Empirical Software Engineering (2023) 28:140
https://doi.org/10.1007/s10664-023-10351-6

Computation offloading for ground robotic systems communicating over WiFi – an empirical exploration on performance and energy trade-offs

Milica Đorđević[1] · Michel Albonico[2] · Grace A. Lewis[3] · Ivano Malavolta[1] · Patricia Lago[1]

Accepted: 30 May 2023 / Published online: 13 October 2023

Abstract

Context Robotic systems are known to perform computation-intensive tasks with limited computational resources and battery life. Such systems might benefit from offloading heavy workloads to the Cloud; however, in some cases, this implies high network traffic that degrades performance and energy consumption.

Goal In this study, we aim at evaluating the impact of different computation offloading strategies on performance and energy consumption in the context of autonomous robots.

Method We conduct two controlled experiments involving a robotic mission based on the Turtlebot3 robot and ROS 1. The mission consists of three tasks that are recurrent in robotics and good candidates for computation offloading in research, namely, SLAM mapping, navigation stack, and object recognition. Each of the tasks is either executed on board or offloaded in a full-factorial experiment design. The obtained measures are then statistically analyzed.

Results The results show that offloading the object recognition task causes a more significant decrease in resource utilization and energy consumption than both SLAM mapping and navigation. However, object recognition affects the volume of network traffic significantly to the extent that it can easily cause network congestion.

Conclusions In the context of our experiments (*i.e.,* those involving small-scale ground ROS-based mobile robots operating under WiFi networks), offloading object recognition is beneficial in terms of performance and energy consumption. Nevertheless, large network bandwidth needs to be available for object recognition offloading. While the image resolution and frame rate have a significant impact on not only the network traffic but also energy consumption and performance, these parameters need to be carefully set so that the results of this task can be always received in time, which is particularly crucial in real-time systems.

Keywords Robotic system · Computation offloading · Software engineering · Empirical evaluation · Energy consumption

Communicated by: Neil Ernst

✉ Ivano Malavolta
i.malavolta@vu.nl

Extended author information available on the last page of the article

1 Introduction

Robotic systems can be resource-constrained in terms of onboard computing and storage capabilities (Koubaa 2020). Such constraints are typically dictated by robot size, shape, payload requirements, and working environment, especially for drones and small terrestrial mobile robots (Doriya et al. 2012). Additionally, mobile robots usually have a limited power supply (Koubaa 2020) (Doriya et al. 2012), where computation-intensive tasks account for a substantial part of the total energy consumption (Sarker et al. 2019). To compensate for such limitations, computation could be offloaded from the robot to the Cloud, in situations where connectivity is available (Tripathy et al 2022).

Despite overcoming hardware limitations, computation offloading may have downsides. For instance, some offloaded tasks may process a large amount of sensor data (e.g., laser scans, camera images) which will need to be exchanged over the network. In such cases, network latency and variable network bandwidth may have a significant effect on the overall performance (Kehoe et al. 2015), which may also impact the energy consumption (Gomez et al. 2011). High latency may also be a constraint for other software quality aspects, such as safety, where the robot must react quickly to avoid accidents; for example, an autonomous vehicle must recognize pedestrians and traffic signs on-the-fly. Therefore, fundamentally, design decisions for offloading must consider the trade-off between performance and other quality aspects required, and the energy consumed for the task execution (Hu et al. 2012).

The main **goal** of this study is to perform an exploratory characterization on how computation offloading strategies affect the performance and energy consumption of ground robotic systems operating under WiFi networks. Our study focuses on a robotic mission involving an autonomous driving robot, and relies on three basic tasks: *mapping (SLAM)*, *navigation* and *object recognition* (Doriya et al. 2012; Hu et al. 2012; Koubaa 2020). The robotic system is implemented on *Robot Operating System* (ROS), a de-facto standard for both research and industry, with packages released and recommended by its community.

With the experiment results, we aim at answering a **main research question**:

- **RQ1** – What are the trade-offs between performance and energy consumption for computation offloading of ground ROS-based systems operating under WiFi networks? Which is answered after two more specific questions.
  - *RQ1a* – What is the impact of computation offloading on the performance of ground ROS-based systems operating under WiFi networks?
  - *RQ1b* – What is the impact of computation offloading on the energy consumption of ground ROS-based systems operating under WiFi networks?

All the tasks of our robotic system are highly configurable, where many of the parameters may directly influence their respective computation load. Therefore, we perform an additional set of experiments to determine what is the effect that some of the most common parameters have on performance and energy consumption. To this aim, we pose an **additional research question**:

- **RQ2** – How do specific parameter configurations influence the offloading design of ground ROS-based systems operating under WiFi networks? Which also relies on two more specific questions.
  - *RQ2a* – What is the impact of parameter configuration on the performance of ground ROS-based systems operating under WiFi network?
  - *RQ2b* – What is the impact of parameter configuration on the energy consumption of ground ROS-based systems operating under WiFi networks?

Among others, the experiments' **results** indicate that the most affected task was object recognition, where its offloading reduces significantly the robot's resource utilization and energy consumption. Object recognition also performs more efficiently when offloaded due to the remote higher computation power (which may also be due to the limited resources of the used robot). On the contrary, mapping and navigation could be executed efficiently on both, onboard and offloaded. The results also indicate that object recognition parameters should be carefully configured so that each image frame can be processed at the desired rate. It is important to note that our experiments have been carried out using a specific type of ground robot (*i.e.,* a Turtlebot3 Burger[1]) operating under a continously-available WiFi network. As such, the reader is invited to consider the following points to better understand the context and validity of the obtained results: (i) the implemented robotic mission is intended to be *minimalistic* (*e.g.,* there are no moving obstacles during the mission) in order to keep control of the main factors under experimentation; (ii) latency and execution time are measured primarily as a performance metric, we are planning to further explore them as thresholds that trigger the reconfiguration of offloading strategies at run-time; (iii) we consider a stable and isolated WiFi network, with sufficient bandwidth, to avoid uncontrollable influences on the mission execution and to facilitate future replications of the experiment; (iv) the ROS packages used in this study are based on ROS1, which is being superseded by ROS2 at the time of writing; nevertheless, the applicability of the results of this study for practitioners/researchers is still reasonably wide since the ROS1 distribution we used has as a long-term support valid until 2025 and it currently has a broad documentation and a large ecosystem of third-party plugins[2].

In summary, the main **contributions** of this paper are:

– an empirical evaluation of the performance and energy trade-offs of computation offloading for ground ROS-based systems operating under WiFi networks;
– directions on how to architect ROS-based systems with regards to offloading common compute-intensive tasks in the context of ground ROS-based systems operating under WiFi networks;
– takeaways for future studies on offloading robotic tasks in the context of ground ROS-based systems operating under WiFi networks;
– an open-source implementation of a robotic system configured via widely used ROS packages;
– a complete replication package including both raw data and source code related to the conducted experiment (Dordevic et al. 2022).

The **target audience** of this work consists of ROS developers and robotics researchers targeting ground ROS-based systems operating under WiFi networks. Because the implemented robotic system has been developed on top of widely popular ROS packages, it is of great importance for ROS developers and researchers to understand what are the possible benefits and downsides of offloading. The experiment design and our analysis of the results can also guide other research questions and hypotheses for future work, which could benefit from the publicly-available implementation of the used robotic system. It is important to note that the robotic system is implemented using ROS1, and the obtained results might not directly translate to ROS2 systems.

The remainder of this paper is organized as follows. Section 2 presents background information for understanding the context of this study. Section 3 defines the goals of the study,

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

[2] https://roboticsbackend.com/ros1-vs-ros2-practical-overview/

its research questions, and used metrics. Section 4 presents the experiment design. Section 5 describes the setup of the hardware and software used in the experiments. Sections 6 and 7 present the results of the first and second experiments, respectively. In Section 8 we elaborate on the obtained results, contextualize them, and elicit the key takeaway messages. Section 9 presents threats to validity. Section 10 presents related work, and finally Section 11 closes the paper and provides directions for future work.

2 Background

In this section, we provide a short introduction to the Robot Operating System and briefly describe tasks that are candidates for computation offloading, namely, SLAM, localization, navigation, and object recognition. They are common in autonomous robot navigation stacks and tend to be compute-intensive.

2.1 Robot Operating System

Robot Operating System (ROS) is an open-source, multi-language middleware for robotics software development that abstracts the underlying hardware (Quigley et al. 2009). A typical ROS-based system is organized as a component-based set of *nodes*, i.e., processes that exchange messages over an abstraction called *topic* in a *publish-subscribe* model. However, ROS also offers a *client-server* communication model in the form of *services*. One of the main reasons for this expansion of ROS is its focus on collaborative robotics software development and code reuse. Domain experts can implement tasks organized within a modular abstraction called *package* that can be further reused by other community members.

ROS currently has two different versions, ROS version 1 (or simply ROS1) and ROS version 2 (or simply ROS2). ROS1 has an academic nature (St-Onge and Herath 2022) and is still largely used in recent studies (Albonico et al. 2023). Since 2017, its core has been modernized, resulting in the first stable release of ROS2 in 2020 (St-Onge and Herath 2022). While ROS1 relies on a central component (*i.e.,* the *master* node), ROS2 distribution relies on the Data Distribution Service (DDS) framework, which is a better option for scalability and maintenance. ROS2 also introduces the concept of QoS profiles and security features such as encryption and authentication. Despite the great difference in their implementations, both versions have a common ground in terms of packages and philosophy. They can also work together through a proxy called *ROS bridge*[3].

2.2 Offloadable Tasks

Robotic systems usually consist of computation-intensive tasks with many of them being good candidates for computation offloading. According to previous research, some of the most commonly used tasks in robotic systems are SLAM (Doriya et al. 2012; Hu et al. 2012; Koubaa 2020), localisation (Koubaa 2020), navigation (Doriya et al. 2012; Hu et al. 2012; Koubaa 2020) and object recognition (Doriya et al. 2012; Koubaa 2020), which are briefly described below.

[3] http://wiki.ros.org/rosbridge_suite

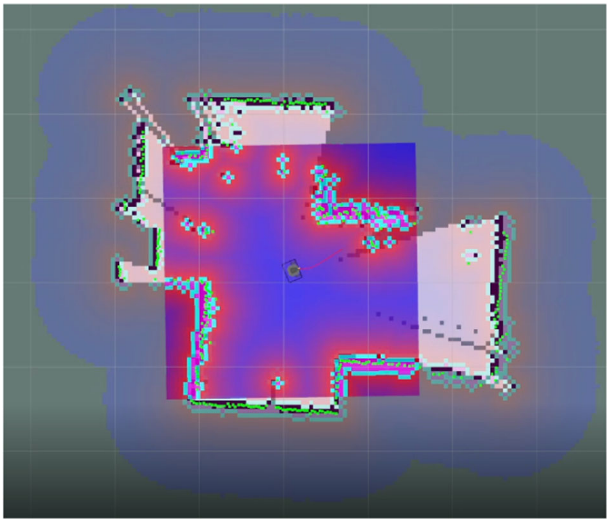


Fig. 1 Example of map generated by a SLAM algorithm

2.2.1 SLAM

Simultaneous Localization and Mapping (SLAM) is a core task for any robotic system that performs automatic navigation. Being aware of its own position and surrounding objects is necessary for planning the robot's future movement. A robot needs to construct a map of the unknown environment, such as the one in Fig. 1[4], and localize itself within that map (the central black spot in the figure).

There are numerous SLAM algorithms, many of which are available as ROS packages and thus reusable by ROS community members. According to the findings of Santos et al. (Santos et al. 2013), Gmapping, KartoSLAM, and HectorSLAM produce the most accurate maps, while the promising results presented in the work of (Grisetti et al. 2007) indicate that Gmapping can perform well on limited processing power robotic systems. For the aforementioned reasons, we opt for offloading Gmapping in this work.

Gmapping is a laser-based SLAM, where it is possible to create a 2D grid map and define the robot's position in the map from the data acquired from a laser scanner, such as a Light Detection and Ranging (LIDAR) and odometry. Using the LiDAR scanner, the algorithm (i) creates a 2D grid map of the environment and (ii) calculates the probability of the robot position being correct using a Rao-Blackwellized particle filter (RBPF) (Grisetti et al. 2007). A Particle Filter is a type of localization that chooses to represent individual randomly-estimated positions on a grid map as *particles*, where a grid map counts on numerous particles. By matching the particles to the scan data coming from the robot, the algorithm is able to identify which particles are most likely to occupy the true position of the robot on the map. For example, once the robot moves within the environment, the sensor captures obstacles (*e.g.,* a wall) and certain particles can be ruled out as they do not match the sensor data. However, imagine the robot in an open area, with no obstacles around; then, it would be difficult to identify which are the likely particles because the sensing data provides no reference. In this

[4] https://msadowski.github.io/hands-on-with-slam_toolbox/

case, the robot relies on other sensors, such as odometry, which tracks the rotation of each of the wheels on a wheeled robot.

A common ROS package is the *gmapping* package[5], which basically wraps the *OpenSLAM* Gmapping algorithm[6] to be transparent to the ROS system.

2.2.2 Localization

When a robot operates in a known working environment (already having its map), it is still necessary for the robot to localize itself on the map so that it can navigate to other locations. This task is performed by localization algorithms.

There is a variety of different localization approaches available, but the most commonly used one in ROS is Adaptive Monte Carlo localization (AMCL). Such algorithms are probabilistic particle filter algorithms used in 2D spaces, which were originally proposed in the work of (Thrun et al 2001), which work similarly to the SLAM localization explained in the previous section. Basically, the robot's location is estimated using a set of particles, i.e., weighted samples that represent a probable pose (dos Reis et al. 2019).

A common package is the AMCL ROS[7], which implements the adaptive Monte Carlo localization approach. ACML ROS has an additional adaptive characteristic, meaning that the number of particles used in the algorithm increases with the uncertainty of the robot's pose (dos Reis et al. 2019). On the contrary, when the uncertainty is low, the number of particles is reduced so that the algorithm performs efficiently.

2.2.3 Navigation

Navigation is a feature where the robot moves autonomously to a goal location on a map. It depends on multiple sensors and localization algorithms to define which is the best path for the robot to go through. This takes into account, for example, obstacle avoidance and behavior recovery (when the robot is unable to perform a goal).

A navigation stack is a set of ROS packages that can be configured together to execute a complex task of driving a robot to a goal location autonomously (ROS wiki 2021). The central part of the navigation stack is the *move_base* package, which provides an interface for configuration and running. Components that perform specific tasks while interacting with each other can be configured as plugins to the *move_base* node. Many of the plugins are available to the ROS community, but ROS developers can write their own plugins that suit their needs.

Figure 2 shows a high-level overview of the *move_base* node, its internal components, and interactions. At a high level of abstraction, this node receives a goal location as a ROS action and sends velocity commands to the robot to navigate to the goal. Internally, it consists of two cost maps that store information about obstacles in the working environment: the *global costmap* that is used for global planning and long-term plans, and the *local costmap* is used for local planning and obstacle avoidance. The *global costmap* receives a map of the environment via *map_server*, and the *local costmap* is fed by sensor sources, *e.g.,* laser scans or depth camera inputs so that dynamic obstacles are taken into account. Another component, the *global planner* is in charge of finding the path between the robot's current position and the

[5] http://wiki.ros.org/gmapping

[6] https://openslam-org.github.io/gmapping.html

[7] http://wiki.ros.org/amcl

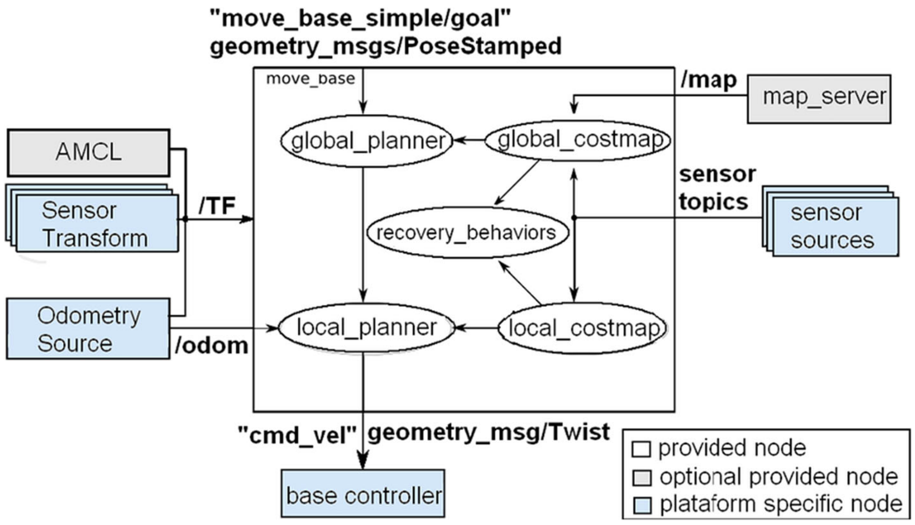


Fig. 2 Navigation stack setup Garro et al. (2014)

goal location. This component is based on one of the path-finding algorithms (e.g., Dijkstra, A*). Once the path is found, the *local planner* sends velocity commands to the robot. The navigation stack can also be configured with *recovery behavior* that will be triggered when the robot gets stuck or, for any reason, is unable to reach its goal location. The detailed guide for configuring the ROS navigation stack is available on the ROS wiki page (ROS wiki 2021).

2.2.4 Object Recognition

Computer vision tasks are used to provide more sophisticated vision in robotic systems. Robots are equipped with cameras, as well as depth cameras, which take action when the surrounding environment cannot be deduced by laser scans. This includes object recognition, face recognition, or even SLAM algorithms that are entirely based on camera images (visual SLAM or V-SLAM).

OpenCV is a commonly used computer vision library, mostly due to its efficiency. The ROS package *find_object_2d* (Labbe 2021) is based on OpenCV[8], which consists of a ROS wrapper around a *FindObject* application, a simple Qt interface that can be used for trying OpenCV implementations of feature detectors and descriptors. Due to its simplicity, maintainability, and widespread usage within the ROS community, the *find_object_2d* package will be used for the object recognition tasks in this work.

3 Experiment Definition

Figure 3 depicts the relationships between the experiment goals, main research questions, and metrics, as proposed by Basili et al. in their *Goal-Question-Metric* (GQM) framework (Basili et al. 1994), which are further explained in this section. This work aims at two goals, which are presented in the next two sections as *main goal* (see Section 3.1) and *secondary goal*

[8] https://opencv.org

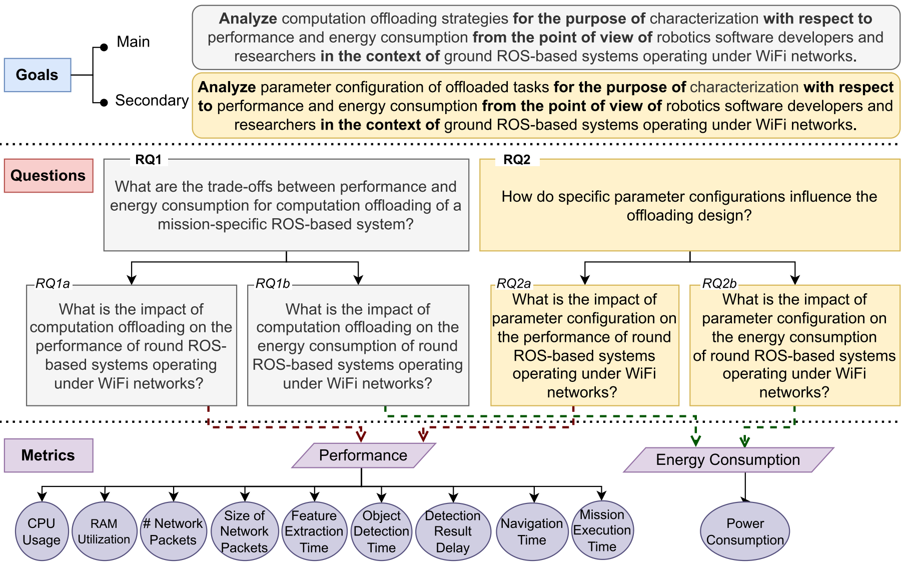

Goals
Main
Secondary
Analyze computation offloading strategies for the purpose of characterization with respect to performance and energy consumption from the point of view of robotics software developers and researchers in the context of ground ROS-based systems operating under WiFi networks.
Analyze parameter configuration of offloaded tasks for the purpose of characterization with respect to performance and energy consumption from the point of view of robotics software developers and researchers in the context of ground ROS-based systems operating under WiFi networks.
Questions
RQ1
What are the trade-offs between performance and energy consumption for computation offloading of a mission-specific ROS-based system?
RQ2
How do specific parameter configurations influence the offloading design?
RQ1a
What is the impact of computation offloading on the performance of round ROS-based systems operating under WiFi networks?
RQ1b
What is the impact of computation offloading on the energy consumption of round ROS-based systems operating under WiFi networks?
RQ2a
What is the impact of parameter configuration on the performance of round ROS-based systems operating under WiFi networks?
RQ2b
What is the impact of parameter configuration on the energy consumption of round ROS-based systems operating under WiFi networks?
Metrics
Performance
Energy Consumption
CPU Usage
RAM Utilization
# Network Packets
Size of Network Packets
Feature Extraction Time
Object Detection Time
Detection Result Delay
Navigation Time
Mission Execution Time
Power Consumption

Fig. 3 GQM representation of experiment definition

(see Section 3.2). The main and secondary goals have their own research questions and both of them consider exactly the same metrics (*e.g.,* CPU usage, navigation time, power consumption).

