer robot runtime. Each run will have a significant amount of overhead of manual labour involved:

1. Fit the robot with a new, completely charged, battery so that each run is started with the same amount of battery potential.

   (a) To make this process easier, we use **three** batteries and their dedicated chargers. Otherwise, this overhead would have made the experiments extremely hard to perform within reasonable time.

2. Read the robot's SD card with the data of the previous run into the computer manually, format the SD card and put it back into the robot.

3. Put the robot on the designated start position and start the robot up.

Table 3.2: Experiment Execution Run Schedule.

Movement	*Tactics*	*Environment*	*Amount of Runs*
No Movement		**Empty**	
No Movement	baseline	Empty	10
No Movement	EE1	Empty	10
No Movement	EE2	Empty	10
No Movement	EE3	Empty	10
No Movement	EE4	Empty	10
No Movement	combined	Empty	10
Fixed Movement		**Empty**	
Fixed Movement	baseline	Empty	10
Fixed Movement	EE1	Empty	10
Fixed Movement	EE2	Empty	10
Fixed Movement	EE3	Empty	10
Fixed Movement	EE4	Empty	10
Fixed Movement	combined	Empty	10
Fixed Movement		**Obstacles**	
Fixed Movement	baseline	Obstacles	10
Fixed Movement	EE1	Obstacles	10
Fixed Movement	EE2	Obstacles	10
Fixed Movement	EE3	Obstacles	10
Fixed Movement	EE4	Obstacles	10
Fixed Movement	combined	Obstacles	10
Autonomous Movement		**Empty**	
Autonomous Movement	baseline	Empty	10
Autonomous Movement	EE1	Empty	10
Autonomous Movement	EE2	Empty	10
Autonomous Movement	EE3	Empty	10
Autonomous Movement	EE4	Empty	10
Autonomous Movement	combined	Empty	10
Autonomous Movement		**Obstacles**	
Autonomous Movement	baseline	Obstacles	10
Autonomous Movement	EE1	Obstacles	10
Autonomous Movement	EE2	Obstacles	10
Autonomous Movement	EE3	Obstacles	10
Autonomous Movement	EE4	Obstacles	10
Autonomous Movement	combined	Obstacles	10
Total:			300

4. Wait for the robot to have started and connected to the wireless network, SSH into the robot and start the mission - defined in 3.4 - and wait for all ROS Nodes, Topics, and Services to be available.
5. Start the mission controller on the computer.
6. Wait for the Run Duration of 120 seconds, and repeat for 300 runs.

This overhead of manual labour took, on average, about 2 to 5 minutes. This brought the total experimentation time needed to complete the 300 runs to $\approx 50_h$. The total time spent performing the experiments was 7 days of 8 hours per day. Which would total $\approx 56_h$, roughly correct including a coffee and lunch break here and there.

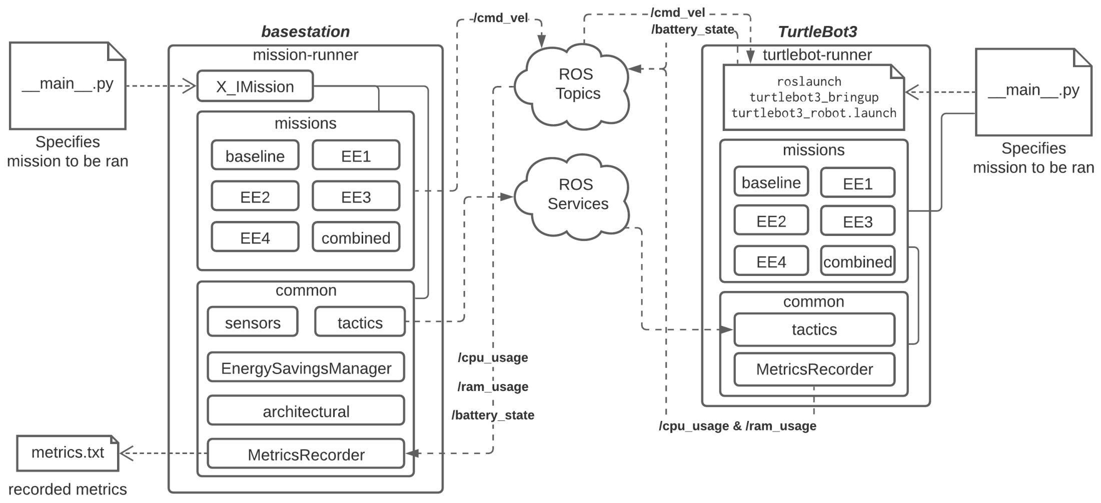


Figure 3.9: The framework developed, in a compact overview.

3.4 Mission Implementation

In this section the missions as implemented in software, and the framework built to support them and their execution are explained.

3.4.1 Framework

For the execution of the missions, a framework has been built, this framework is depicted in a compact overview in figure 3.9. In this subsection the various parts are explained and further depicted in detail.

To annotate figure 3.9;

1. ROS Topics - are used to communicate asynchronous, continuous data (e.g., *CPU and RAM usage and the current state of the battery at any given moment in time.*).

2. ROS Services - are used to communicate synchronously, one-way, much like RPC's[1]; to execute tasks required for the correct functioning of the tactics in the mission context (e.g., *'/camera/spawn' - a single, synchronous, one-way RPC to spawn the Camera Component as part of tactic **EE4***).

[1]Remote Procedure Call

This framework has been built with reusability in mind and therefore all code that is not mission-specific is abstracted away into the *Common* module which is imported and used by the mission-specific files.

The framework has been split into two parts; *mission-runner* and *turtlebot-runner*. Where mission-runner is the part that runs on the basestation (*i.e.,* computer / laptop) and turtlebot-runner, as the name implies, runs on the TurtleBot3. The turtlebot-runner part of the framework is however also not at all TurtleBot3 specific and all reusable code has been split away into the *Common* module.

3.4.1.1 mission-runner

The mission-runner part of the framework is designed to be the controlling part of the framework. It controls the TurtleBot, or any other ROS-based system for that matter, through predetermined ROS Nodes, Topics and Services.

Technical Overview:

In the technical overview, as shown in figure 3.10, only the mission-runner part of the framework is shown. The turtlebot-runner part is shown in the next subsection. It should also be noted, that to depict the usability and the *actual* implementation of the framework, the mission-specific details have, in its most abstract form, been incorporated in the overview.

Some important aspects of the overview are given:

1. The EnergySavingsManager is the controller, using the BatterySensor, to provide the information of a sufficient energy budget to any dependent of the EnergySavingsManager. In this manager, the threshold for the BatteryState can be given, which compared to the actual BatteryState will tell any user of the EnergySavingsManager if the energy budget is sufficient.
   (a) **Note:** That for this experiment, the treshold has been set to 100%, so that every tactic, which enacts its custm logic if the energy budget is found to be insufficient, actually enacts its logic from the start of the mission embodying that tactic.
   (b) This is logical, as the *baseline* mission emobodies the normal operation of the robot, to which the results of the runs of the tactics are compared.

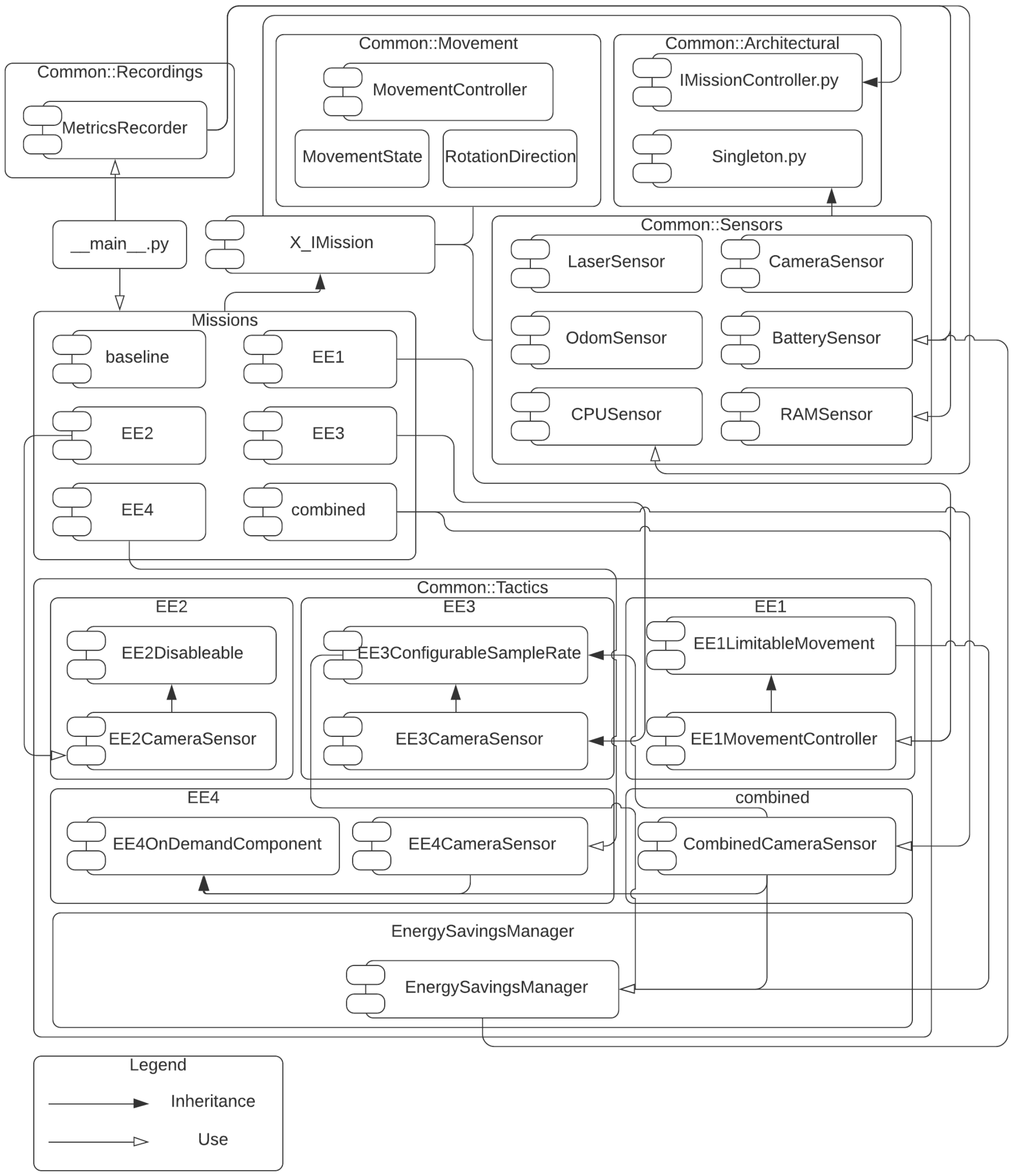


Figure 3.10: The mission-runner part of the framework shown in a technical overview.

