Energy Efficiency of ROS Nodes in Different Languages: Publisher–Subscriber Case Studies

Michel Albonico, Paulo Júnior Varela, Adair José Rohling
Federal University of Technology, Paraná (UTFPR)
Francisco Beltrão, Brazil
michelalbonico,paulovarela,adairrohling@utfpr.edu.br

Andreas Wortmann
Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW)
University of Stuttgart, Germany
andreas.wortmann@isw.uni-stuttgart.de

ABSTRACT

The Robot Operating System (ROS) is a de facto standard for programming robotic systems. It currently provides well-established client libraries for two major languages: C++ and Python. Different programming languages are known for their different abstraction levels, and as a consequence, their resource usage, including energy consumption. With energy efficiency being recurrently a quality requirement, it is important to understand how programming in those two languages may affect the energy consumption of robotic systems. In this study, we analyze the impact on energy consumption when programming ROS nodes in those two main supported languages. We design and conduct an empirical experiment on ROS 2 nodes implemented in C++ and Python for simple and well-documented topic-based examples. We statistically assess to what extent energy consumption is affected by language choice, where nodes programmed in C++ presented a consistently better energy efficiency. A deeper analysis of the measured variables indicates that the energy efficiency difference between the two client libraries is closely related to the underlying architecture.

CCS CONCEPTS

• **Software and its engineering** → **Software design tradeoffs**; • **Computer systems organization** → *Robotics*; • **Hardware** → *Impact on the environment*.

KEYWORDS

ROS, Programming Languages, Energy Efficiency, Robotic, Green Software

ACM Reference Format:
Michel Albonico, Paulo Júnior Varela, Adair José Rohling and Andreas Wortmann. 2024. Energy Efficiency of ROS Nodes in Different Languages: Publisher–Subscriber Case Studies. In *2024 ACM/IEEE 6th International Workshop on Robotics Software Engineering (RoSE '24), April 15, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3643663.3643963

RoSE '24, April 15, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0566-3/24/04…$15.00
https://doi.org/10.1145/3643663.3643963

1 INTRODUCTION

Robots play an important role in many areas of our society. They are commonly used in the manufacturing industry, transportation (including self-driving vehicles), and as domestic allies (e.g. vacuum cleaners) [5]. A great part of those robots depends on increasingly complex software, where the Robotic Operating System(ROS) [6, 21] appears as a key contributor to writing complex robotic systems.

ROS is considered the de-facto standard for robotic systems in both, research and industry [13]. It provides an abstraction layer that enables specialists from different areas to integrate their software into one robotic system. In addition, ROS comprises a comprehensive set of open-source libraries and packages. With over half a billion ROS packages downloaded in 2020, it has also encouraged code reuse [21]. ROS has two main versions, ROS 1 and ROS 2, with ROS 1 end-of-life (EOL) programmed for 2025. In this paper, we focus only on ROS 2, the only supported distribution in the near future, using ROS as nomenclature.

Software energy efficiency has been a recurrent concern among software developers [19]. This is stimulated by factors that include environmental impact, budget, and battery-dependent devices [6, 15, 22], which also applies to the robotic domain. Simple software architectural decisions can make an impact on the energy efficiency of robotic software [4], where the programming language is known to be a determinant factor [17]. In the case of ROS, C++ and Python are the two main programming languages thoroughly supported and documented by the community. Therefore, practitioners usually start by choosing one of them, which must be done without further understanding their impact on ROS systems' energy efficiency.

In **this paper**, we systematically investigate the energy consumption and power consumption of ROS nodes implemented in both languages, C++ and Python. We choose to address only well-documented ROS algorithms, which provide concrete examples on ROS tutorials Wiki page[1]. As a preliminary study, we only investigate ROS communication using topics, the most common ROS communication paradigm in recent studies [2]. All the algorithms are rigorously revised to keep their complexity and package dependencies as close as possible. We experiment with four different ROS nodes, each implemented in the two selected languages, making it a 4-factor and 2-treatment (4F-2T) experiment.

The **target audience** of this study are researchers and practitioners working with ROS-based systems. This can aid other researchers and practitioners in making informed decisions, optimizing their ROS systems, and conducting meaningful experiments in the field

[1] https://docs.ros.org/en/galactic/Tutorials.html

of energy efficiency robotic systems. It teases researchers to conduct further investigations, going from different ROS communication models to architectural node design, such as service-based communication and composing multiple nodes in a single process. Moreover, it aids practitioners in making informed decisions about which language to use for their specific robotics projects and thus enables greener robotic software.

The paper not only **contributes** with valuable insights into the energy and power consumption of ROS nodes but also offers a methodological framework and practical guidance for conducting further experiments in this domain. We also provide a complete replication package and experimental data, which can benefit both, researchers and practitioners. Finally, the paper also works as a warning for greener robotic software, which may contribute to more environmentally aware robotic software development.

2 BACKGROUND

This section presents the fundamental concepts of ROS, its programming premises, and the principles of the Running Average Power Limit (RAPL).