3.1 Main Goal And Research Question

By following the goal definition template provided by (Basili et al. 1994), the *main goal* of this study can be formulated as: ***analyze*** *the computation offloading strategies* ***for the purpose of*** *characterization* ***with respect to*** *performance and energy consumption* ***from the point of view of*** *robotics software developers and researchers* ***in the context of*** *ground ROS-based systems operating under WiFi networks*.

Based on the goal reported above, we aim at answering the following main research question (**RQ1**):

What are the trade-offs between performance and energy consumption for computation offloading of ground ROS-based systems operating under WiFi networks?

This question is further divided into the following two more specific sub-questions:

- **[RQ1a]** What is the impact of computation offloading on the performance of ground ROS-based systems operating under WiFi networks?
- **[RQ1b]** What is the impact of computation offloading on the energy consumption of ground ROS-based systems operating under WiFi networks?

The main motivation behind computation offloading is to reduce a heavy workload on the already limited robotic system's resources. Therefore, we define the research question *RQ1a* to evaluate if computation offloading indeed has a significant impact on resource utilization, in particular, CPU usage and RAM utilization. Moreover, one of the main drawbacks of computation offloading lies in the heavy network exchange of sensor data. Thus, we aim to evaluate the number and the size of network packets exchanged between the robot and the offload target (*i.e.,* remote PC).

The efficiency of the tasks themselves can be potentially improved when they are offloaded, because there are more computational resources available within the offload target. However, ROS1 has already known communication latency, depending on network traffic (Shakhimardanov et al. 1985). Therefore, we evaluate if object recognition and navigation perform the main operations faster when they are offloaded.

Robotic systems are equipped with batteries of a limited lifetime. Particularly for mobile robots, energy consumption needs to be optimized in such a way that they have enough power supply to perform the intended mission. The energy consumed for the onboard execution of computation-intensive tasks can be spared with their offloading to remote machines. Nevertheless, computation offloading comes with the energy consumption overhead for the network exchange of sensor data. To that aim, we define research question *RQ1b* to evaluate if the energy savings, accounted for offloading intensive tasks, can compensate for the energy consumed for the data exchange over the network. The trade-off between the two energy expenses can be crucial when deciding if a task should be offloaded or not.

3.2 Secondary Goal

The *secondary goal* of this study is to ***analyze*** *the parameter configuration of offloaded tasks* ***for the purpose of*** *characterization* ***with respect to*** *performance and energy consumption* ***from the point of view of*** *robotics software developers and researchers* ***in the context of*** *ground ROS-based systems operating under WiFi networks.*

The tasks that were subject to computation offloading in the primary experiment are highly configurable, with many of the parameters having a direct influence on their respective computation and/or communication workload. Therefore, we might get different results, or even conclusions, if another parameter configuration is used. For this reason, we aim to perform an additional set of experiments to answering the following research question (**RQ2**): *How do specific parameter configurations influence the offloading design of ground ROS-based systems operating under WiFi networks?*

RQ2 is further structured into the following (more specific) sub-questions:

– **[RQ2a]** What is the impact of parameter configuration on the performance of ground ROS-based systems operating under WiFi networks?
– **[RQ2b]** What is the impact of parameter configuration on the energy consumption of ground ROS-based systems operating under WiFi networks?

It is important to note that during the execution of the experiment we are keeping the parameter configurations fixed for the whole duration of the mission. This decision stems from the fact that we aim to (i) isolate their potential impact on the dependent variables of the study and (ii) facilitate the comparison of the obtained results against those of future replications of the experiment.

3.3 Metrics

We use ten metrics as dependent variables to evaluate the performance and energy efficiency of the subjects of our experiment. As illustrated in Fig. 3, these metrics are classified in two main groups: *performance*, for answering *RQ1a* and *RQ2a*, and *energy consumption*, for answering *RQ1b* and *RQ2b*.

The *Performance* metrics are chosen according to the following rationales. *CPU Usage* and *RAM Utilization* are defined so that the impact of different offloading strategies on CPU and RAM can be evaluated. CPU and Memory are very common metrics in other performance studies, as these resources are necessary for running the software under experimentation (Maruyama et al. 2016; Shamshiri et al. 2018; Wienke et al. 2018). *Number* and *Size of Network Packets* are selected for the purpose of evaluating the network overhead (Profanterb et al. 2019; Maruyama et al. 2016) and latency (Shakhimardanov et al. 1985) that comes with the price of computation offloading. Those metrics also go towards the evaluation of adequate network transportation, which is a principle of robotic framework design (Reichardt et al. 2013). We also consider other latency-related metrics, which reflect task efficiency besides network performance. *Feature Extraction Time*, *Object Detection Time* and *Detection Result Delay* are defined to compare the efficiency of the object recognition task within different offloading strategies, whereas *Navigation Time* is included to assess the efficiency of the navigation task.

Energy Consumption relies only on the *Power Consumption* metric, which measures the total power consumption within a certain period of time (*i.e.,* experiment/mission execution).

All the metrics are described in Table 1, and Section 4.2 provides details on the measurement techniques.

Table 1 Metric descriptions

Name	Unit of Measurement	Description
CPU Usage	*percentage*(%)	The average CPU usage during the entire mission execution.
RAM utilization	*megabytes*(MB)	the average RAM utilization during the entire mission execution. The memory has the same.
Number of Network Packets	*count*(#)	The total number of network packets exchanged between the robot and remote PC during the mission execution.
Size of Network Packets	*megabytes*(MB)	The total size of network packets exchanged between the robot and remote PC during the mission execution.
Feature Extraction Time	*milliseconds*(ms)	The average time needed for the features to be extracted from a received image frame.
Object Detection Time	*milliseconds*(ms)	The average time needed for comparing the extracted features against images in the database to conclude if any of the objects in the database are recognized in a received image frame.
Detection Result Delay	*milliseconds*(ms)	The average delay of transferring object detection outcome from the object recognition node to a node that logs the recognition result on the robot.
Navigation Time	*milliseconds*(ms)	The average time it takes for a robot to navigate from its current location to a goal location.
Mission Execution Time	*milliseconds*(ms)	The total duration of the mission.
Power Consumption	milliWatts (mW)	The instantaneous power consumption sampled at 200Hz during the entire mission execution. The energy consumption (in Joules) is then calculated as the integral of power consumption (mW) over the mission execution time.

3.3.1 Main Factors

In this study, we consider three main factors as offloading candidates, representing the three main robotic tasks of the mission used in this study (see Section 4.1:

SLAM/localization offloaded: defines if SLAM/localization is executed on-board or offloaded to a remote PC. This is a nominal variable with possible values *false* and *true*. SLAM/localization will be the subjects of offloading in two different setups of the experiment which will be further elaborated in Section 4;

Navigation offloaded: defines if navigation is executed on-board or offloaded to a remote PC. This is a nominal variable with possible values *false* and *true*;

Object recognition offloaded: defines if object recognition is executed on-board or offloaded to a remote PC. This is a nominal variable with possible values *false* and *true*.

Each robotic task can be operated under different configurations, each with different internal parameters (*e.g.,* the image frame rate for object recognition, the number of particles used by the SLAM algorithm). The values of those parameters refer to two main configurations of the Object Recognition, SLAM, and Navigation factors, namely: (i) the default values of the involved tasks (*e.g.,* 20fps for the object recognition task) and (ii) the maximum values supported by the used hardware (*e.g.,* 60fps for the object recognition task). This leads to the three different configurations shown in Table 2.

Configurations 1 and 2 are the ones used in the first round of experiments (necessary to answer RQ1). The only difference between them is the value of the map parameter, which is *known* for Configuration 1 and *unknown* for Configuration 2. In *Configuration 1*, the map is provided by a map server (which we call *known*), and therefore we do not run SLAM to map the environment. Instead, we still run a localization algorithm, which is then the offloading candidate instead of SLAM. In contrast, in *Configuration 2*, SLAM is used to map the environment and is considered as the offloading candidate. Thus, besides evaluating the impact of offloading each of the tasks, we are also able to evaluate how an environment mapping approach influences the dependent variables. The other factors are set with minimal parameter values that allow the robot to perform its tasks in the mission, and the execution time is fixed with the necessary time for the robot to perform a complete mission.

Configuration 3 has the highest possible values for each parameter. In contrast to previous configurations, which aim at evaluating the effects of offloading, *Configuration 3* aims at estimating the impact of re-configuring some offloaded task parameters. *Configuration 3* is considered in the second round of experiments (necessary to answer RQ2).

Table 2 Parameters under experimentation and their values

Factor	Parameter	Conf. 1	Conf. 2	Conf. 3
Object Recognition	*image resolution*	640x480px	640x480px	1280x960px
	image frame rate	20fps	20fps	60fps
SLAM	*number of particles*	–	5	30
	temporal updates	–	off	on
Navigation	*velocity samples*	10x20	10x20	20x40
	simulation time	1.5s	1.5s	3s
	map	*known*	*unknown*	*unknown*

The two parameters analyzed with respect to object recognition are *image resolution* and *image frame rate*. The image resolutions correspond to 480p (640x480 pixels) and 960p (1280x960 pixels) in a 4:3 aspect ratio. The 4:3 aspect ratio is known for allowing to capture more content in a picture/frame [9]. On the other hand, 480p is the upper resolution for standard definition (SD), while 960p is the upper high definition (HD) resolution supported by the camera device. Despite 20 frames-per-second (FPS) being a bit lower than the 24 FPS indicated as the lowest level for good realism[10], it is still acceptable because in our mission we only consider static images. With these parameters, we want to inspect the magnitude of the effect that the increased image resolution and frame rate, respectively, have primarily on communication intensity[11], but also on resource utilization on the robot side.

The parameters under experimentation regarding SLAM are *number of particles* and *temporal updates*. As reported by (Abdelrasoul et al. 2016), the increase in the number of particles in RBPF filter in *gmapping* has a significant effect on both CPU usage and RAM utilization. We want to confirm if this is indeed the case, and thus we increase the number of particles from 5 (the minimum necessary for an acceptable robot location estimation) to 30 (the value that allows a precise robot location estimation). While (Abdelrasoul et al. 2016; Putra et al. 2019) report the effect of linear and angular updates, *temporal updates* remain unexplored. This is the time interval for laser scan processing in *gmapping*, regardless of the robot's movements. We chose to run our experiment with it turned off (default or value equals -1) or on (value equals 0.5).

Finally, the number of translation and rotation *velocity samples* in the DWA approach of the navigation stack local planner plugin, as well as *simulation time*, are listed in the tuning guides as influential with respect to computation intensity (Zheng 2017; ROS wiki 2021). To confirm this, in *Configuration 3*, we set them as twice the values of the first configurations.

4 Experiment Setup

In this section we describe the robotic mission performed in this experiment, the metric measurement process, and the experiment orchestration. We also define the experimental hypothesis and data analysis procedure. Note that we provide additional details on the hardware and software used in Section 4.5.2.

4.1 Robotic Mission And Laboratory Arena

In this study we consider a single terrestrial robot called Turtlebot3 Burger[12]. The Turtlebot robot is a widely-known research and education ground robot and a recent systematic mapping study on ROS confirmed that Turtlebot is the most used robotic platform for carrying out software engineering research (Albonico et al. 2023). Moreover, the Turtlebot is open-source (both hardware and software) and thus highly customizable, affordable, available worldwide,

[9] https://www.makeuseof.com/4-3-vs-16-9-aspect-ratio-photos-videos/

[10] https://www.redsharknews.com/technology-computing/item/3881-why-24-frames-per-second-is-still-the-gold-standard-for-film

[11] We anticipate that due to technical ROS-specific configurations used in this experiment, the total number and size of network packets exchanged are not directly impacted by the higher image resolution and image frame rate (more details about this aspect of our experiment are provided in Sections 7.1 and 7.2).

[12] https://www.turtlebot.com

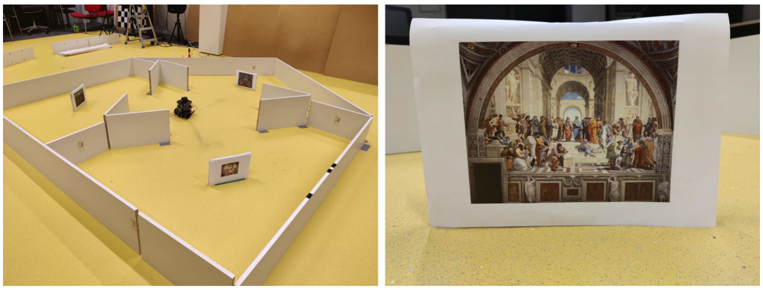

(a) Robot at the starting position. (b) One of the paintings.

Fig. 4 Robot arena where the mission is executed

small-sized, and designed to be fully compatible with ROS; all together those characteristics allow independent researchers to easily replicate/reproduce our study.

We hypothetically name our robot *Sherlock* (given its investigative nature), and it is located in a *gallery* where some of the most famous paintings in the world are stored. Sherlock has knowledge about the 50 most famous paintings of all time[13] in its database. Three randomly-chosen paintings are exhibited in separate rooms of the gallery. Sherlock's task is to autonomously navigate from its starting position, in the center of the gallery, to each of the rooms and recognize which painting is exhibited. As we will further report in the remainder of this paper, such behavior allows the robot to go through different paths, and recognize distinct paintings.

Despite the hypothetical nature of our mission, our robot uses exactly the same algorithms and software packages which are widely-used in real-world robotic systems [14], such as the `navigation` (Marder-Eppstein et al. 2010), `gmapping`[15], and the `find_object_2d`[16] ROS packages. During its mission, Sherlock is able to map the environment (*mapping*), mainly by using its LIDAR sensor, estimate its position on the map (*localization*), trace a route, and navigate to various destinations (*navigation*), while recognizing objects during the execution of the whole mission (*object recognition*). To keep the experiment under control and reduce the number of variables, we choose to avoid dynamic elements, such as other objects crossing the robot's way, and use pictures to mimic object recognition.

Figure 4a represents the arena used as a gallery in the experiment. The gallery is a pentagon whose sides are 3x2x2x2x2m long. Three randomly chosen paintings are printed in high resolution on A4 size paper and they are all of equal size. Paintings are placed in the corners of a triangle whose sides are equal (2m) and the robot's starting position is in the center of the triangle. From that position, the robot traverses a 0.8m distance to each of the rooms and inspects the exhibited painting, where it stops for 1s and returns to its starting position. All three rooms are visited two times each, which yields 2.5 minutes of the total mission execution time. Because the navigation is performed autonomously, the total duration of the mission depends on how global and local paths are chosen by the navigation stack. It is

[13] https://listsurge.com/top-50-most-famous-paintings

[14] https://discourse.ros.org/t/2022-ros-metrics-report/29594

[15] http://wiki.ros.org/gmapping

[16] http://wiki.ros.org/find_object_2d

noticed that this time is approximately 2.5 minutes for most of the runs, with some slight variations. The video of mission execution is also available[17].

4.1.1 Experiment Orchestration

All the experiment execution is orchestrated by **Robot Runner**, a tool for setting up the execution of measurement-based experiments involving robotics software (Swanborn and Malavolta 2001). Robot Runner executes on a PC and interacts with the ROS system under experimentation by launching the files that represent the system itself via the *roslaunch* command[18]. As the files need to be launched from the Raspberry Pi, we first establish the SSH connection between the PC and the Raspberry Pi via the *paramiko*[19] Python library, upon which the *roslaunch* commands are executed for the respective launch files.

In addition to relying on the ROS system stack, we also developed six Robot Runner custom plugins for measurement/profiling, namely *profilers*:

Resource Profiler: a ROS note for measuring CPU usage and Memory utilization.

Network Profiler: a plugin for measuring network activity.

Time-based Profilers are the two plugins used to measure time-based metrics, namely *move_base profiler* which measures navigation and experiment time, and *find_object_2D profiler* which measures time-based metrics of the object recognition package/task.

All the profilers are further explained in the next section. In Section 4.3 we also give a complete overview of the Robot Runner configuration for the experiment executions.

4.2 Profilers

In this section we explain in detail the implementation and execution details of the profilers we use for collecting the measures in this study (see Table 1).

4.2.1 Resource Profiler

We implemented a dedicated ROS node for profiling CPU usage and memory utilization. The ROS node provides dedicated ROS services for starting/stopping the CPU usage and RAM utilization measurement. These ROS node is booted by a *ROS launch file*, while the services are started and stopped remotely by a *ROS service client* running inside the *experiment orchestrator* (*i.e.,* Robot Runner). The whole measurement process is performed without sending any data throughout the network to avoid extra traffic; the collected measures are returned to the *ROS service client* only when the measurement collection is stopped. The measures are collected via the *psutil* Python package[20], which allowed us to collect measures at a higher frequency than other tools, such as *Linux top*. The CPU and Memory measures are collected with a frequency of 200Hz. The CPU usage and Memory utilization are measured globally, without monitoring a specific process. The ROS launch file used to bring up the *Resource Profiler* ROS node allows the user to programmatically set the sampling frequency before bringing up its ROS node. The source code of the Resource Profiler is available online[21].

[17] https://youtu.be/S5rqGRa1qMQ

[18] http://wiki.ros.org/roslaunch

[19] http://www.paramiko.org/

[20] https://pypi.org/project/psutil/

[21] https://github.com/IntelAgir-Research-Group/ros_melodic_profilers

4.2.2 Power Profiler

The Power Profiler is also implemented as a ROS node, which works similarly to the CPU and Memory Profiler, being launched remotely and only sending the measurement data to the ROS service client at the end, with a sampling reate of *200Hz*.

The Power Profiler is assembled and programmed on top of the prototyping board *Arduino Nano*[22]. The device is customized for the purpose of the experimentation. All the communication between the Turtlebot and the device passes through Arduino's serial port, without using ROS communication. We use the communication via the serial port in order to do not interfere with the network conditions during the experiment. Therefore, we count on an external device that measures all the energy that goes from the battery to the robot. The device validation is done by comparing its results with the ones from a professional hardware multimeter and their collected measures are in line with those of the multimeter. The power consumption measurement is done by the INA219 Adafruit sensor[23], the top-ranked power sensor at TinyTronics[24] at the time of writing. More specifically, the INA219 documentation reports that the device has 99% precision, and can measure potential energy/voltage (V, *volts*), current (A, *amperes*), and power (mW, *milliWatts*) at a speed that can reach *400kHz*. We decided to keep our sampling rate at 200Hz since in preliminary runs of the experiment we noticed that too frequent readings could influence the experiment results. All the measured data is stored in a local Secure Digital (SD) card, also attached to the Arduino. Despite the SD card configuring a bottleneck in the measurement rate, it still allows us to write at a speed of ≈*10MBs*, which fits our purpose of millisecond-spaced measurement.

4.2.3 Network Profiler

For the network, we first make sure that all ROS-based communication takes place within an independent WiFi network. Then, we sniffed all the packages between the remote PC and Turtlebot interfaces.

We instrument the *Network Profiler* with the *pyshark* [25] Python package (based on the *Wireshark* [26] tool), a well-known package analyzer. This is run as a Robot Runner plugin because, differently from the Resource and Power Profilers, it runs in the remote PC and does not need to be isolated. The information about the network protocol, sender and receiver IP addresses and ports, and the size of packets is recorded when measurement collection is taking place.

4.2.4 Time-Based Profiler

The *Time-based Profiler* is run offline, extracting data from log files. It is implemented with a main component, the *Abstract Profiler*, which serves as a base for other profilers that process log files from ROS nodes, *i.e.,* finding and fetching log files of a particular node, either local or remote. When the node whose log file is requested executes remotely, the file is fetched via *Secure File Transfer Protocol* (SFTP).

[22] https://docs.arduino.cc/hardware/nano

[23] https://www.adafruit.com/product/904

[24] https://www.tinytronics.nl/

[25] https://github.com/KimiNewt/pyshark

[26] https://www.wireshark.org/

The **move_base profiler** processes the log files of *move_base* and *sherlock_controller* and parses information regarding navigation times in a single run of the experiment.

The **find_object_2D profiler** processes the log files of *find_object_2d* and *sherlock_obj_recognition* and parses information regarding feature extraction times, object detection times, and detection result delays in a single run of the experiment.