2. The MetricsRecorder is responsible for recording any data published in any topic that is needed for gaterhing metrics.
   (a) These topics are provided by the *Sensors* and consist of the; *BatterySensor*, *CPUSensor* and the *RAMSensor* which supply the data to the mission-runner part of the framework as a representation of the data published by the turtlebot-runner framework on the respective topics; */battery_state*, */cpu_usage* and */ram_usage*.
3. All Sensors are Singletons, as a ROS Topic is also a single channel (*i.e.,* address; *e.g.,*'/BatteryState') and so the code representing those topics is inherently Singleton.
4. The tactics are located in the *Common* module.
   (a) Tactics and their arbiter - as described in section 3.1.2 - are defined as abstract classes, enabling their functionality for whatever class inherits them. This code is also **resuable** and can be used *right now* in any codebase for ROS-based systems.
   (b) Each tactic is a folder containing the abstract class representing the *Arbiter*, and the controller, controlled by the *arbiter*, responsible for the control over the entity (*e.g., EE1MovementController, EE2CameraSensor* representing the implementations of *EE1* and *EE2* respectively).
   (c) **Note:** That under any 'normal' circumstance, these abstract classes should be inherited by the default controllers controlling the entity. However, as this study explicitly studies four tactics, six treatments (including *baseline* and *combined*), while sharing a common codebase, these had to be placed away from the baseline, common code and placed in their respective folders.
   (d) **Note:** That thanks to this abstract, modular construction, EE4 is very easily implemented by defining the *EE1MovementController* as the default MovementController in the *X_ Combined.py* file for whatever movement treatment is considered. Furthermore, the default CameraSensor is changed to the *EE4CameraSensor* which inherits the abstract *EE3ConfigurableSampleRate* and *EE4OnDemandComponent* classes. Enabling these tactics automatically, using one codebase.
5. X_IMission, where X stands for the *Movement* factors. So; N_IMission, F_IMission and A_IMission for the treatments *No Movement, Fixed Movement* and *Autonomous Movement* respectively.

(a) It is inherited by the six *Tactics* treatments, in the overview grouped under 'missions'.

(b) Each implements the mission as needed, and associated with that combination of *Tactic* and *Movement* treatment.

(c) X_IMission always inherits the *IMissionController* which is not mission-specific. It allows for the rest of the framework to make assumptions about what constitutes a MissionController and defines the universal methods and variables needed by each mission.

(d) X_IMission always makes the following controllers are available:

   i. MovementController, for each mission will have to be able to move the robot.

   ii. The basic Sensors; OdomController and CameraController, for each mission will have to be able to rotate, and thus have Odometry data available, and each mission will have to be able to control the Camera.

   iii. **Note:** That the LaserSensor is the only other sensor that would otherwise be necessary to be able to perform missions. The other Sensors are only for the MetricsRecorder to record them. But considering they are part of the *Common* module, can of course be used and read out by any other file.

6.

3.4.1.2 turtlebot-runner

The turtlebot-runner part of the framework is designed to be the part that is controlled by the mission-runner part of the framework. It controls the TurtleBot directly, through libraries directly influencing the hardware itself, in accordance with commands received from the mission-runner. Note that also for this part of the framework, all code that is not mission-specific is abstracted away into the *Common* module and that this code, in conjunction with the mission-runner *Common* module, is completely independent and resuable for any other ROS-based system, through predetermined ROS Nodes, Topics and Services.

Technical Overview: In the technical overview, as shown in figure 3.11, only the turtlebot-runner part of the framework is shown. It should also be noted, that to depict the usability and the *actual* implementation of the framework, the mission-specific details have, in its most abstract form, been incorporated in the overview.

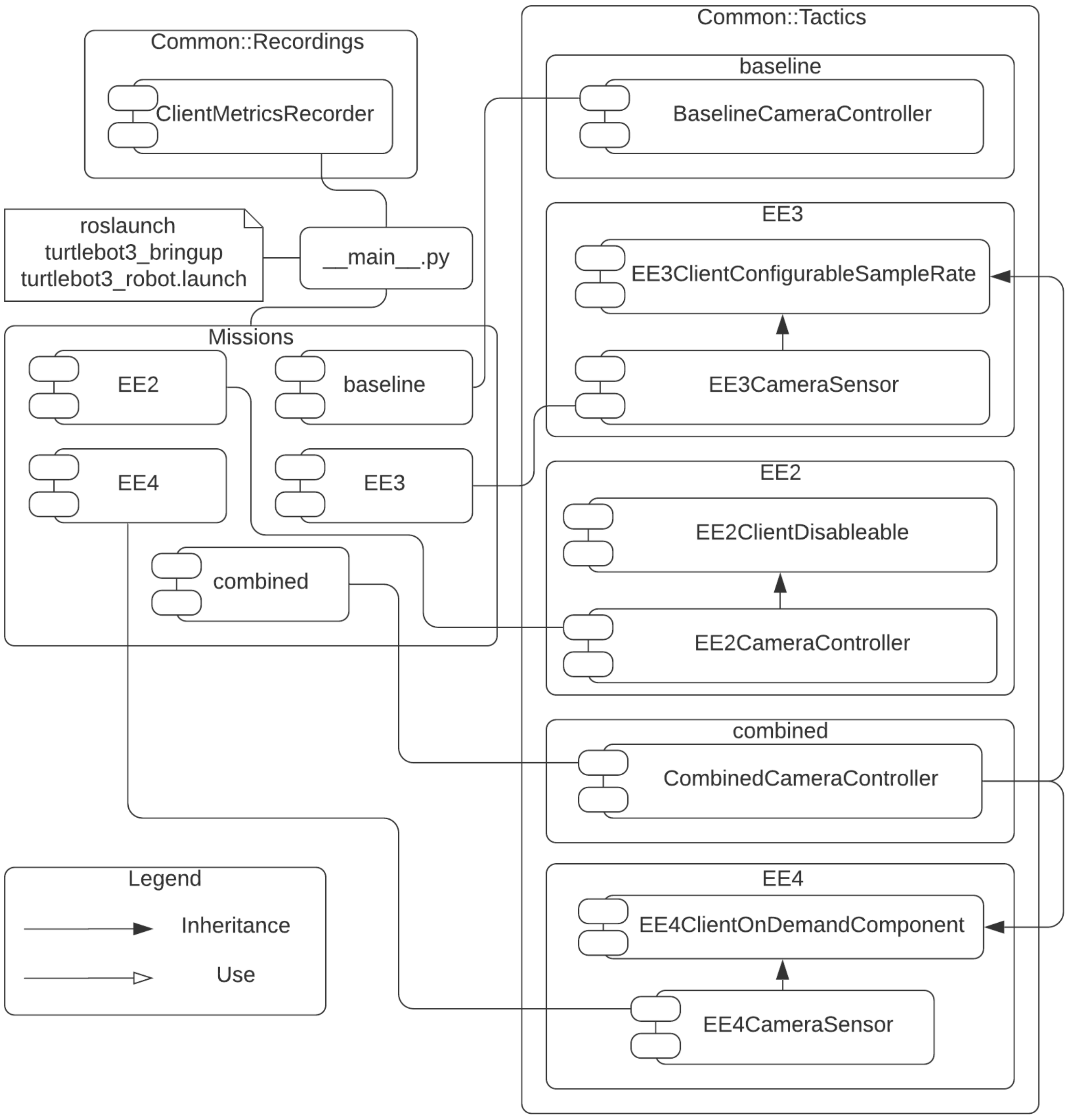


Figure 3.11: The turtlebot-runner part of the framework shown in a technical overview.

Some important aspects of the overview are given:

1. The turtlebot-runner part of the framework relies heavily on the de-facto mission for a TurtleBot3 as given in the ROBOTIS E-Manual; turtlebot3_bringup[1]. This command will start a ROS program for the TurtleBot3, enabling the TurtleBot3 ROS node. This node is responsible for publishing data such as the Laser, Odometry and BatteryState data. it is also responsible for responding to any movement commands.

2. __main__.py imports the missions and runs them directly, as the codebase for each mission specific file is so small. These files only enable the required CameraController according to their tactic.

3. __main__.py runs the ClientMetricsController which publishes data such as the CPU and RAM usages in percentages on the dedicated *'/cpu_usage'* and *'/ram_usage'* topics respectively. So that it can be recorded to a file directly on the computer by the MetricsRecorder of mission-runner on the other side.

4. The reusable codebase once more proves itself by the way the tactics have been implemented. The *Arbiters* of the tactics have been implemented as abstract classes which can be inherited by any other class what needs to implement any of the tactics. The *Combined* treatment for the *Tactics* factor is also here a great example of this. The *CombinedCameraController* only needs to inherit the *EE3ClientConfigurableSampleRate* class and the *EE4ClientOnDemandComponent* class to incorporate all four tactics *(EE1 - EE4)*.

5. *EE1* Is missing from this overview, as the tactic is implemented solely on the mission-runner side. As processing the movement commands remains the same, they are altered based on the available energy budget on the mission-runner side before sending.

3.4.1.3 robot-runner

In this section, a brief explanation of robot-runner is given and the rationale for why it was not used and why this is of importance. In January of this year (2020), we developed robot-runner; a framework enabling the automatic execution of experiments defined as mission files and a config.JSON. This framework was to be used and for a long time that made sense.

[1]https://emanual.robotis.com/docs/en/platform/turtlebot3/bringup/

However, as specifics of the experiment and its requirements became clear; robot-runner was built to automate the overhead of stopping and starting any experiment run. However, this experiment would require a new, completly charged battery and the manual reading and formatting of the SD card. Only if we would have been able to develop a second robot, which could automate these things, we would have to do these things manually.

Therefore, robot-runner; a framework solely focused on automating the execution of the experiment by starting a new run when another one ended, did not make sense to use anymore. On top of this, the robot-runner framework was thus no longer suited for this experiment as the code overhead, normally used for a succesful automatic end and start of runs, was no longer needed and now a cumbersome, possibly result-influencing, cumbersome risk.

It was this reason that lead to the decision to develop a new, more lightweight, framework; mission-runner, in conjunction with turtlebot-runner. Both with the sole goal to create code as reusable and lightweight as possible, with the assumption that a Human, manually orchestrating the process, would ensure things like the availability of the correct ROS nodes before starting the mission. This saved valuable lines of code which would check for this, as present in the robot-runner framework, saving not only time, but also the risk of influencing the results of the experiments as a result of running code which was not useful, nor necessary to the experiment and reduced the share of the researched tactic in the used codebase.

Next to this, the danger of the experimental robot-runner framework messing up a substantial amount of runs was present and rather avoided than engaged with the limited timeframe of this technical report in mind.

Hence, the resulting mission- and turtlebot-runner frameworks developed for and presented in this technical report.

3.4.2 Missions in Code

To make clear what the missions look like in code, two files (*F_Combined.py* and *F_IMission.py*) are given here as examples and any mentionable aspects are considered here:

3.4.2.1 Mission Implementation Example

F_Combined.py:

```python
from F_movement.F_IMission import F_IMission
from common.tactics.ee1.movement.EE1MovementController import EE1MovementController
from common.tactics.combined.camera.CombinedCameraSensor import CombinedCameraSensor
```

```
4
5  class F_Combined(F_IMission):
6      def __init__(self):
7          super().__init__()
8          self.mvmnt_controller = EE1MovementController(self.ros_rate)
9          self.camera_controller = CombinedCameraSensor()
10
11     def do_mission(self) -> None:
12         self.do_mission_camera_recording_only_turns()
```

For F_Combined.py it can be noted that:

1. The MovementController and CameraController are defined for each misison by default in the IMissionController, inherited by each mission-specific file through the inheritance of the X_IMission.py file where *X* stands for the treatment of the *Movement* factor.