2.1 ROS

ROS is a standard robotics framework in both, industry and research, for the effective development and building of a robot system [20]. Currently, there are many distributions of ROS available, grouped into two main versions (ROS 1 and ROS 2).

ROS architecture consists of several component types: nodes, topics, messages, and services. *Nodes* are executable processes, usually implementing a well-defined functionality of the ROS system. ROS nodes exchange data in an asynchronous publish/subscribe model. Messages are transmitted on a *topic* by a node with a publisher role. Other nodes that want to receive that information subscribe to that topic. If the nodes need to communicate synchronously, they can also be implemented with *service* calls.

2.2 ROS Programming

ROS is renowned for its flexibility in language choice, allowing developers to work with the language that best suits their needs. C++ and Python are the most used in ROS tutorials, counting on the two major client libraries maintained by the ROS 2 community, `rclcpp`[2] and `rclpy`[3], both built on top of `rcl` library[4]. As we can observe in Figure 1, all the client libraries share the same underlying software layers, starting with the ROS middleware, which is then overlapped by the `rmw` layers that abstract the ROS middleware layer enabling to switch between different communication middlewares without ROS 2 modifications. At the top, the `rcl` layer provides a uniform and high-level API for programming ROS 2 applications.

2.3 Running Average Power Limit (RAPL)

Modern processors provide a Running Average Power Limit (RAPL) interface for power management, which reports the processor's accumulated energy consumption, and allows the operating system to dynamically keep the processor within its limits of thermal design power (TPD) [11]. RAPL is a recurrent profiling tool in previous related work [8, 12, 24, 26]. It keeps counters that can provide power consumption data for both, processor and primary memory. CPU is proven to be one of the most energy-consuming parts of a computer system [9, 17, 24]. Despite the primary memory usage not being a usual determinant factor in other studies [17, 24], it is one of the main RAPL metrics and in this work will be used to determine whether it is still the case for ROS programming.

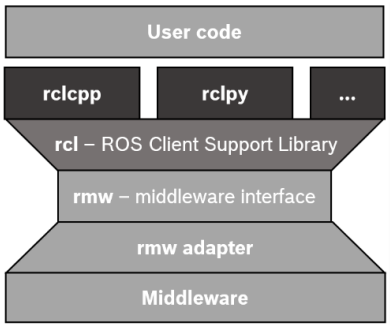


Figure 1: Underlying layers of a ROS node programming [7].

3 EXPERIMENT DEFINITION

The experiment of this paper is defined after the Goal Question Metric (GQM) model [3]. It starts with a well-defined *goal*, which is then refined into *research questions* that are answered by measuring the software system using objective and/or subjective *metrics*.

3.1 Study Goal

This study *goal* is to ***analyze*** *ROS programming with C++ and python* ***for the purpose of*** *understanding the extent* ***with respect to*** *energy efficiency* ***from the point of view*** *robotics researchers and practitioners* ***in the context of*** *publisher and subscriber ROS nodes.*

The goal is defined after a previous research [17] that spots C++ as one of the most energy-efficient programming languages. However, in their work, the algorithm complexity is the major factor that influences energy consumption, among which neither depends on the massive underlying software layers illustrated in Figure 1 nor is ROS-based.

ROS algorithms depend on canonical client libraries, such as *rclcpp* and *rclpy*. Despite those APIs being developed in parallel (for both languages) with the same underlying design principles, both languages are very different in terms of abstraction. This may impact directly ROS algorithm's performance and energy consumption, which should be thoroughly investigated.

3.2 Question

From our goal, we derive the following two research questions:

RQ1: What is the effect of C++ and python on the energy efficiency of a publisher/subscriber ROS node?

RQ2: How does the underlying software stack influence ROS nodes' energy efficiency?

Answering research RQ1 can help practitioners and researchers ponder which programming language to choose for future green

[2]https://github.com/ros2/rclcpp
[3]https://github.com/ros2/rclpy
[4]https://github.com/ros2/rcl

ROS projects. The RQ2 answer helps us to understand how much the ROS node algorithm itself and the `rclpy` and `rclcpp` client libraries impact global energy consumption in conjunction with all other software layers.

3.3 Metrics

Table 1 describes the metrics used for measurements during the experiments. *Energy consumption*, *power* and *execution time* are the key metrics used to assess the energy efficiency of a ROS node, while *CPU* and *memory usage* are metrics that help us to understand how intensive is the ROS node in terms of computational processing, and then reason about the measured *energy efficiency*.

Table 1: Experiment metrics.

Metric	Unit	Description
Energy Consumption	millijoules (mJ)	Amount of energy necessary to run the ROS node.
Power	milliwatts (mW)	Energy consumption rate when running the ROS node.
Execution time	milliseconds (ms)	Total time spent to run a ROS node.
CPU usage	percentage (%)	Average CPU percentage used during a ROS node execution.
Memory usage	megabytes (KB)	Amount of memory used during a ROS node execution.