4.3 Experiment Orchestration Setup

Apart from the two nodes that are added for measurement collection, *i.e., Resource Profiler* and *Power Profiler*, the robotic system under experimentation is completely independent of the Robot Runner. Therefore, these nodes execute on-board the robot, on the Raspberry Pi, as CPU, RAM, and battery (as the subject of the measurement) are located on the robot itself. We made the decision to start and stop measurements via service calls, in a client-server fashion, to avoid transmitting outputs of the measurement collection process during the mission execution, as that would influence the measured network traffic exchange, but it would also account for certain CPU, RAM, and energy utilization. With the client-server communication model, the entire results are transmitted from the robot to the Robot Runner after the experiment mission ends.

The experiment is orchestrated by Robot Runner events according to the graphical flow depicted in Fig. 5, where each event and its purpose are:

• **Start Run** event is called before the measurements are started; thus we use this opportunity to clean up the environment and launch all the nodes that require a stabilization period. For instance, it takes 10 seconds to start up and calibrate the laser scanner, and a few seconds to calibrate the camera. Then, we start these nodes prior to the mission execution and ensure that calibration is done.

• **Start Measurement** event is called right before mission execution to start the measurement profilers.

• **Launch Mission** event launches the files regarding the experiment setup. Further parameters, such as offloading configuration, are passed by the *roslaunch* command as parameters (see Section 5.3).

• **Stop Measurement** event stops measurement profilers. The measurements collected by each profiler are received and stored in their respective CSV files within the folder of the current run. We wait until all the data is transferred, which may take 1 minute.

• **Stop Run** event terminates all the nodes that were launched in *start run*. This is done to ensure that no ROS node is running in between the two runs and that the robot's resources can cool down before the new run starts.

• **Populate Run Data** event aggregates or sums the results from the CSV files of the respective profilers, and stores it as a Robot Runner run table row that abstracts the previously executed run.

4.4 Experiment Execution

As explained in Section 3.3.1, the experiment is full-factorial, which results in multiple configurations. Each configuration consists of a single execution, where executions are completely randomized so that we mitigate any potential influence of system conditions that are out of our control, such as battery level and warm motors. To mitigate any potential voltage differences due to battery level, after each run, the battery is replaced with a fully charged one. For that purpose, three batteries were always charging at the station, while the fourth

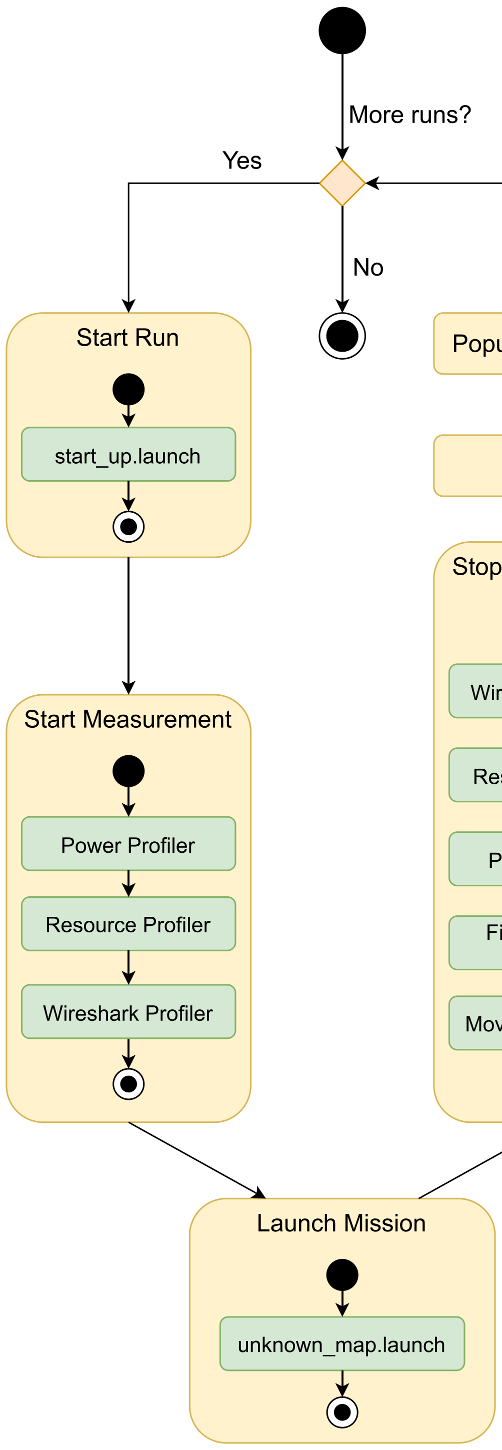


Fig. 5 Orchestration flow

one was used in the current run. We also add a 1-minute delay between run executions so that system resources can cool down.

The experiment is conducted in two rounds: the *first round* provides data for *RQ1a* and *RQ2a* observations, while the *second round* provides extra data that can be used to answer *RQ1b* and *RQ2b*.

While the pure mission execution time is estimated to be 2.5 minutes (see Section 4.1), with additional time for stabilization and data transfer and processing, we have an estimation of 5 minutes per run in total, which sums up to 23 hours with the repetitions of the *first* and *second* rounds. We did not opt for longer missions in order to keep feasibility purposes. Indeed, we had one researcher (i) continuously monitoring the telemetry of the robot and the behavior of the robot itself during each run of the experiment and (ii) replacing the battery at the beginning of each run with a fully-charged one (in order to always have a fully-charged battery during the mission execution – this is relevant for being able to compare the energy measures collected across runs).

4.4.1 First Round

The experiments for both *Unknown map* and *Known map* setups are designed with three independent variables, each having two possible values (see *Configurations 1* and *2* in Section 3.3.1). This yields $2^3 = 8$ possible treatments, with each treatment being repeated 10 times, resulting in 80 runs in total. The 80 runs yield 6.5 hours of experimentation for *Unknown map* and *Known map* setup, respectively, for 13 hours in total.

4.4.2 Second Round

We perform the additional round of experiments only with the *Unknown map* setup, for keeping the experiment within a reasonable duration (see *Configuration 3* in Section 3.3.1). The decision is justified by the fact that the results for both setups were fairly similar in the primary experiment. In addition, the primary experiment results have shown that the benefits of object recognition offloading, with respect to both energy consumption and performance, are greater when compared to both SLAM and navigation. For that reason, we decided to perform parameter effect analysis with SLAM and navigation executed onboard and only object recognition offloaded.

We decided to perform an additional set of experiments, rather than introducing parameters as factors in the primary experiment, for three reasons. First, the number of treatment combinations in full-factorial design would explode. Second, the data analysis would be cumbersome if the experiment has a large number of main factors because the factorial ANOVA test with more than three factors is rarely performed Kassambara (2021). Third, it would be unfeasible to conduct the experiment within a reasonable time.

The dependent variables remain the same as those used in the primary experiment. In the experiments, each of the parameters has two different values and we opt for 10 repetitions per treatment. This yields six additional experiments, with $2 \cdot 10 = 20$ runs per experiment. The mission, system design, and experiment infrastructure remain the same; the only varying factor is the configuration of the nodes under experimentation. The duration per run remains 5 minutes, hence the total duration of all six experiments yields $6 \cdot 20 \cdot 5 = 600$ minutes or 10 hours of experimentation.

4.5 Data Analysis

We conduct distinct data analyses according to the specificities of each round of the experiment. Both are explained in sequence.

4.5.1 First Round Analysis

Because there are three independent variables defined in the experiment, we opt for three-way ANOVA (Lars et al 1989) to compare differences in means as main effects and inspect the existence of significant two-way and/or three-way interactions.

However, there are three conditions that need to be fulfilled prior to the three-way ANOVA execution (Kassambara 2021). The *first assumption* of having independent observations is met inherently according to the way the experiment itself is conducted. Runs are executed randomly and without dependencies with previous and upcoming runs. The *second assumption* states that there should be no extreme outliers, otherwise such observations should be removed. The *third assumption* implies normal distribution of observations. In Kassambara's guide for ANOVA test (Kassambara 2021), it is stated that the normality assumption can be checked in two ways: i) normality is satisfied if the residuals are normally distributed, or ii) check the normal distribution for each of the treatment combination groups. However, it is stated that the residuals normality approach is more suitable for a relatively low number of samples per sample group, which is the case in this experiment (10 samples for each treatment combination in both setups). Nevertheless, we opted for both approaches to analyze if they would lead to different conclusions. In both approaches, the normality assumption is checked via QQ-plot analysis and complemented with Shapiro-Wilk test. The third and final assumption refers to homogeneity of variances (*i.e.,* homoscedasticity), which is checked with Levene's test (Schultz et al. 1985).

In the data analysis process, we may get into situations where one or more assumptions for three-way ANOVA test execution are not fulfilled. In such cases, we turn to a non-parametric alternative to factorial ANOVA, namely, the permutation test (Anderson and Braak 2023). However, ANOVA is considered robust to the violation of its assumptions when the sample sizes per group are all of the equal size, which is indeed the case in this experiment (Ito 1980). For this reason, we opt to perform both permutation tests and three-way ANOVA when the assumptions are violated and compare the obtained results. Indeed, the conclusions from both of the tests performed are always slightly different in the p-values obtained.

We choose the 95% confidence interval and thus we consider the obtained p-values significant when they are lower than a 0.05 threshold. This is the default confidence interval in R, the most widely used by researchers. In cases when three-way interaction is considered significant, we computed simple two-way interactions with the Bonferroni adjustment (Bonferroni 1963). The significant simple two-way interactions are further analyzed with simple main effects, followed by the pairwise comparison with *estimated marginal means* to identify statistically significant groups. In cases when three-way interaction is insignificant, we performed pairwise comparisons with the *estimated marginal means* (EMM) test for any two-way interaction to compare the means among different groups.

4.5.2 Second Round Analysis

Because each of the experiments has only one independent variable, *i.e.,* the parameter under experimentation, with two treatments, we opt for Welch's t-test (Welch 1947) for comparing the difference between the means of the two treatments. We choose Welch's over Student's t-test because we do not want to assume anything about the equality of variances of the samples among the two treatments; hence we selected the more robust test.

However, Welch's t-test also implies a number of assumptions (Kassambara 2019). The first assumption of having independent observations, *i.e.,* each subject belongs to only one group and there are no relationships between observations within the two groups, is satisfied inherently by the way in which the experiment is conducted. Similar to ANOVA, there should be no significant outliers and the samples for both treatments should be normally distributed. Normality is again established with QQ-plot analysis combined with the Shapiro-Wilk test.

5 Hardware And Software Configuration

In this section we provide detail about the hardware used and each software setup for the experiment.

5.1 Hardware Components

As already anticipated, in this experiment we use a Turtlebot3 Burger, which is a widely-used robotic platform in robotics software engineering studies Albonico et al. (2023). The Turtlebot version we use in the experiment has the following configuration:

- 1 sensor *LDS-01 (HLS-LFCD2)* capable of sensing 360 degrees, attached to the top part of the robot, as designed by ROBOTIS;
- 1 board *OpenCR1.0* equipped with the energy-efficient 32-bit ARM Cortex-M7 processor with floating point unit, which works as the main controller of the robot;
- 1 *Raspberry Pi 4 Model B* SBC with 8GB of RAM running Ubuntu 18.04.5, with the ROS system;
- 1 *Raspberry Pi Camera Module v2*, with an 8mp camera module capable of taking 3280x2464 pixel static images.
- 1 *Arduino Nano Every* micro-controller board connected to the Raspberry Pi. It is used to control and program the current sensor attached to it.
- 1 *Adafruit INA219* current sensor connected to the Arduino board that measures high side voltage and DC current draw.

The other hardware used in the experiment are:

- 1 laptop **Lenovo Legion Y540** equipped with Intel Core I7-9750H processor, 16GB of RAM and runs Ubuntu 18.04.5, which works as the remote PC where the tasks are offloaded, but it is also used for experiment orchestration.
- 1 WiFi router **Netgear R6220** used to connect the robot and the remote PC in a local, isolated network. Its speed goes up to 1.2Gbps and it has a 5GHz channel to which devices were connected during the experiment.

5.2 Software Design

As mentioned in Section 3.3.1, there are two broad types of navigation approaches, namely, *map-less* and *map-based* (Hu et al. 2012). These two variations of map-based navigation approaches are the subjects of the experiment.

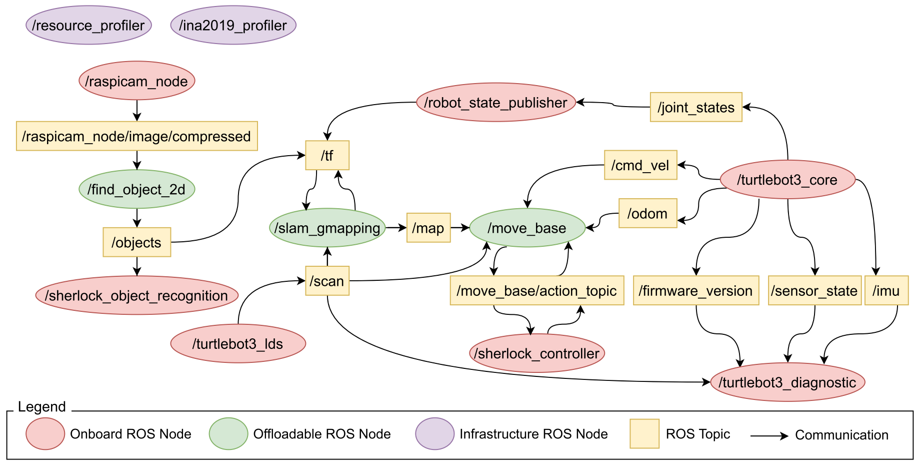


Fig. 6 Computation graph of the *Unknown map* experiment setup

In the first system setup, the offloadable tasks are *SLAM*, *navigation*, and *object recognition*, referred to as *Unknown map* setup in this study because the map of the environment is not known in advance, yet constructed simultaneously while the robot navigates through the unknown environment. In the second system setup, referred to as *Known map* setup, the offloadable tasks are *localization*, *navigation*, and *object recognition*, where localization is necessary for navigation even though the mapping process is already performed. There are two reasons for considering SLAM/Localization, Navigation, and Object Recognition as offloadable components in our experiment:

– They implement tasks that are known to be computationally expensive in robotic systems (Dey and Mukherjee 2016; Lee et al. 2017; Indelman et al. 2012);
– The other potentially-expensive components in terms of computation either cannot be offloaded to the Cloud since they have a strong dependency to the hardware of the robot (*e.g.,* the Turtlebot controller or the driver of the camera) or they are meaningful only when running in the Cloud (*e.g.,* the robot diagnostic receiver, which is used for telemetry visualization).

In this section, we explain the design of the system from the perspective of the offloadable tasks and their relationships with the rest of the system. For such a relationship, we consider computation graphs, common in the ROS environment, which show ROS nodes and their communication over topics. Figures 6 and 7 depict the computation graphs of the system under experimentation for both, *Unknown map* and *Known map* setup. The images represent the graphical notation created with *rqt* [27], a *Graphical User Interface* (GUI) development framework in ROS, but with customized colors, where green nodes are offloading candidates.

5.2.1 SLAM (slam_gmapping node)

This node is responsible for the **SLAM** layer. It processes the incoming laser scans from the *scan* ROS-topic and, as a result, periodically sends occupancy grid messages to the *map* ROS-topic. The occupancy grid is a message format that represents the map of the environment in

[27] http://wiki.ros.org/rqt

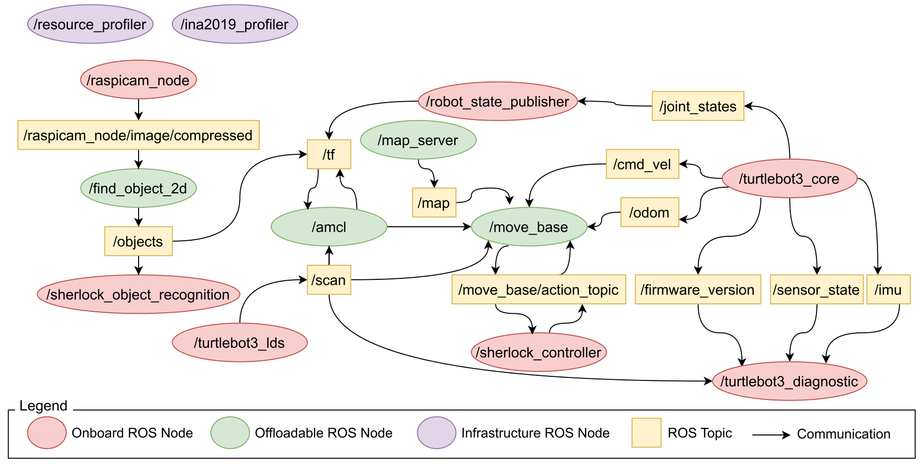


Fig. 7 Computation graph of the *Unknown map* experiment setup

the form of an evenly spaced field of binary variables. While SLAM is indeed a computation-intensive task, the scans coming from the LiDAR sensor (on the robot) and relatively large and frequent occupancy grid messages may lead to a high communication load if SLAM is offloaded.

A robotic system typically consists of many 3D coordinate frames, each referencing different parts of the robot and its sensors. For instance, sensor readings are referenced from the coordinate frame attached to the sensor itself. If the robot needs to take any action based on the sensor readings, it is important to know, at all times, the relative position of the frame attached to the sensor to the frame attached to the robot's base. In ROS, all transformations between the coordinate frames are managed via the *tf* package (Foote et al. 2021). It provides a dedicated ROS-topic called *tf* where relative positions, i.e., *transformations*, between coordinates frames of the robotic system are periodically published.

5.2.2 Navigation (move_base node)

This node implements the **navigation layer** in both, *Unknown map* and *Known map* setups. A custom node (*sherlock_controller*) orchestrates the mission and sends goal locations to the *move_base* node. The navigation stack then finds a path to a goal location and publishes velocity commands over the *cmd_vel* ROS-topic to the *turtlebot3_core* node so that the robot starts moving according to the planned trajectory.

The navigation stack requires laser scan readings over the *scan* ROS-topic so that dynamic obstacles can be cleared out or added to the map. Furthermore, this task also requires odometry information, which is transferred over the *odom* ROS-topic from the *turtlebot3_core* node. As both laser scans and odometry are always published from the robot, the network overhead could be very high when navigation is offloaded.

The map of the environment is provided over the *map* ROS-topic, either by the *slam_gmapping* or *map_server* node in the *Unknown map* and *Known map* setup, respectively. The navigation stack uses the map to find the path with no obstacles between the robot's current location and the goal. In the *Known map* setup, the map is provided only once, as it will not be changed during the mission. However, in the *Unknown map* setup, the map is periodically published as the output of SLAM, meaning that there is a substantial communication dependency between SLAM and navigation. Therefore, we may expect reduced network overhead when both SLAM and navigation are offloaded simultaneously, as the maps will not be exchanged over the network between the two nodes when they run on the same machine.

Like SLAM and localisation, the navigation stack also heavily uses transformations. In addition to the two aforementioned transformations required by *slam_gmapping* and *amcl*, it requires the *map* → *odom* frame transformation, provided by these two nodes in their respective setups. As discussed in Section 2.2.3, numerous plugins within the navigation stack indicate that the *move_base* node is indeed computation-intensive. While this makes this task a good candidate for computation offloading, we can conclude that laser scans, odometry, and frame transformations potentially impose a high communication load when it is offloaded.

5.2.3 Object Recognition (find_object_2d node)

This node represents the object recognition artifacts and is completely independent of the other offloadable tasks in both setups. It receives camera images from the *raspicam_node* (the driver for the Raspberry Pi Camera Module) over the *raspicam_node/image/compressed* ROS-topic. Unfortunately, the driver is not available for the ROS Melodic distribution, which is used in this experiment, as it depends on the Multimedia Abstraction Layer (*mmal*) library that is not available in Ubuntu 18.04 for the amd64 architecture. We solved this issue by compiling the *mmal* library from the source and configuring it manually to enable *raspicam_node*. The latest commits to the library source still break the amd64 support, so we compiled the library from the commit that is used in Ubuntu 20.04, where the *mmal* library for this architecture is available.

As a result of object recognition process, the *find_object_2d* node publishes messages on the *objects* ROS-topic. Such messages contain information regarding the IDs of the objects that are recognized and their positions within the received image frame. A custom node (*sherlock_object_recognition*) receives the messages over this ROS-topic and logs the results of object recognition. Apart from indisputable computation intensity, image frames that have to be transferred from the robot's camera sensor indicate very intense communication if object recognition is offloaded.

5.2.4 Localisation (amcl and map_server nodes)

The *amcl* node encapsulates the localisation task, while the *map_server* provides the already known maps, a large difference from the *Unknown map* graph. The *amcl* node relationships with other nodes, as well as required and provided transformations, are almost identical to the ones of the *slam_gmapping*. Therefore, the implications of localisation offloading are fairly similar to those previously discussed for SLAM. The only difference is that the map of the environment is now the input to the *amcl* node. Because the map is already created and it will not be changed during the mission, the *map_server* node publishes the map over the *map* ROS-topic to the *amcl* node in a one-time communication. The lack of the mapping process results in a significantly lower exchange of messages over the *map* ROS-topic with localisation, as compared to SLAM, which potentially makes this task more suitable for offloading with respect to network overhead. In case of offloading, both nodes must be on the same device.