   (a) It can be observed that for the *Combined* mission to be defined, only the default controllers for the MovementController and CameraController need to be overwritten to the required controllers for that tactic to be implemented, as seen on lines 8 and 9.

2. All X_IMission.py files have the abstract method *do_mission()*, which needs to be overridden, but also provides two versions of the baseline mission (the baseline mission is alays executed, that what differs according to the tactic is only changed); *do_mission_camera_recording_only_turns()* and *do_mission_camera_recording_everything()*.

3.4.2.2 Tactic Implementations

In this subsection, the implementations of the tactic arbiters is explained (*i.e., EE1*, *EE2*, *EE3*, *EE4*, *Combined*). The tactic implementations involve two parts, a part running on the mission-runner and a part that is running on the turtlebot-runner.

Combined Tactic As example, the *Combined* tactic is chosen as it combines and implements all the individual tactics. In figure 3.12 the structure can communication lines can be observed between the two framework paths. The figure is briefly discussed here:

1. Communication lines are highly unusual in this kind of diagram, but essential for understanding the connectivity of the system. Just like it is important to understand

that one class inherits and uses the functions of another class, so too is it important to understand how the CombinedCameraSensor communicates with the actual component it represents on the TurtleBot3.

   (a) On the TurtleBot3, the tactic arbiters (*e.g., EE3ClientConfigurableSampleRate*, etc.) communicate an invocation of their provided ROS Services by invoking the *callback method* as provided at initialisation of the class.

2. ServiceProxies are different from Services as a Service is that which is registered, at a specific address, where the ServiceProxy connects to that service at that specific address. Allowing two-way, one-to-one, communication.

3. The ServiceCall text notations are an example of what could be transferred over the ROS Service connection. For the possible Remote Procedure Calls over the Service connection one can look at the provided Service channel addresses as given as attributes of the corresponding classes (*i.e., Service('/address')*).

4. **Note:** ROS only allows for the initialisation of *one* ROS node per OS process. This also means, that despawning the ROS node (destroying the node) to then initialise it again is impossible to do in the same process.

   (a) This hindered EE4 from implementation as it requires to destroy and spawn the node.

   (b) As a workaround, it can be observed that in turtlebot-runner, the EE4 tactic is represented by a controller, rather than an inheritable parent class. Now the Combined entity can use the EE4OnDemandController to register a ROS node, providing the necessary *Spawn* and *Despawn* Services. On invocation, the controller invokes the callback method set by the Combined entity, which then spawns the CameraController in a subprocess, or terminates that subprocess, and with that the ROS node, respectively.

3.4.2.3 Mission Variables

In this section all mission variables as declared in the code, and easily changeable as a result, are given so that the execution of the missions based on these values is clear. The values are given in table 3.4.

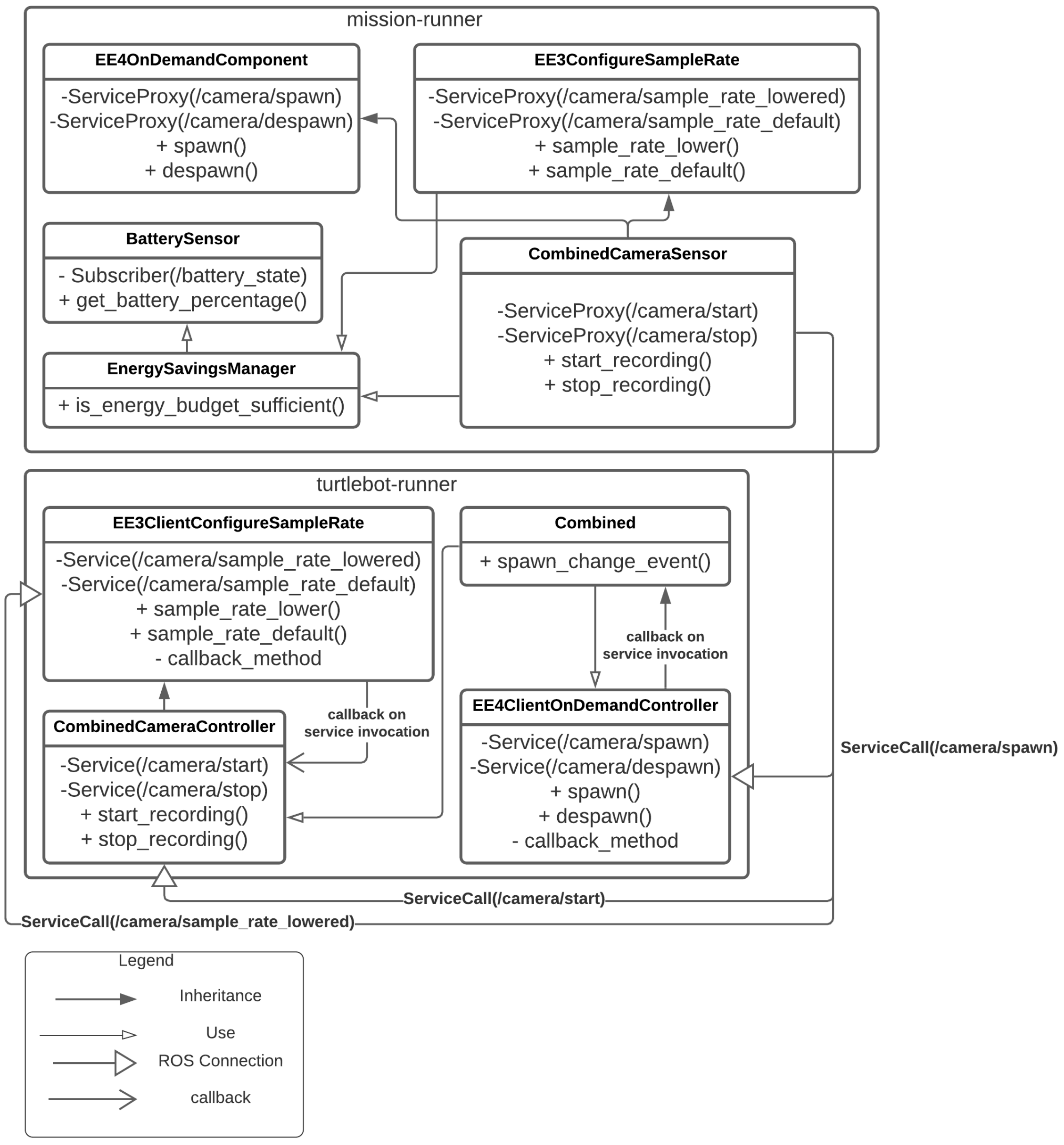

mission-runner
EE4OnDemandComponent
-ServiceProxy(/camera/spawn)
-ServiceProxy(/camera/despawn)
+ spawn()
+ despawn()
EE3ConfigureSampleRate
-ServiceProxy(/camera/sample_rate_lowered)
-ServiceProxy(/camera/sample_rate_default)
+ sample_rate_lower()
+ sample_rate_default()
BatterySensor
- Subscriber(/battery_state)
+ get_battery_percentage()
CombinedCameraSensor
-ServiceProxy(/camera/start)
-ServiceProxy(/camera/stop)
+ start_recording()
+ stop_recording()
EnergySavingsManager
+ is_energy_budget_sufficient()
turtlebot-runner
EE3ClientConfigureSampleRate
-Service(/camera/sample_rate_lowered)
-Service(/camera/sample_rate_default)
+ sample_rate_lower()
+ sample_rate_default()
- callback_method
Combined
+ spawn_change_event()
callback on service invocation
CombinedCameraController
-Service(/camera/start)
-Service(/camera/stop)
+ start_recording()
+ stop_recording()
callback on service invocation
EE4ClientOnDemandController
-Service(/camera/spawn)
-Service(/camera/despawn)
+ spawn()
+ despawn()
- callback_method
ServiceCall(/camera/spawn)
ServiceCall(/camera/start)
ServiceCall(/camera/sample_rate_lowered)
Legend
Inheritance
Use
ROS Connection
callback

Figure 3.12: The implementation of the Combined tactic, communicating with the turtlebot-runner part of the framework.

Table 3.3: Experiment Mission Variables.

Variable	*Value*	*In File*
rotation_interval_in_seconds	int = 20	IMissionController
minimal_turn_in_degrees	int = 60	IMissionController
maximal_turn_in_degrees	int = 140	IMissionController
forward_stopping_distance_threshold	float = 0.5	IMissionController
default_speed	float = 0.6	MovementController
rotation_base_speed	float = 0.8	MovementController
percentage_limit	float = 0.3	EE1LimitableMovement
battery_percentage_threshold	int = 100	EnergySavingsManager

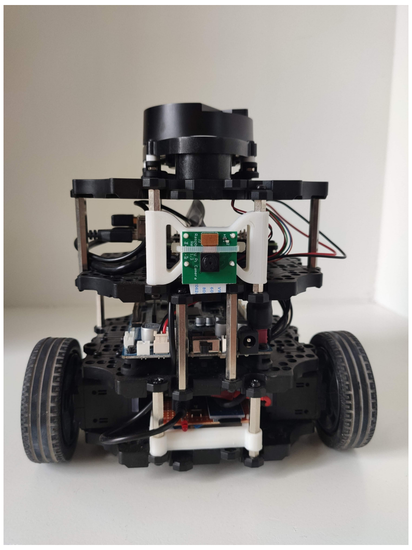


Figure 3.13: The TurtleBot3 as seen from the front with the camera and circuit mounted

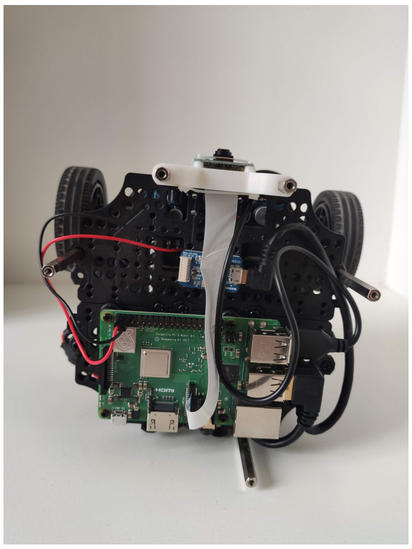


Figure 3.14: The camera mount and connection to the Raspberry Pi from close up.

3.5 Hardware Additions

For the experiments to be feasible, considering the need for an additional sensor compared to the TurtleBot3 out-of-the-box, some hardware additions were made. These are explaned here, pictures are shown and everything required to be able to understand and recreate the additions is given.

As an overview of the hardware additions, figures 3.13, 3.14, 3.15 and 3.16 depict pictures taken of the robot as it was used in the execution of the experiments.

3.5.1 Energy Measuring Circuit - Schematic

Considering that the TurtleBot3 topic published; */battery_state* provides very rough data in the form of the current percentage of charge hold by the battery, the need for a circuit that allows for fine-grain measurements of the consumed energy at any given point arose.