All the measurements refer to the ROS node operating system process. The energy consumption measurements take into account the two main processing factors: CPU and memory. After the energy consumption is measured, we calculate the power with the following formula: $P = \frac{EC}{t}$, where P is power, EC is energy consumption, and t is the total ROS node execution time in seconds. In parallel with the energy consumption, we also measure the average CPU and memory usage. We give more details of the measurement process and related calculations measurements in Section 5.3.

4 EXPERIMENT PLANNING

We choose to implement ROS nodes after well-documented publish/subscribe algorithms from ROS Tutorials Wiki pages[1], which provide concise and compatible examples in C++ and Python. The algorithms are also selected to be independent of any physical robot, which increases the experiment's controllability.

Table 2 depicts the four selected algorithms, with a short description, and details of their implementation, including their dependency on other packages and libraries, and algorithm complexity metrics (i.e., logical lines of code – LLOC, and the algorithm McCabe's cyclomatic complexity – MCC). The dependencies and algorithm metrics are used to illustrate the compatibility between Python and C++ algorithm implementations.

The algorithms cover a basic topic-based asynchronous communication scenario, with two different `publisher` and `subscriber` node implementations, which also deal with two different message types. Nodes 1 and 2 implement a basic publish/subscribe topic communication, where Node 1 (`publisher`) sends plain-text messages to a topic, and Node 2 (`subscriber`) reads those messages by subscribing to the topic. Node 3 implements a `publisher` that sends coordinates instead of a plain-text message, which results in a distinct ROS message package requirement (i.e., `geometry_msgs` instead of `std_msgs`). Finally, Node 4 implements the `turtlesim` `subscriber`, which reads the coordinates sent by Node 3. We do not render the `turtlesim` on the screen avoiding extra processing that is not related to the `subscriber` node. Although, we validate Node 3 with the official `turtlesim` node[5]. We provided a replication package with all the implementations, where we can also check the compatibility among different language algorithms[6].

The experiment is planned with the following variables.

- **Independent variables**: the independent variable under investigation in this study is the ROS algorithm. The values for this variable correspond to the four algorithms subject of this study, their corresponding message types, and the two languages they are implemented with.
- **Static variables**: static variables are the same for all runs, which reflect the ROS system deployment, except the ROS algorithm under study. Therefore, the static variables are the *host device*; the *ROS distribution*; and the *operating system*.
- **Dependent variables**: these are the metrics that depend on the experiment execution, previously described on Table 1.

The factors of this study are the four ROS algorithms with two treatment values (C++ and Python) each, making our experiment design 4 factors - 2 treatments (4F-2T) experiment [25]. The experiment is balanced, with respect to its factor, as every treatment has been applied for 100 runs. In Section 5.2, we give more details of how nodes are executed during the experiments.

5 EXPERIMENT EXECUTION

In this section, we define the hardware and software components and aspects of experiment execution.

5.1 Instrumentation

All the experiments were conducted on a single desktop with the following specifications: Linux Ubuntu 22.04 operating system, kernel version 6.2.0-33-generic, with 8GB of RAM, and a Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz.

All the ROS packages necessary to run the experiments are installed locally under the ROS humble distribution[7], with an end-of-life (EOL) in May 2027. For each algorithm, we create a dedicated ROS package, and implement it as a ROS node. All the procedures are inside the node's callback functions, so ROS can spin them, taking care of underlying threading[8]. Algorithm executions are orchestrated by the `ros2 run` command, which speeds up automation and guarantees the same underlying layers for every execution.

To avoid concurrency, all the experiments were executed with a dedicated machine, without running other end-user apps. The operating system power saving mode was set as `performance` which should not limit the power usage. Finally, we gave priority level -20 (the highest) to the process corresponding to the experimented

[5] https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Introducing-Turtlesim/Introducing-Turtlesim.html
[6] https://github.com/IntelAgir-Research-Group/energy-ros2-cpp-python
[7] https://docs.ros.org/en/humble/index.html
[8] http://wiki.ros.org/roscpp/Overview/Callbacks%20and%20Spinning

Table 2: Algorithms subject of investigation

Node	Description	Dependencies	LLOC Python	LLOC C++	MCC Python	MCC C++
1.Simple Publisher (Node1)	ROS node (`talker`) that continuously sends messages to a `topic` for a constrained period of time.	`rclpy/rclcpp`, `std_msgs`	34	39	2	2
2.Simple Subscriber (Node2)	ROS node (`listener`) that subscribes to a `topic` and reads messages published by (Node1) until no more messages are received.	`rclpy/rclcpp`, `std_msgs`	34	42	2	2
3.Teleoperation Publisher (Node3)	ROS node that sends coordinates via teleoperation messages to supposedly drive a simulated robot forward and backward.	`rclpy/rclcpp`, `geometry_msgs`	35	38	3	3
4.Teleoperation Subscriber (Node4)	ROS node that reads coordinates published by Node 3.	`rclpy/rclcpp`, `geometry_msgs`	30	36	2	2

algorithms to give it the priority to access the machine resources. We wait 1 minute between each experiment execution waiting for the machine to cool down. This is greater than the observable time for the CPU to go back to its initial usage percentage.