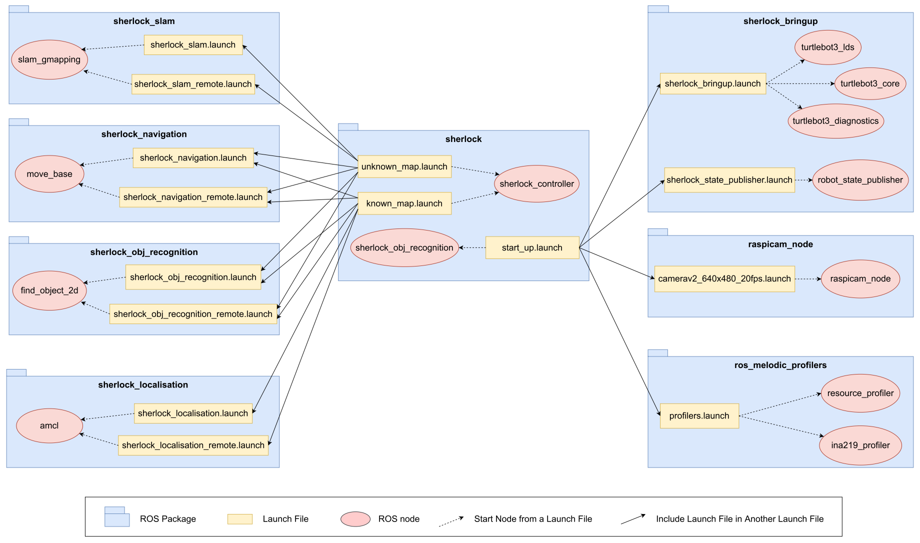


Fig. 8 Ros Package Organization Of The System

5.3 Additional Software Packages

ROS offers a reusable abstraction called *package* for the modular organization of the robotic software. Source code of ROS nodes, parameter configurations structured in *Yet Another Markup Language* (YAML) or other file formats, but also message and service definitions, are contained inside ROS packages.

ROS also offers a powerful *launch file* mechanism, a file format based on *Extensible Markup Language* (XML), that primarily serves for running multiple ROS nodes at once. Because ROS-based systems consist of many ROS nodes that communicate with each other, it is very convenient to configure all nodes in a single place and run them via a single *roslaunch* command. Launch files are also contained inside ROS packages. They can be customized with parameters that are passed in the *roslaunch* command.

The ROS package organization of the system under experimentation, along with the contained ROS nodes and launch files, is presented in Fig. 8. The notation is an adaption of the *Unified Modeling Language* (UML) package diagram, adapted to depict the modular organization of ROS packages. The organization conforms to the design principle of *separation of concerns* (SoC), where each ROS package organizes a dedicated functional unit of the system. System packages and their purpose are presented in the following:

• **sherlock** package[28] represents the entire system as a whole and it is positioned the highest in the package hierarchy. It contains the source code of two ROS nodes, namely, *sherlock_controller* and *sherlock_obj_recognition*. *sherlock_controller* node orchestrates the robot's movements during the mission as it sends goal locations to the *move_base* node via the ROS action communication model. *sherlock_obj_recognition* node receives the results of object recognition in a publish-subscribe fashion, over the ROS topic, and logs the results. The package also contains three launch files. The *start_up.launch* file that initiates other TutleBot3 specific nodes. The other two launch files, namely, *unknown_map.launch* and *known_map.launch*, include and launch the files that configure and run nodes that are subject

[28] https://github.com/IntelAgir-Research-Group/sherlock

of computation offloading in the *Unknown map* and *Known map* setup, respectfully. These launch files are configured with parameters that indicate if each of the nodes is executed on-board or offloaded.

• **sherlock_bringup** package[29] contains the *sherlock_bringup.launch* file that configures and runs TurtleBot3-specific nodes. These include node *turtlobot3_core*, which interacts with the external commands and transmits them to the robot's firmer, *turtlebot3_lds*, the driver for LiDAR sensor, and *turtlebot3_diagnostics*, a node for the robot's state diagnostics. The package contains file, which runs and configures the *robot_state_publisher* node, which publishes static frame transformation based on the description of TurtleBot3 robot's links and joints, also known as the robot's *model*. The description of the TurtleBot3 model is expressed in XML-based *Unified Robot Description Format* (URDF).

• **raspicam_node** package[30] contains the source code of the ROS driver for the Raspberry Pi Camera Module, namely, *raspicam_node*. The node is configured with a resolution of 640x480px and 20fps frame rate, calibration file, and other parameters, and launched within.

• **sherlock_slam** package[31] contains two launch files, namely, and *sherlock_slam_remote.launch*, for configuring and running the *slam_gmapping* node locally and on the remote machine, respectively. The remote machine, where the node should be executed, is passed as a parameter to the *machine* attribute of the *node* tag in the *sherlock_slam_remote.launch* file (ROS wiki 2021). The two files will be launched when SLAM is executed on board and offloaded, respectively.

• **sherlock_navigation** package [32] contains two launch files, namely, *sherlock_navigation.launch* and *sherlock_navigation_remote.launch*, for configuring and running the *move_base* node locally and on remote machine, respectively. The two files will be launched when navigation is executed on-board and offloaded, respectively.

• **sherlock_obj_recognition** package[33] contains two launch files, namely, *sherlock_obj_recognition.launch* and *sherlock_obj_recognition_remote.launch*, for configuring and running *find_object_2d* node locally and on remote machine, respectively. The two files will be launched when object recognition is executed on board and offloaded, respectively.

• **sherlock_localisation** package[34] contains two launch files, namely, *sherlock_localisation.launch* and *sherlock_localisation_remote.launch*, for configuring and running *amcl* node locally and on remote machine, respectively. The two files will be launched when localisation is executed on board and offloaded, respectively.

• **ros_melodic_profilers** package[35] contains source code of two nodes that serve for CPU usage, RAM utilization, and power consumption measurement collection: *resource_profiler* and *ina219_profiler*. Both nodes are configured and launched in the *profilers.launch* file that resides in the same package. *resource_profiler* node exposes two ROS services, namely, for initiating the CPU usage and RAM utilization measurements, and for termination of measurement collection, where the measurement outputs are returned as a result of the service call. CPU usage and RAM are sampled with the *psutil*[36] Python library at 50Hz. *ina219_profiler* node also exposes two ROS services, for initiating and terminating power consumption mea-

[29] https://github.com/minana96/sherlock_bringup

[30] https://github.com/minana96/raspicam_node

[31] https://github.com/minana96/sherlock_slam

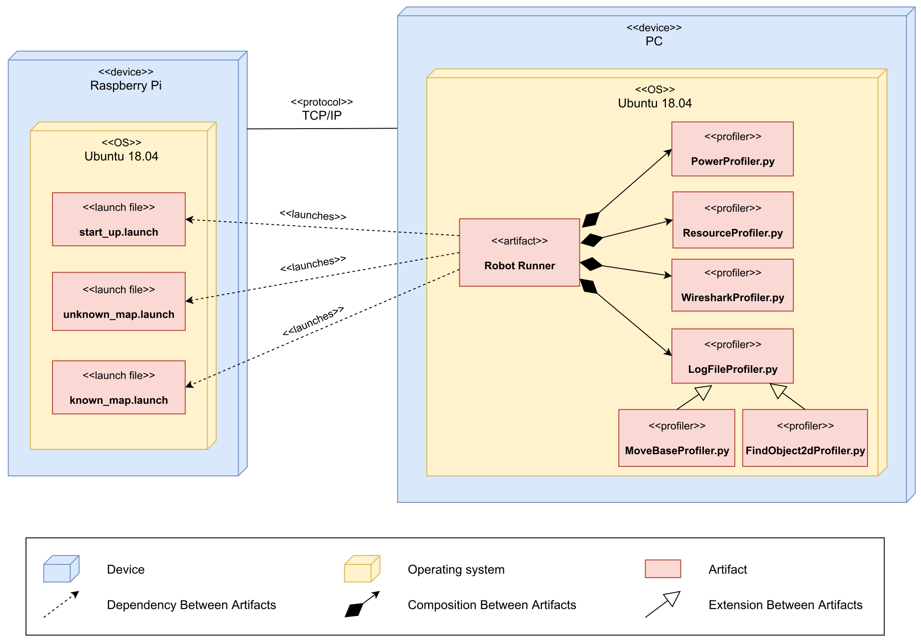
[32] https://github.com/minana96/sherlock_navigation

[33] https://github.com/minana96/sherlock_obj_recognition

[34] https://github.com/minana96/sherlock_localisation

[35] https://github.com/IntelAgir-Research-Group/ros_melodic_profilers

[36] https://psutil.readthedocs.io/en/latest/



<<device>> Raspberry Pi
<<OS>> Ubuntu 18.04
<<launch file>> start_up.launch
<<launch file>> unknown_map.launch
<<launch file>> known_map.launch
<<protocol>> TCP/IP
<<launches>>
<<launches>>
<<launches>>
<<device>> PC
<<OS>> Ubuntu 18.04
<<artifact>> Robot Runner
<<profiler>> PowerProfiler.py
<<profiler>> ResourceProfiler.py
<<profiler>> WiresharkProfiler.py
<<profiler>> LogFileProfiler.py
<<profiler>> MoveBaseProfiler.py
<<profiler>> FindObject2dProfiler.py
Device
Operating system
Artifact
Dependency Between Artifacts
Composition Between Artifacts
Extension Between Artifacts

Fig. 9 Deployment Diagram

surement collection. Upon receiving the service request for measurement initiation, the node sends a request to the Arduino board over a serial connection to start the power measurement process.

5.4 Software Deployment

The deployment diagram of the experiment infrastructure is presented in Fig. 9. The system is deployed on the Raspberry Pi (on board of TurtleBot3 robot) and on the PC. Both devices are connected to an isolated local network, and they run on the Ubuntu 18.04 operating system.

The time on both machines is synchronized via *chrony*[37], as advised in the network setup for distributed systems in ROS (ROS wiki 2021). The remote PC is configured as a *Network Time Protocol* (NTP) server, while the robot is the NTP client. This decision was made to ensure that the timestamps for the measurements on both machines are synchronized, but also to ensure that the current time on the robot will be updated in an isolated network, without an internet connection. Otherwise, this would not be possible because Raspberry PI does not possess a real-time clock. The PC also works as our Cloud environment, where candidate tasks are offloaded.

5.4.1 Ros System

The Raspberry Pi device launches the three main launch files from the **sherlock** package. The nodes started by those launch files are also running on the Raspberry Pi, with the exception of four tasks (SLAM, localisation, navigation, and object recognition), when they are subject

[37] https://chrony.tuxfamily.org/

to computation offloading. In such scenarios, the respective nodes are deployed on the PC via the *machine* attribute mechanism in the *node* tag of launch files, resulting in an SSH remote command. We opted for this ROS launch setting, as opposed to the simultaneous launching of two separate files on the Raspberry Pi and PC, mainly for maintainability. The tasks reside in a single launch file that is always launched on the Raspberry Pi, as opposed to maintaining two launch files on two different devices.

ROS needs a master node, called *roscore*, which orchestrates the other nodes. In this experiment, the ROS master node is running on the Raspberry Pi. This decision was made so that the entire system is completely independent of the PC when the offloadable tasks are executed onboard because only those nodes can be executed on the PC.

6 Results Of The First Round Of Experiments (RQ1)

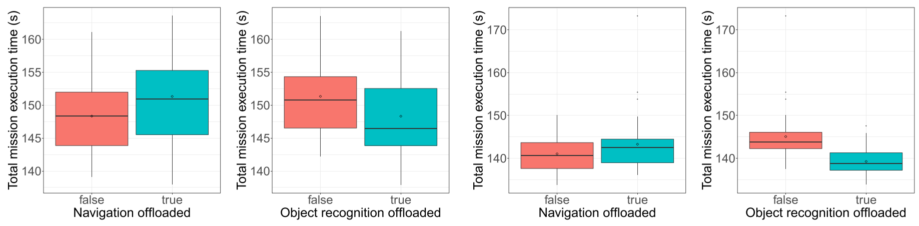
In this section, we will present the most important results of the performed statistical analyses and observations. The complete data analysis process, along with the verified assumptions for the performed statistical tests, and the obtained test results, are available in the replication package (Dordevic et al. 2022).

6.1 Mission Execution Time

The results for the total mission execution time are presented in Fig. 10. In the *Unknown map* setup results, there were neither significant three-way nor two-way interactions among the three factors. However, the results yielded a significant main effect of navigation and object recognition offloading, respectively, on the total mission execution time. When navigation is offloaded, the mean of mission execution time slightly increases from 148.338 to 151.320 seconds (Fig.10a). In contrast, offloading object recognition causes a decrease in total mission execution time, from 151.337 to 148.321 seconds, on average (Fig. 10b).

Very similar results are obtained in the *Known map* setup, where the mean mission execution time increased from 141.038 to 143.272 seconds when navigation is offloaded (Fig. 10c), but decreased from 145.062 to 139.248 seconds when object recognition is offloaded (Fig. 10d).

Because the total mission execution time is mostly dependent on how the navigation package finds and executes paths, it is expected that navigation offloading has an effect on this variable. Network delay in laser scan and command velocity message exchange between



(a) Offloading navigation in *Unknown map* setup (b) Offloading object recognition in *Unknown map* setup (c) Offloading navigation in *Known map* setup (d) Offloading object recognition in *Known map* setup

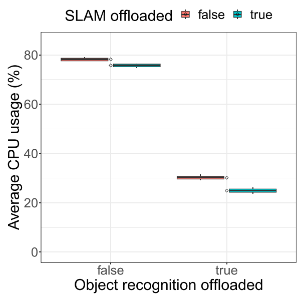
Fig. 10 The effect of computation offloading strategies on the total mission execution time

the robot and remote PC prolongs mission execution time when navigation is offloaded and the potential of more efficient remote execution does not compensate for the network delay. The effect of object recognition is, however, surprising at first, given that the object recognition task is completely independent of navigation. The vast amount of resources that object recognition utilizes when executed on-board slows down the execution of command velocities received from the navigation stack, thus the total mission execution time increases.

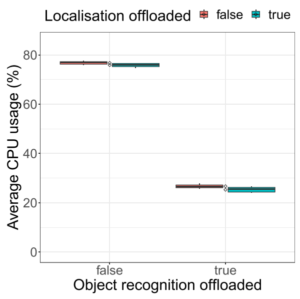
6.2 Cpu Usage

The results for the average CPU usage are presented in Fig. 11. The test results in the *Unknown map* setup yielded significant main effects for all three tasks, but there was also a significant two-way interaction between SLAM and object recognition. The pairwise comparisons have shown that offloading SLAM causes a statistical decrease in average CPU load when object recognition is both executed on-board (from 78.197% to 75.746%) and offloaded (from 30.176% to 24.933%) (Fig.11a). However, the estimates of effect size in pairwise comparisons show that the magnitude of the SLAM offloading effect is more than two times greater when object recognition is offloaded (2.451% difference in EMM, compared to 5.243%). This is expected, given that most of the CPU usage is used for object recognition when executed onboard. This is quite evident when we compare the two groups on the left and right of Fig. 11a.

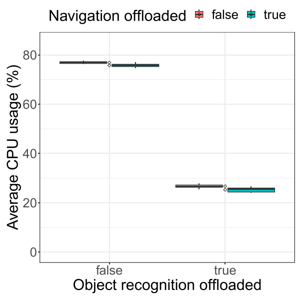
Very similar results are obtained in the *Known map* setup, with all three main effects being significant, but with two significant two-way interactions. When we performed a pairwise comparison for significant two-way interaction between localization and object recognition, we confirmed that the effect of localization offloading is significant when object recognition is both executed on-board and offloaded (Fig. 11b). However, the decrease in CPU usage caused by localization offloading is slightly larger when object recognition is offloaded (from 26.708% to 25.281%, and from 76.772% to 75.874%, when object recognition is offloaded and not, respectively). Similar results are confirmed for significant two way-interaction between navigation and object recognition (Fig. 11c), where the decrease from 26.713% to 25.276% when object recognition is offloaded is slightly greater when compared with 76.886% to 75.760% decrease when object recognition is executed on-board (estimates of 1.127% and 1.437% differences in EMM, respectively). Even though both,



(a) Offloading object recognition and SLAM in *Unknown map* setup



(b) Offloading Object recognition and localisation in *Known map* setup



(c) Offloading object recognition and navigation in *Known map* setup

Fig. 11 The effect of computation offloading strategies on average CPU usage

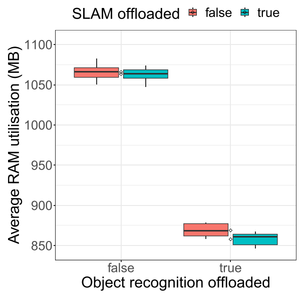
offloading localization and navigation, cause a statistically significant decrease in CPU load, we can conclude from both Figs. 11b and 11c that this decrease is almost insignificant when compared to the decrease caused by object recognition offloading.

We can conclude that object recognition accounts for much more significant CPU usage, as compared to other tasks. Therefore, offloading object recognition significantly reduces the CPU workload. Even though the results show that offloading other tasks, namely, SLAM, localization, and navigation, also results in more optimal CPU usage, the gains are negligible when compared to offloading object recognition. In fact, object recognition is so computationally intensive that the benefits of offloading localization and navigation in the *Known map* setup are not even statistically significant when object recognition runs on board. This is because object recognition reclaims the computation power that was used for localization and navigation once they are offloaded.

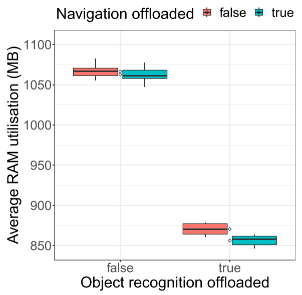
6.3 Memory Utilization

The results for the average RAM utilization are presented in Fig. 12. The test results in the *Unknown map* setup yielded significant main effects for all three tasks, but there were also two significant two-way interactions. The pairwise comparisons for significant two-way interaction between SLAM and navigation have shown that offloading SLAM causes a statistically significant decrease in average RAM utilization when object recognition is offloaded (from 868.939 MB to 857.226 MB), but not when object recognition is executed on-board (Fig. 12a). Similar results are obtained for significant two-way interaction between navigation and object recognition, yielding a statistically significant decrease when navigation is offloaded (from 870.495 MB to 855.670 MB) only when object recognition is offloaded, but not when it is executed on-board (Fig. 12b). When we compare the two groups on the left and right in both Figs. 12a and 12b, we can see that the decrease in RAM utilization caused by object recognition offloading is indisputable.

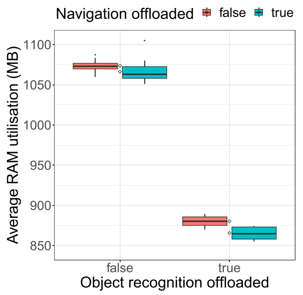
The results obtained in *Known map* setup are quite similar, with all three main effects being significant, but with one significant two-way interaction between navigation and object recognition. When we performed a pairwise comparison, we confirmed that the effect of navigation offloading is significant when object recognition is both executed on-board and offloaded (Fig. 12c). However, the the decrease in RAM utilization caused by navigation



(a) Offloading object recognition and SLAM in *Unknown map* setup



(b) Offloading object recognition and navigation in *Unknown map* setup



(c) Offloading object recognition and navigation in *Known map* setup

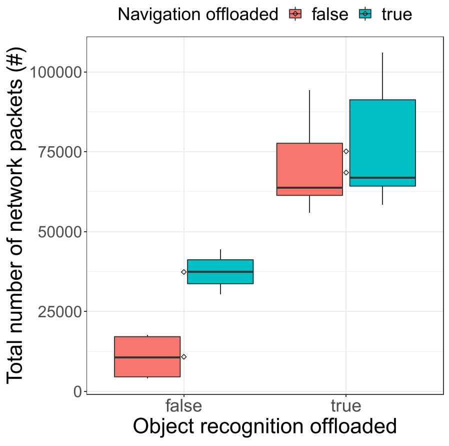
Fig. 12 The effect of computation offloading strategies on average RAM utilization

offloading is almost two times larger when object recognition is offloaded (from 880.225 MB to 865.590 MB, and from 1073.734 MB to 1066.304 MB, when object recognition is offloaded and not, respectively). Even though the test yielded a significant main effect of localization offloading on RAM utilization, the resulting effect size of only $\eta^2 = 0.003$ shows a slight decrease in RAM utilization. On the other hand, object recognition has an indisputable effect on RAM utilization, which is evident in Fig. 12c.