After research and discussion, it was found that the INA219[1] with its detailed documentation[2] would be perfect for the job.

This had the following reasons:

1. Adafruit is a respected supplier of TinyTronics (*Tiny Electronics*) such as microcontrollers, sensors, etc.

[1]https://www.adafruit.com/product/904
[2]https://cdn-learn.adafruit.com/downloads/pdf/adafruit-ina219-current-sensor-breakout.pdf

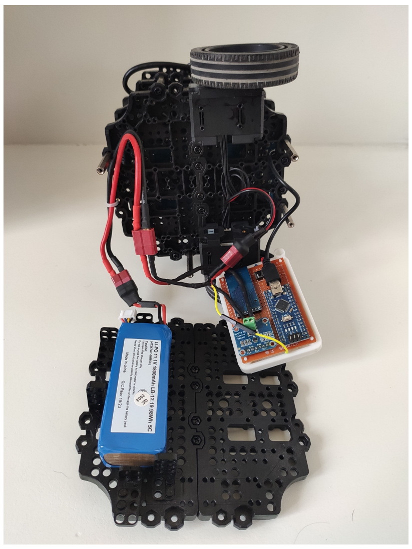


Figure 3.15: The circuit mount and connection with the battery and robot from close up.

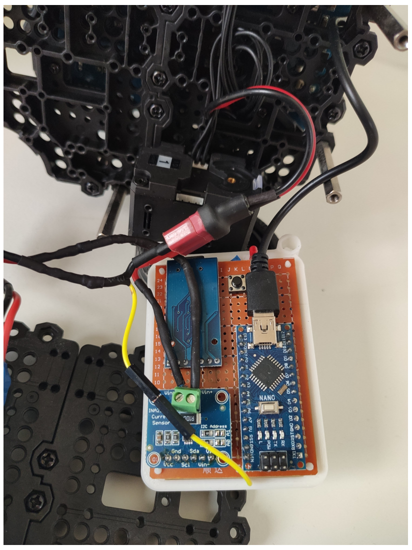


Figure 3.16: The circuit from close up.

2. The INA 219 is a respected, de-facto board for energy measurements in the IoT (*Internet of Things*) and TinyTronics movements.

3. The INA 219 met the requirements of the project perfectly;

   (a) The INA 219 has 1% precision in measurement as reported in its documentation; *"you can use this breakout to measure both the high side voltage and DC current draw over I2C with 1% precision"*.

      i. Vital for the validity of this technical report and the results of the empirical evaluation.

   (b) The INA 219 was built to gather readings precisely in the, for this technical report, required range (up to 26VDC, 3.2A).

   (c) The INA 219 could be powered by 5V or 3.3V and is specifically designed to be used with microcontrollers. This allowed the creation of a simple but effective circuit, **and** guaranteed that the circuit would have minimal impact on the energy consumption drawn from the battery.

   (d) The INA 219 can measure Voltage, Amperes and power in milli-Watts simultaneously and at incredible speeds (*up to 400kHz*[1]).

[1]As said however, this speed was unachievable as the SD card formed the bottleneck.

Now that the circuit for measuring the energy consumption was picked, complementary hardware was required to actually be able to use the INA 219. For this the following combination of hardware has been chosen:

1. The Arduino NANO
   (a) Specifically chosen for its small power signature, smaller than the Arduino Uno, and to still offer enough computing power and IO pins necessary for a successful circuit.
2. A Generic TinyTronics SD Card read/write module.
   (a) This did not have to be anything fancy, as long as it was small, had a low power signature and was able to do the job quickly and effectively.
3. A standard-issue LED
   (a) This functions as a status LED to show if the energy measurements are being written to the SD card and thus if it is safe to remove the SD card.
4. A standard-issue button
   (a) This functions as a START/STOP button, which controls if the measurements are being written to the SD card or not.
5. Battery leads (T Plug Connector) for connecting the battery to the circuit and the robot to the circuit.

The schematic was created, drawn and realised by soldering the circuit to a PCB; this schematic is given in figure 3.17.

The code created and used for the circuit is given in the replication package of this technical report. For mounting this circuit to the TurtleBot3, a custom 3D print has been designed and printed with a Creality CR10S 3D printer. A picture showcasing the mount is shown in figure 3.18. The *.stl* file for this 3D print is also given in the replication package.

3.5.2 Raspberry Pi Camera Module

As mentioned before, the TurtleBot3 required an additional sensor for the experiment design to be able to be executed on it. For this, a standard issue Raspberry Pi Camera

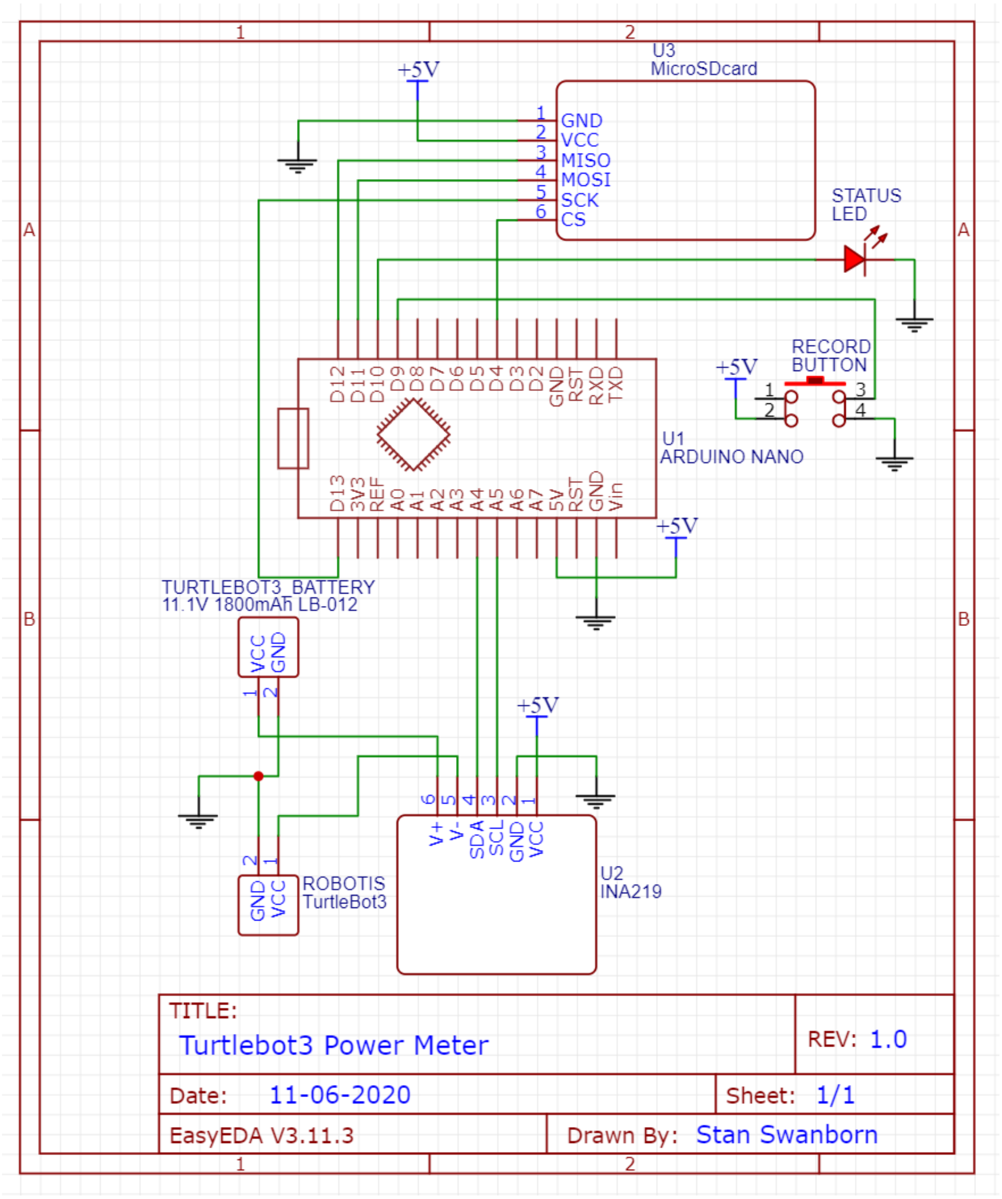


Figure 3.17: The schematic of the power meter circuit.

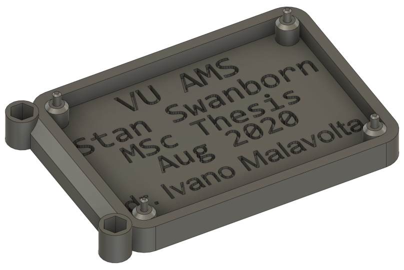


Figure 3.18: The circuit mount as designed and 3D printed.

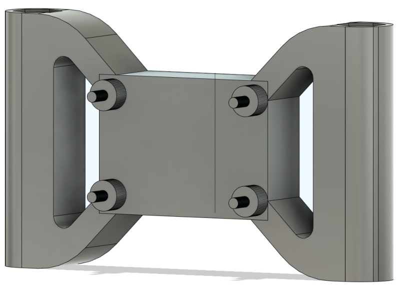


Figure 3.19: The camera mount as designed and 3D printed.

Module V1.3 [1] has been chosen. It also has excellent documentation [2] and most importantly; it is extensively programmatically controllable with the provided Python library **picamera**[3].

The Raspberry Pi Camera Module was chosen specifically for these reasons:

1. It uses quite a lot of power compared to any other TinyTronics sensors (*350mA*[4])
   (a) This was of importance, as the tactics would be manipulating this sensor and thus its power usage; clear results in the form of bigger differences are desirable for a valid conclusion and good, conclusive data analysis.
2. The Raspberry Pi Model 3B+ on the TurtleBot3 has a dedicated port for such a camera and was thus easily installed.
3. It is extensively programmatically controllable.
4. It has excellent documentation and widespread use.
5. It is inexpensive.

For this hardware addition a 3D printed mount was also designed and printed and is shown in figure 3.19.

[1] https://www.tinytronics.nl/shop/nl/raspberry-pi/raspberry-pi-compatible-camera-5mp-v1.3
[2] https://www.raspberrypi.org/documentation/hardware/camera/
[3] https://picamera.readthedocs.io/en/release-1.13/
[4] https://tinyurl.com/yy7fux9z

3.5.3 TurtleBot3 Battery Specifications

Table 3.4: TurlteBot3 LiPo Battery Specifications.

Variable	*Value*
Voltage	11.1V
Capacity	1800mAh = 1.8Ah
Volt-Amp-Hours	$11.1V \cdot 1.8Ah = 19.98VoltAmpHours$
Joules (J)	$19.98VoltAmpHours \cdot 3600J = 71.928J$

Volt-Amp-Hours are of course the equivalent of its more prominent term: Watt-Hours (Wh). As 1 Watt = 1 Joule per second it follows that 1 Watt-Hour = 3600 Joules, as there are 3600 seconds in 1 hour. The total capacity of the battery, in Joules, is thus; **71.928J**.