5.2 Nodes Executions

The nodes are executed in pairs (a `publisher` and its respective `subscriber`): `node1` with `node2`, and `node3` with `node4`. Keeping in mind that for each `subscriber` the `publisher` must fork the message transmission, we also want to check how much CPU overhead this implies. For this, we re-executed the experiment four times, going from one `subscriber` node, and increasing `subscriber` node instances by ten at each execution. During the re-executions, we only keep track of Node 1 energy consumption and resource usage since it is the one that is directly affected.

5.2.1 Nodes 1 and 2. in this pair execution, Node 1 periodically (every 0.5 seconds) publishes a message on a specific topic, while Node 2 keeps reading the published messages. After 100 messages, both nodes are destroyed.

5.2.2 Nodes 3 and 4. Node 3 publishes coordinates to the topic `/turtle1/cmd_vel` each second, interchanging messages for a robot to be driven forward and then backward. This is repeated 50 times, which for a matter of comparison, intends to sum up the same number of messages exchanged between Nodes 1 and 2.

5.3 Resource Profiling

We profile the energy consumption with a customized Python that relies on `pyRAPL` library[9]. In the script, a `pyRAPL` measurement wraps up the algorithm launching command and, therefore, considers its global energy consumption, including `ros2 run` orchestration. Our script measurements have been validated against the ones from an existing RAPL-based tool by Pereira et al. [17]. We opt for customizing our own script for a matter of maintenance and adaptation. `pyRAPL` also counts the node's total execution time.

For CPU usage, we rely on `ps` Linux command[10] and calculate the average CPU usage for each node execution while repeating it until the node execution completion. We use a similar approach for memory by using the `pmap` Linux command[11].

5.4 Data Analysis

To facilitate our analysis, we first explore the distribution of power consumption for both groups visually with data plotting. In our data analysis, for each of the ROS nodes experimented, we aim to investigate potential differences in energy efficiency (energy and power consumption) against two primary independent variables, the programming languages `Python` and `C++`, using the one-way ANOVA test. The null hypothesis ($H0$) posits that there is no significant difference in the mean energy and power consumption between these two groups, while the alternative hypothesis ($H1$) suggests otherwise. Our objective is to rigorously test these hypotheses to determine whether the programming language used has an impact on power consumption.

6 RESULTS

This section presents the results of the experiments according to their grouped execution.

6.1 Nodes 1 and 2

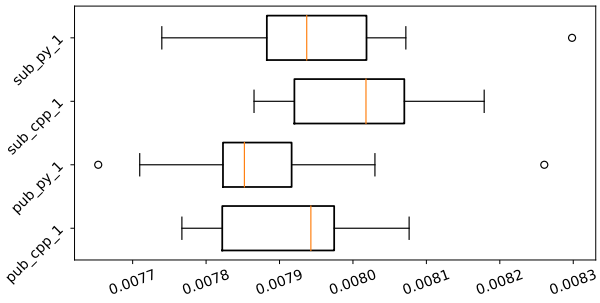
Figure 2a illustrates the power consumption of Nodes 1 (`n1`/publisher) and 2 (`n2`/subscriber) with only one instance of `n2`. Despite the measurements resulting in boxplots that cover nearer areas, their mean values point nodes written in Python consume less energy than the ones written in C++. This differs from previous research in the area of programming language energy efficiency [17] that pointed C++ as one of the most energy-efficient languages. However, it is understandable considering that both run on top of ROS stack, sharing basically the same underlying software layers, which' energy consumption is aggregated in the measurements. In this case,

[9] https://pypi.org/project/pyRAPL/

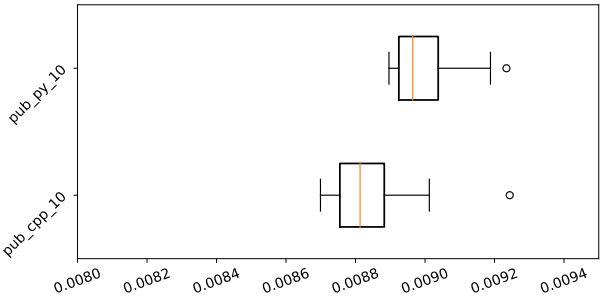
[10] https://man7.org/linux/man-pages/man1/ps.1.html

[11] https://man7.org/linux/man-pages/man1/pmap.1.html

the language should have little impact for such a simple communication as in the experiment. Furthermore, the one-way ANOVA test results in a *p*-value of 0.3103, which is considerably higher than the significance level, and *fails to reject the null hypothesis*. This indicates that the power consumption of both languages is close when with a single subscriber, where we cannot statistically confirm any of the languages is more energy-efficient in that case.