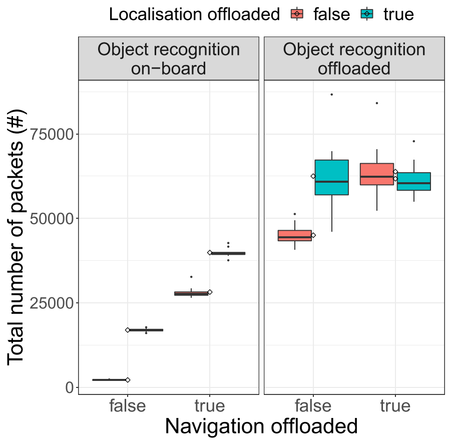
Similar to CPU usage, it is evident that object recognition accounts for much more significant RAM utilization when compared to other tasks. Therefore, offloading object recognition significantly reduces RAM utilization. Even though the results show that offloading other tasks, namely, SLAM, localization, and navigation, also results in a more optimal RAM utilization, the gains are negligible when compared to offloading object recognition. In fact, object recognition is so intensive that the benefits of offloading SLAM and navigation in *Unknown map* setup are not even statistically significant when object recognition runs on board. This is because object recognition reclaims the parts of the RAM that were utilized by SLAM and navigation once they are offloaded.

6.4 Number Of Packets

The results for the total number of network packets exchanged between the robot and the PC are presented in Fig. 13. The test results in the *Unknown map* setup yielded significant main effects for all three tasks, but there was also a significant two-way interaction between navigation and object recognition. The pairwise comparisons have shown that offloading navigation causes a statistically significant increase in the total number of packets when navigation is both executed on-board (from 10823.45 to 37365.30 packets) and offloaded (from 68487.55 to 75142.85 packets) (Fig. 13a). However, the estimates of effect size in pairwise comparisons show that the magnitude of the navigation offloading effect is more than four times greater when object recognition is executed onboard (26541.85 packets difference



(a) Offloading object recognition and navigation in *Unknown map* setup



(b) Offloading object recognition, navigation and localisation in *Known map* setup

Fig. 13 The effect of computation offloading strategies on the total number of network packets exchanged

in EMM, compared to 6655.30). This result indicates that network congestion happens when both navigation and object recognition are offloaded and some network packets are lost. Nevertheless, object recognition accounts for the greatest volume of packet exchange, which is evident when we compare the two groups on the left and right of Fig. 13a.

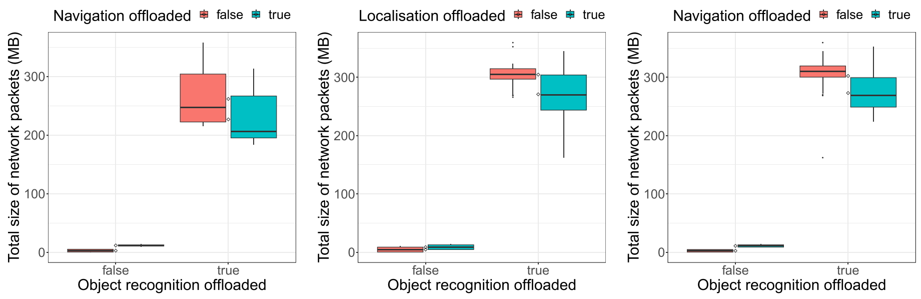
However, the results in the *Known map* setup are more complex, with all three main effects being statistically significant, but also all two-way and three-way interactions significant as well. The significant three-way interaction is stemming from the left part of the Fig. 13b. The results, in this case, have shown that when both object recognition and navigation are offloaded, offloading localization actually causes a slight decrease in the total number of network packets exchanged. The most significant increase in the total number of packets transferred definitely stems from object recognition offloading. However, we can notice in Fig. 13b that the traffic exchanged by only simultaneous localization and navigation offloading is almost as high as the one caused by object recognition offloading alone.

Unlike the CPU usage and RAM utilization results, where the effect of object recognition is the dominant compared to other tasks, we can see that navigation and localization account for a number of network packets that is more comparable to those exchanged by object recognition, but still lower. It is surprising that the magnitude of network exchange caused by SLAM is not as significant as for localization, given that these two tasks require and send almost the same data. In fact, SLAM should be even more communication-intensive, because it periodically produces maps as a result of the mapping process. The relatively high map update interval of 5 seconds, but also the relatively small arena, which is very accurately mapped early in the mission, definitely accounts for a lower exchange of maps.

The most important finding is the network congestion that happens when both localization and navigation are offloaded together with object recognition. Because all three tasks account for a large number of packets exchanged over the network, the network bandwidth is not large enough the handle the volume of network traffic when all tasks are simultaneously offloaded.

6.5 Size Of Packets

The results for the total size of network packets exchanged between the robot and the PC are presented in Fig. 14. The test results in the *Unknown map* setup yielded significant main effects only for object recognition, but there was also one significant two-way interaction between



(a) Offloading object recognition and navigation in *Unknown map* setup

(b) Offloading object recognition and localisation in *Known map* setup

(c) Offloading object recognition and navigation in *Known map* setup

Fig. 14 The effect of computation offloading on total size of network traffic exchanged

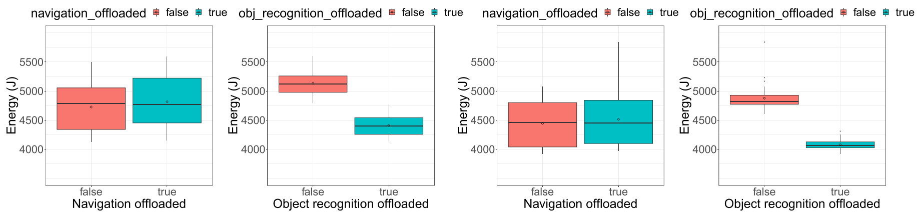
navigation and object recognition. The pairwise comparisons have shown that offloading navigation has no significant effect on the total size of network packets exchanged when object recognition is executed on-board, but it causes a statistically significant decrease in the total size of packets when object recognition is offloaded (from 302.442 MB to 273.076 MB) (Fig. 14a). Nevertheless, it is evident in Fig. 14a that object recognition exchanges a very large volume of data over the network.

The results in the *Known map* setup are quite different, yielding a significant main effect for all three independent variables and two significant two-way interactions. The pairwise comparisons for significant two-way interaction between localization and object recognition have shown that offloading localization causes a statistically significant decrease in the total size of packets exchanges when object recognition is offloaded (from 304.646 MB to 270.872 MB), but not when object recognition is executed on-board (Fig. 14b). Similar results are obtained for significant two-way interaction between navigation and object recognition, yielding a statistically significant decrease when navigation is offloaded (from 302.442 MB to 273.076 MB) only when object recognition is offloaded, but not when it is executed on-board (Fig. 14c). Unlike the *Unknown map* setup, the main effects of localization and navigation on the overall size of packets exchanged are yielded as statistically significant. However, the magnitude of the effect for localization and navigation, respectively, is larger than that caused by object recognition offloading. The magnitude of the increase in the total size of network traffic exchanged due to object recognition offloading is indeed evident in Figs. 14b and 14c.

Despite the more significant effect that both localization and navigation offloading have on the total number of packets, their effect on the total size is not as significant. This is expected because the size of laser scans, odometry, and other data that these tasks exchange, is indeed incomparable to the size of images that object recognition receives when offloaded, even though the images are sent over the network as compressed. The decrease in the total size of network packets when either navigation or localization are offloaded simultaneously with object recognition indeed confirms that there is network congestion in such strategies. As a result, some images are lost during the exchange, as was also confirmed in the results for the total number of packets in Section 6.4.

6.6 Energy Consumption

The results for total energy consumption are presented in Fig. 15. In the *Unknown map* setup results, there were neither significant three-way nor two-way interactions among the three factors. However, the results yielded a significant main effect of navigation and object recog-



(a) Offloading navigation in *Unknown map* setup (b) Offloading object recognition in *Unknown map* setup (c) Offloading navigation in *Known map* setup (d) Offloading object recognition in *Known map* setup

Fig. 15 The effect of computation offloading strategies on total energy consumption

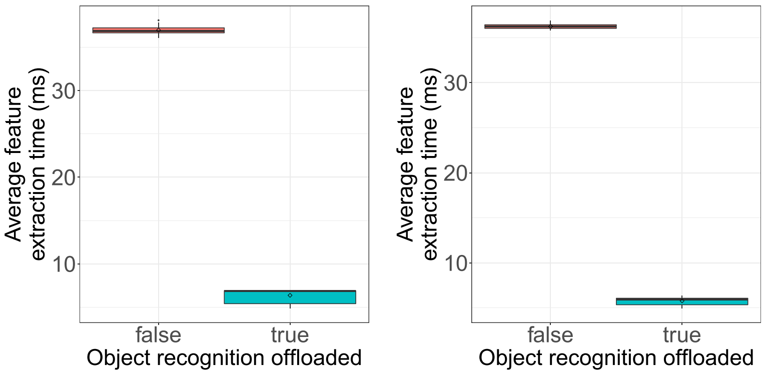
nition offloading, respectively, on total energy consumption. When navigation is offloaded, the mean of energy consumption slightly increases from 4725.150 J to 4815.433 J (Fig. 15a). On the contrary, offloading object recognition causes a significant decrease in total energy consumption, from 5132.966 J to 4407.616 J, on average (Fig. 15b).

Very similar results are obtained in the *Known map* setup, where the mean energy consumption slightly increased from 4441.430 J to 4513.235 J when navigation is offloaded (Fig. 10c), but significantly decreased from 4876.787 J to 4077.878 J when object recognition is offloaded (Fig. 10d).

The results for energy consumption are fairly similar to those obtained for mission execution time (see Section 6.1) because energy consumption is measured over the entire execution time. This result can be seen as an indication that the differences in the energy consumption is primarily due to a shorter mission execution time. Nevertheless, the magnitudes of the impact are much larger for object recognition yet smaller for navigation when compared to those obtained for mission execution time. This is noticeable for both setups, but especially in the *Known map* setup. Therefore, we can conclude that the energy consumption for network exchange when navigation is offloaded is greater when compared to the energy consumed when this task is executed onboard. On the contrary, offloading object recognition does yield a significant reduction of the total energy consumption, despite the very high volume of network traffic exchanged when this task is offloaded.

6.7 Feature Extraction Time

The results for average feature extraction time are presented in Fig. 16. The results for average feature extraction time are almost identical for both setups. The only significant main effect is object recognition, with neither significant two-way nor three-way interactions. In the *Unknown map* setup, the average feature extraction time when object recognition is executed on-board is 36.986 ms and only 6.365 ms when it is offloaded. Similarly, the average feature



(a) Offloading object recognition in *Unknown map* setup (b) Offloading object recognition in *Known map* setup

Fig. 16 The effect of computation offloading strategies on average feature extraction time

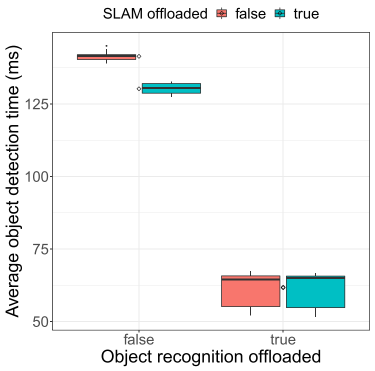
extraction time in the *Known map* setup is 36.233 ms when object recognition is executed on-board and 5.791 ms when it is offloaded (Fig. 16b).

We can conclude that features are extracted much faster when object recognition is offloaded because there are more computation resources available on the PC as compared to the Raspberry Pi running onboard the robot.

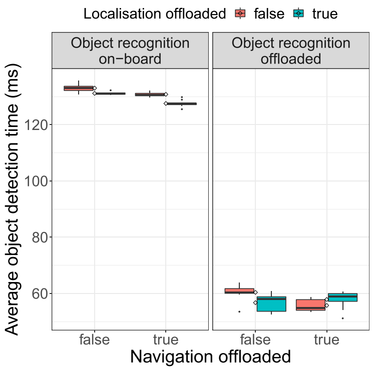
6.8 Object Detection Time

The results for average object detection time are presented in Fig. 17. The results in the *Unknown map* setup yield a significant main effect for all three independent variables, but also a significant two-way interaction between SLAM navigation. The pairwise comparisons for significant two-way interaction between SLAM and object recognition have shown that offloading SLAM causes a statistically significant decrease in object detection time when object recognition is executed on-board (from 141.398 ms to 130.215 ms), but not when object recognition is offloaded. The main improvements in the object detection time stem from object recognition offloading, which is evident in Fig. 17a.

The results in the *Known map* setup are slightly more complex because apart from all three significant main effects, a significant three-way interaction emerged. With further analysis of significant three-way interaction, it is established that both localization and navigation yield statistically significant improvements in average object detection time when object recognition is executed onboard. This is evident in the left part of Fig. 17b, but it is also noticeable that these improvements are of small magnitude. The significant three-way interaction was traced to the fact that when both object recognition and navigation are offloaded, the average extraction time actually increases when localization is offloaded as compared to its onboard execution. This is caused by network congestion, where due to the packet loss, it takes more time for incoming image frames to arrive at the remote PC, thus the object detection time



(a) Offloading object recognition and SLAM in *Unknown map* setup



(b) Offloading object recognition, navigation and localisation in *Known map* setup

Fig. 17 The effect of computation offloading strategies on average object detection time

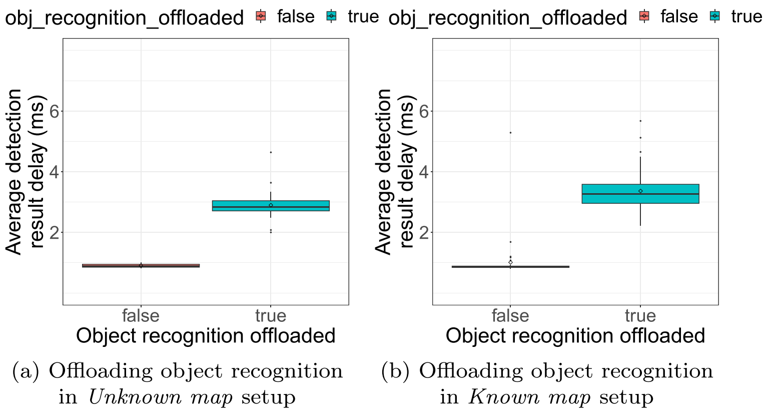


Fig. 18 The effect of computation offloading strategies on average detection result delay

is prolonged. The speed up in object detection is mainly the result of object recognition offloading, as evident in Fig. 17b.

It is evident in both setups that object detection is performed more efficiently when object recognition is offloaded, as there are more computation resources available on the remote PC. However, when object recognition runs on board, the object detection time is slightly improved when SLAM and localization are offloaded, as there are more computation resources available for object recognition in these offloading strategies. Such improvements are still insignificant as compared to the efficiency caused by object recognition offloading.

6.9 Detection Result Delay

The results for average detection result delay are presented in Fig. 18. The results for the average detection result delay are almost identical for both setups. The only significant main effect is the one of object recognition, with neither significant two-way nor three-way interactions. In the *Unknown map* setup, the average detection result delay is prolonged from

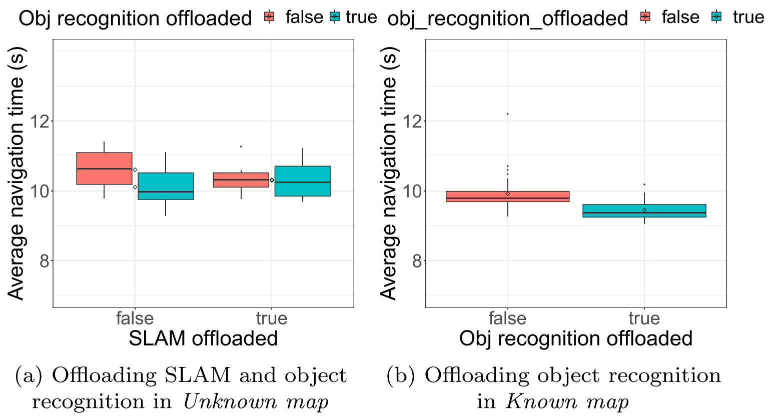


Fig. 19 The effect of computation offloading strategies on average navigation time

0.904 ms to 2.899 ms when object recognition is offloaded (Fig. 18a). Similarly, the average detection result delay time in *Known map* setup is prolonged from 1.012 ms to 3.605 ms when object recognition is offloaded (Fig. 18b).

We can conclude that the average detection result delay is higher when object recognition is offloaded because the results have to be exchanged over the network, from the remote PC to the node running onboard the robot. Nevertheless, the delay of around 3 ms when object recognition is offloaded is not substantially larger when compared to around 1 ms delay, when the results are not transmitted over the network.

6.10 Navigation Time

The results for average navigation time are presented in Fig. 19. The results in the *Unknown map* setup show that offloading object recognition has a significant main effect on average navigation time, but there is also a significant two-way interaction between SLAM and navigation. The pairwise comparisons for significant two-way interaction have shown that offloading object recognition only causes a statistically significant decrease in the average navigation time when SLAM is executed onboard (from 10.606 seconds to 10.107 seconds), but not when it is offloaded.

Similar results are obtained in the *Known map* setup, with the only significant main effect being the one of object recognition, with neither significant two-way nor three-way interactions. The average navigation time is reduced from 9.920 seconds to 9.448 seconds when object recognition is offloaded.

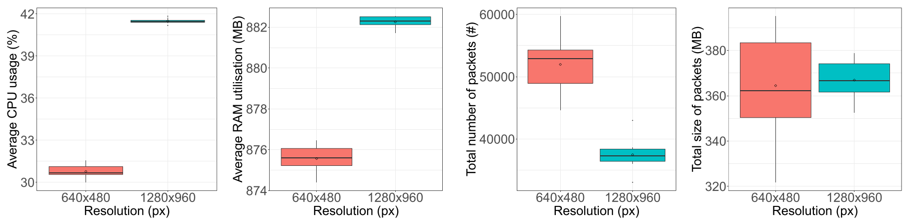
It is quite surprising that navigation offloading has no significant effect on the average navigation time in either of the setups, yet it is faster when object recognition is offloaded. As we also concluded in the case of mission execution time, the availability of resources when object recognition is offloaded is the reason why navigation performs more efficiently in such cases.

7 Results Of The Second Round Of Experiments (RQ2)

In this section, we present the most important results of the performed statistical analyses and observations. The complete data analysis process, along with the assumptions analysis of the performed statistical tests and the obtained test results, are available in the replication package.

7.1 Image Resolution

Addendum – We anticipate that, for technical reasons (explained below), the results obtained for network-related metrics (*i.e.,* total number of packets and total size of network packets) are not directly related to the change in the image resolution used in our experiment. Specifically, in Figs. 20c and in 20d we observe that the total number of packets exchanged is unexpectedly *lower* for images with higher resolution, while the total size of network packets remains *similar* to those with low-resolution images. In the following, we explain why this phenomenon is happening. By referring to Figs. 6 and 7, the `raspicam_node` sends *compressed* images to the `find_object_2d` node for both image resolutions. Indeed, the message definition



(a) Average CPU usage. (b) Average RAM utilization. (c) Total number of network packets. (d) Total size of network packets.

Fig. 20 The effect of image resolution parameter on system resources and network traffic

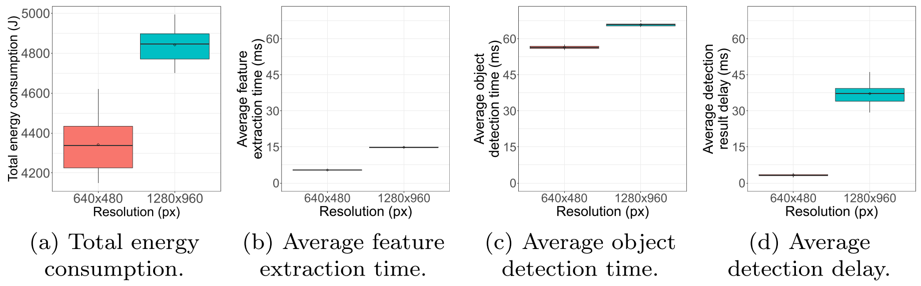
of the ROS topic we use for such communication is *sensor_msgs/CompressedImage*[38]; in our system, the compression is based on the JPEG algorithm and it is done consistently with the same degree of compression, independently of the resolution of the produced images. We also manually analyzed the source code of all the ROS nodes involved in the object recognition task and confirm that image compression is not happening in other ROS nodes or in message exchanges within our system. Because the size of images compressed using the JPEG algorithm is not necessarily linear with respect to the size of the input image, we speculate that this indeed might be a possible explanation for the fact that we do not observe the size of network packets doubling when using 1280x960 images. Nevertheless, despite the technical limitation of this experiment with respect to network-related metrics, in the remainder of this section the reader will see that using images with higher resolution lead to higher resource utilization and energy consumption.