3.6 Empirically Evaluating the Green Tactics (RQ2)

Data analysis – Firstly, we explore the collected energy measures via violin plots and summary statistics. Then, we analyze the distribution of the energy measures in order to check if a parametric test (*e.g.,* the one-way ANOVA) can be applied, which can potentially lead to higher statistical power w.r.t. non-parametric tests (25). However, a visual analysis of Q-Q Plots and the application of the Shapiro-Wilks test (29) with $\alpha = 0.05$ reveal that the energy measures across tactics are not normally distributed. Even after applying several data transformations (*e.g.,* squared, reciprocal, log (30, 31)), energy measures are still not normally distributed.

We apply the Kruskal-Wallis test (with $\alpha = 0.05$), a rank-based non-parametric test for testing whether two or more samples all come from identical populations (32). In the context of our study, we use the Kruskal-Wallis to determine if there are statistically significant differences of energy consumption for every treatment of the *tactic* variable. The *magnitude* of the difference of energy consumption is estimated via the Eta-squared statistic and interpreted according to (33).

In order to identify which tactics lead to significantly different energy consumption, we perform a pairwise comparison between each tactic and the baseline using the Wilcoxon test (34) with Benjamini-Hochberg correction (35). The comparison is carried out both globally and across all possible combinations of movement strategy and physical environment.

We assess the *magnitude* of the difference of each tactic via the Cliff's Delta effect size measure (36). The values of the Cliff Delta measures are interpreted according to (37).

4

Results

4.1 Tactics for Energy-Efficient Robotics Software (RQ1)

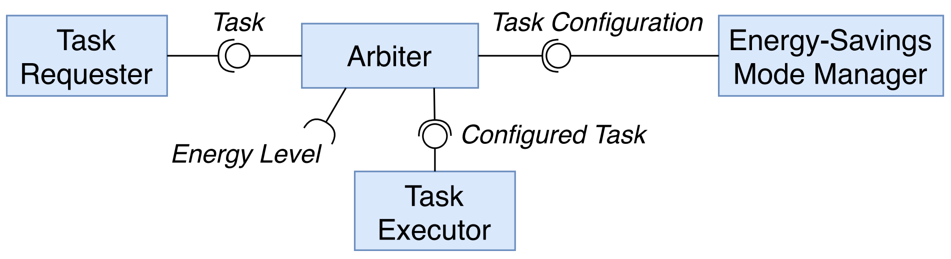
The four green tactics identified in this study share the common goal of saving the energy consumed by a robot. Each tactic description below includes the motivation for using the tactic, a component-and-connector (C&C) view that shows the main components of the tactic (see Figure 4.1), a description of the tactic based on the C&C view, and an example of how it is used in one of the data points considered in this study.

EE1: Limit Task – There are robot tasks that consume a significant amount of energy (*e.g.,* streaming large videos or robot navigation). Therefore, limiting these tasks when a robot reaches a critical energy level is important for extending the time that a robot is operational. One way to limit execution of energy-hungry tasks is to place the robot in energy-savings mode once the energy level reaches an established threshold. For each of these tasks there is a default mode and an energy-savings mode, as shown in the examples in Table 4.1.

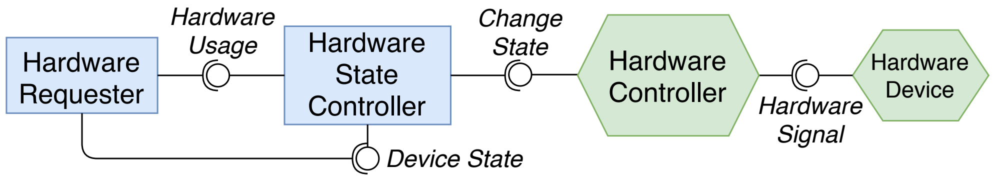
Table 4.1: Default and Energy-Saving Mode for Robot Tasks

Default Mode	**Energy Savings Mode**
Move in any direction at the max power rate	Adjust power rate to 50% of set max power rate
Publish any type of data	Do not publish PCL point clouds
Send video stream to the operator display	Do not send any video streams

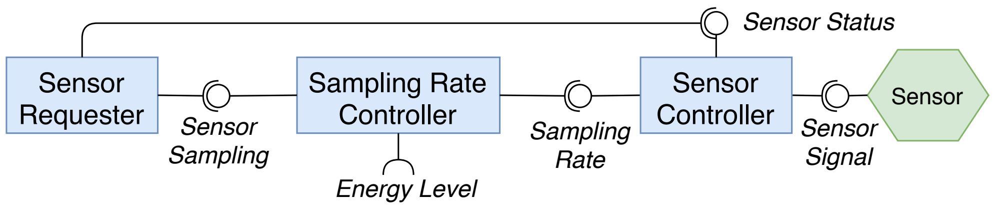
The *Limit Task* tactic configures a robot's task to execute in energy-savings mode when energy levels reach an established threshold (see Figure 4.1(a)). The *Task Requester* is responsible for requesting to execute a task, the *Arbiter* decides whether to execute the task in default mode or energy-savings mode, the *Energy-Savings Mode Manager* provides



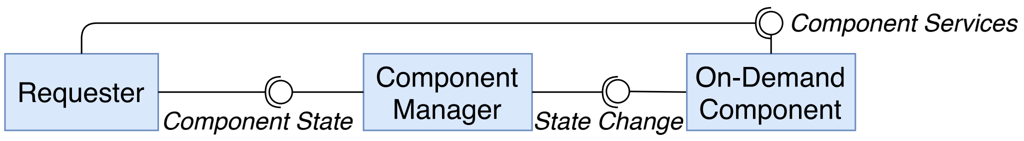
(a) *Limit Task* tactic (EE1)



(b) *Disable Hardware* tactic (EE2)



(c) *Energy-Aware Sampling* tactic (EE3)



(d) *On-Demand Components* tactic (EE4)

Figure 4.1: C&C view for the identified green tactics

the task configuration for energy-savings mode, and the *Task Executor* executes the task in either default or energy-savings mode, as such:

1. The *Task Requester* sends a task to the *Arbiter*.
2. After receiving the task, the *Arbiter* checks the energy level of the robot (provided by another component).
3. If energy level is below the established threshold, the *Arbiter* requests the energy-savings mode task configuration from the *Energy-Savings Mode Manager*.
4. The *Arbiter* forwards the received task to the *Task Executor* for execution.
5. The *Arbiter* continues checking the energy level during the execution of the task.
6. If the energy level is below the threshold, the *Arbiter* obtains the task configuration from the *Energy-Savings Mode Manager* and instructs the *Task Executor* to continue execution of the task in its energy-savings mode.
7. Similarly, if the energy level rises above the threshold, the *Arbiter* instructs the Task Executor to continue execution of the task in its default mode.
8. Once the task is completed the *Task Requester* is notified.

An example of the use of this tactic is a ROS-based system that uses haptic devices (data point 36). Haptic teleoperation allows a user to perform manipulation tasks in distant, scaled, hazardous, or inaccessible environments (38). In this system, the haptic device controller is a ROS node that communicates tasks to another ROS node which in turn controls the robot. When the energy of the robot arm reaches a critical level, the robot arm controller node adjusts the received task based on its configuration for energy-savings mode. In addition, because the haptic device controller is subscribed to an arm feedback topic, it can inform the user about the arm's battery state as an indication for why it is operating at a slower speed so that haptic user feedback can be adjusted.

EE2: Disable Hardware – Hardware components (*e.g.,* servos, drivetrains, manipulator arms) of a robot often consume a significant amount of energy. For example, in addition to the energy required to power a motor such as the Dynamixel XC430-W240 (39), the controller of the motor also consumes energy due to the CPU usage to process the produced data (*e.g.,* about its current velocity and temperature). It is therefore important to prevent unnecessary utilization of hardware resources in order to extend the operation time of the robot.

The *Disable Hardware* tactic disables hardware components when they are not strictly needed, which results in less energy consumption by the robot and more efficient power management (Figure 4.1(b)). The tactic is implemented to manage the state of the actual *Hardware Device*, as such:

1. The *Hardware Requester* notifies the *Hardware State Controller* whether or not the hardware device is needed for a certain task.

2. The *Hardware State Controller* instructs the *Hardware Controller* to disable or enable the *Hardware Device*.

3. Before enabling or disabling the *Hardware Device*, the *Hardware Controller* checks if it is safe to change the state of the HW device (*e.g.,* to toggle a hardware pin).

4. If it is safe, the Hardware Controller enables or disables the *Hardware Device*.

5. Note that it is also possible for the *Hardware Requester* to obtain the state of the *Hardware Device* at any time via the *Hardware State Controller*.

An example of the use of this tactic is the *ros_ control* package (40), one of the most used packages within the ROS ecosystem which includes ROS-based controller managers and controller-hardware interfaces (data point 23). The *controller_ manager* node advertises two services: a *load_ controller* service (enable hardware) and an *unload_ controller* service (disable hardware). If a node needs to enable the robot hardware, it sends a request to the *load_ controller* service. If it wishes to disable the robot hardware, it sends a request to the *unload_ controller* service. After receiving a request, the *controller_ manager* node performs the request by either enabling or disabling the robot hardware. The requesting node is notified with the result to ensure that it is aware of the robot hardware status.

EE3: Energy-Aware Sampling – In robotics, many sensors are designed to provide a continuous stream of data (*e.g.,* accelerometers, LIDARs, cameras) (41). However, sampling data from sensors is an energy consuming task, especially as incoming data is continuously processed (CPU usage).

The *Energy-Aware Sampling* tactic shown in Figure 4.1(c) adjusts the rates for sensor sampling based on the energy level of the robot, as such:

1. The *Sensor Requester* asks the *Sampling Rate Controller* to start sampling the *Sensor* at a given rate.

2. The *Sampling Rate Controller* instructs the *Sensor Controller* to start sampling the *Sensor* at the given rate and continues checking the energy level during the execution of the sampling task.

3. If the energy level reaches a critical threshold, the *Sampling Rate Controller* instructs the *Sensor Controller* to start sampling at a lower rate and informs the *Sensor Requester* of the adjusted sampling rate.

4. Note that it is also possible for the *Sensor Requester* to obtain sensor status at any time from the *Sensor Controller*.

An example of the use of this tactic is a ROS-based driver for InvenSense's 3-axis gyroscope (42) (data point 51). An MPU controller node subscribes to a *battery_state* topic to check the battery level and a *sampling_rates* topic to which sensor sampling rates are published by an MPU node. Based on battery levels, the MPU Controller adjusts sensor sampling rates accordingly by sending a request to the sampling action advertised by the MPU node that controls the actual sensor.

EE4: On-Demand Components – Continuously running a software component (*e.g.,* a ROS node) requires the spawning of an operating system (OS) process which is an energy-consuming task in terms of CPU/memory usage (*i.e.*, executing a CPU-intensive loop) and other resources (*e.g.,* sensors, motors, fans for cooling). Therefore, it is necessary to ensure that OS processes are not running if they are not needed.