(a) *Distribution of power consumption (mW) with 1 subscriber.*



(b) *Distribution of power consumption (mW) of* n1 *with 10 subscribers (*n2 *instances).*

Figure 2: Power consumption of Nodes 1 (n1) and 2 (n2).

As explained in the experiment execution section, we also experiment n1 with multiple n2 instances to verify its power consumption scalability. In Figure 3, we observe that the power consumption converges as the number of n2 instances increases, when C++ becomes more energy-efficient, and the power consumption difference is visually perceptible. Since each subscriber requires the publisher to create a new thread, a plausible reason for the Python scalability issue is its well-known inefficient multi-threading implementation[12]. This is confirmed in Figure 2b, which shows the measurement distribution with 10 n2 nodes. For that experiment execution, the one-way ANOVA test results in a 0.0002 *p*-value, which successfully *rejects the null hypothesis*, and confirms our observation.

A further investigation indicates that the energy efficiency difference between the two languages may not be strictly related to the CPU and memory usage, which are the determinant factors in other research [9, 17]. Figure 4 illustrates the CPU and memory

[12]https://blog.devgenius.io/why-is-multi-threaded-python-so-slow-f032757f72dc

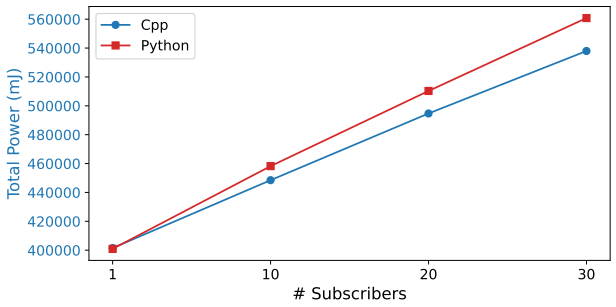


Figure 3: Power consumption scalability for Node 1.

usage for multiple subscribers (n2 nodes). We observe that memory usage increases slightly for both languages and the difference remains consistent and does not compare to their power consumption growth. In contrast, the CPU usage difference decreases as the number of n2 nodes increases and both become the same with 30 n2 nodes. This may also be related to a better efficiency of C++ multi-threading, resulting in better use of multiple CPU cores. The results indicate that the energy efficiency may be related to other aspects besides the programming languages, including the client libraries (`rclpy` and `rclcpp`) architectures.

By using the `perf top` tool during a complete experiment execution, we notice that while C++ uses a total of 12 shared objects while Python uses 17. This may be one of the causes of increased energy/power consumption due to architectural characteristics, such as system-level and instruction cache usage, code path among objects, and idle time. This tends to be more intensive as the number of subscribers increases, since objects related to libraries such as `libfastrtps`, necessary for inter-nodes communication, are requested more frequently.

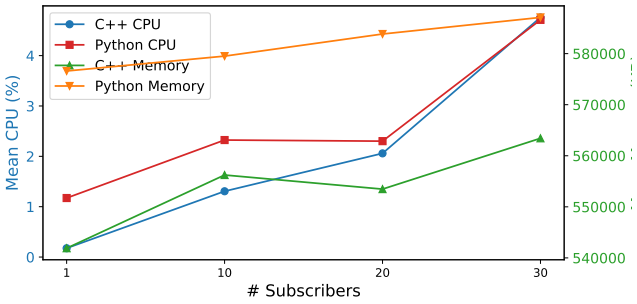


Figure 4: CPU and memory scalability of Node 1.

6.2 Nodes 3 and 4

Figure 5 illustrates the power consumption distribution for both, n3 and n4 with one instance each. We observe a consistent difference, where the nodes programmed in the C++ language consume less energy. In Figure 6, we observe that for n3 the difference is constant

even when the number of n4 node instances increases. After running the one-way ANOVA test we have a p-value equal to 0.0000, which rejects the null hypothesis and confirms our observation, indicating a statistically significant difference in the mean power consumption between the two groups. Different than for n1, the difference seems to stabilize from 10 subscribers.

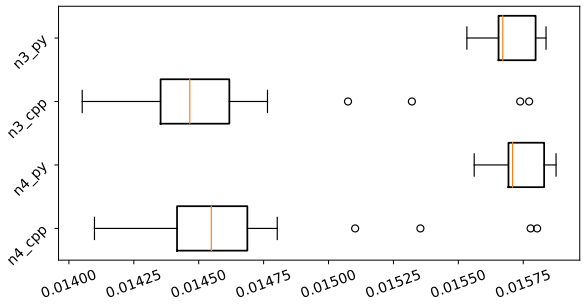


Figure 5: Power consumption (mW) of Nodes 3 (n3) and 4 (n4) with one instance each.

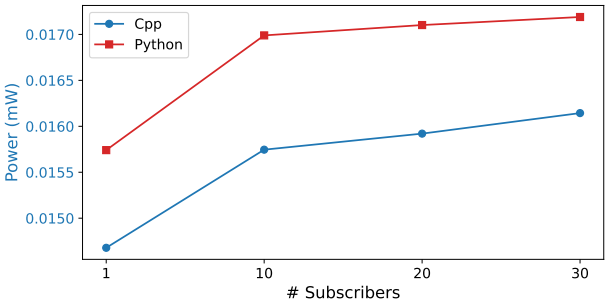


Figure 6: Power consumption scalability of Node 3.

In Figure 7, we observe that CPU and memory decrease as the number of subscribers increases, which is later discovered to be due to overloading. A further investigation reveals high CPU waiting rates from 10 n4 nodes, which indicates a processing bottleneck. For instance, C++ nodes result in the following CPU waiting rates: 30% with 10 n4 nodes; 55% with 20 n4 nodes; and 65% with 30 n4 nodes. With Python nodes, the overload seems to be even worse, where the waiting rates start with 45% with 10 n4 nodes and goes up 75% with 30 n4 nodes.

It is important to state that after identifying this, we planned to run another round of experiments with fewer subscribers, increasing one by one. However, this is not possible since even with 2 subscribers we already face waiting rates of 0.55%.

Despite the inconclusiveness of the experiment due to overloading, we also observe in this experiment that the power consumption is not strictly dependent on CPU and memory usage. The CPU waiting rates suggest that despite Python starting with a better CPU performance than C++, it tends to result in worse scalability as the publisher node needs to deal with more subscribers.

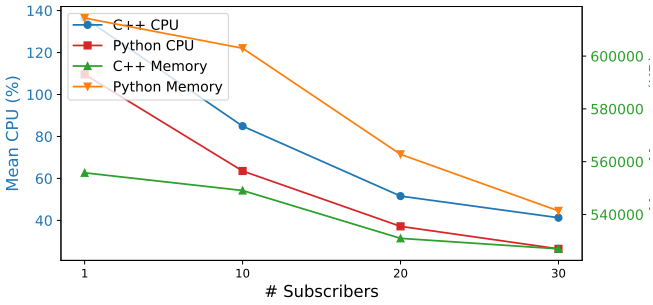


Figure 7: CPU and memory scalability of Node 3.

The identified overloading corroborates with Figure 6, which shows that after 10 n4 instances the power consumption stabilizes, consistently related to the lack of CPU power. When we investigate deeper into process shared objects, such as in the previous experiment, we observe a difference in the number of objects: 17 for C++ and 20 for Python. If compared with the previous experiment, the increased number of shared objects is related to `geometry_msgs` libraries, more numerous than for `std_msgs` one. This can be explained by the more complex data structures, which require specialized computation to serialize and deserialize messages transmitted over the network. The serialization/deserialization of such complex messages also explains the high CPU usage.

7 DISCUSSION

The experimental results provide valuable insights into the energy efficiency of ROS nodes programmed with different languages. The discussion section will focus on interpreting the results and drawing conclusions based on the findings.

Energy Efficiency of ROS Nodes: The experiment revealed that the energy efficiency of ROS nodes is indeed influenced by the choice of programming language. Contrary to previous research, which pointed to C++ as one of the most energy-efficient languages, the current study found that for ROS it may not be always the case since nodes in both languages resulted in similar power consumption in some simple publisher/subscriber scenarios. However, in more complex scenarios, going from 10 subscriber nodes and more complex message formats, C++ is still the language programming with better energy efficiency.

Impact of Underlying Software Stack: The results also shed light on the influence of the underlying software stack, particularly the ROS client libraries (*rclcpp* and *rclpy*), on the energy efficiency of ROS nodes. It was observed that the energy consumption was closely related to the architectural aspects of the client libraries, with C++ nodes demonstrating better energy efficiency. This suggests that the design and implementation of the client libraries play a crucial role in determining the energy efficiency of ROS nodes.

Practical Significance: While statistical significance is essential for hypothesis testing, practical significance is also crucial in determining the real-world impact of the findings. The experiment demonstrated that choosing C++ over Python as a programming

language when implementing ROS nodes results in significant energy savings over time. Considering the first experiment, from 10 subscribers, `Node 1` in C++ demonstrates a better energy efficiency. Choosing that node would reduce the power consumption in $\approx 0.36mWh$ (milliwatt-hours, if we consider it runs for one hour), which for a matter of comparison, would be enough to light up a $1W$ lamp for ≈ 22 minutes. This number becomes exponential, and therefore, more significant if we think of a large robotic system, with multiple robots, each of them relying on multiple topics, or even hundreds of robotic systems relying on C++ nodes. This becomes more significant if we think that implementing ROS nodes in C++ is very similar to in Python, despite the difficulties of the languages themselves, and that both languages rely on the same underlying software stack. Therefore, it is a very basic and initial design decision that directly contributes to the robotic system's energy efficiency.

7.1 Answering the Research Questions

Based on the result analysis, we can answer the research questions.
Answer of Research Question 1: the experiments reveal that the choice of programming language, specifically C++ and Python, has an impact on the energy efficiency of ROS nodes. We observe that *C++ is the most energy-efficient language when programming ROS nodes that communicate over topics*. However, the difference is not as significant as in other studies with algorithms that run without an underlying framework (i.e., ROS). In a very simple scenario with textual messages and only one subscriber, Python even seems to consume less power, despite no statistical significance. In both experiments, CPU and memory usage scale differently than the power consumption, which indicates that there is no direct correlation among them. One of the determinant factors may be the underlying software layers, which is discussed while answering RQ2.
Answer of Research Question 2: the results emphasized the significant influence of the underlying software stack, particularly the ROS client libraries (`rclcpp` and `rclpy`), on the energy consumption of ROS nodes. The design and implementation of the client libraries are determinant factors for the energy efficiency of ROS nodes. Ground-truth programming language theory puts Python as a language with high levels of abstraction, which is given it overlaying other low-level languages, including C. We identified that its client library (i.e, `rclpy`) also relies on a greater number of shared underlying objects than `rclcpp`. As discussed in the results section, this is a reasonable explanation of the increased power consumption for nodes implemented with it.

The importance of a correct client library design is reinforced in some ROS Discourse discussions. For instance, one of those helped us with a workaround solution when we faced unexpected CPU overloading during the experiment executions[13]. The solution was to use a static-single-threaded spinning for `rclcpp`[14], which is officially available since 2020 and documented on ROS Documentation forum[15]. Despite the intense discussion, we did not identify any mention of how poor ROS client library design could compromise energy efficiency. This indicates that the current study and possible extensions can help the ROS community to develop more energy-efficient software.

[13]https://discourse.ros.org/t/singlethreadedexecutor-creates-a-high-cpu-overhead-in-ros-2/10077
[14]https://github.com/ros2/rclcpp/pull/873
[15]https://docs.ros.org/en/foxy/Concepts/About-Executors.html

8 THREATS TO VALIDITY

Here, we discuss the threats to the validity of the experiments.

8.1 External Validity

The generalizability of the findings may be limited due to the specific nature of the ROS nodes and the chosen algorithms. The results may not be applicable to all types of ROS-based systems and may not fully represent the energy efficiency of more complex robotic applications. However, it brings important insights into the most common communication in the ROS system, the publish/subscribe architecture.

8.2 Internal Validity

The internal validity has been guaranteed by a strictly controlled environment, aiming at mitigating variations in system load, background processes, or hardware differences. Their impact is also tackled by repeating the experiments a hundred times.

The tools and libraries for measurements have been thoroughly used in previous research [8, 12, 24, 26]. The measurements of the only customized tool, the one for energy and power consumption, are validated against the measurements of another tool also used in previous research.

The experiment focused on the energy efficiency of ROS nodes programmed with different languages, but the influence of the underlying client library architecture (`rclcpp` and `rclpy`) on energy consumption may not have been fully isolated from the language-specific effects. This should not be a problem since the languages imply the use of such libraries and we also analyze their potential impact on the measurements.

The observed overloading issues with CPU usage and memory may have impacted the results, potentially confounding the comparison of energy efficiency between C++ and Python nodes. Therefore, the results of that experiment were considered inconclusive and their discussion is used only for a matter of instigation of new experiments that can prove the raised hypothesis.

8.3 Construct Validity

The experiments are carefully planned and executed with metrics that are representative of this type of work [8, 17]. The power consumption metric plays a crucial role in construct validity by providing a direct measure of the rate at which energy is consumed when running the ROS nodes. It can capture subtle differences in energy efficiency resulting from language-specific factors and underlying software architecture and is not confused by other dependent variables, such as the execution time.

9 RELATED WORK

The energy efficiency of software has been a recurrent matter of investigation in recent years, going from deployment infrastructures to programming languages [23], which is more aligned with the interests of this paper's research. To the best of our knowledge after a search on the main research indexers, there is no previous

work that studies the energy efficiency of programming languages in the ROS ecosystem context.

Pereira et al. [17] conducted a large study that most resembles ours, where they experiment with a set of different algorithms programmed with 27 of the most popular programming languages. Their work also led to a rank of the most energy-efficient programming languages [18]. Despite the purpose of the research being similar to our approach, the nature of the studied algorithms is very distinct. Our algorithms are ROS domain-specific, where the algorithms depend on a running underlying ROS stack, while the authors consider a set of diverse programming problems, which run natively, without any middleware or framework layer.

Other studies with fewer similarities with ours investigate the energy efficiency of specific programming languages in varied contexts [10, 14]. Others focus on other aspects of the programming language, such as the compiler [1] or programming paradigm [16].

10 CONCLUSION

In this paper, we start an investigation of how the chosen programming language may impact the energy efficiency of a ROS system. We lead two pairs of ROS nodes to repeatedly exchange messages over a publish/subscribe communication pattern while measuring CPU and memory usage and each node's power consumption.

The experiment results indicate that for simple communication scenarios with only one subscriber, there is no significant difference between the two studied languages, i.e., C++ and Python. However, with multiple subscribers, the C++ client library becomes more energy efficient and scales better, which we reason to be related to native multi-threading implementation. Therefore, the choice of programming language must be one of the preliminary points to be discussed in ROS projects that want to achieve good energy efficiency. The results also reveal that programming language client libraries' design may compromise energy efficiency directly, where their broader discussion among development teams could also help in making robotic systems more energy efficient.

Future work should focus on a deeper investigation of the energy efficiency of ROS nodes with more complex scenarios. Such scenarios could, for instance, involve services, action servers, multi-threading, and compositions. Additionally, it is also important to have a further and more detailed understanding of the architectural aspects of each of the client libraries.

ACKNOWLEDGMENTS

The completion of this research was made possible through the support and funding from the following grants: National Council for Scientific and Technological Development (CNPq), Grant Process 200006/2024-0, and InnovationsCampus Mobilität der Zukunft, Mid-term Fellowships for Experts.

REFERENCES

[1] Sarah Abdulsalam, Donna Lakomski, Qijun Gu, Tongdan Jin, and Ziliang Zong. 2014. Program energy efficiency: The impact of language, compiler and implementation choices. In *International Green Computing Conference*. IEEE, 1–6.

[2] Michel Albonico, Milica Đorđević, Engel Hamer, and Ivano Malavolta. 2023. Software engineering research on the Robot Operating System: A systematic mapping study. *Journal of Systems and Software* 197 (2023), 111574. https://doi.org/10.1016/j.jss.2022.111574

[3] Victor Basili, Gianluigi Caldiera, and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.

[4] Katerina Chinnappan, Ivano Malavolta, Grace A Lewis, Michel Albonico, and Patricia Lago. 2021. Architectural Tactics for Energy-Aware Robotics Software: A Preliminary Study. In *European Conference on Software Architecture*. Springer, 164–171.

[5] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Jana Tumova. 2017. Engineering the software of robotic systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 507–508.

[6] Steve Cousins. 2011. Exponential growth of ros [ros topics]. *IEEE Robotics & Automation Magazine* 1, 18 (2011), 19–20.

[7] ROS 2 Documentation. [n. d.]. *Executors*. https://docs.ros.org/en/foxy/Concepts/About-Executors.html

[8] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review* 40, 3 (2012), 13–17.

[9] E. Hirao, S. Miyamoto, M. Hasegawa, and H. Harada. 2005. Power Consumption Monitoring System for Personal Computers by Analyzing Their Operating States. In *2005 4th International Symposium on Environmentally Conscious Design and Inverse Manufacturing*. 268–272. https://doi.org/10.1109/ECODIM.2005.1619220

[10] Håvard H Holm, André R Brodtkorb, and Martin L Sætra. 2020. GPU computing with Python: Performance, energy efficiency and usability. *Computation* 8, 1 (2020), 4.

[11] Intel. [n. d.]. Running Average Power Limit Energy Reporting CVE-2020-8694,... https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html

[12] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (mar 2018), 26 pages. https://doi.org/10.1145/3177754

[13] Anis Koubaa. 2015. ROS as a service: web services for robot operating system. *Journal of Software Engineering for Robotics* 6, 1 (2015), 1–14.

[14] Mohit Kumar. 2018. *Energy Efficiency of Java Programming Language*. Wayne State University.

[15] Ivano Malavolta. 2021. https://www.ivanomalavolta.com/mining-the-ros-ecosystem-for-energy-efficient-robotics-software/

[16] Sepideh Maleki, Cuijiao Fu, Arun Banotra, and Ziliang Zong. 2017. Understanding the impact of object oriented programming and design patterns on energy efficiency. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. IEEE, 1–6.

[17] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN international conference on software language engineering*. 256–267.

[18] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021), 102609. https://doi.org/10.1016/j.scico.2021.102609

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

[20] André Santos, Alcino Cunha, Nuno Macedo, and Cláudio Lourenço. 2016. A framework for quality assessment of ROS repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4491–4496.

[21] Stanford Artificial Intelligence Laboratory et al. [n. d.]. *Robotic Operating System*. https://www.ros.org

[22] Stan Swanborn and Ivano Malavolta. 2020. Energy efficiency in robotics software: A systematic literature review. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 144–151.

[23] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol de Vries. 2021. Green IT and Green Software. *IEEE Software* 38, 6 (2021), 7–15. https://doi.org/10.1109/MS.2021.3102254

[24] Jóakim von Kistowski, Hansfried Block, John Beckett, Cloyce Spradling, Klaus-Dieter Lange, and Samuel Kounev. 2016. Variations in CPU Power Consumption. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/2851553.2851567

[25] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[26] Huazhe Zhang and H Hoffman. 2015. A quantitative evaluation of the RAPL power control system. *Feedback Computing* 6 (2015).