While analyzing Figs.20 and 21, we identify that the magnitude of the image resolution effect on the utilization of system resources is still significant. In Fig. 20a, we can see that the higher resolution causes a significant increase in average CPU usage, from 30.774% for 640x480px to 41.497% for 1280x960px, with the resulting effect size of d = -30.006. The effect on RAM utilization is similar; higher resolution causes an increase from 875.567 MB to 882.260 MB on average, with d = -13.118 effect size (Fig. 20b). The mean of total energy consumption also increased from 4342.202 J to 4841.340 J, with an effect size of d = -4.077 (Fig. 20a), even though the resolution increase has no significant effect on total mission execution time.

As anticipated at the beginning of this section, the total number of packets decreased, from 51974.4 to 37486.1 packets, with an effect size of d = 4.181. Even though the number of packets has decreased, the tests show that the total size of network traffic has not changed significantly with the resolution improvement (Fig. 20d). The slight increase from 364.473 MB to 366.913 MB in total is not statistically significant.

High resolution also has a significant effect on object recognition performance. As we can see in Fig. 21b, the average feature extraction time has increased from 5.402 ms to 14.859 ms, with an effect size of d = -83.838. This result is not surprising, since more significant features can be extracted from a high-resolution image frame as compared to the lower resolution one. Similarly, the average object detection time has increased from 56.450 ms to 65.872 ms on average, with the effect size of d = -13.339 (Fig. 21c). Finally, high resolution has a significant effect on the average delay for detection result message transfer. The delay has increased from only 3.254 ms to 37.189 ms on average, which could cause issues in real-

38 http://docs.ros.org/en/api/sensor_msgs/html/msg/CompressedImage.html



(a) Total energy consumption. (b) Average feature extraction time. (c) Average object detection time. (d) Average detection delay.

Fig. 21 The effect of image resolution parameter on energy consumption and object recognition performance

time applications, where the deadline for receiving object detection results may be of great importance (Fig. 21d). The effect size of this increase is d = -10.003.

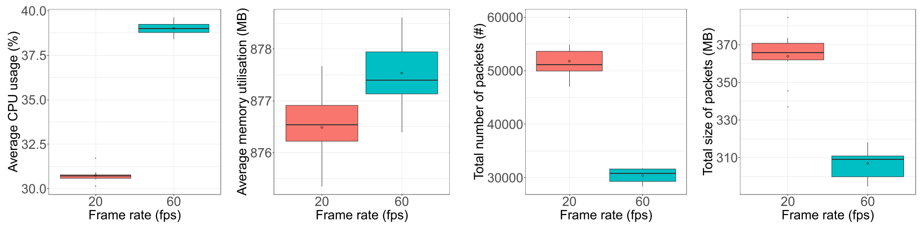
Regardless of whether object recognition is offloaded or not, the images are captured on board the robot because the camera module is attached to the robot itself. This is why the higher image resolution results come with the price of increased CPU usage, RAM utilization, and energy consumption. Moreover, there are more significant features in images of better quality, thus it takes longer to perform feature extraction and object detection.

However, the lower number of network packets exchanged is rather surprising because the image rate remains the same for both resolution treatments. With the inspection of log files and *find_object_2d* source code, we conclude that this ROS package is not multi-threaded, thus the newly received image frame are not be processed until the previous one has finished. With the input frame rate for object recognition set to 10fps, this means that it should take no more than 100ms to process each received image frame to keep up with the desired frame rate. As the results show, feature extraction and object detection are performed in 14.859 ms and 65.872 ms on average, respectively, with 1280x960px image resolution, which is already more than 80 ms of the average image processing time. For those image frames that take more than 100 ms to be processed, the next frame will not be received, which, in fact, causes package loss.

7.2 Image Frame Rate

Addendum – We anticipate that, due to technical reasons specific to this experiment, our results for network-related metrics are not directly related to the change in the frame rate as well. Indeed, as shown in Figs. 22c and in 22d, the total number of exchanged packets and their total size are unexpectedly *lower* when adopting a frame rate of *60fps*. In order to better understand this phenomenon, we first closely inspected the *raspicam_node* node and we confirm that it produces images at the defined rates (*i.e.,* 20fps and 60fps). However, we also observed that (i) the *raspicam_node* node is configured to publish the latest produced frame only if the previous one is already sent through the network and (ii) the default size of the queue of the *find_object_2d* ROS node is only 10. It is important to note that for each topic the ROS communication mechanism has two queues (one for the publisher and one for the subscriber) and internally the ROS communication mechanism prevents messages from traveling across the network if one of those two queues is filled[39]; in our system, this mechanism results in both queues being filled by the higher number of input images in the

[39] http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers



(a) Average CPU usage. (b) Average RAM utilization. (c) Total number of network packets. (d) Total size of network packets.

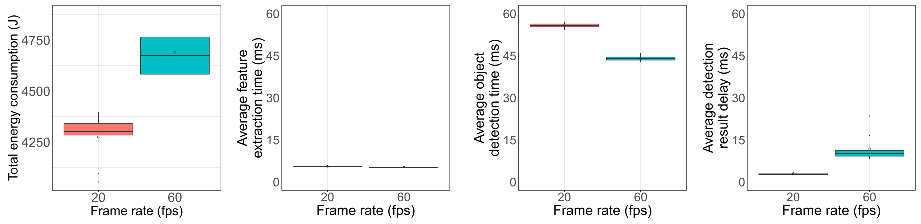
Fig. 22 The effect of image frame rate parameter on system resources and network traffic

case of 60fps, thus leading to lower network traffic (in terms of both number and total size of the exchanged network packets).

The effect of the image frame rate parameter is presented in Fig. 22 and 23. The test shows that image frame rate has a significant effect on the utilization of system resources. In Fig. 22a, we can see that CPU usage increased from 30.745% to 39.020%, on average, when the frame rate was increased from 20fps to 60fps, with the resulting effect size of r = 0.846. The effect on RAM utilization is similar; a higher frame rate causes an increase from 876.487 MB to 877.536 MB on average, with d = -1.478 effect size (fig. 22b). The mean total energy consumption also increased from 4274.945 J to 4689.950 J, with an effect size of r = 0.846 (Fig.23a), even though the increase in image frame rate has no significant effect on total mission execution time.

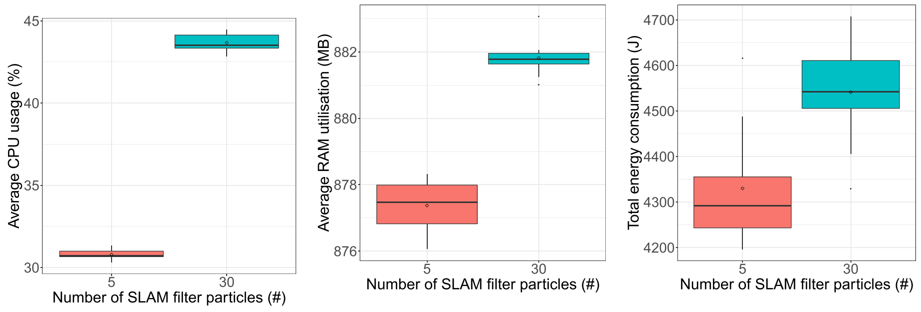
As depicted in Fig. 22c, the total number of packets decreased from 51786.3 to 30397.2 packets with the higher image frame rate. As discussed above, this is due to the filling of the ROS messages queues being filled prematurely in the specific configuration of our experiment. The magnitude of this decrease is d = 7.544 effect size. Because the resolution remains the same for both frame rate treatments, the size *per packet* does not change, thus the total size of all exchanged packets also decreased from 363.838 MB to 306.942 MB on average (Fig. 22d). The effect size of the total packet size change is d = 5.092.

However, the effects that increased frame rate has on object recognition performance are completely opposite of those caused by resolution increase. As we can see in Fig. 23b, the average feature extraction time has decreased from 5.445 ms to 5.318 ms. The size of this effect is d = 0.507. Similarly, the average object detection time has decreased from 55.899 ms to 44.053 ms on average, with the effect size of d = 13.572 (Fig. 23c). In contrast, we can conclude that image transfer at the high frame rate causes a significant increase in the average



(a) Total energy consumption. (b) Average feature extraction time. (c) Average object detection time. (d) Average detection delay.

Fig. 23 The effect of image frame rate parameter on energy consumption and object recognition performance



(a) Average CPU usage. (b) Average RAM utilization. (c) Total energy consumption.

Fig. 24 The effect of number of particles parameter on system resource and energy consumption

delay for detection result messages, from only 2.850 ms to 11.802 ms on average (Fig. 23d). The effect size of this increase is r = 0.845. Regardless of whether object recognition is offloaded or not, the images are captured on board the robot; thus sampling more images per second comes with the price of increased CPU usage, RAM utilization, and energy consumption.

7.3 Number of Particles

The effect of the number of RBPF particles in *gmapping* is presented in Fig. 24. The increased number of particles causes a significant increase in system resource utilization. In Fig. 24a, we can see that CPU usage has increased from 30.783% to 43.671%, on average, when the number of particles is increased from 5 to 30, with the resulting effect size of d = -29.562. The effect on RAM utilization is similar; a higher number of particles cause an increase from 877.375 MB to 881.817 MB on average, with r = 0.845 effect size (Fig. 24b). The mean of total energy consumption also increased from 4329.966 J to 4541.058 J, with an effect size of d = -1.709 (Fig. 24c), even though the increase in the number of particles has no significant effect on total mission execution time.

Each particle in the RBPF algorithm carries a piece of information about a single part of the environment, thus higher number of particles entails more computation effort. The results presented in the work of (Abdelrasoul et al. 2016) have shown that CPU usage and RAM utilization indeed increase with more particles. The results of this study thus confirm these findings. However, the authors do not consider the energy consumption of the robotic system, whereas we can conclude that it also increases with the number of particles.

7.4 Temporal Updates

The effect of temporal map updates, with periodical laser scan processing regardless of the robot's movements, is presented in Fig. 25. The results show that map updates caused by laser scan processing every 0.5 seconds do not have a significant effect on any of the parameters but the average CPU usage. As we can see in Fig. 25, CPU usage slightly increased from 30.734% to 31.824%, on average. While this increase does not seem that significant at first glance, its effect size is d = -3.030.

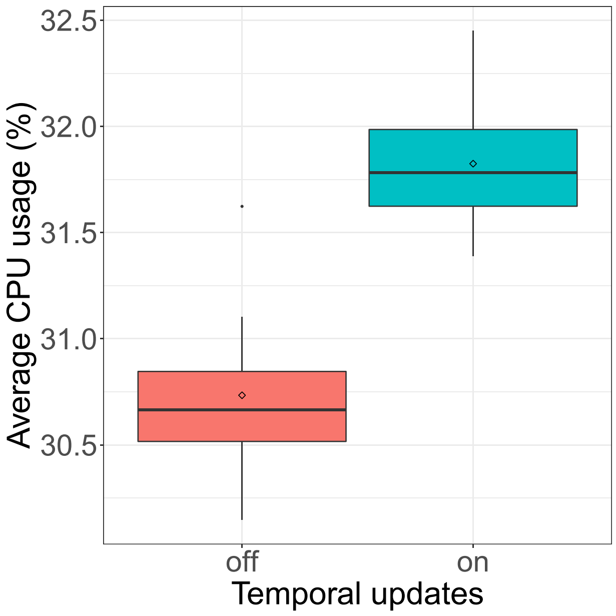


Fig. 25 The effect of temporal updates on average CPU usage

The results presented in the work of Abdelrasoul et al. Abdelrasoul et al. (2016) show that the map updates triggered when the robot traverses a certain distance, either with translation or rotation, cause an increase in CPU usage and RAM utilization when the distance is set to lower values. However, the authors do not consider periodical updates at regular time intervals. We can conclude that temporal map updates cause a significant increase in CPU usage, yet they do not have a significant effect on RAM utilization.

7.5 Velocity Samples

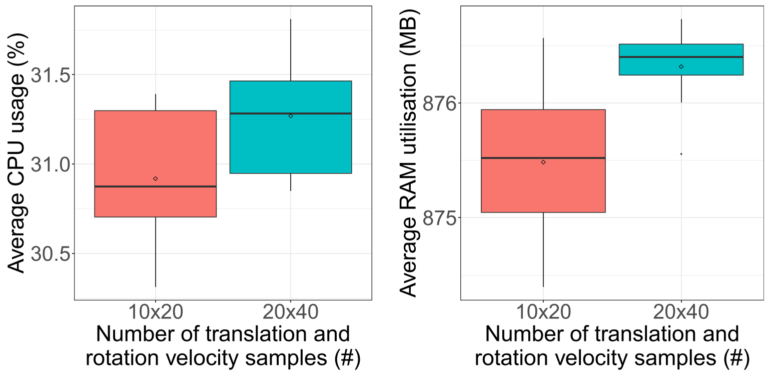
The effect of the number of velocity samples parameter is presented in Fig. 26.

The increase in the number of translation and rotation velocity samples in the local planner from 10 and 20 to 20 and 40, respectively, causes a significant increase in CPU usage and RAM utilization. In Fig. 26a, we can see that the CPU usage slightly increased from 30.919% to 31.269%, on average, with the resulting effect size of d = -0.957. The effect on RAM utilization is similar; a higher number of velocity samples causes a bit more significant increase from 875.484 MB to 876.318 MB on average, with d = -1.581 effect size (Fig. 26b).

In the DWA algorithm, the larger sample of velocities means that more possible trajectories will be simulated when choosing the next one for the robot to follow. The results confirm that the sampling process requires large computation effort, as both CPU usage and RAM utilization increase with the higher number of velocity samples.

7.6 Simulation Time

The effect of the simulation time parameter is presented in Figs. 27 and 28. The increase in simulation time in the DWA algorithm from 1.5 to 3 seconds causes a significant change in



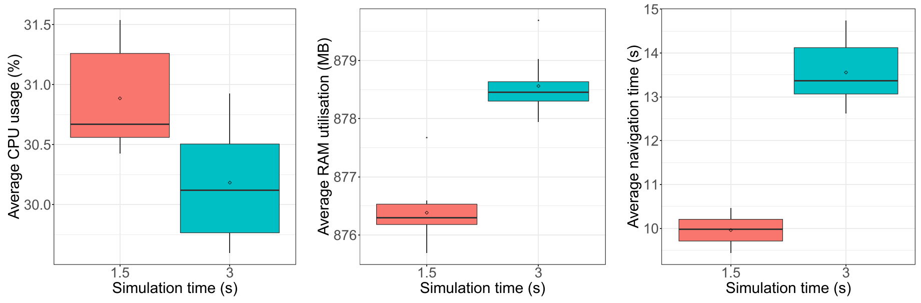
(a) Average CPU usage. (b) Average RAM utilization.

Fig. 26 The effect of number of velocity samples parameter on system resources

CPU usage and RAM utilization. In Fig. 27a, we can see that the CPU usage decreased from 30.886% to 30.183%, on average, contrary to intuition. While this change does not seem that significant, the resulting effect size of d = 1.580. In contrast, simulation time increase does cause a statistically significant increase in RAM utilization, from 876.383 MB to 878.560 MB on average, with r = 0.845 effect size (Fig. 27b).

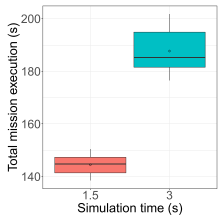
As shown Fig. 27c, the average navigation time increased from 9.960 to 13.556 seconds, with the effect size of d = -6.557. As a result, the total mission execution time increased from 144.488 to 187.716 seconds, with an effect size of d = -6.543 (Fig. 28a). Consequently, the increased mission execution time triggered an increase in total energy consumption, as well as in the total number and size of network packets exchanged. The total energy consumption increased from 4277.45 J to 5461.64 J, on average, with an effect size of d = -6.143 (Fig. 28b). The total number of network packets increased from 44863.6 to 56137.1 (Fig. 28c), whereas the total size of packets increased from 318.425 MB to 435.180 MB (Fig. 28d), with effect sizes of d = -2.527 and r = 0.845.

The navigation tuning guides claim that the increase in simulation time requires higher computation load (Zheng 2017; ROS wiki 2021). The principle of the DWA algorithm goes in hand with these claims, as the simulation of longer trajectories implies more computation power. However, the results show only a significant increase in RAM utilization for longer

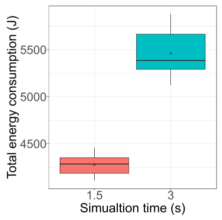


(a) Average CPU usage. (b) Average RAM utilisation. (c) Average navigation time.

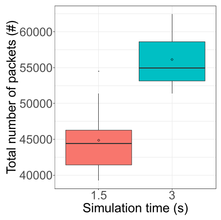
Fig. 27 The effect of simulation time parameter on system resources and navigation time



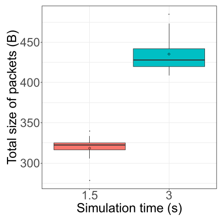
(a) Mission execution time.



(b) Energy consumption.



(c) Number of network packets.



(d) Size of network packets.

Fig. 28 The effect of simulation time parameter on mission execution time, energy consumption, network packets, and size of network packets

simulation time, whereas CPU usage is slightly decreased. This may be due to the small environment in which the experiment is run, which has no obstacles and only the map boundaries.

The most important observation is the increased navigation time for longer simulation time, as it takes around 40 seconds longer for the robot to traverse the same trajectory. The way in which the robot navigates to its goal location has completely changed and we notice that the robot approaches the goal slowly as compared to the original simulation time of 1.5 seconds[40]. Consequently, the change of this single parameter triggered the increase in energy consumption, but also network traffic exchange.

8 Discussion

In this section, we provide an overview of the main findings emerging from our experiments that can be of use to both practitioners and researchers. It is important to highlight the fact that the findings reported in this section are emerging from our specific Turtlebot-based experiments and are valid within the scope of similar software stacks and robotics configurations (*e.g.,* ground robots running the *gmapping*, *move_base*, and *find_object_2d* ROS packages and offloading their computation via a WiFI network). Further research (and experiments replications) is needed to provide evidence that the emerging findings apply also in the context of our types of robots and configurations (*e.g.,* manufacturing robots, flying robots, ROS2-based robots, etc). In the remainder of this section we firstly discuss the main findings for RQ1 (see Section 8.1), then we discuss those for RQ2 (see Section 8.2), and finally we provide more detailed reflection points for each considered robotic task (see Section 8.3.

For the sake of readability, we give a unique identifier to each of the most actionable findings, where the ones targeting researchers are marked as R_x, and the ones targeting practitioners are marked as P_x.

8.1 Offloading Impact on Performance and Energy Consumption (RQ1)

The obtained results show that object recognition is a greedy resource consumer and, when associated with SLAM, they consume together about 80% of available CPU (Fig. 11), and the memory utilization is also about 25% higher when running onboard (Fig. 12). In such a configuration, SLAM seems to have a lower impact, whereas object recognition is the most

40 https://youtu.be/hVS775ITt4o

computing-intensive task. Object recognition is also the task that results in higher network traffic, when offloaded (Fig. 13). The execution time is not considerably affected by any of the offloading configurations (Fig. 10), which might be seen also as an indication of the stable WiFi network used in our experiments. Finally, the object recognition task is the one influencing the most the energy consumption of the robot (Fig. 15). This behavior is understandable if we look at Fig. 16, which shows the object recognition feature extraction is about four times faster when running remotely, and object detection time almost doubles when running onboard. We checked all the data, and the CPU usage was not exhausted at any moment, which indicates that this phenomenon is not due to a lack of available resources.

RQ1a Answer: The computing-intensive task, *i.e.,* object recognition, had its processing time reduced when offloaded to the Cloud, mainly thanks to the higher processing power available. For the other tasks, there was no considerable performance difference. The only critical aspect of offloading was the network delay in responding after a task/processing is completed, which may be an impediment for time-sensitive tasks.

RQ1b Answer: Offloading did not affect the energy consumption negatively and should not be considered a constraint. The main source of energy consumption is the already well-known CPU usage (followed by memory utilization), and as expected computing-intensive tasks exhibited more consistent energy savings when offloaded.

R_1 - Use offloading or GPUs for improving processing time of vision tasks

For vision tasks (*e.g.,* object recognition), offloading or the usage of a single-board computer (SBC) that includes GPU (*e.g.,* the NVIDIA Jetson[a] board) can improve processing time considerably.

[a] `https://www.nvidia.com/pt-br/autonomous-machines/embedded-systems/`

P_1 - Offloading might lead to higher latency

The most relevant drawback of offloading a task is the latency to transmit processed data throughout the network, which may be a constraint, depending on the robotic system features. Resource consumption is not affected negatively in our experiments.

P_2 - Computation-intensive tasks are good candidates for offloading

In the case of compute-intensive tasks, such as object recognition, offloading reduces robot resource consumption considerably.