The *On-Demand Components* tactic shown in Figure 4.1(d) starts new components only when their functionality is needed. The *Requester* represents a component that requires the functionality of the *On-Demand Component.* The *Component Manager* acts as a controller that either starts up or shuts down a component based on requests, as such:

1. The Requester indicates to the *Component Manager* that it needs the *On-Demand Component* to be in either the online or offline state.

2. The *Component Manager* starts up (online) or shuts down (offline) the *On-Demand Component* if the request is different from its current state.

3. The *Component Manager* notifies the *Requester* of the state of the *On-Demand Component.*

4. If the *On-Demand Component* is online, the *Requester* can start using its services.

5. Once the *Requester* no longer requires the functionality of the *On-Demand Component* it goes back to Step 1) to change its state to offline.

An example of the use of this tactic is the way that data point 14 uses cameras. In order for the camera to operate, it requires the *camera_driver* ROS nodelet to be up and running. The requester nodelet publishes the required state for the camera (online/offline) to a *camera_status* topic, which is subscribed to by the nodelet manager. Based on the published required status, the nodelet manager either starts up or shuts down the *camera_driver*. Once the *camera_driver* is up and running, it advertises a service that can be called by the requester.

4.2 Empirical Evaluation of the Tactics (RQ2)

We implement the green tactics into our Turtlebot as follows:

- EE1: limit the movement of the robot by waiting 5 seconds before each 360°rotation;
- EE2: disable the camera of the robot (*i.e.,* with no video acquisition) when the robot is moving among locations;
- EE3: lower the frame rate of the camera to 30 FPS;
- EE4: kill the ROS node of the camera when the robot is moving and bring it up before each 360°rotation.

As discussed in Section 3.6, our experiment also includes a *baseline* treatment where no tactics are applied and a *combined* treatment where all the tactics are applied simultaneously.

Data exploration. The energy consumption across all tactics ranges between 1067.08 and 1429.11 Joules (see Table 4.2), with a median (mean) of 1277.74 (1271.80) Joules.

Table 4.2: Descriptive statistics of the energy consumption in Joules (SD=standard deviation, CV=coefficient of variation)

Tactic	Min.	Max.	Median	Mean	SD	CV
Baseline (B)	1151.93	1416.81	1336.96	1318.11	60.92	4.62
EE1	1164.58	1386.37	1293.06	1291.62	51.70	4.00
EE2	1089.45	1369.67	1258.91	1255.11	62.44	4.97
EE3	1130.56	1429.11	1337.52	1313.29	72.78	5.54
EE4	1084.92	1321.92	1250.00	1239.13	63.67	5.14
Combined (C)	1067.08	1322.60	1225.36	1213.56	59.18	4.88
Global	1067.08	1429.11	1277.74	1271.80	72.77	5.72

The standard deviation of the collected energy measures is non negligible and ranges from 51.70 for the EE1 tactic to 72.78 for the EE3 tactic; overall, the values of the standard deviation are mainly due to the robot performing different movements during the execution of the mission and to the intrinsic fluctuation of energy and it justifies our design choice of repeating the runs of the experiment 10 times for each trial. Nevertheless, the coefficient of variation remains between 4% and 6%, making us reasonably confident about the reliability of the measurement infrastructure we setup for the experiment.

Result 1 – On *average* all green tactics improve energy efficiency, however not all tactics impact the energy consumption of the system with the same magnitude.

As shown in Figure 4.2, tactic EE4 is the one impacting energy the most, making the Turtlebot consume an average of 78.55 Joules less than the baseline treatment, followed by EE2 and EE1 with an average saving of 63.0 and 26.49 Joules, respectively. The EE3 tactic shows a slightly different behaviour; even though on average it saves 4.82 Joules, when it is applied the system tends to consume the same (or even more) energy w.r.t. the baseline (the median energy consumption of EE3 is 0.56 Joules *higher* w.r.t. the baseline). This result may seem surprising, however it can be explained by the way we implemented the EE3 tactic. Indeed, EE3 just changes the frame rate of the sensor to 30 FPS but the Turtlebot does not use the acquired video stream, *e.g.,* by persisting, manipulating, or streaming the recorded video. We decided to implement EE3 in this way so to completely isolate the application of the tactic from the business logic managing the data produced by the camera. In summary, in the specific context of our experiment the application of EE3 did not lead to energy savings (the is also statistically confirmed). This phenomenon is also confirmed in the mobile apps domain (43), where lowering the frame rate of a camera does not impact its energy consumption *per se*, rather energy is impacted the most by *how* the recorded video stream *is used* in other components of the system (*e.g.,* streaming the recorded video to the cloud).

These results are statistically confirmed, with the Kruskal-Wallis test producing a *p*-value of 2.74×10^{-18} (with *large* effect size), which allows us to reject the null hypothesis that the energy measures at each tactic come from identical populations (44). The pairwise comparison between each tactic and the baseline with the Wilcoxon test further confirm our results; the *p*-value for EE1, EE2, and EE4 is lower than 4×10^{-3}, thus rejecting the null hypothesis that the median difference between the baseline-EE1, baseline-EE2, and baseline-EE4 pairs is zero. We find a *medium* effect size for EE1 (0.34) and a *large* effect size for EE2 and EE4. These results provide evidence about the fact that the application of the

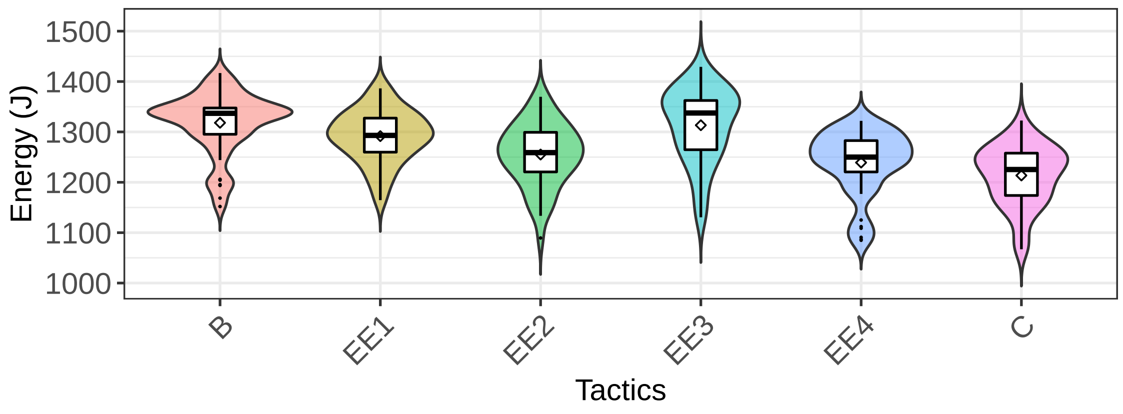


Figure 4.2: Energy consumption across architectural tactics (B=baseline, EEi=ith applied tactic, C=combined, $\diamond$=mean)

EE1, EE2, and EE4 tactics lead to a significantly different amount of energy consumption in the context of our experiment.

Result 2 – The combination of all green tactics improves energy efficiency more than each tactic in isolation.

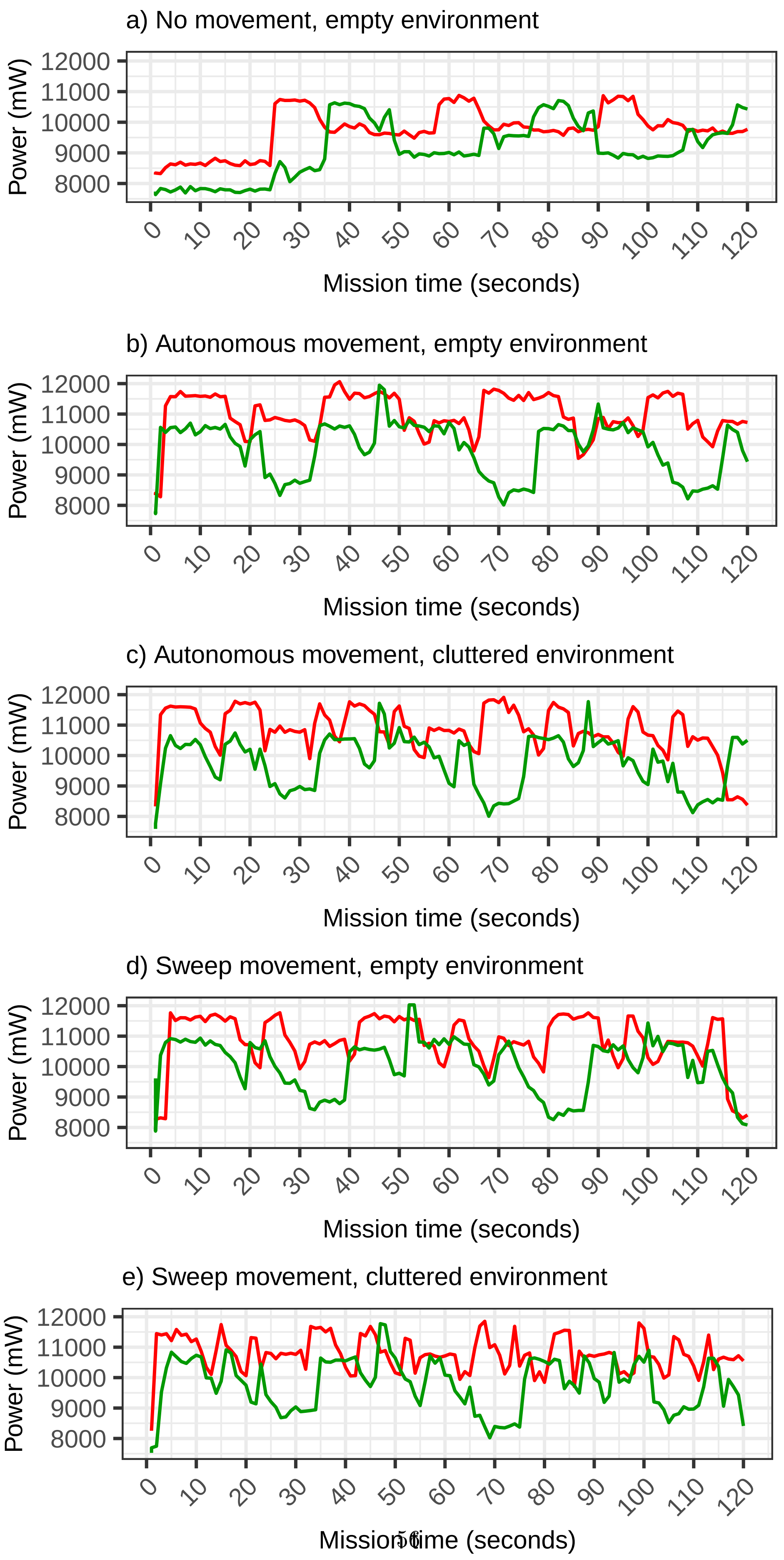
The median (mean) energy consumed by the Turtlebot with all combined tactics is 111.6 (104.55) Joules less than the baseline. This difference is far higher than those related to the individual tactics (see Table 4.2 and the right-most violin in Figure 4.2): the application of the tactics leads to a 7.9% energy saving on average. To put this result into perspective, considering that the total energy of the battery of the Turtlebot is 71928 Joules and that on average our 2-minute missions consume 1271.8 Joules, the total lifetime of a Turtlebot without tactics is about 109 minutes, whereas the application of the tactics would like to a total lifetime of about 119 minutes (a 10-minute improvement over a mission of less than 2 hours). The previously mentioned Wilcoxon tests statistically confirm this result with a p-value of 5.75×10^{-11} and the Cliff's delta measure reveal a *large* effect size (0.78).

Result 3 – The movement strategy and the physical environment influence how energy is consumed during the mission.

Figure 4.3 shows the power measurements collected during one randomly-chosen mission for each combination of movement strategy and physical environment. Among others, here we can clearly notice (i) the generally lower power consumption of the robot with the applied tactics with respect to the baseline, (ii) the low power consumption of the robot with the *noMovement* strategy in the first 20 seconds of the mission, where the robot still does not use at all the wheel actuators (Figure 4.3a), and (iii) the more chaotic power consumption in the *cluttered* environment due to the avoidance of the encountered obstacles (Figures 4.3c and 4.3e).

By looking at the combinations of tactic, movement strategy, and physical environment (see Figure 4.4), we can witness that different amounts of energy are consumed when the robot is moving. This result is expected since additional energy is consumed by the two actuators for rotating the wheels of the robot. More interestingly, we can also confirm the general results obtained when discussing results 1 and 2. Specifically, almost all tactic-movement-environment combinations lead to a statistically significant difference in terms of energy consumption, with the exception of EE3. Moreover, when the results are statistically significant, their effect size is always *large*. This gives evidence about the fact that applying tactics EE1, EE2, and EE4 (and their combination) likely leads to higher energy efficiency, with a *large* effect on it.

We have three other exceptions to the main trend, namely the *EE1-noMovement-empty*, *EE1-sweep-empty*, and *EE2-autonomous-cluttered* combinations. About the *EE1-noMovement-empty* combination, we speculate that it is due to the fact that movement is a fundamental component of EE1, thus having the robot standing still for the whole duration of the mission (the *noMovement* treatment) would have penalized the EE1 tactic. At the time of writing we do not have hard evidence for explaining the results about the other two combinations since they all involve different combinations of factors. Further analysis and replications of the experiment are already planned for clarifying this specific part of the study.


a) No movement, empty environment
b) Autonomous movement, empty environment
c) Autonomous movement, cluttered environment
d) Sweep movement, empty environment
e) Sweep movement, cluttered environment
Power (mW)
12000
11000
10000
9000
8000
0
10
20
30
40
50
60
70
80
90
100
110
120
Mission time (seconds)

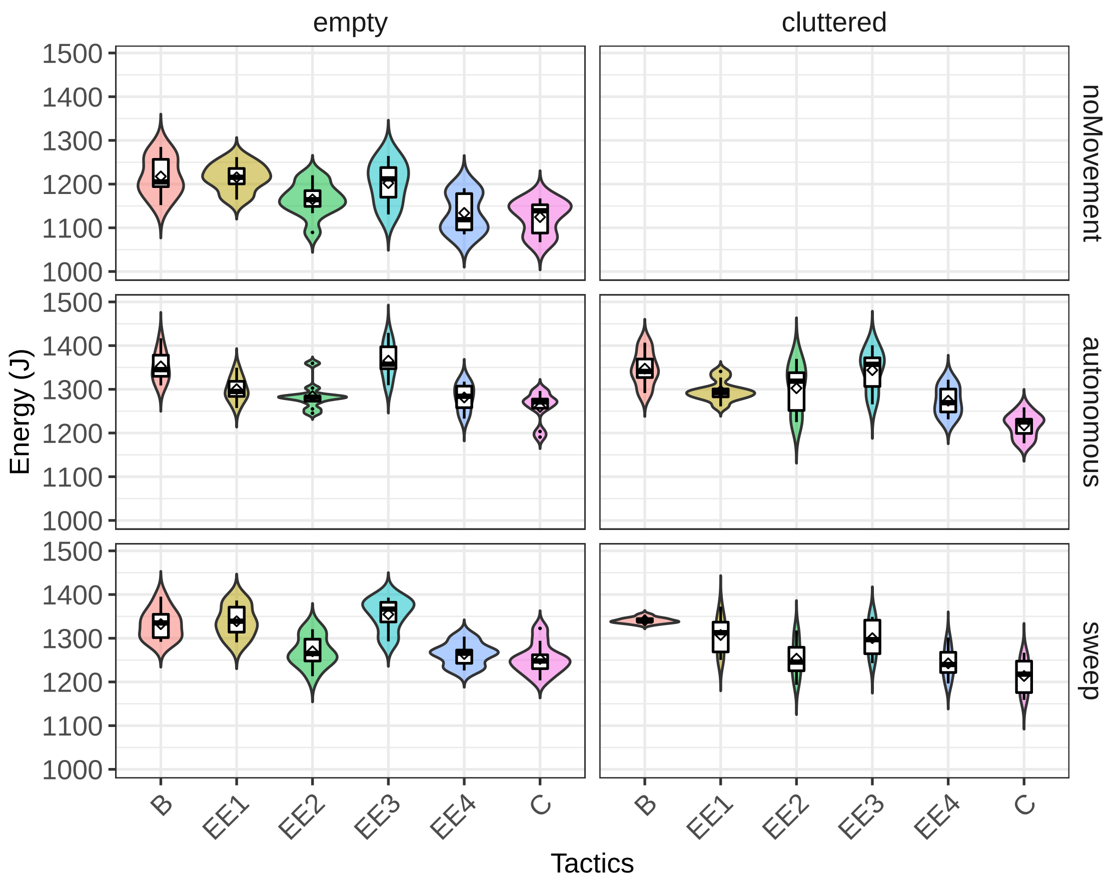


Figure 4.4: Energy consumption across all movements strategies and environments

4.3 Discussion

Energy is infrequently discussed by roboticists. Of the 339,563 ROS data points only 562 (0.17%) mention energy-related topics. This result is quite counter-intuitive, considering that (i) the majority of the discussed systems involve battery-powered mobile robots and (ii) the lifetime of current battery-operated robots is low. This phenomenon might also relate to the lack of testing/debugging tools for energy-efficient or at least energy-aware robotics software (*e.g.,* accessible energy measurement tools, libraries for energy-aware programming for robots); concern also confirmed in other domains, *e.g.,* mobile apps (45), where two main problems are developer awareness and lack of tools. We do not have hard evidence explaining whether the infrequence of energy-related data points is due to lack of awareness, tools, or simply interest by roboticists. A follow-up qualitative study might shed light on this phenomenon. Our answer to RQ2 empirically demonstrates that the software design choices of roboticists do impact the energy efficiency of robots, often with large effects. This should motivate researchers on energy efficiency to focus on robotics software, and developers to adopt the green tactics we identified and seek (and document) new ones.

Roboticists tend to not document architecture. The advantages of architecture documentation are widely reported (46), including facilitating the onboarding of newcomers to open-source projects, and being able to discuss/reason about tradeoffs between system-level quality attributes like energy and performance. However, none of the 97 analyzed data points include a documented architecture, *e.g.,* via a diagram or a thorough description of the involved components, connectors, and their configuration; finding also confirmed by another study on the architecture of ROS-based systems (18). We suggest roboticists to document the architecture of the (part of) system they want to discuss in order to better clarify their general points, design decisions, and rationale to the reader of their posts in discussion and social coding platforms and code/comments in their own source code.

There are other green tactics out there. The green tactics we identified are not meant to be complete. We designed our study so to let the green tactics *emerge* from the practice; there may be other sources for the tactics, like robotics textbooks, interviews with robotics experts, grey literature. Our choice is motivated by two main forces: (i) to empirically assess how and to which extent *practitioners* deal with energy efficiency at the architectural level, and (ii) to focus on tactics that are applied in real robots and and robotics contexts. In principle, the latter point makes the green tactics directly applicable in real projects, which, together with the empirical assessment in RQ2, should motivate practitioners to adopt them.

The green tactics should be refined and used in context. The tactics should be used and refined depending on the specific context of the system being developed. For example, the main component of tactic EE1 (*i.e.,* Limit Task) is the Arbiter, which decides whether to execute a task in default or energy-saving mode. However, there are many different types of tasks (*e.g.,* paint a wall, drive to a point of interest) and many definitions of modes depending on the specific robot at hand (see Table 4.1). Thus, roboticists must understand the context-sensitive tradeoffs implied by the system under development and apply the tactics accordingly. This observation is specially true when considering *performance* and *maintainability* since the presented tactics can involve having additional tactic-specific components/roles (*e.g.,* the Arbiter in EE1, the Component Manager in EE4), potentially leading to (i) communication and computation overhead and (ii) higher complexity of the system, thus hindering future improvements over time.

Know the Physics of your robot. One of the lessons learned during the execution of our experiment is that sensors and actuators might behave in counter-intuitive ways from the perspective of software developers. For example, the first implementation of tactic EE1 consisted in limiting the robot to 30% of its nominal speed; the intuition was that slower robots would make less "work" than faster robots, thus saving energy. Some pilot runs showed that this assumption was completely wrong. Indeed, the electric motors for rotating the wheel of the Turtlebot actually was consuming more energy at slower speeds! This is mainly due to the fact that the majority of the input power at slower speeds was used to overcome the dynamic friction inside the motors and as the speed was increasing, friction played a smaller and smaller role in their overall efficiency (39, 47). We suggest researchers and roboticists to have the energy-related behaviour of their robots under control by (i) carefully studying the technical specifications of all hardware components of the robots and, based on that, (ii) benchmarking the energy consumption of their robots under different conditions and configurations.

5

Threats to Validity

The threats to the validity of the results of this empirical evaluation are listed here.

5.1 External Validity

This threat is introduced by part (i) of this two-part research-effort:

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.

Considering this empirical evaluation is based on the tactics extracted from part (i), it suffers from the same External Validity threats and thus its mitigation is also the same.

This empirical evaluation's external validity is threatened by the fact that it might not be representative for cases where another robot than the ROBOTIS TurtleBot3 is used. This threat is mitigated by the use of ROS, which serves as the sole base for the framework and the implementation of the tactics; the robotic system it is deployed on is therefore not of interest. This empirical evaluation is thus valid for any ROS-based robotic system.

This empirical evaluation's external validity is threatened by the fact that it might not be performed in a representative environment. This threat is mitigated as the robotic arena constructed is constructed in such a fasion that it is comparable to robotic arena's widely used in robotic research. Furthermore, the robotic arena was static, and did not change and was not altered during the execution of the experiments. The robotic arena served as a controlled environment, in which any events that would threaten the validity of the results were controlled.

This empirival evaluation's external validity is threatened by the fact that it might not perform a representative set of missions. This threat was mitigated by first extensively reading into scientific literature on robotic research, applications of robotic systems and taking the mined datapoints as part (i) of this study into account. Based on this set of data, the missions were rigorously and iteratively constructed.

5.2 Internal Validity

This threat has been mitigated as much as possible by defining the experiment, as explained in section 3, as rigorously as possible. Iteratively defining it by discussing it after each iteration.

This empirical evaluation's internal validity is threatened by the fact that the tactics might not be correctly implemented. This threat is mitigated by the fact that the tactics were implemented according to well defined sequence diagrams which had to be followed to the letter. Any changes made, as given in section 3.4, were to be discussed. The tactic implementations were also rigorously and iteratively defined, going through two iterations.

This empirical evaluation's internal validity is threatened by the fact that the measurements might not be correct. This threat is mitigated by the choice for the AdaFruit INA 219; a de-facto standard in energy measurement for robotics and TinyTronics.

This empirical evaluation's internal validity is threatened by the fact that all the other software involved in the experiment might influence the results. This threat was mitigated by reducing the amount of software involved; apart from the developed framework and the tactics, the only software used is the standard, de-facto ROS 'bringup' mission from ROBOTIS which allows control over the TurtleBot3. This is developed by ROBOTIS themselves and can be considered rigorous and well tested. The other software used is the AdaFruit library to gather the metrics from the INA 219 in the power meter circuitry. For this software, the standard, de-facto AdaFruit INA 219 library was used, which serves the metrics as single float values which can then be directly written to the SD card; for which the standard Arduino SD card library was used. Mitigating the threat of any 'faulty', 'wrong' or 'experimental' software interfering negatively with the results of this evaluation.

5.3 Construct Validity

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 experiment design was carefully designed and carried out;

as part of part(i): 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.

For the empirical evaluation, the following threats were mitigated:

Inadequate preoperational explication of constructs

This sub-threat deals with constructs not being well defined before being translated into measures. To mitigate this sub-threat, the experiment was defined using the GQM method, meaning that the goal, the underlying questions and the metrics required to answer these questions were defined in a cascading fashion.

Mono-method bias

This sub-threat occurs when there is only a single type of measurement or observation used.

For this experiment the de-facto energy sensor for TinyTronic; AdafruitINA 219 was used to measure the metrics. This has the potential to introduce the mono-method bias into the experiment. This threat is mitigated by the fact that the readings from the INA 219 have been retrieved redundantly, meaning; not only was the power in milli-Watts measured, but also the values consituting that value, in order to be able to automatically validate the data.

Additionally, the CPU usage, RAM usage and Battery Percentage as reported by the TurtleBot3 have been measured by a seperate system; the 'Remote PC' controlling the experiments.

5.4 Conclusion Validity

The threat as introduced by part (i) of this two-part research-effort was mitigated:

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.

For this empirical evaluation, the threat has been mitigated by extensive measures in the experiment setup;

(i) The screwed-in-place arena which is unable to alter the physical setting in which the experiments are performed. (ii) The marked start point in which the robot is to be placed before each mission is started, guaranteeing the same starting position for each run. (iii) The marked positions of the obstacles, guaranteeing the obstacles are always in the **exact same position** for all runs.

The measures above describe the validity of the retrieval of the metrics. However, another threat to this empirical evaluation's conclusion validity needs to be addressed; the correctness of the statistical analysis.

The statistical analysis has been conducted according to well-accepted methodological guidelines for statistical analysis.

6

Conclusion

Based on an extensive mining of the ROS software ecosystem, in this paper we identify and empirically evaluate a first body of architectural tactics for energy-efficient robotics software. The results show that (i) the green tactics significantly help improve the energy efficiency of the robot and (ii) context and SW-HW interplay play an important role for their most-effective selection. Given the surprising lack of focus of the studied roboticists on energy-related topics, this green-tactics body of knowledge can help roboticists start changing their practice by (i) adopting these green tactics and (ii) becoming aware of the benefit of documenting and communicating their architecture design decisions, possibly leading to a new (energy-aware) development mindset.

References

[1] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. **Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing**. *Future generation computer systems*, **28**(5):755–768, 2012. 1

[2] Meiyappan Nagappan and Emad Shihab. **Future trends in software engineering research for mobile apps**. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, **5**, pages 21–32. IEEE, 2016. 1

[3] Roy Paulissen, Sandy Kalisingh, Jesse Scholtes, Alex van Geldrop, and Anne-Lize Hoftijzer. **Robotics in the Netherlands**. *Netherlands Foreign Investment Agency (NFIA) report*, 2016. 1

[4] H Lasi, P Fettke, and H Kemper. **Industry 4.0**. *Bus Inf Syst Eng 6*, pages 239–242, 2014. 1

[5] International Energy Agency. Office of Energy Technology and R&D. and Group of Eight (Organization). *Energy technology perspectives.* International Energy Agency, 2006. 1

[6] US Energy Information Administration. **Manufacturing energy consumption survey (MECS)**. 2018. 1

[7] A Fysikopoulos, D Anagnostakis, K Salonitis, and G Chryssolouris. **An empirical study of the energy consumption in automotive assembly**. *Procedia CIRP 3*, pages 477–482, 2012. 1

[8] IFR Statistical Department. **Executive summary of World Robotics**. 2010. 1

[9] J Engelmann. **Methoden und Werkzeuge zur Planung und Gestaltung energieeffizienter Fabriken**. 2009. 1

[10] J Evans. **An Autonomous Mobile Robot Courier for Hospitals**. *IROS*, pages 1695–1700, 1994. 1

[11] R Aylett. **Robots: Bringing Intelligent Machines To Life**. 2002. 2

[12] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012. 2, 4

[13] **PLATFORM - TurtleBot 3 - ROBOTIS**, Jul 2020. [Online; accessed 30. Jul. 2020]. 2

[14] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. **ROS: an open-source Robot Operating System**. In *ICRA workshop on open source software*, **3**, page 5. Kobe, Japan, 2009. 3

[15] **ROS Community Metrics**. http://wiki.ros.org/Metrics, Jul 2020. [Online; accessed 28. Jul. 2020]. 3

[16] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. **The Robot Operating System: Package reuse and community dynamics**. *Journal of Systems and Software*, **151**:226–242, 2019. 3

[17] F Bachman, L Bass, and M Klein. **Deriving architectural tactics: A step toward methodical architectural design**. 2003. 4

[18] Ivano Malavolta, Grace Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. **How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems**. In *ACM/IEEE International Conference on Software Engineering*, 2020. 5, 7, 58

[19] Shaiful Alam Chowdhury and Abram Hindle. **Characterizing energy-aware software projects: Are they different?** In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 508–511, 2016. 6

[20] Haroon Malik, Peng Zhao, and Michael Godfrey. **Going green: An exploratory analysis of energy-related questions**. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 418–421. IEEE, 2015. 6

[21] Irineu Moura, Gustavo Pinto, Felipe Ebert, and Fernando Castor. **Mining Energy-Aware Commits**. *Working Conference on Mining Software Repositories*, 2015. 6

[22] Gustavo Pinto, Fernando Castor, and Yu David Liu. **Mining questions about software energy consumption**. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31, 2014. 6

[23] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. **What do programmers know about software energy consumption?** *IEEE Software*, **33**(3):83–89, 2015. 6

[24] ISO. *ISO/IEC/IEEE 42010, Systems and software engineering — Architecture description*, December 2011. 7

[25] C Wohlin, P Runeson, M Höst, M.C. Ohlsson, B Regnell, and A Wesslén. **Experimentation in software engineering**. 2012. 7, 17, 46

[26] D. S. Cruzes and T. Dyba. **Recommended Steps for Thematic Synthesis in Software Engineering**. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 275–284, Sep. 2011. 7

[27] Faheem Ullah and Muhammad Ali Babar. **Architectural Tactics for Big Data Cybersecurity Analytics Systems: A Review**. *Journal of Systems and Software*, **151**:81–118, 2019. 7

[28] Grace Lewis and Patricia Lago. **Architectural tactics for cyber-foraging: Results of a systematic literature review**. *Journal of Systems and Software*, **107**:158–186, 2015. 7

[29] S.S. Shapiro and M.B. Wilk. **An analysis of variance test for normality (complete samples)**. *Biometrika*, pages 591–611, 1965. 46

[30] Sira Vegas. **Analyzing software engineering experiments: everything you always wanted to know but were afraid to ask**. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 534–535, 2018. 46

[31] Ryan A. Peterson. **Using the bestNormalize Package**, Jun 2020. [Online; accessed 27. Aug. 2020]. 46

[32] William H Kruskal and W Allen Wallis. **Use of ranks in one-criterion variance analysis**. *Journal of the American statistical Association*, **47**(260):583–621, 1952. 46

[33] Maciej Tomczak and Ewa Tomczak. **The need to report effect size estimates revisited. An overview of some recommended measures of effect size**. 2014. 46

[34] Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*, **751**. John Wiley & Sons, 2013. 46

[35] David Thissen, Lynne Steinberg, and Daniel Kuang. **Quick and easy implementation of the Benjamini-Hochberg procedure for controlling the false positive rate in multiple comparisons**. *Journal of educational and behavioral statistics*, **27**(1):77–83, 2002. 46

[36] Norman Cliff. **Dominance statistics: Ordinal analyses to answer ordinal questions.** *Psychological bulletin*, **114**(3):494, 1993. 46

[37] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005. 46

[38] Paulo Rezeck, Bruna Frade, Jessica Soares, Luan Pinto, Felipe Cadar, Hector Azpurua, Douglas G Macharet, Luiz Chaimowicz, Gustavo Freitas, and Mario FM Campos. **Framework for Haptic Teleoperation of a Remote Robotic Arm Device**. In *2018 Latin American Robotic Symposium, 2018 Brazilian Symposium on Robotics (SBR) and 2018 Workshop on Robotics in Education (WRE)*, pages 170–175. IEEE, 2018. 49

[39] ROBOTIS. **Dynamixel XC430-W240 manual**. https://emanual.robotis.com/docs/en/dxl/x/xc430-w240, Aug 2020. [Online; accessed 20. Aug. 2020]. 49, 59

[40] **ros_control - ROS Wiki**. http://wiki.ros.org/ros_control, Aug 2020. [Online; accessed 20. Aug. 2020]. 50

[41] Michael Beetz. *Plan-based control of robotic agents: improving the capabilities of autonomous robots*, **2554**. Springer Science & Business Media, 2002. 50

[42] **3-Axis | TDK**. https://invensense.tdk.com/products/motion-tracking/3-axis, Aug 2020. [Online; accessed 20. Aug. 2020]. 51

[43] Swaminathan Vasanth Rajaraman, Matti Siekkinen, and Mohammad A Hoque. **Energy consumption anatomy of live video streaming from a smartphone**. In *2014 IEEE 25th Annual International Symposium on Personal, Indoor, and Mobile Radio Communication (PIMRC)*, pages 2013–2017. IEEE, 2014. 53

[44] Michael Sullivan and JCM Verhoosel. *Statistics: Informed decisions using data*. Pearson Boston, MA, 2013. 53

[45] Gustavo Pinto and Fernando Castor. **Energy efficiency: a new concern for application software developers**. *Communications of the ACM*, **60**(12):68–75, 2017. 58

[46] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002. 58

[47] **Why is an electric motor more efficient at higher loads?** https://physics.stackexchange.com/questions/46113/why-is-an-electric-motor-more-efficient-at-higher-loads, Aug 2020. [Online; accessed 27. Aug. 2020]. 59