8.2 The Influence of Parameter Configuration (RQ2)

Different from offloading tasks, we see that changing parameter configuration is way more critical, reflecting different levels of resource utilization in all cases. While a higher image resolution in object recognition results in more CPU usage (increasing by $\approx 25\%$), memory utilization only increases by $\approx 1\%$, the size of network packets remains constant, and the total of exchanged network packets is decreased by $\approx 37\%$. On the other hand, other parameters, such as the number of particles for *Gmapping*, result in a more predictable performance and resource consumption. In the object recognition case, it was concluded that most of the unpredictable behaviors are due to a lack of synchronization between the robot output (such as video frames per second) and the task input setup. This reflects the particularities of each task and its algorithms and goes back to one of the ROS facilitators, also a vision for robotic future[41]: *building technologies that combine knowledge across previously unrelated disciplines*.
RQ2a Answer: Performance is directly impacted by parameter configuration, which is a critical factor. The impact is not straightforward either; this is expected since tasks have their own loading or warming-up costs. We also observe an unexpected reduction in the number and size of exchanged packets when increasing the frame rate of the camera node from 20fps to 60fps; this is mainly due to the queue of the *find_object_2d* ROS node being filled by the higher number of input images. This experiment provides evidence that queue sizes and publishing/subscribing rates of ROS nodes must be properly considered by practitioners since they might lead to severe consequences in terms of system's behaviour.
RQ2b Answer: Since parameter configuration affects resource usage in most of the performed trials, it also leads to considerable variations in terms of energy consumption. Therefore, choosing the tuning of ROS tasks seems to be a critical design choice in case energy is a scarse resource.

8.3 Considerations For The Specific Robotic Tasks

8.3.1 Object Recognition

Despite the fact that this task was highly optimized for low-processing-power devices, with a relatively low image frame rate, image resolution, and a maximum number of features, the results still show that the benefits of offloading object recognition outweigh on-board execution. Feature extraction and object detection are also much faster when they are offloaded, possibly given the onboard GPU of the used laptop model (Rad et al. 2021). Unfortunately, the amount of network traffic exchanged is also significant due to the large volume of image frames exchanged, even when they are compressed, as is the case in the experiment setup. The results, however, indicate that the trade-off between energy consumed due to network transfer and onboard execution still leans toward offloading because the total energy consumption is significantly decreased.

[41] https://foxglove.dev/blog/the-future-of-robotics

P_3 - Offload object recognition tasks to save energy

When executing object recognition tasks, offloading consumes significantly less energy than executing the tasks onboard the robot.

One of the most important observations in the performed experiments is network congestion. When only navigation is additionally offloaded along with object recognition, there is already packet loss, even in a completely isolated local network, without an Internet connection. This is why it is important for practitioners to ensure that the available network bandwidth can handle such a large volume of exchanged image frames, especially if additional tasks are offloaded together with object recognition. This is a significant improvement opportunity for researchers, as there is a need for network protocols that can enable efficient real-time exchange of such large volumes of data as images. Some improvements towards this goal are already reported in research in robotics (Wu et al. 2012).

P_4 - Deploy Network QoS mechanisms for supporting multiple task Offloading

It is important to deploy mechanisms for ensuring the Quality of Service (QoS) in the network to avoid congestions when multiple tasks are offloaded.

R_2 - Research opportunities on ROS2 systems exchanging a large volume of data

The DDS protocol used in ROS2 partially tackles network congestion (especially for large volumes of data), so (i) further experiments are advised to provide evidence about this solution and (ii) there is room for exploring different DDS-based network protocols that can handle large volumes of data on ROS2 systems.

Another important observation for practitioners and researchers using the *find_object_2d* task is that the combination of three parameters, namely, image resolution, input frame rate, and a maximum number of features, has a significant effect on how efficiently an object recognition is executed. Image resolution and a maximum number of features should be carefully chosen so that all image frames are processed at the desired input frame rate. Otherwise, as we witness in the parameter effects experiments (RQ2), some images will not be transmitted to the object recognition node at all. In the context of this experiment, a resolution of 640x480px, an input frame rate of 10fps, and a maximum number of 500 features give promising results as all images are processed at the set rate. However, increasing either resolution or frame rate to two times higher values already causes image losses. An interesting behavior is observed when the input frame rate is increased to 30fps (with two times higher output frame rate), where the *find_object_2d* node, in the beginning, tries to speed up the image processing to keep up with the set input rate by extracting a lower number of features from the image frame. However, the node gives up on this optimization very soon and it starts extracting a higher number of features, but at the cost of losing some images. This is an opportunity for researchers to improve the *find_object_2d* task so that the parameters are automatically configured, with the main goal of meeting set user requirements during the entire mission execution. For instance, if a user wants to achieve a maximum processing time for all image frames, the system should configure the set of parameters, and automatically adapt them during mission execution, so that the desired time is always met. The automation of parameter configuration can be also generalized for other robotics tasks, without restrictions to object recognition only.

P_5 - The parameters of find_object_2d have a significant effect on its efficiency.

The combination of three parameters, namely, image resolution, input frame rate, and a maximum number of features, have a significant effect on how efficiently object recognition is executed.

R_3 - find_object_2d parameter reconfiguration at runtime

It is necessary to implement an optimized strategy that allows parameter reconfiguration of find_object_2d at runtime for relevant preset metrics.

The choice of the frame rate is not straightforward, as it is not only guided by the efficiency of the object recognition process, but rather practitioners should choose the frame rate depending on the speed at which the robot operates and the dynamism of the working environment. If the captured part of the working environment significantly changes with each image frame, the frame rate should be increased. In our experiment, since we operate the TurtleBot3 at a relatively low speed, we found 20fps satisfactory. If the nature of the mission requires a higher frame rate, it must be ensured that the available network bandwidth is high enough to handle the increased network traffic exchange.

P_6 - Set the image frame rate based on the robot's performance and the environment of the mission

Practitioners should choose the frame rate depending on the speed at which the robot operates and the dynamism of the working environment.

While we find the resolution of 640x480px to be satisfactory in our experiment, practitioners should choose the resolution with many other factors in mind. The choice depends on the size of the objects that should be recognized and the robot's distance from them. The lighting in the working environment is an important factor, which can be especially problematic in outdoor missions, but also the vibrancy of the background around the objects that should be detected. Our experiment was performed with closed curtains, so the lighting effect was not problematic. However, we notice during the experiment trials, that lighting and distinctive background behind the objects indeed caused for some pictures to be detected better than others. Finally, during experiment execution, we noticed that some of the three pictures can be recognized already from the robot's starting position, at 0.8m distance, while the others have to be approached more closely to be successfully detected. The choice of resolution should also be driven by the number of features that can be extracted from an object. Colorful pictures with distinctive elements are recognized more accurately than those containing just a couple of plain geometric shapes. If the object has no significant features that are easy to extract and detect, a higher resolution is necessary. Alternatively, practitioners may turn to sophisticated reinforcement learning algorithms, which can give good results in such cases. Algorithms based on *Deep Neural Networks* (DNNs) may perform object recognition efficiently, even at low resolution. Even though object recognition based on DNNs is not a subject of this study, it would be an interesting point for future research to evaluate its effects on performance and energy consumption in robotic systems.

R_4 - Experiments on alternative object recognition algorithms

It would be an interesting point for future research to evaluate the effects of different algorithms, such as Deep Neural Network (DNN) or even Video Processing as an alternative to object recognition.

A possibility for research includes splitting DNNs between the robot and the remote machine so that some part of the image processing can be efficiently performed onboard. This can result in reduced network exchange between the robot and the remote machine, as there would be no need to transfer the entire image over the network.

Finally, it should be noted that in the system used in the experiments, object recognition results are only logged. If practitioners need to make further decisions based on object recognition results, the network delay for sending the image and receiving the result back should be taken into account. Unfortunately, we were not able to calculate the delay for image sending based on the logs of the camera driver and object recognition nodes, but we do report the average delay between the moment when the object is detected in the Cloud and the result received on the robot's side (3 ms). This delay becomes 1 ms when the object recognition task is executed onboard. However, this delay increased to 37ms and 12ms when resolution and frame rate are higher, respectively. It should also be noted that the experiment was performed in an isolated network, with no Internet connection. The experiment trials, performed with the router connected to the University's WLAN, show much more variability in object recognition delay, ranging from 3ms to 300ms. This effect would be even more amplified if the tasks were offloaded to the cloud, where the distance to the servers and variability of network conditions may play a much more significant role.

P_7 - Consider high network latency when offloading object recognition

Practitioners shall expect high network latencies, even in controlled networks, such as the one we used in our experiments.

8.3.2 SLAM and Localization

Because SLAM was a subject of computation offloading in several research studies, we were surprised to see that there are no significant benefits of its offloading. In fact, according to our experiment, practitioners can execute either localization or SLAM onboard efficiently without utilizing vast amounts of energy and resources. We did not observe significant improvements in map accuracy when we increased the number of particles from 5 to 30, which is indeed aligned with the results reported in the work of (Abdelrasoul et al. 2016, This can be the case because we use an environment of a similar size as Abdelrasoul *et al.,* yet the implications may be different for larger and more dynamic environments. Researchers have an opportunity to evaluate the effects of the number of filter particles in the *gmapping* SLAM algorithm with respect to map accuracy in the context of environments of different sizes and shapes.

P_8 - SLAM and Localization tasks can be executed efficiently both onboard and in the Cloud

SLAM and Localization can be run onboard without much resource utilization and can also be offloaded without affecting much the performance and energy consumption.

R_5 - SLAM and Localization tasks shall be evaluated in dynamic and large environment

SLAM and Localization tasks should be further evaluated in a dynamic and larger environment. One idea that should be also considered is an environment where moving obstacles are present.

Since no study performed in the context of *gmapping* configuration evaluation is focused on the temporal updates parameter, we were interested to evaluate if it had any effect on resource utilization. As it turns out, that effect is not significant, yet we do notice that the edges of the map tend to be smoother with periodical temporal updates and the overall accuracy of the map is marginally better. With no significant effect on resource utilization, we can conclude that *gmapping* can benefit from temporal updates when it is executed onboard the robot.

P_9 - Gmapping temporal updates

Gmapping can benefit from temporal updates when it is executed onboard the robot.

8.3.3 Navigation

Given the complexity of the navigation stack, where several plugins are in charge of different computation-intensive tasks (see Section 2.2.3), we expected that its offloading would lead to significant benefits. Surprisingly, navigation offloading lead only to longer mission execution time and caused an increase in the total energy consumption, whereas there was no significant reduction in CPU usage and RAM utilization. Those results lead to the observation that the navigation task performs more efficiently onboard the robot.

P_{10} - The navigation task performs more efficiently on-board

In our experiment, even with a simple mission, the navigation task performed more efficiently onboard the robot, mainly due to avoidance of network delays.

We could observe also interesting results when looking at the configuration parameters of the navigation task. The navigation tuning problem is widely known in the ROS community, as confirmed by the guides particularly focused on navigation tuning (Zheng 2017; ROS wiki 2021. Our experience confirms that navigation tuning is not a straightforward task in practice, as single localized changes to only one of many parameters can completely change the way in which the robot navigates. We witnessed this effect in the parameter effect analysis, where the average navigation time increased by around 40 seconds when the simulation time is increased from 1.5s to 3s. What we expected was only the increase in CPU usage and RAM utilization, as it stated in the guides that longer simulation time requires more computation power.

P_{11} - Different configurations of the Navigation task leads to large differences in terms of resource usage

Researchers shall experiment with different parameters of the Navigation task since even slight changes to only one of those parameters can lead to completely different ways in which the robot navigates (and to significantly different behaviors in terms of performance and energy consumption).

The reconfigurability of the Navigation task is a double-edged sword. While practitioners have more control, it is not easy to configure diverse parameters. Even experienced ROS developers struggle with grasping what each of the many parameters is for, but more importantly, how their values may affect the robot's behavior at runtime (Zheng 2017). This introduces the need for a ROS Navigation task that is more usable for ROS developers, allowing seamless control over the desired navigation behavior in the form of less, but more comprehensive parameters.

R_6 - Raise the level of the navigation task parameters

Researchers are suggested to investigate on methods and techniques for raising the level of abstraction of the configuration of Navigation task parameters, so to make them more usable and intuitive for ROS developers.

R_7 - Autonomous reconfiguration for the Navigation task

The complex behavior of the Navigation task reinforces the need for an automated reconfiguration strategy with less dependency on practitioner setup and optimized parameter configuration.

Finally, even though the Navigation task performs efficiently on-board, without utilizing large amounts of system resources and energy, a higher number of translation and rotation velocity samples in the DWA algorithm cause a significant increase in CPU usage and RAM utilization. The onboard execution can be potentially improved with more efficient algorithms for local path planning, as it may not be needed to simulate the full trajectories for numerous velocity samples when choosing the next local path. Optimizations in local planning algorithms are a good point for further research.

R_8 - Investigate on path planning optimization techniques

Optimize local path planning in order to consume less CPU and memory, which could be started by reducing or reordering the DWA space of translations and velocities.

9 Threats To Validity

This section provides an overview of the main threats to validity, as defined in the classification framework proposed by (Wohlin et al. 2012).

9.1 Internal Validity

The experiment is conducted in a controlled environment with a completely randomized order of execution, which mitigates the effect of particular execution time for different treatments. Each run is executed independently from any previous or following run executions, with a one-minute break between runs. Because all nodes running on either the robot or the remote PC are terminated at the end of each run, this gives enough time for the system resources to stabilize before the next run execution begins.

Our choice of keeping the duration of each run of the experiment to 5 minutes (including the time to swap the batteries and the idle waiting time) might be an internal threat to the validity of the experiment, especially for what concerns the energy consumption of the robot. As discussed in Section 4.4, we did not opt for longer missions in order to keep feasibility purposes since the experiment required to have at least one researcher always present during the missions execution. Replications of the experiment involving longer missions are planned for future work.

Detection result delay is calculated as the difference between the timestamp when the object detection process is completed and the timestamp when the result of the object detection result is received at the robot's side. When object recognition is offloaded, the differences in time synchronization between the remote PC and the robot can have an effect on the obtained result. This effect is mitigated with time synchronization of the PC and the robot in the distributed system via *chrony*, with the remote PC being the NTP server. The *max delay* configuration parameter on the client's side, *i.e.,* the robot, is set to 0.03, indicating that all measurements with a round-trip delay of 0.03 seconds or more will be ignored. This configuration should provide quite high accuracy regarding time synchronization in a distributed system.

Dedicated nodes run onboard the robot for sampling CPU usage, RAM utilization, and energy consumption, namely, *resource_profiler* and *ina219_pro-filer*. While they also account for certain utilization of CPU, RAM, and energy, there is no other way to sample the required metric without performing the sampling process onboard the robot. Whereas the frequency at which the INA219 current sensor sends the measurements is fixed to 200Hz, we opted for keeping the frequency of CPU usage and RAM utilization sampling at 50Hz. This frequency provides a fair amount of samples per second to accurately estimate the effect of offloading strategies on both CPU usage and RAM utilization, whereas it does not account for large resource consumption with the higher frequencies.

Finally, we measured all packets exchanged between the robot and the remote PC, without restriction. In this way, the SSH messages exchanged between the PC and the robot for the purpose of automated mission launching from Robot Runner are also included in the measurements. These SSH messages, where the PC is an SSH client and the root SSH server, are not excluded from the measurements, as the SSH connection and the remote commands for mission automation remain the same for all treatments. This is the reason why there is a number of network messages exchanged between the robot and PC in the obtained results for the treatment in which all tasks are running onboard. Moreover, there is an SSH connection in the opposite direction, with the robot being an SSH client and the PC an SSH server when tasks need to be offloaded to the remote PC. Such SSH messages are also not excluded from the measurement data, as they can be considered as an overhead that comes as a consequence of computation offloading.

9.2 External Validity

The mission we implemented for this study might not be representative of all possible real-world robotic missions. For example, the networking conditions of real-world mobile robotic missions are not always as performant and stable as the one we had during our experiments (*e.g.,*in the case of outdoor search-and-rescue missions). Nevertheless, there are several cases where it can be relatively safely assumed to have a stable network connection with large bandwidth and speed. Examples of such cases include: robots used as museum guides (Hellou et al. 2007; Webster and Ivanov 2022) (this is the case we are reproducing in our experiments), warehouse automation, service robots at home, in hospitals, or in the workplace, etc So, despite the relatively strong assumption of having a fast and stable WiFi network, the network conditions of our experiment can still be considered as representative of a large number of robotic systems used in production. Moreover, we are aware that operating robotic systems in real museums is a more nuanced and complex endeavor than the missions performed in our experiment (Hellou et al. 2007; Webster and Ivanov 2022). The mission performed by the robot in our experiment is a reduced representation of real-world robotic systems used in actual museums; this is by design since we (i) needed to be in *full control* of all relevant aspects of the system being measured and (ii) want to allow third-party researchers to independently verify and replicate our experiment with relatively low effort.

Moreover, despite the Turtlebot is one of the most used robots within the robotics software engineering scientific community (Albonico et al. 2023), we are aware that its hardware specs can be different from those of robots used in the wild (Macenski et al. 2022). This threat is mainly due to the feasibility of the performed experiment and we mitigated it by performing tasks that are common in the robotics domain, such as navigating within an environment, recognizing objects, avoiding obstacles, etc This threat is further mitigated by having a complete replication package with all the details and code related to the experiment, thus allowing future replications to use different robots, missions, and configurations.

The offloadable tasks that are subjects of offloading are selected according to the findings of previous research studies. To that end, we reuse some of the most popular ROS packages that encapsulate the defined tasks. This means that the obtained results are relevant for the majority of ROS community members as good representatives of computation and communication load. The reuse also mitigates possible biases that our own implementation of the tasks would have on the obtained results. The three offloadable tasks have been implemented in such a way as to have little to no dependencies on each other. Such architecture was chosen for the sake of better evaluation of the effects that each of the tasks has on the various collected measures (*e.g.,* mission completion time, energy consumption).

While the *Known map* setup of the experiment is a very typical setup in which the navigation stack is used (with the already constructed map of the environment), the *Unknown map* setup involves an additional frontier exploration step. In such scenarios, the navigation stack does not receive hard-coded coordinates of the goal location, yet they are sent by the exploration package as a result of map processing. Such a setup was not used in the experiment for several reasons. First, it would create a dependency between SLAM and navigation nodes that is undesirable, as explained in the previous paragraph. More importantly, it would include an exploration node in the system design, which could highly influence the measured performance and energy consumption due to its inevitable computation intensity.

The mission considered in this study is executed on ROS1, which is in line with recent work in academia (Albonico et al. 2023; Reichardt et al. 2013). Despite its widespread use in academia, the documentation of ROS1 is also vaster at the time of this study. ROS2 has been largely adopted in recent years (St-Onge and Herath 2022); however, ROS1 is still largely used even in recent projects[42] and will be supported until 2025. It is also reasonable to believe that ROS1 will be active even longer than that (St-Onge and Herath 2022). Furthermore, the studied algorithms are still used in ROS2, and their provided features have not changed. We are aware that ROS2 counts on a more sophisticated network layer with built-in Quality of Service (QoS)[43], which could impact our results, especially those related to network traffic. A replication of this study in the context of ROS2 is considered for future work.

In this study we represent the Cloud to where we do offloading via a laptop within the same WiFI network of the robot. This may affect both the network-dependent and the time-based metrics. A different network configuration, with a real Cloud service, relying on authentication and packet routing could compromise the number and size of packets, and as a consequence, the delay in receiving package responses. Additionally, the real-world network can not be as stable as in the controlled environment of the experiments. This is a requirement for a completely new experiment and should be addressed in future work. In this experiment, we explicitly consider the measurements of the robot, which would not be directly affected by where offloaded computation is executed, and the network latency has already been shown as a critical factor.

[42] https://dev.to/admantium/robotic-projects-reasons-for-switching-from-ros2-to-ros1-4ka9

[43] https://design.ros2.org/articles/ros_on_dds.html

Finally, we are aware that the specific configuration of the DWA algorithm (*i.e.,* the number of velocity samples and the simulation time – see Table 2) might not be fully transferrable to other conditions/setups than those used in this study. We partially mitigated this threat by setting their values to the default ones of the *gmapping* ROS package (which in principles should be covering the most recurrent needs) and to twice the default values (in order to collect evidence about their influence on the dependent variables of this study). Experimenting with additional configurations of the DWA algorithm is planned for future work.

9.3 Construct Validity

It can be noticed in the replication package that, even when all three tasks are executed on board, there are still packages exchanged between the PC and the robot. These are the SSH messages stemming from the automated execution of the mission launch file from the Robot Runner. While their effect on the total number of network packets exists, it should be of equal magnitude for all treatments and hence SSH messages were not filtered out during network traffic measurements. Simultaneously, the offloading process also causes the exchange of SSH messages because the *machine* tag in the launch file opens an SSH connection from the robot to the PC. Such SSH messages were also not filtered out since they can be considered as an overhead that the computation offloading process implies.

While the metrics regarding object recognition performance are divided into three parts, namely, feature extraction, object detection, and transfer of results, in that order, there is an image transfer prior to all these operations. Unfortunately, the metric expressing the delay for image-receiving could not be extracted from the processed log files, yet it is important and it should be taken into account, especially in real-time systems.

Navigation time is calculated as the delay between the moment when the navigation goal is sent from the controller node, always executed on the robot, until the moment when the same node receives the result from the navigation stack that the goal has been reached. The initial idea was to break this metric into three parts, expressed as delay for goal sending, pure navigation processing time, and delay for receiving results. However, the navigation stack does not log the moment when the navigation goal is received, yet it logs the moment when the first global path is found. For this reason, we decided to express total navigation time as one metric, which does include the navigation delay that is undoubtedly more significant when the navigation stack is offloaded.

The effects that a particular parameter configuration used in the primary experiment may have on the obtained results and conclusions drawn are mitigated in the second round of experiments, where the evaluation of parameter effects is performed.

Finally, object recognition CPU usage is higher when it runs locally, even though we confirm that the resources were never exhausted. Experiment results showed that when running in hardware with more processing power and GPU, the package was able to extract image features faster. This may be due to the package features being aligned to the GPU nature, or even due to CPU multithreading. Therefore, other hardware configurations, such as onboard GPU or a more powerful SBC could have a direct impact on the results. However, the results are still sound for small and resource-limited robots, which are considered in this study.

9.4 Conclusion Validity

The most severe threat regarding conclusion validity is the application of the parametric three-way ANOVA test even when assumptions are not fulfilled. However, its non-parametric alternative, *i.e.,* permutation test, is applied first and considered as ground truth, yet only complemented with the three-way ANOVA execution for the sake of comparison. Despite the violation of assumptions, three-way ANOVA always led to the same conclusions as the permutation test, with differences in obtained values mostly seen in the second or third significant digit. The non-parametric alternative to Welch's t-test is always applied when either of its assumptions is not fulfilled in the parameter effects evaluation. Significant two-way and three-way interactions analysis in posthoc test of three-way ANOVA is always performed with Bonfferoni adjustment.

Alternately, the MANOVA test (French et al. 2008) could be performed instead of ANOVA for additional inspection of the effects that dependent variables have on each other. Due to the very large number of 10 dependent variables, it would be very difficult to perform MANOVA, thus we decided to perform a separate ANOVA test for each dependent variable to only compare differences in means of different treatments of the main factors.

10 Related Work

In this section, we give a brief overview of prominent work in computation offloading in the context of robotic systems. All work presented in this section focuses on cloud offloading, in particular, as opposed to this study, where tasks are offloaded to a remote PC.

One of the first works to explore the benefits of computation offloading for robotic systems is by (Arumugam et al. 2010). The main motivation behind the proposed DAvinCI framework is to facilitate the handling of a large environment, particularly for teams of heterogeneous robots, by dividing tasks among robots in multi-robot environments. The authors provide only a proof of concept by offloading the FastSLAM algorithm which they implement as a map/reduce task in Hadoop. They do not provide details regarding the experiment performed to prove their concept, and the only metric considered is execution time. The potential effect of computation offloading on network latency and energy is not reported. (Shakhimardanov et al. 1985) conducted experiments to compare the ROS1 communication layer to the ZeroMQ messaging library. In their study, they measure the communication latency on a publisher/subscriber pattern by changing parameters like the message size, message frequency, and the number of subscribers. However, the study only considers message latency and does not further investigate neither how this impacts the execution time of the ROS tasks nor how this can reflect on the energy consumption of the robot.

(Wendt et al. 2018) study the communication protocol of ROS 1 among Edge containers. They employ different container network modes and analyze their implications on ROS deployments. The authors also propose a proxy server architecture for resolving the identified problems, such as port assignment and bidirectional connection establishment over a bridged container network. Despite the broad approach to the network layer, the study does not consider any ROS packages specifically.

(Parra et al. 2021) propose a domain-specific language to facilitate a model-driven approach to set up QoS profiles in the context of ROS-based systems. The authors also adapt an existing framework to ROS2, enabling QoS monitoring and constraints check. While their work allows the authors to set up QoS profiles at design time, they do not perform QoS measurements.

The main task in the work of (Mukhandi et al. 2010) is real-time moving object recognition and tracking. Three modules in their system, namely, motion detection, object recognition, and stereo-vision, are offloaded to the server or computed locally, depending on the estimated computation and communication, so that the real-time constraints are always satisfied. Variability of network conditions is a huge problem in real-time systems and it must be taken into account when deciding whether to offload a module or not. The authors choose to offload three tasks, as in this study, yet the tasks they offload are all concerned with computer vision. Interestingly, the path planning model was not a subject of computation offloading and it is always executed on board the robot in their system. However, the authors do not provide details about this module whatsoever, thus it is not known why navigation is not a subject of offloading.

Computer vision tasks are a very common subject of computation offloading. (Bistry and Zhang 2010) propose a system where part of the image processing is executed on a smart camera while the other part is offloaded to the cloud. They chose the SIFT algorithm for object detection due to its efficiency, but its high computational effort is the motivation for proposing a distributed architecture where the computation effort is shared between a smart camera and the cloud. While sharing computation effort between the robot and the remote machine can have potential benefits on resource and energy utilization, the approach presented in this work requires the installation of a smart camera. Such a sophisticated sensor most likely accounts for additional energy consumption overhead when attached to the robot, which is not evaluated by the authors.

(Wu et al. 2012) have chosen another computer vision algorithm, SURF, to prove the proposed concept. The main contributions of their work consist of a real-time transport protocol intended for transmitting a large volume of image data to the cloud, as well as a control law and scheduling strategy that can reduce the network communication overhead.

V-SLAM represents a combination of two commonly offloaded tasks: SLAM and computer vision. This task was a subject of research conducted by (Basili et al. 2015). In V-SLAM, the map is constructed according to camera images that the robot records while moving around the environment. For precise maps, these images have to be recorded at a very high frequency, which also means that a large amount of data would be transmitted over the network if this task is offloaded. At the same time, feature detection algorithms extract key features that need to be compared with images in the database in real time, which requires high computation power, but also large storage for the images. The conflicting requirements make the decision to offload V-SLAM very difficult. The authors use the Oriented FAST and Rotated BRIEF (ORB) algorithm for feature detection and the brute force matching algorithm to search the images and construct the map, particularly optimized for the purpose of V-SLAM.

Researchers in the field of Cloud computing carried out studies related to ours. (Mukhandi et al. 2019) propose an approach for securing ROS-based systems using MQTT communication. This allows the integration of robotic systems with other types of systems, such as Internet of Things (IoT) devices/systems. The authors also perform real-world experiments on the performance of the additional security layer. Despite a security layer being a basic requirement in any system and is a strong candidate for further experiments, their work branches ROS, while we seek to analyze a more transparent/non-invasive deployment focussing on the official ROS distributions. (Masaba et al. 2019) propose the ROS Communication Benchmarking Tool (ROS-CBT), a tool that simulates communication among multiple robots. Their tool allows testing a multi-robot system before real-world deployment, and can specifically evaluate communication link performance at run time. However, they do not conduct an empirical evaluation of their proposed tool via real robots. (Fukui et al. 2022) study a multi-SLAM deployment with ROS working as a middleware. They also use Turtlebot3 as the robotic platform in their experiments. Their work has a different purpose, considering the communication among multiple SLAM nodes, instead of offloading the SLAM node into the Cloud.

While cloud computing has been exploited in the robotics field for a decade now, researchers are turning to other paradigms as well, such as Edge and Fog computing. The main idea in the work of (Sarker et al. 2019) is to add Edge and Fog computing layers between the robot and the cloud because these layers can provide the computation power of the cloud yet they would be located closer to the robot, thus minimizing network latency and variability. Their motivation is to enhance the energy efficiency and operational performance of small indoor robots with limited processing power. This is one of the few works that is centered around power consumption and reports the power measurements in the experiment.

In very recent and cutting-edge work, (Chinchali et al. 2021) advocate that the process of deciding when to offload a task and when not, given current network conditions, can be modeled as a sequential decision-making problem. The decision problem is thus based on the learning approach, using deep reinforcement learning. As a proof-of-concept, the authors have used state-of-the-art DNNs to implement the offloading strategy for a computer vision task and they report a performance improvement of up to 2.6 times.

11 Conclusion

In this study, we empirically characterized the impact of computation offloading strategies on the performance and energy consumption of ROS-based systems. To this aim, we developed a robotic system that consists of three tasks that are considered good candidates for computation offloading, namely: SLAM/localization, navigation, and object recognition. The offloading strategies consist of offloading each of those tasks to a remote machine. The experiment is automated and orchestrated independently from the mission itself via a dedicated experimental infrastructure. We complemented the main finding with an additional set of experiments to evaluate the effect of different parameter configurations on the performance and energy consumption of the measured robot.

The experiment results show that offloading object recognition significantly reduces the resource utilization and energy consumption of the robot. Additionally, object recognition is also more efficient when offloaded due to the availability of higher computation power on the remote machine. However, the large volume of images exchanged over the network when offloading object recognition requires compatible network bandwidth. The additional experiment shows that the choice of image resolution and image frame is a sensible one. Differently from object recognition, SLAM/localization and navigation can be executed efficiently on board, as they do not utilize as many resources and energy as compared to object recognition.

Several future research directions are reported in the Discussion section of this paper. Other possible directions for future work include a replication of this study for comparing the current results with the ones on ROS2. It is also important to evaluate further image processing techniques, including those based on reinforcement learning and DNN would also lead us to a deeper understanding of the current results. Object recognition could be potentially performed more efficiently with other techniques. New hardware configurations are also important, for which we highlight the NVIDIA Jetson[44] as the main SBC, which might impact the results because a more significant workload could be run onboard.

Data Availibility Statement The datasets generated and analysed during the current study are available in the following GitHub repository: https://github.com/S2-group/EMSE-2023-ros-offloading-tradeoffs-rep-pkg.

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

Abdelrasoul Y, Saman ABSHM, Sebastian P (2016) A quantitative study of tuning ros gmapping parameters and their effect on performing indoor 2d slam. In 2016 2nd IEEE international symposium on robotics and manufacturing automation (ROMA), pp 1-6. IEEE

Albonico M, Đorđevic M, Hamer E, Malavolta I (2023) Software engineering research on the robot operating system: A systematic mapping study. J Syst Softw 197:111574

Anderson M, Braak CT (2003) Permutation tests for multi-factorial analysis of variance. J Stat Comput Simul 73(2):85–113

Arumugam R, Enti VR, Bingbing L, XiaojunW, Baskaran K, Kong FF, Kumar AS, Meng KD, Kit GW (2010) Davinci: A cloud computing framework for service robots. In 2010 IEEE international conference on robotics and automation, pp 3084–3089. IEEE

Basili V, Caldiera G, Rombach H (1994) Goal question metric paradigm. Encycl Softw Eng 1:528–532

Benavidez P, Muppidi M, Rad P, Prevost JJ, Jamshidi M, Brown L (2015) Cloud-based realtime robotic visual slam. In 2015 Annual IEEE Systems Conference (SysCon) Proceedings, pp 773–777. IEEE

Bistry H, Zhang J (2010) A cloud computing approach to complex robot vision tasks using smart camera systems. In 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp 3195–3200. IEEE

Bonferroni C (1963) Teoria statistica delle classi e calcolo delle probabilita. Pubbl del R Ist Super di Sci Econ e Commericiali di Firenze 8:3–62

Chinchali S, Sharma A, Harrison J, Elhafsi A, Kang D, Pergament E, Cidon E, Katti S, Pavone M (2021) Network offloading policies for cloud robotics: a learning-based approach. Auton Robot, pp 1–16

Dey S, Mukherjee A (2016) Robotic slam: a review from fog computing and mobile edge computing perspective. In Adjunct Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing Networking and Services, pp 153–158

Doriya R, Chakraborty P, Nandi GC (2012) Robotic services in cloud computing paradigm. In 2012 International Symposium on Cloud and Services Computing, pp 80–83. IEEE

dos Reis WPN, Morandin O, Vivaldini KCT (2019) A quantitative study of tuning ros adaptive monte carlo localization parameters and their effect on an agv localization. In 2019 19th International Conference on Advanced Robotics (ICAR), pp 302–307. IEEE

Foote T, Marder-Eppstein E, Meeussen W (2021) tf. Available: http://wiki.ros.org/tf. Accessed on: 03 Aug 2021 [Online]

French A, Macedo M, Poulsen J, Waterson T, Yu A (2008) Multivariate analysis of variance (manova)

[44] https://www.nvidiastore.com.br/jetson

Fukui M, Ishiwata Y, Ohkawa T, Sugaya M (2022) Iot edge server ros node allocation method for multi-slam on many-core. In 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), pp 421–426

Garro R, Orozco J, Ordinez L, Santos R (2014) Estrategias de diseño basadas en patrones de un subsistema de movimiento para un robot pulverizador. In 2014 IEEE Biennial Congress of Argentina (ARGENCON), pp 405–410. IEEE

Gomez K, Riggio R, Rasheed T, Granelli F (2011) Analysing the energy consumption behaviour of wifi networks. In 2011 IEEE Online Conference on Green Communications, pp 98–104

Grisetti G, Stachniss C, Burgard W (2007) Improved techniques for grid mapping with rao-blackwellized particle filters. IEEE Trans Robot 23(1):34–46

Hellou M, Lim JY, Gasteiger N, Jang M, Ahn HS (2022) Technical methods for social robots in museum settings: An overview of the literature. In J Soc Robotics, pp 1–20, 2022

Hu G, Tay WP, Wen Y (2012) Cloud robotics: architecture, challenges and applications. IEEE Netw 26(3):21–28

Indelman V, Williams S, Kaess M, Dellaert F (2013) Information fusion in navigation systems via factor graph based incremental smoothing. Robot Auton Syst 61(8):721–738

Ito PK (1980) 7 robustness of anova and manova test procedures. Handb Stat 1:199–236

Kassambara A (2019) Anova in r: The ultimate guide. Available: https://www.datanovia.com/en/lessons/anova-in-r/, Nov 2019. Accessed on: 03 Aug 2021 [Online]

Kassambara A (2019) T-test in r: The ultimate guide. Available: https://www.datanovia.com/en/lessons/t-test-in-r/, Nov 2019. Accessed on: 03 Aug 2021 [Online]

Kehoe B, Patil S, Abbeel P, Goldberg K (2015) A survey of research on cloud robotics and automation. IEEE Trans Autom Sci Eng 12(2):398–409

Koubâa A (2020) Service-oriented computing in robotic. Encycl Robot, pp 1–12

Labbe M (2021) find_object_2d. Available: http://wiki.ros.org/find_object_2d. Accessed on: 03 Aug 2021 [Online]

Lee J, Wang J, Crandall D, Šabanović S, Fox G (2017) Real-time, cloud-based object detection for unmanned aerial vehicles. In 2017 First IEEE International Conference on Robotic Computing (IRC), pp 36–43. IEEE

Macenski S, Foote T, Gerkey B, Lalancette C, Woodall W (2022) Robot operating system 2: Design, architecture, and uses in the wild. Sci Robot, 7(66):eabm6074

Marder-Eppstein E, Berger E, Foote T, Gerkey B, Konolige K (2010) The office marathon: Robust navigation in an indoor office environment. In 2010 IEEE international conference on robotics and automation, pp 300–307. IEEE

Maruyama Y, Kato S, Azumi T (2016) Exploring the performance of ros2. In Proceedings of the 13th International Conference on Embedded Software, pp 1–10

Masaba K, Li AQ (2019) Ros-cbt: Communication benchmarking tool for the robot operating system: Extended abstract. In 2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS), pages 1–3

Đorđevi M, Albonico M, Lewis GA, Malavolta I, Lago P (2022) Study Replication Package. https://github.com/S2-group/EMSE-2023-ros-offloading-tradeoffs-rep-pkg

Mukhandi M, Portugal D, Pereira S, Couceiro MS (2019) A novel solution for securing robot communications based on the mqtt protocol and ros. In 2019 IEEE/SICE International Symposium on System Integration (SII), pp 608–613

Yamini Nimmagadda, Karthik Kumar, Yung-Hsiang Lu, and CS George Lee (2010) Real-time moving object recognition and tracking using computation offloading. In 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp 2449–2455. IEEE

Parra S, Schneider S, Hochgeschwender N (2021) Specifying qos requirements and capabilities for component-based robot software. In 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE), pp 29–36

Profanterb S, Tekat A, Dorofeev K, Rickert M, Knoll A (2019) Opc ua versus ros, dds, and mqtt: Performance evaluation of industry 4.0 protocols. In 2019 IEEE International Conference on Industrial Technology (ICIT), pp 955–962

Putra IA, Prajitno P (2019) Parameter tuning of g-mapping slam (simultaneous localization and mapping) on mobile robot with laser-range finder 360° sensor. In 2019 International Seminar on Research of Information Technology and Intelligent Systems (ISRITI), pp 148–153. IEEE

Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Wheeler R, Ng AY et al (2009) Ros: an open-source robot operating system. In ICRA workshop on open source software, volume 3, pp 5. Kobe, Japan

Rad PA, Hofmann D, Mendez SAP, Goehringer D (2021) Optimized deep learning object recognition for drones using embedded gpu. In 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), pp 1–7

Reichardt M, Föhst T, Berns K (2013) On software quality-motivated design of a real-time framework for complex robot control systems. Electron Commun EASST, 60

Reichardt M, Föhst T, Fleischmann P, Arndt M, Berns K (2013) Principles in framework design applied in networked robotics. IFAC Proc 46(29):150–155

Santos JM, Portugal D, Rocha RP (2013) An evaluation of 2d slam techniques available in robot operating system. In 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR), pages 1–6. IEEE

Sarker VK, Queralta JP, GiaTN, Tenhunen H, Westerlund T (2019) Offloading slam for indoor mobile robots with edge-fog-cloud computing. In 2019 1st international conference on advances in science, engineering and robotics technology (ICASERT), pp 1–6. IEEE

Schultz BB (1985) Levene's test for relative variation. Syst Zool 34(4):449–456

Shakhimardanov A, Hochgeschwender N, Reckhaus M, Kraetzschmar GK (2011) Analysis of software connectors in robotics. In 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp 1030–1035

Shamshiri RR, Hameed IA, Pitonakova L, Weltzien C, Balasundram SK, Yule IJ, Grift TE, Chowdhary G (2018) Simulation software and virtual environments for acceleration of agricultural robotics: Features highlights and performance comparison. Int J Agric Biol Eng 11(4):15–31

St Lars, Wold Svante et al (1989) Analysis of variance (anova). Chemom Intell Lab Syst 6(4):259–272

St-Onge D, Herath D (2022) The Robot Operating System (ROS1 &2): Programming Paradigms and Deployment, pp 105–126 09

Swanborn S, Malavolta I (2021) Robot runner: A tool for automatically executing experiments on robotics software. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp 33–36. IEEE

Thrun S, Fox D, Burgard W, Dellaert F (2001) Robust monte carlo localization for mobile robots. Artif Intell 128(1–2):99–141

Tripathy A, van Deventer J, Paniagua C, Delsing J (2022) Interoperability between ros and opc ua: A local cloud-based approach. In 2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS), pp 1–5

Webster C, Ivanov S (2022) Public perceptions of the appropriateness of robots in museums and galleries. J Smart Tour 2(1):33–39

Welch BL (1947) The generalization of 'student's' problem when several different population varlances are involved. Biometrika 34(1–2):28–35

Wendt A, Schüppstuhl T (2022) Proxying ros communications - enabling containerized ros deployments in distributed multi-host environments. In 2022 IEEE/SICE International Symposium on System Integration (SII), pp 265–270

Wienke J, Wigand D, Koster N, Wrede S (2018) Model-based performance testing for robotics software components. In 2018 Second IEEE International Conference on Robotic Computing (IRC), pp 25–32

ROS wiki (2021) Basic navigation tuning guide. Available: http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide. Accessed on: 03 Aug 2021 [Online]

ROS wiki (2021) <machine> tag. Available: http://wiki.ros.org/roslaunch/XML/machine. Accessed on: 03 August 2021 [Online]

ROS wiki (2021) Network setup in ros. Available: http://wiki.ros.org/ROS/NetworkSetup. Accessed on: 03 Aug 2021 [Online]

ROS wiki (2021) Setup and configuration of the navigation stack on a robot. Available: http://wiki.ros.org/navigation/Tutorials/RobotSetup. Accessed on: 03 Aug 2021 [Online]

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers

Wu H, Lou L, Chen CC, Hirche S, Kuhnlenz K (2012) Cloud-based networked visual servo control. IEEE Trans Ind Electron 60(2):554–566

Zheng K (2017) Ros navigation tuning guide. arXiv preprint arXiv:1706.09068

Authors and Affiliations

Milica Đorđević[1] · Michel Albonico[2] · Grace A. Lewis[3] · Ivano Malavolta[1] · Patricia Lago[1]

Milica Đorđević
m.djordjevic@student.vu.nl

Michel Albonico
michelalbonico@utfpr.edu.br

Grace A. Lewis
glewis@sei.cmu.edu

Patricia Lago
p.lago@vu.nl

1 Vrije Universiteit Amsterdam, Amsterdam, Netherlands

2 Federal University of Technology, Paraná (UTFPR), Francisco Beltrão, Brazil

3 Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA