Department of Informatics
Technical University of Munich

Master's Thesis in Robotics, Cognition, Intelligence

Analysis of Edge Computing Offloading Strategies for Autonomous Mobile Robots

Analyse von Edge Computing Offloading Strategien für autonome mobile Roboter

Supervisor Prof. Dr.-Ing. habil. Alois C. Knoll

Advisor Robin Dietrich, M.Sc.

Author Xiyan Su

Date May 12, 2023 in Garching

Disclaimer

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, May 12, 2023

(Xiyan Su)

Abstract

AMRs run various computationally intensive algorithms on their onboard systems, such as SLAM, perception, navigation, and path planning. However, their onboard systems usually have limited computation and energy resources. To investigate the effects of different offloading strategies, an offloading framework is designed and implemented using ROS2 for an object detection task using YOLOv5. Then, baseline offloading strategies are implemented and evaluated on defined metrics with experiments. After evaluating the baseline strategies, a dynamic offloading strategy that makes offloading decisions according to the states of the system is developed and implemented. In the end, an experiment on an actual robotic system using both Ethernet and Wi-Fi connections is carried out to evaluate the dynamic offloading strategy against the baselines. The results show that the dynamic offloading strategy not only improves the average precision of the object detection task but also reduces the resource usage of the onboard system compared to the baseline strategies.

Zusammenfassung

AMRs führen verschiedene rechenaufwändige Algorithmen auf ihren Onboard-Systemen aus, wie z.B. SLAM, Wahrnehmung, Navigation und Pfadplanung. Ihre Onboard-Systeme verfügen jedoch in der Regel über begrenzte Rechen- und Energieressourcen. Um die Auswirkungen verschiedener Offloading-Strategien zu untersuchen, wurde ein Offloading-Framework für eine Objekterkennungsaufgabe mit YOLOv5 entworfen und mit ROS2 implementiert. Dann wurden Baseline-Offloading-Strategien implementiert und anhand definierter Metriken mit Experimenten bewertet. Nach der Bewertung der Baseline-Strategien wurde eine dynamische Offloading-Strategie entwickelt und implementiert, die Offloading-Entscheidungen gemäß dem Zustand des Systems trifft. Schließlich wurde ein Experiment an einem Robotersystem mit Ethernet- und Wi-Fi-Verbindungen durchgeführt, um die dynamische Offloading-Strategie mit den Baseline-Strategien zu vergleichen. Die Ergebnisse zeigen, dass die dynamische Offloading-Strategie nicht nur die durchschnittliche Präzision der Objekterkennungsaufgabe verbessert, sondern auch den Ressourcenverbrauch des Onboard-Systems im Vergleich zu den Baseline-Strategien reduziert.

Acknowledgement

The work of this thesis is carried out at Autonomous Mobile System Research Lab (AMSRL) at Intel Labs in Karlsruhe, Germany. The main research code base and the hardware in use on which this thesis is built are provided by AMSRL. The thesis is co-supervised by AMSRL and the chair of Robotics, Artificial Intelligence and Real-time Systems at Technical University of Munich (TUM).

First, I would like to thank AMSRL at Intel Labs for providing me with the opportunity to finish my thesis work and for providing me with the needed materials to implement and conduct the experiments.

Furthermore, I would like to thank Frederik Pasch, M.Sc. (Intel Labs), and Robin Dietrich, M.Sc. (TUM) for their continuous supervision and insightful inputs in the work and writing of the thesis.

In the end, I would also thank my partner Pingjun Hong, M.A. for her loving support throughout the entire thesis.

Contents

List of Algorithms

List of Figures

List of Tables

Special Terms

cAdvisor cAdvisor (Container Advisor) provides container users an understanding of the resource usage and performance characteristics of their running containers. 27

Docker Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. 24, 26, 32

Fast-DDS eprosima Fast DDS (formerly Fast RTPS) is a C++ implementation of the DDS (Data Distribution Service) standard of the OMG (Object Management Group). 10, 32, 49

Gazebo Gazebo is an open-source 3D robotics simulator. 10, 11, 29–31, 47, 48

Linux GNU/Linux is a family of open-source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds. 26, 27, 32, 33

OpenVINO OpenVINO is an open-source toolkit for optimizing and deploying deep learning models on Intel hardware. It provides boosted deep learning performance for vision, audio, and language models from popular frameworks like TensorFlow, PyTorch, and more. 8, 24

protobuf Protocol Buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. 11

psutil Process and system utilities (psutil) is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. 27

Python Python is a high-level, general-purpose programming language. 9

PyTorch PyTorch is a machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing. 8, 9, 24, 25, 33

Wi-Fi Wi-Fi is a family of wireless network protocols based on the IEEE 802.11 family of standards. 33, 36, 46, 47

YOLO YOLO, an acronym for You Only Look Once, is a one-stage object detection algorithm. 8, 9, 24, 25, 29, 36, 47, 48

Acronyms

A2C Advantage Actor Critic. 15

AMR Autonomous Mobile Robot. 1–3, 7, 13, 17, 19, 20, 23–25, 27, 29–33, 35, 36, 39, 43, 45, 47, 48

AMSRL Autonomous Mobile System Research Lab. v, 27, 29, 34

AP Average Precision. 34–37, 39, 43, 45, 46, 49

CNN Convolutional Neural Network. 8

COCO Common Object in Context. 25, 48

CPU Central Processing Unit. 18–20, 24–27, 31–33, 39, 47

DDS Data Distribution System. 9, 10, 26, 33

Deep RL Deep Reinforcement Learning. 14

DL Deep Learning. 7, 8

DNN Deep Neural Network. 5, 7, 8, 14, 15, 24

DQN Deep Q-learning Network. 14, 15

FPS frames per second. 25, 36, 39

GPU Graphics Processing Unit. 24–26, 32, 33

IFB Intermediate Functional Block. 11

IO Input/Output. 26

IoT Internet of Things. 7

KPI Key Performance Indicator. 7

mAP Mean Average Precision. 24, 25

MCC Multi-Access Cloud Computing. 5

MEC Multi-Access Edge Computing. 5

NE Nash Equilibrium. 14

NetEm Network Emulator. 11, 24, 32, 33, 45, 49

NMS Non-maximum Suppression. 25

NUC Intel Next Unit of Computing. 24, 31, 33, 39

QoS Quality of Service. 10, 31–34, 49

RAN Radio Access Network. 5

RAPL Running Average Power Limit. 27

RoI Region of Interest. 8

ROS Robot Operating System. 2, 9–11, 23–27, 30–35, 45, 47, 49

RPN Region Proposal Network. 8

RTPS Real-time Publish-Subscribe. 9

RTT Round-Trip Time. 20, 34, 36, 38, 39, 44–47, 49

SDFormat Simulation Description Format. 11

SLAM Simultaneous Localization and Mapping. 1, 7, 19, 23

SMA Simple Moving Average. 23, 25

TCP Transmission Control Protocol. 9

TSN Time-Sensitive Network. 7

TUM Technical University of Munich. v

UDP User Datagram Protocol. 9, 26

VM Virtual Machine. 5

WAN Wide Area Network. 5

Chapter 1

Introduction

1.1 Problem Statement

Autonomous Mobile Robots (AMRs) have gained enormous significance in the industrial sector over the last decade. They are used to improve operational efficiency, precision, and safety. In order to understand and navigate through the environment independently, the AMRs are required to run computation- and memory-intensive algorithms related to image processing, path planning, Simultaneous Localization and Mapping (SLAM), and learning [Sae+21]. However, this poses a challenge for the robot's onboard system, which has limited computational capabilities due to size and cost limitations as well as battery life [Bax+22]. Furthermore, these algorithms are time-sensitive and can cause operational failure for the AMRs if the latency restrictions cannot be satisfied. It becomes essential for large automated factories using AMRs to address this problem.

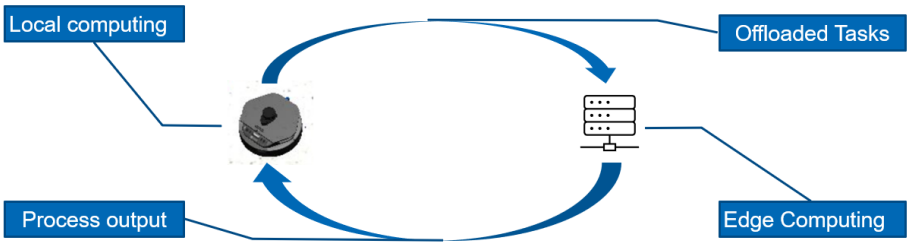

Figure 1.1: This figure shows the computation offloading between the AMR and the edge computer. The AMR can choose to compute the tasks locally or to offload them to the edge computer. If the AMR chooses to offload, the edge computer will compute the tasks and send the processed output back to the AMR via the network

Edge computing as an evolution of cloud computing brings application hosting from data centers down to the edge of the network, where the data were initially collected, to achieve low latency and bandwidth efficiency [Lin+19]. For latency-sensitive tasks on AMRs, such as perception and navigation, Edge computing offers an opportunity to enable the AMRs with limited resources by offloading costly computation tasks to the edge, while only a small portion of the computation remains on the AMR's on-board system. An illustration of computation offloading between the AMR and the edge computer is shown in fig. 1.1.

However, depending on the application scenarios, offloading certain tasks from AMRs to the edge at all times may not be possible, beneficial, or even feasible due to the network's latency, dynamic network changes, and resource availability. Baxi et al. [Bax+22] point out that a simple perception task using an RGB-D camera can cause over 100 ms sensing-to-actuation round-trip latency and over 400 Mbps network bandwidth usage. On the other hand, offloading computational workloads to the edge could also impact the AMR's safety,

availability as well as on-board resources. The exact influence on the robotic system will depend on the chosen offloading strategy. Therefore, it is necessary to investigate the effects of certain offloading strategies on the AMR's safety, availability, and task performance.

1.2 Motivation

A number of works have investigated the computation offloading strategies as a purely mathematical problem and have proposed different algorithms to solve the problem. However, few of them consider the real application in AMRs, where the dynamic network changes and onboard resource limitations can affect the performance of the offloading strategies greatly. Therefore, the purpose of this thesis is to investigate the influence of different offloading strategies using actual robotic systems and edge computers in an experimental way.

With the results from the experiments, this thesis is trying to answer the following research questions: What effects do different offloading strategies have on the specified metrics? Furthermore, this thesis is trying to answer the question: If and how the effects of different offloading strategies vary under different circumstances and if they have any (application-specific) constraints. Finally, this thesis is trying to gain insights if more complex strategies to achieve better results on the metrics.

Furthermore, this thesis aims to develop a dynamic offloading strategy that can detect the limitations of the available resources on different systems and can adapt to dynamic changes in network conditions, in order to minimize the execution latency of the offloaded computation task and improve its performance by using edge computers equipped with more computation resources.

1.3 Research Methodology

In order to carry out the experiments, an offloading framework for robot perception is implemented using Robot Operating System (ROS)2 [Mac+22] and used to analyze the effects of different offloading strategies. More specifically, this thesis considers a 2D object detection task using the YOLOv5 models, proposed by Jocher et al. [Joc+22]. YOLOv5n, a smaller variant of the perception model, will be deployed on the AMR's on-board system, while the edge counterpart uses a more accurate but also a more complex variant, e.g., YOLOv5l. The environment is an industrial warehouse containing different objects as obstacles, which is a common use case for AMRs. The AMR is equipped with camera sensors and uses navigation 2, proposed by Macenski et al. [Mac+20], to navigate through a user-defined route.

First, the baseline strategies are investigated. More specifically, an "edge only" strategy that only offloads to the edge computer, a "robot only" strategy that only uses the AMR's onboard system, and a series of strategies that offload to the edge with different ratios are used as the baseline strategies. This step aims to investigate the influence of different offloading strategies on the performance of the perception task and the resource usage of the AMR and the edge computer as well as the network. Then, a dynamic offloading strategy that makes decisions based on runtime parameters is developed and implemented, such as latency and AMR's CPU usage and power consumption. This dynamic offloading strategy aims to minimize latency and improve the performance of the object detection task, inspired by the problem formulation of Ning et al. [Nin+19]. Additional constraints subjected to power consumption and network bandwidth are applied to the algorithm.

In the end, an evaluation framework is implemented based on important metrics of the

object detection task performance and the AMR's onboard resources. The baseline offloading strategies are first evaluated in simulation to verify the functionalities and the robustness of the offloading framework. Then, the experiments are carried out with an actual robotic system and evaluate the baseline strategies on the defined metrics. With results from the experiments with the baseline strategies, the dynamic offloading strategy is developed and evaluated against the baseline strategies by conducting experiments using Ethernet and Wi-Fi connections.

Chapter 2

Background

This chapter describes the fundamental background in different fields that is needed to understand the content of the thesis. An introduction to edge computing is presented in section 2.1. In section 2.2, the Deep Neural Network (DNN) approach to detect objects in images is introduced. Finally, the frameworks and software tools used for the implementation are introduced in section 2.3.

2.1 Edge Computing

Edge computing as an evolution of cloud computing brings the application hosting from centralized data centers down to the network edge, closer to the consumers and the data generated by applications, in order to reduce the latency and improve the bandwidth efficiency [Kek+18]. An illustration of the relation among cloud, edge, and end devices is shown in fig. 2.1

2.1.1 Development of edge computing

Over the last two decades, cloud computing starts to show its power in industry, business, and daily life by leveraging powerful infrastructures, such as remote data centers, to augment the computation capabilities of less powerful devices [Lin+19]. Especially with the booming of mobile devices, e.g., smartphones, Multi-Access Cloud Computing (MCC), as the integration of cloud computing and mobile computing, has become recognized as the new generation of computing. However, cloud computing often cannot satisfy the latency requirements for time-sensitive tasks, because the computation task has to go through multiple hops, i.e., multiple network segments separated by routers. As a result, many efforts have gone into reducing the latency by placing the computing units closer to the end devices.

To leverage the Wide Area Network (WAN) delays, jitters, congestion, and failures, Satyanarayanan et al. [Sat+09] first used Virtual Machines (VMs) to instantiate customized service on a nearby "cloudlet", which is a resource-abundant computer or cluster of computers placed on-premise, i.e., at the edge of the Internet near the end devices, as a proof of concept for edge computing. Later on, Bonomi et al. [Bon+12] introduced fog computing, which allows the end devices to use both edge and cloud computing and defines the connection between the end devices, the edge, and the cloud. In recent years, as an effort to standardize edge computing, ETSI [Kek+18] announced the Multi-Access Edge Computing (MEC) focusing on integrating edge computing with Radio Access Network (RAN), especially with 5G network. On the other hand, various companies, including Intel, founded the OpenFog Consortium

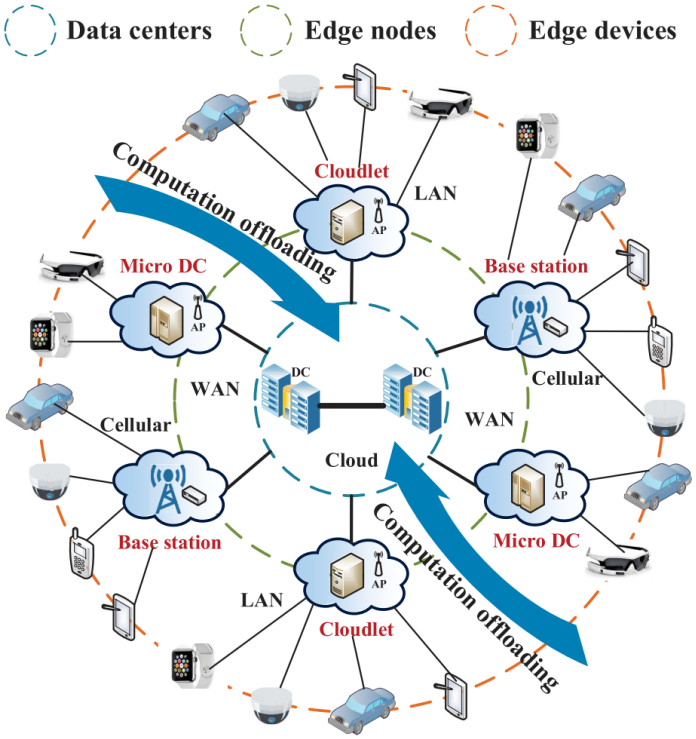

Figure 2.1: The figure shows a typical three-tier architecture of edge computing with the flow of computation offloading [Lin+19]. The edge devices, sometimes also called end devices or mobile devices, are the devices computing at the edge of the network. An edge node is a computing node located at the edge of the network near the edge devices. The cloud usually consists of data centers that are far away from the edge devices. The tree-tier computation offloading architecture allows edge devices to offload to edge nodes and/or the cloud.

and introduced the OpenFog as an open architectural framework for fog computing. Then, as a reconciliation of the two standards, ETSI and OpenFog consortium start to collaborate on the application of edge and fog computing. Another approach is the Time-Sensitive Network (TSN) [Sat+23]. With a focus on reducing network latency and jitters, TSN was initially proposed as a standard for wired communication. However, in recent years, researchers have also gone into the TSN-5G integration to achieve Key Performance Indicators (KPIs) of Industrial 4.0 in network latency and jitters.

In the industrial domain, Internet of Thingss (IoTs) supported by Industrial 4.0 have been a key enabler for industrial automation. Therefore, Intel published the edge infrastructure handbook providing ready-to-deploy time-sensitive solutions for IoT applications [Cor21].

2.1.2 Edge robotics

AMRs run a great number of modules on their onboard system, such as perception, SLAM, navigation, and path planning. However, the robotic onboard resources are usually limited due to cost and spatial reasons. Offloading some of the computation to the edge can alleviate the workload of the robot's onboard system. This approach, sometimes also called edge robotics, is an ideal use case for edge computing.

Many works have already proven the concept of edge robotics. Huang et al. [Hua+22] show that using edge in multi-robot SLAM algorithm reduces the processing latency. Sossalla et al. [Sos+22] took a step further by offloading navigation, SLAM, and control functionalities to the edge and ensured the network latency and throughput requirements by using a 5G connection. Xie et al. [Xie+21] achieved real-time instance segmentation for mobile robots with limited onboard resources. In the works of Fu et al. [Fu+19] and Tanwani et al. [Tan+], they both show that offloading the object recognition tasks reduces the execution latency and improves the task performance.

Although edge computation offloading in robotics does show promising results in reducing execution latency and improving task performance, Saeik et al. [Sae+21] pointed out that the dynamic network conditions and the resource allocation are still open challenges in edge robotics. This leaves several open questions of application partitioning, offloading decision-making, and distributed task execution.

2.2 Object Detection

Object detection, as an essential part of robotic perception, is one of the algorithms the AMR requires to understand the environment. It provides bounding boxes and classification of potential objects in the robot's view. In the last two decades, object detection has undergone a paradigm shift from statistical classifiers using hand-crafted features to Deep Neural Network (DNN) using general-purpose learning procedures and large datasets [RLS18]. Deep Learning (DL)-based methods not only improve the performance of object detection but also increase the computation workload of the robotic system, making it harder to achieve real-time object detection on the robot with complex DNN models. Instead of increasing the computation capabilities of the onboard hardware, edge robotics, as a method to alleviate the computation workload of the robot's onboard system, can help achieve real-time execution in a more cost-efficient way.

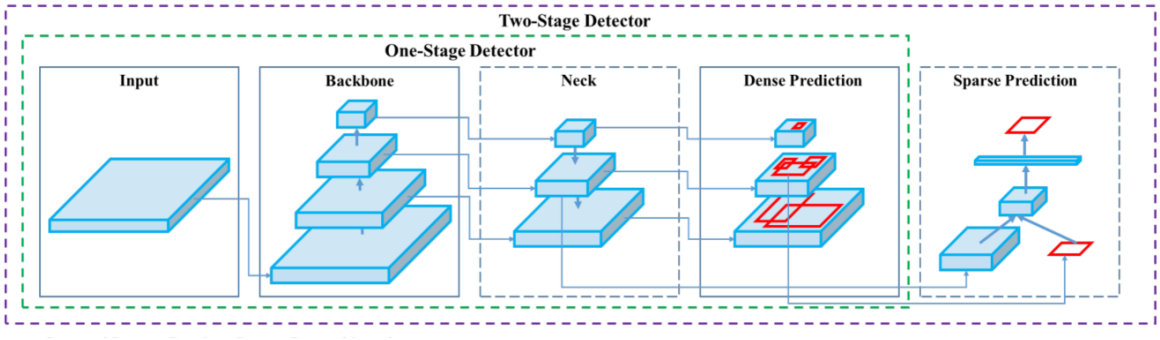

Figure 2.2: Object detection algorithms can be divided into two categories based on the architecture: two-stage detector and one-stage detector. Both architectures consist of a backbone, a neck, and a head. [BWL20]

2.2.1 DNN-based object detection

State-of-the-art object detection methods usually consist of a backbone, a neck, and a head, as shown in fig. 2.2. The backbone is usually made up of pre-trained Convolutional Neural Network (CNN) layers that are used for feature extraction, such as VGG16 [SZ15] and ResNet-50 [He+16]. The neck usually consists of some layers used to collect feature maps from different stages. Finally, the head is responsible for predicting bounding boxes and classes of objects.

As shown in fig. 2.2, the object detection algorithms can be divided into two categories based on the structure of the head: one-stage detectors and two-stage detectors. Two-stage detectors, such as Faster R-CNN [Ren+15] and Mask-RCNN [He+17], have a Region Proposal Network (RPN) that generates proposals for Region of Interest (RoI). Then, the regressor and classifier detect a bounding box and a class for that RoI. On the other hand, one-stage detectors, such as SSD [Liu+16] and YOLO [Red+16], have only one network to do both. Benefiting from the simple structure of the head, one-stage detectors are faster but usually come with a trade-off of slightly lower precision in detection. However, thanks to their faster inference time, the one-stage detectors can satisfy the real-time requirements of robot perception.

With the development of DL algorithms, DL frameworks have also undergone major development in the last decade. Frameworks, such as TensorFlow [Mar+15] and PyTorch [Pas+19], have already been the de facto standards for implementing and training the DNN. Moreover, toolkits like OpenVINO [Dem+19] can optimize a DNN and deploy it on Intel hardware.

2.2.2 YOLOv5

The YOLO architecture [Red+16] as a one-stage detector uses one network to generate the proposals for RoI and compute regression and classification of objects by dividing the image into many grids and generating detection relative to the center of the grid. This is also known as the anchor-based detector. With the initial success, the YOLO architecture has also undergone a fruitful development. As of the time of the works of this thesis, YOLOv5 [Joc+22] is the current state-of-the-art algorithm for one-stage object detections.

YOLOv5 also provides models of various sizes with different object detection performances and inference times. This is easy for deploying them on different platforms. For example, since the robot's onboard system has limited resources, it can use a simple model like the YOLOv5n. For the edge, complex models, such as YOLOv5l and YOLOv5x, can be

used. Furthermore, YOLOv5 can be easily deployed with PyTorch, which supports Nvidia GPU deployment via CUDA toolkit.

2.3 Frameworks and Tools

During the implementation, several frameworks and software tools are used for the offloading framework and the experiments. This section gives a short introduction of the involved frameworks and software tools.

2.3.1 Robot Operating System

ROS is an open-source robotic operation system, proposed by Quigley et al. [Qui+09]. As described by the author himself, ROS is not an operating system in the traditional sense, but rather a structured communication layer above the host operating systems. Nevertheless, the emergence of the ROS has propelled the advancement of robotics immensely for research and commercial applications. However, ROS starts to show limitations because of its research foundations when its commercial usage transitioned into products [Mac+22]. Therefore, a redesigned ROS2 is introduced by Macenski et al. [Mac+22] to tackle the challenges, such as security, reliability, and scalability.

The redesigned ROS2 differentiates itself from the original ROS (or ROS1) in three aspects. Foremost, ROS2 uses Data Distribution System (DDS) as its communication layer, unlike ROS1, which supports Transmission Control Protocol (TCP) communication. Instead, DDS uses the Real-time Publish-Subscribe (RTPS) protocol, which is built on User Datagram Protocol (UDP) [OMG]. ROS2 can benefit from this when the network is lousy, such as wireless communications. Furthermore, using DDS allows ROS2 to get rid of the master in ROS1, since DDS allows multi-cast and discovery. This allows the deployment of the ROS nodes on multiple platforms without further implementation. Second, unlike ROS1 where the Python library "rospy" and the C++ library "roscpp" are written separately in their own languages, ROS2 has a common library written in C language "rcl" on which both the Python library and the C++ library depend. Therefore, the performance of the two libraries is more consistent in ROS2.

ROS provides a standardized abstraction of communication for robotic applications over multiple machines. This is ideal for use cases like cloud robotics and edge robotics. ROS provides three communication patterns: topics, services, and actions. A typical ROS2 application can be illustrated in fig. 2.3. A ROS node is responsible for a single, module purpose, such as controlling the camera and doing image processing [ROS21] Each node can send and receive data to other nodes via the three communication patterns. The most common pattern is topics. The ROS nodes communicate by publishing and subscribing to messages on topics. Depending on what information needs to be communicated between the ROS nodes, the messages use the defined type. With the publish-subscribe pattern, the topics allow many-to-one, one-to-many, as well as many-to-many communication. The services provide a request-response communication. Clients submit requests to the service server, which processes the request and sends a response back to the client asynchronously. Actions provide goal-oriented and asynchronous communication interfaces with a request, a response, and periodic feedback. This communication pattern is normally suitable for long-term complex tasks, such as navigation. Other than the communication patterns, the ROS node can also be configured via the ROS parameters at startup or during the runtime without changing the code. A ROS parameter usually needs to be first declared and then used somewhere in

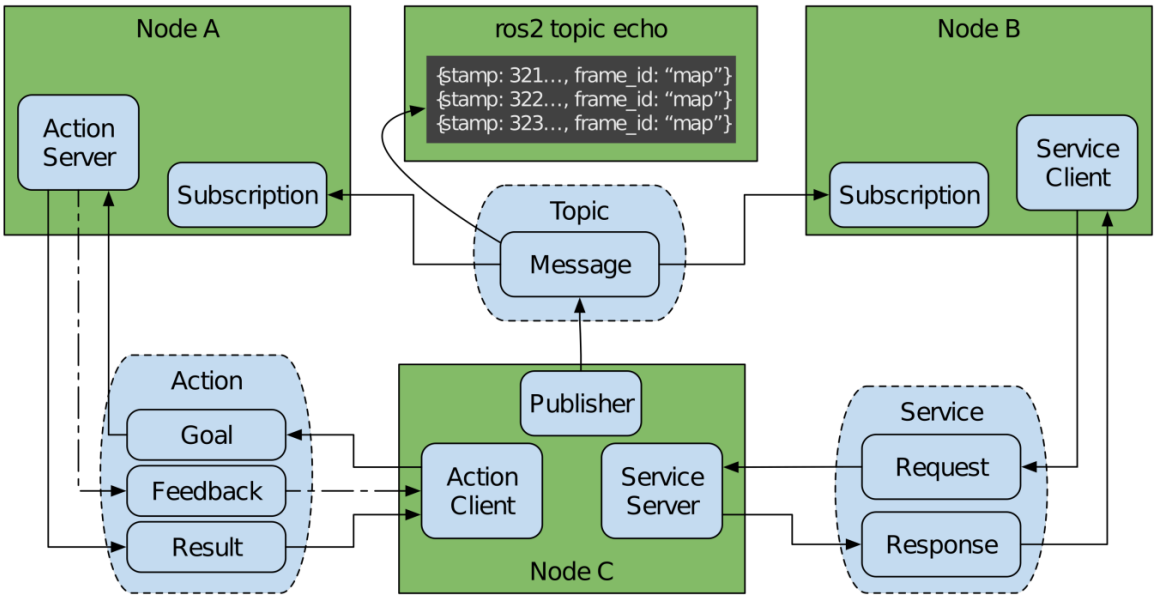

Figure 2.3: Communication patterns between ROS2 nodes [Mac+22]

the ROS node.

ROS2 also provides the functionality to record the topics to a data file called ROS bag and allows users to replay it. During replaying, the messages on different topics are published to the same topics in the same order as they are recorded. This allows the users to save the states of the experiment and recreate it later multiple times.

ROS2 also allows the users to define the Quality of Service (QoS) settings and the underlying DDS can be further modified. Publishing and subscribing to the topics have two reliability policies: best-effort and reliable. The reliable policy ensures the receipt of the message while the best-effort policy only sends the messages regardless of whether they are received. A message can be sent multiple times over a bad network connection under the reliable policy. On the other hand, the best-effort policy does not consider the receipt of the messages. It sends the message and then returns, which is more suitable for a bad connection. Due to the underlying implementation of the DDS software, only Fast-DDS can achieve the real best-effort policy when the environment variable "RMW_FASTRTPS_PUBLICATION_MODE" is set to "ASYNCHRONOUS", which means the publishers will not block the execution of the ROS node and return immediately after publishing the message. The queue size of the message buffer can also be defined by the users. This parameter defines how many messages can be restored if the publisher cannot publish the messages in time. An appropriate queue size can ensure few message drops and reasonable memory usage of the system.

Since ROS2 Galactic is the state-of-the-art version during the work of this thesis, the name "ROS" refers to ROS2 Galactic for the rest of the thesis unless specified otherwise.

2.3.2 Gazebo Simulation

Gazebo is an open-source 3D simulator for robotics. It integrates a physics engine and a rendering engine, and support code for sensor simulation and actuator control [Wik]. Therefore, the camera sensor can generate realistic RGB images from the simulation. The Gazebo simulation has undergone a series of name changes during its development, as shown in fig. 2.4. The original Gazebo, also now known as Gazebo Classic, is discontinued after the

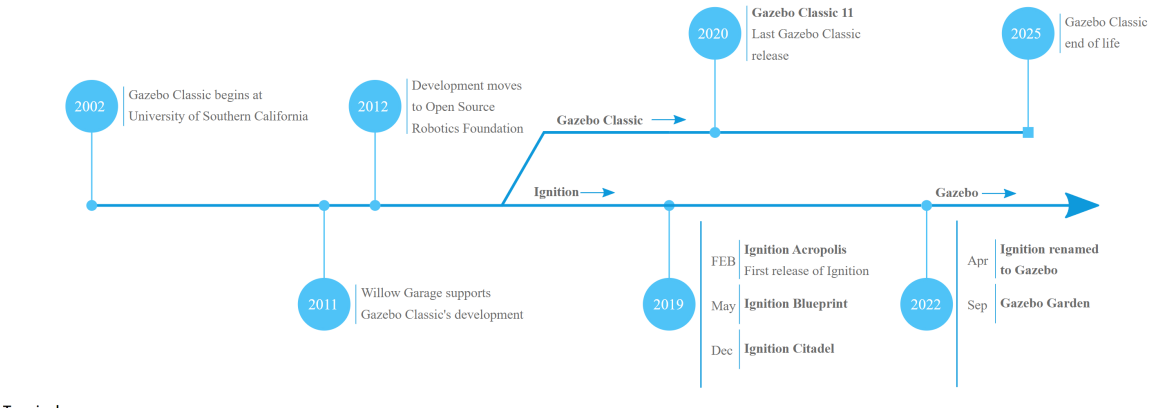

Figure 2.4: The development of Gazebo simulation [Ope]

last generation released in 2020. The Ignition name is initially used for another approach to differentiate itself from the original Gazebo, i.e., Gazebo Classic, is changed back to Gazebo again in 2022 for the current state-of-the-art Gazebo Garden. However, since this thesis uses Gazebo Fortress, due to a dependency of ROS2 Galactic, the name Gazebo refers to Gazebo Fortress for the rest of the thesis for simplicity.

In Gazebo, the simulation scenarios are implemented as world files and model files, which are written in Simulation Description Format (SDFormat) [Ope20]. The Gazebo server loads the world file on startup and it can load the model files either on startup or during runtime on demand. The Gazebo client runs a user interface for the server. Camera sensors and bounding box cameras sensors can generate RGB image stream and bounding boxes during the simulation. The messages are published in protobuf format and they can be bridged to ROS messages with ROS-Gazebo bridge [Ope].

2.3.3 Network policies

Network Emulator (NetEm) is an enhancement of the Linux traffic control facilities that allow one to add characteristics to packets outgoing from a selected network interface [The]. It can add delay packet loss, duplication, as well as rate onto an existing network interface. An outgoing policy can be set up for network interface "eth0" as follows:

```
tc qdisc add dev eth0 root netem delay 100ms loss 10% rate 160Mbit
```

In this example, the Network Emulator (NetEm) set the outgoing policy of the network interface "eth0" to 100 ms delay, 10% packet loss and 160 Mbps bandwidth. However, it is worth noticing that NetEm can only set up outgoing policies. For incoming policies, a Intermediate Functional Block (IFB) device needs to be set up and all traffic needs to be redirected to the new network interface. Then, the NetEm can be used to modify the outgoing policies of the IFB network interface, which will act as the incoming policies of the original network interface.

Chapter 3

Related Work

A plethora of research has investigated offloading strategies of edge computing. Offloading strategies decide when and how the AMRs should offload the computation tasks to the edge. They can be mainly divided into three categories: optimization approaches, game theory approaches, and learning-based approaches. This chapter gives a short introduction to these approaches.

3.1 Optimization Approaches

Optimization approaches formulate the computation offloading problem as an optimization problem with the goal to minimize energy consumption and/or execution latency. The multi-robot computation offloading can be usually formulated as an integer linear programming problem. The offloading strategy is derived from the problem solution.

Zhao et al. [Zha+15] formulate the problem as a joint optimization problem of the radio and computation resources aiming to reduce the overall energy consumption of mobile devices. The energy consumption model consists of two components: energy to transmit data and energy to compute the task locally. Zhao et al. also consider task partitioning in computation offloading. In the mathematical model, the computation task can be partitioned and partially offloaded without overhead for partitioning. In the end, Zhao et al. propose a heuristic algorithm to find the sub-optimal solution. Chen et al. [CLD15] formulate the problem as a non-convex quadratic program with the goal of minimizing an offloading cost which consists of weighted energy consumption and execution latency. Chen et al. solve the problem by applying a semi-definite relaxation and a randomization mapping method to the optimization problem. It is worth mentioning that the aforementioned works still focus on the mobile cloud computing paradigm, which only considers the cloud as the destination of computation offloading. However, the approaches are transferable to edge computing.

With the emergence of edge computation offloading, edge computers can also be used to perform computational tasks. Guo et al. [GL18] consider the case where the end devices can choose to offload either to centralized data centers or to edge computers. The problem is formulated as an optimization problem with the goal of minimizing energy consumption with a constraint for maximal execution latency. An exhaustive algorithm is proposed to find the optimal solution. Ning et al. [Nin+19] consider not only the three computation options but also the computation task partitioning and distribution among the local system, edge computer, and the cloud. Therefore, the problem is formulated as a mixed integer linear programming problem aiming to reduce the execution latency, which also inspired the problem formulation of the work of this thesis. An iterative heuristic algorithm is proposed to solve the problem.

The formulated optimization problem with a joint optimization goal is, in general, a mixed integer linear programming problem and cannot be solved in polynomial time. Therefore, on the one hand, heuristic algorithms are proposed to find the sub-optimal solution. On the other hand, relaxation is applied to the problem to find the approximate optimal solution. Furthermore, the optimization approaches assume a central decision-making unit that has full knowledge of the entire system. However, with dynamic network conditions and time-sensitive computation tasks, this may not be the case for the edge and the robot. Therefore, an offloading strategy with the ability to adapt to dynamic network changes and satisfy real-time requirements of the computation task is needed.

3.2 Game Theory Approaches

Unlike the optimization approaches, the game theory approaches formulate the problem as a game where end devices are players of the game. The offloading decision is made by finding an optimum or a Nash Equilibrium (NE) of the game. A NE is a group of strategies where no player can profit by modifying his strategy while the other players' strategies are kept unchanged [Cha+22]. Therefore, the game theory approaches are intrinsically distributed since the players make their own decisions.

Chen et al. [Che+16] formulate the distributed computation offloading decision-making problem among mobile device users as a multi-user computation offloading game. The approach aims to minimize a cost function consisting of energy consumption and execution latency. In the end, a distributed computation offloading algorithm is proposed to solve the problem. Furthermore, Pham et al. [Pha+18] also consider the scenario where multiple users offload to multiple servers. The problem is formulated as a joint optimization problem with the goal of optimizing the transmit power of the users and the computation resources at the servers. Two distributed matching algorithms are proposed to choose the server and the sub-channel. Hong et al. [Hon+19] take another step further by considering the computation offloading routing. The problem is formulated as a multi-hop cooperative computation offloading game. They further proposed a distributed algorithm to attain the NE of the game for partitioned and unpartitioned tasks. In addition, Xu et al. [XYL20] investigate how the weight factor in a cost function consisting of different optimization goals affects the overall system performance.

The game theory approaches allow distributed decision-making. However, the algorithms usually require several iterations for the cost function to converge and reach the NE. The communication overhead will increase with the number of end devices. Furthermore, the works lack experiments with real applications and only approach the problem as a purely mathematical problem.

3.3 Learning-based Approaches

In recent years, Deep Reinforcement Learning (Deep RL) has been used widely for decision-making problems. In edge computation offloading, the offloading strategy can also be modeled as a DNN. Different offloading options are treated as the action space. Network conditions and robot onboard resources are considered as the state space. Since the learning-based approaches do not need a model of the system, they are more capable of dealing with dynamic changes in the network.

Huang et al. [Hua+19] train a Deep Q-learning Network (DQN) agent to make offloading

decisions for multi-users computation offloading. The reward function is formulated to reduce energy consumption and execution latency. Lu et al. [Lu+20] take another step further with the DQN agent by considering task partitioning for multi-user and multi-server computation offloading problems. The reward function is formulated to improve energy consumption, load balancing, and execution latency.

Several works from these approaches also demonstrate real application performance. Chinchali et al. [Chi+19] train a Advantage Actor Critic (A2C) agent to perform computation offloading with face recognition algorithm. The agent shows improved performance in accuracy and execution time compared with naive offloading strategies, such as only offloading to the edge, only computing locally, and random offloading. Moreover, Penmetcha et al. [PM21] offload navigation tasks to the cloud. The trained DQN agent is capable of making offloading decisions by considering the data size of the task. Finally, Ruggeri [Rug21] trains a DQN agent to make offloading decisions for scene understanding, which is a safety risk evaluation task. The safety risk is reduced compared to naive offloading strategies.

Learning-based approaches show promising results in dealing with dynamic network changes and can make offloading decisions in a distributed manner. However, the computation of the DNN also causes additional execution latency and increases the computation workload of the robot's onboard system. For time-sensitive tasks, such as object detection, the additional execution latency can cause the performance to deteriorate.

In the end, it is also worth noticing that a plethora of literature has already investigated the mathematical problem of computation offloading and various approaches have been proposed to tackle the problem. However, few investigate the influence of different strategies on the robotic system metrics in an experimental way. Therefore, this thesis intends to investigate this problem by implementing an offloading framework and conducting experiments on the influence of different offloading strategies.

Chapter 4

Methodology

This chapter describes the methodology to solve the problem. In section 4.1, the system design of the perception offloading is presented and explained. Then, the offloading strategies are described in section 4.2.

4.1 System Design

A perception offloading framework and different offloading strategies are designed and implemented in order to carry out the experiments. This section first describes the design for an offloading framework based on perception tasks. Then, different offloading strategies are presented. Finally, this section describes what system states are chosen to represent the state of the perception offloading.

4.1.1 Perception offloading

The designed offloading framework is illustrated in fig. 4.1. As shown, only the scenario where one AMR offloads the perception task to one edge computer is investigated. Therefore, the offloading modules are located solely on the robot's onboard system. To ensure the safety of the robot in case of a complete network connection loss, a perception module is also included on the robot's onboard system. However, since the robot onboard system usually has limited resources, the onboard perception module uses a less computationally expensive object detection model. In contrast, the edge computer has abundant resources. Therefore, it uses a perception module with a more complex model with higher precision.

To evaluate different offloading strategies, the offloading module adopts a plug-in mechanism to load the offloading strategy at start-up. The offloading module receives different system states from the state monitors located on the robot's onboard system and on the edge computer. When the offloading module receives an image from the camera, it has to make a decision whether to offload the image to the edge computer or to compute it locally on the robot. Therefore, the offloading module passes the system states to the strategy plug-in as runtime parameters and receives the offloading decision.

In addition to the perception modules and the offloading modules, there are also different state monitors that keep track of the onboard resources of the robot and the network condition between the robot and the edge computer. The network monitor measures the network latency between the robot and the edge computer. The resource monitor monitors CPU usage and energy consumption of the robot's onboard system. Furthermore, the perception modules are also responsible for measuring the inference times of the object detection models and sending them back to the offloading module

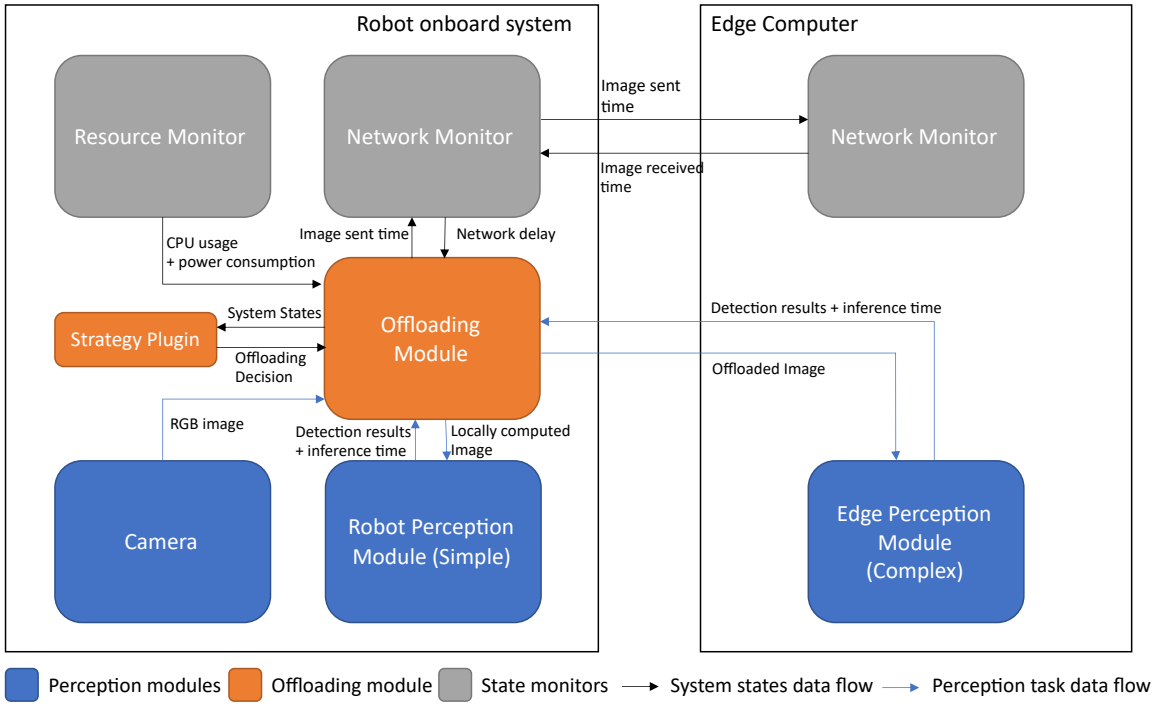

Figure 4.1: Design of perception offloading framework where the robot and the edge systems are separated by rectangles. Each system consists of different modules that are responsible for different tasks. The orange blocks represent modules related to offloading, while the blue blocks represent modules related to perception tasks. The grey blocks represent modules measuring system states. Arrows show the direction of data flows.

4.1.2 System States

Various system states are considered for the perception offloading. They can be mainly divided into two categories: the onboard resources and the network condition. The network condition is measured by the network delay between the robot's onboard system and the edge computer. The outgoing data of the robot include the offloaded images with their sent times, while the incoming data contain an array of object detection results and the times when the edge computer receives the offloaded images and sends the detection back. The network monitors measure both the outgoing and incoming network delays. The total network delay can be calculated by

$$t_{network} = t^{out}_{network} + t^{in}_{network} \tag{4.1}$$

where $t^{out}_{network}$ is the network outgoing delay and the $t^{out}_{network}$ is the network outgoing delay. They can be calculated as

$$t^{out}_{network} = t^{robot}_{sent} - t^{edge}_{received}$$
$$t^{in}_{network} = t^{edge}_{sent} - t^{robot}_{received}$$

For onboard resource usage, the Central Processing Unit (CPU) usage, the power consumption, and the bandwidth usage are measured. The CPU usage and the power consumption can be directly measured with the robot's onboard system. However, the bandwidth usage needs to be calculated with the network IO measurements as follows:

$$bandwidth = \frac{n_{t_2} - n_{t_1}}{t_2 - t_1} \times 10^{-6} \times 8 \tag{4.2}$$

where n_{t_i} represents the number of bytes the network has sent till time t_i. The bandwidth is calculated in megabits per second (Mbps).

The onboard resource usage represents the availability of the AMR. If the CPU usage is too high, the robot will have no resources left for other tasks performed onboard, such as SLAM and navigation, and cannot operate anymore. Similarly, if the power consumption is too high, the robot needs to be recharged more often. Moreover, if the available bandwidth of the robot network is too small, the robot cannot communicate via the network. Therefore, these measurements are used as system states for making offloading decisions.

4.2 Offloading Decision

For the offloading module to make offloading decisions, an offloading strategy is needed. Several naive strategies are first investigated as baseline strategies. Their influence on various metrics of the system is evaluated. Based on their performance, a dynamic offloading strategy that uses the runtime system states, which are described in section 4.1, is implemented and evaluated against the baseline strategies.

4.2.1 Baseline strategies

As baselines, the scenarios where the robot only computes the object detection task locally on its onboard system or only offloads the task to the edge computer are first investigated. These two strategies are named the "robot only" strategy and the "edge only" strategy. Then, the thesis continues to investigate the scenarios where the robot offloads a portion of the frames with a fixed ratio. For example, an offloading ratio of 0.2 will allows the offloading module to offload one image to the edge for every five images. The offloading strategies with a fixed offloading ratio can be realized with algorithm 1. The offloading ratio should be a value ranging from 0 to 1.

Algorithm 1 Algorithm to offload with a fixed ratio

```
1:  function RatioStrategy(r)        // where r is the offloading ratio
2:      c1 ← GetImageCounter()       // Get the counter for the total images received
3:      c2 ← GetOffloadCounter()     // Get the counter for the images offloaded
4:      SetImageCounter(c1 + 1)      // Add one first to image counter to avoid zero division
5:      if c2/c1 ≥ r then
6:          return false             // Compute locally
7:      else
8:          SetOffloadCounter(c2 + 1)
9:          return true              // Offload to edge computer
10:     end if
11: end function
```

With results from the experiments on baseline offloading strategies, this thesis intends to find the limits of the system and analyze its behavior. More specifically, the experiments and evaluation with the baseline strategies intend to investigate the question that under what circumstances the performance of the perception performance starts to deteriorate and offloading the perception tasks is no longer beneficial for the robot. With the results, a dynamic offloading strategy that uses runtime system states to make offloading decisions is then developed and implemented.

4.2.2 Dynamic offloading

The dynamic offloading strategy aims to adapt to the changes in robot and edge systems as well as in network conditions. For the one-robot-one-edge scenario, a dynamic offloading strategy with the goal of minimizing the execution latency and improving the average precision of the object detection task is used. Moreover, this dynamic offloading strategy is subjected to the constraints of other system states. Two runtime system states are considered as constraints: AMR's CPU usage and the network bandwidth. The problem can be formulated as a constraint optimization problem as follow:

$$\min_{(\alpha,\beta)} t_{latency} = t_{inference} + t_{network} \tag{4.3}$$

with

$$t_{inference} = \alpha t^{L}_{inference} + \beta t^{E}_{inference}$$
$$t_{network} = \alpha t^{L}_{network} + \beta t^{E}_{network}$$

s.t.

$$\alpha + \beta = 1$$

$$\alpha = \begin{cases} 1, & \text{if the task is processed locally} \\ 0, & \text{else.} \end{cases}$$
$$\beta = \begin{cases} 1, & \text{if the task is processed by the edge computer} \\ 0, & \text{else.} \end{cases}$$

where the $t_{latency}$ represents the overall execution latency of the object detection task and $t_{inference}$ and $t_{network}$ represent the inference time and the network delay correspondingly. The execution latency is also called the Round-Trip Time. It measures the time for a round trip from the offloading module to the perception modules either located on the robot's onboard system or on the edge computer.

In addition to the execution latency, the offloading decision is also subjected to the AMR's CPU usage and the network throughput. If the CPU usage exceeds a certain value, the AMR only offloads to the edge. Similarly, the AMR only computes the task locally if the network throughput exceeds the threshold. These constraints ensure that the AMR and the network are capable of finishing the task at all. The decision-making strategy can be implemented as algorithm 2.

Algorithm 2 Algorithm to offload with dynamic parameters

```
1: function DecisionMakingStrategy(params)// runtime parameters: params
2:     cpu_max ← GetMaxCPULevel()
3:     if params['cpu_usage'] ≥ cpu_max then
4:         return true                    // Offload if CPU usage exceeds limit
5:     end if
6:     throughput_max ← GetMaxThroughput()
7:     if params['bandwidth'] ≥ throughput_max then
8:         return false                   // Compute locally if bandwidth usage exceeds limit
9:     end if
10:    latency_robot ← params['robot_inference_time'] + params['robot_network_delay']
11:    latency_edge ← params['edge_inference_time'] + params['edge_network_delay']
12:    if latency_robot ≥ latency_edge then
13:        return true                    // Offload if edge RTT is higher
14:    else
15:        return false                   // Compute locally if edge RTT is higher
16:    end if
17: end function
```

Chapter 5

Implementation

This chapter focuses on the implementation of the offloading frameworks designed in chapter 4. In section 5.1, it describes the implementation of the offloading framework in ROS. In section 5.2, it describes how the system states are measured for the robot and edge.

5.1 Offloading Framework

The offloading framework is implemented with ROS2 [Mac+22]. As mentioned in section 2.3, a ROS node is a process with a single purpose that communicates with other nodes through ROS communication patterns, such as topics and services. Therefore, different modules are implemented as ROS nodes and the communication between the nodes uses ROS topics exclusively, as illustrated in fig. 5.1. The RGB images from the simulation are received by the offloading node via ROS topic. Thereon, the offloading node decides whether to offload to the edge or to compute it locally by publishing it to the corresponding ROS topics. The perception nodes receive the image from the ROS topics and publish the processed results to a dedicated ROS topic which is subscribed by a ROS bag recorder and will be eventually used for evaluation. Moreover, the network delays and the inference times are measured by the offloading node and the perception nodes by critical time points during the offloading and the inference.

5.1.1 Offloading Module

The offloading node is the center of the offloading frameworks because it is responsible for making offloading decisions and carrying out the perception offloading. In order to keep track of the system states monitored by different nodes, the offloading node subscribes to different system state topics and stores the current states. To prevent the fluctuation in the systems states from affecting the offloading decision, the offloading node applies a Simple Moving Average (SMA) algorithm to smooth the data. The stored system states are passed to the offloading strategy during run time by the offloading node. In order to evaluate different offloading strategies, the offloading node uses a plugin mechanism to load the offloading strategy and hyper-parameters when the system first starts.

5.1.2 Perception Module

In general, an offloading task can be any computationally intensive algorithms an AMR is required to run, such as perception, navigation, SLAM, path planning, etc. An object detec-

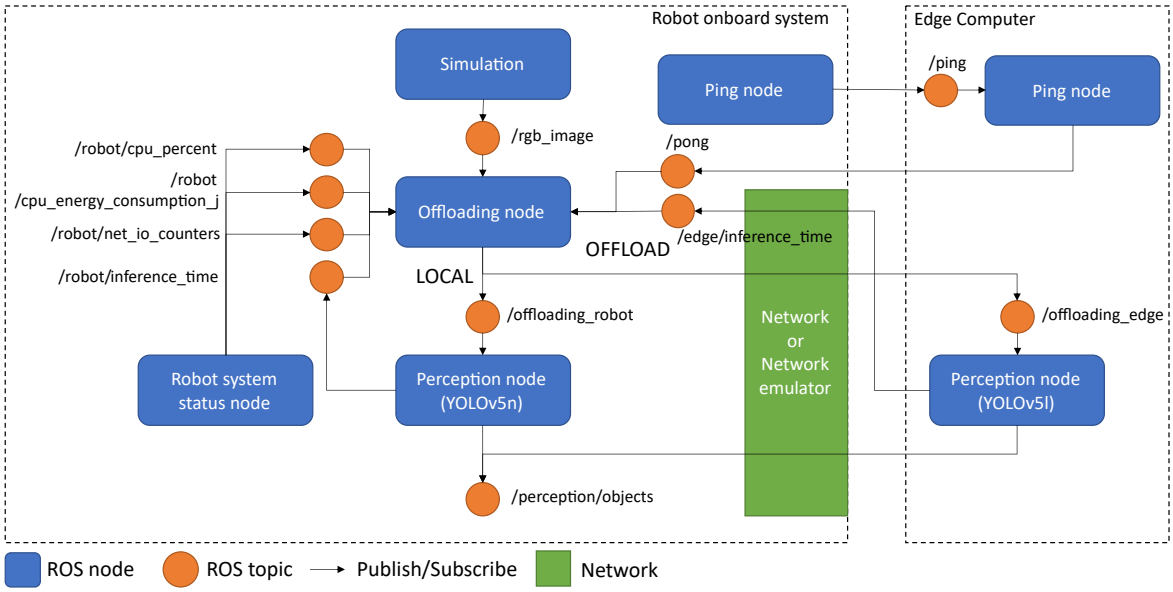

Figure 5.1: This figure shows the offloading framework implementation with ROS. Each blue block represents a ROS node. The orange circles represent ROS topics. The arrows show the direction of messages. The robot onboard system and the edge computer are separated by dotted rectangles. They can either be an actual physical system or a system virtualization such as a Docker container. The two systems can be connected by an actual network or a virtual network interface with NetEm to emulate real network conditions.

tion task based on RGB images is chosen as an example because it can have a significant performance difference between the robot's onboard system and the edge computer, which corresponds to the use cases of edge offloading. The robot is usually equipped with a simple onboard system with only access to CPU, while edge computers are usually resource-abundant computers and data centers on-premise with access to Graphics Processing Unit (GPU), as described in section 2.1. As mentioned in section 2.2, the robots use DNNs to detect objects primarily. With frameworks like PyTorch and OpenVINO that can make use of GPUs, the performance difference between AMR's onboard system and the edge computer is immense. Therefore, object detection is chosen as an example for offloading tasks.

Model	YOLOv5n	YOLOv5s	YOLOv5m	YOLOv5l	YOLOv5x
Robot inference time	50.88 ms	115.75 ms	240.06 ms	464.22 ms	821.93 ms
Edge inference time	8.87 ms	11.51 ms	20.35 ms	31.18 ms	52.55 ms
Model size [Joc+22]	4 MB	14 MB	41 MB	89 MB	166 MB
mAP [Joc+22]	45.7%	56.8%	64.1%	67.3%	68.9%

Table 5.1: Inference times of different YOLOv5 models on AMR's onboard system and edge computer

In order to adapt to the performance difference between the AMR's onboard system and the edge computer, the perception module should have two models available for object detection: a lightweight model that runs on the onboard system and a more complex model that runs on the edge computer. YOLOv5 provides a series of models with different complexities. To find appropriate models for the onboard system and the edge computer, the inference times of different models on different machines are investigated, which can be taken from table 5.1. To simulate the discrepancy of the computation capabilities between the two systems, this experiment uses a Intel Next Unit of Computing (NUC) equipped only with an Intel(R) Core(TM) i3-8109U CPU. On the other hand, the edge computer is equipped with

an Intel(R) Core(TM) i9-7900X CPU and an Nvidia GeForce GTX 1060 6GB GPU. The models are deployed with PyTorch and output an array of bounding box detection. The inference time is measured between the time when the YOLOv5 models receive the image and the time when the perception nodes output an array of bounding box detection. This includes the time for image pre-processing and the time of results post-processing. Furthermore, the image input for the offloading module has a frame rate of around 25 frames per second. The model sizes and the Mean Average Precision (mAP) data are taken from the documentation from Jocher et al. [Joc+22]. The mAP data are evaluated on Common Object in Context (COCO) val2017 [Lin+14]. As a compromise between the inference time and the performance, the YOLOv5n is chosen to be deployed on the AMR's onboard system and YOLOv5l on the edge computer.

The YOLOv5 models are deployed within the ROS nodes. Object detection is carried out in the perception nodes synchronously. This means the perception nodes are blocked during the execution of the inference. If the perception nodes cannot process the inference fast enough, the image messages will start to queue and the network delay will increase. The synchronous inference can achieve low inference time and reduce the CPU usage of the perception nodes, since the ROS nodes do not have to spin continuously to check for processed results on another thread. The models are used in evaluation mode in PyTorch to reduce the overhead of the computation tasks used for training the models. The confidence threshold is set to 0.25 and the Non-maximum Suppression (NMS) threshold is set to 0.45 as the default settings of YOLOv5. Furthermore, the inference size is 640 by 640 pixels and the maximum number of detection per image is set to 1000. The inference is calculated in full precision, i.e., FP32. The detection classes are taken from the COCO data sets, provided by YOLOv5.

5.2 State Monitors

In order to assess the system's status during perception offloading, it is necessary to measure various system states, including network delay, network bandwidth, CPU usage, and energy consumption. This section describes how the system states are measured and what tools are used to measure them.

5.2.1 Network

The network delay and bandwidth are used to represent the network conditions during the perception offloading. The network delay contributes a great amount to the execution latency of the edge perception, which the dynamic offloading strategy described in section 4.2 intends to reduce. Moreover, for the perception offloading task, the offloaded images also require huge network bandwidth. For example, it requires 244 Mbit/s available bandwidth to offload a series of image messages with a resolution of 848 by 480 pixels at 25 frames per second (FPS). Therefore, these two system states are crucial to measuring the network condition of the AMR.

The network delay can be measured with ping-pong nodes between the AMR's onboard system and the edge computer. Since the images are offloaded as ROS messages, the in and out delays can be measured by calculating the difference between the time stamps. The ping pong nodes update once every second to reduce the unnecessary CPU usage of the measuring nodes. To prevent the fluctuation of the network delays affecting the offloading decision, a SMA is applied to the network delay.

The available bandwidth can be measured by sending a series of bytes from the AMR's

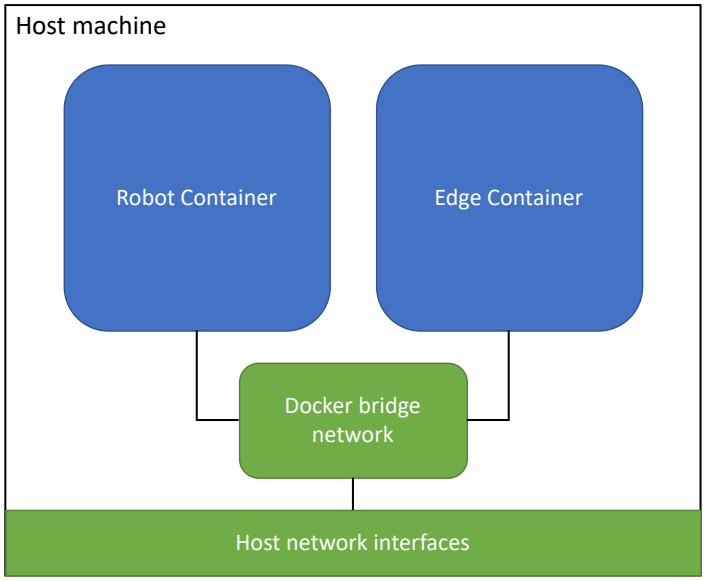

Figure 5.2: This figure shows the deployment of the Docker containers and the docker bridge network within the host machine. The robot and edge systems are virtualized as Docker containers. The containers are connected with each other through the Docker bridge network. The bridge network creates virtual network interfaces for each container.

onboard system to the edge computer and measuring the throughput, as described in section 4.1. However, the measurements per se will use all of the available bandwidth and affect the performance of the perception offloading. Therefore, the measurements should not take place during the perception offloading. The bandwidth is measured with the network bandwidth measurement tool "iPerf". Since ROS2 relies on DDS which uses UDP as its transport [Mac+22], iPerf also is configured to use UDP as communication protocol as well. On the other hand, the bandwidth in use can be calculated by dividing the accumulated network Input/Outputs (IOs) by the time, as described in eq. (4.2). Since the experiments are carried out both in simulation and on an actual robotic system, the measurement methods vary among different virtualization. For Docker containers, the network IOs can be measured with the tool "docker stats" provided by Docker. It measures the data transferred by the network interfaces of the containers. On the robotic system, the network IOs can be measured for specific network interfaces, given that only the perception offloading is running on the robot.

5.2.2 Onboard resources

To make more informed offloading decisions, the measurements of these onboard resources are necessary. Similar to bandwidth usage, the measurements of the onboard resources are also dependent on virtualization. Therefore, the measurements of the onboard resources are discussed in two scenarios: Docker containers and actual robotic systems.

The deployment of the docker containers and the docker bridge network within the host machine can be illustrated in fig. 5.2. For Docker containers, the CPU usage can be measured by tools provided by Docker as well. Since Linux Docker containers rely on control groups, Docker keeps track of the groups of processes of different containers and exposes the metrics, such as CPU usage and network IOs. However, container-level CPU energy consumption relies on power estimation models and cannot be directly measured. Moreover, for the specific use case of this thesis, the edge perception is solely calculated using the GPU. Therefore,

the system-level CPU energy consumption can still reflect the robot's power consumption. Alternatively, the system states of containers can be monitored using cAdvisor. However, the overhead of such a tool is not negligible. Therefore, it is not adopted as one of the measurement methods.

For an actual robotic system, the measurements of system states become simple, since the system-level metrics can be directly used. The system states are measured with the "system status" package as a part of the code base of AMSRL. The "system status" package utilizes the Python package "psutil". It can measure system-level CPU usage and CPU power consumption as well as IOs for network interfaces. In the case of Intel CPUs, which is used in the work of this thesis exclusively, the power consumption can be measured with Running Average Power Limit (RAPL) provided by Linux kernel Power Capping Framework. The system status node reads the system states and makes them available to the offloading node by publishing them to different ROS topics.

Since it is assumed that the edge computer has abundant resources, the system states of the edge computer are not measured and are not taken into consideration during the decision-making and evaluation process. Furthermore, even though only the single-robot single-edge scenario is investigated, the available resources from the edge computers can be affected by numerous factors in real-world applications, such as the number of edge computers, the number of AMRs, and other processes running the edge computers. In addition, with more powerful hardware and easy access to a power supply, the edge computer consumes more energy and computation resources for the same task than the AMR's onboard system. Therefore, it is pointless to compare the consumption of the resources between the two systems. Therefore, only the system states of the AMR's onboard system and the network condition between the AMR and the edge computer are considered.

Chapter 6

Experiment

This chapter describes the experiments conducted in simulation and on a robotic system. First, section section 6.1 describes the simulated scenario in use to carry out the experiments in simulation and on a robotic system. Then, the specifications of the experimental setups are described in section 6.2. After that, the evaluation framework used to analyze the experimental results is presented in section 6.3. Finally, the analysis of the experimental results is presented in section 6.4.

6.1 Simulated scenario

To run experiments both in simulation and on real robots, a simulated scenario of a factory warehouse with one AMR and obstacles is implemented. The simulation is recorded and replayed for each experiment run to reduce the computation overhead of the simulation and to make the experiment repeatable. Furthermore, in order to compare the results from the simulation and the real robot, the simulation recording is used as input for both experiments under the same replay settings.

6.1.1 Environment and AMR simulation

The scenario simulates a modern-day factory warehouse, illustrated in fig. 6.1. The simulated scenario is implemented in Gazebo with the help of a software package called "scenario execution", which the author implemented during his work at AMSRL. The package allows the users to define a series of events during the experiments in parallel and/or serial. For example, the experiment can be considered as successful if only a number of criteria have been met. The package can be used to check for the criteria and shut down the experiment after all criteria are satisfied. The scenario consists of a factory warehouse environment and static obstacles, such as humans, boxes, shelves, and pallets, which are common obstacles in a factory warehouse. Since the performance difference among different YOLOv5 models are negligible when the scene is too simple, ten human obstacles in the simulated scenario are included to increase the complexity of the object detection task. Moreover, some human obstacles are partially occluded by other obstacles in the scenario. Detecting occluded objects has been proven to be a challenging task in object detection [SSV21].

The AMR is implemented as a dynamic object in the scenario and it moves along a pre-defined path. The pre-defined path is made up of a series of waypoints in the simulation. Once the AMR reaches the current waypoint, the scenario execution will assign the next waypoint to the AMR. After all waypoints are successfully reached, the scenario execution

Figure 6.1: The figure shows a bird view of the simulated scenario. The scenario consists of an industrial warehouse and different obstacles. A AMR equipped with camera sensors is navigating in the scene.

shuts down the simulation. In the simulation, the AMR does not collide with other obstacles. The AMR is equipped with an RGB camera that is located on top of the robot and facing forward. The intrinsic parameters of the RGB camera sensor are listed in table 6.1. During the simulation, the camera can observe all of the obstacles in the simulation with partial occlusion of some obstacles. In addition to the RGB images, the simulated camera sensor also outputs an array of ground truth for the bounding boxes in the object detection task using the bounding box sensor from Gazebo. The images are bridged from the Gazebo simulation topics to ROS topics so that the offloading framework can make use of them. The ground truth is also bridged by an implemented custom bridge for experiment evaluation. In fig. 6.2, the point of view of the AMR's camera sensors is shown. On the left, it shows the illustration of the bounding box ground truth data, while the RGB images are shown on the right. The bounding box camera sensor and the RGB camera sensor are positioned the same and intend to simulate an actual point of view of a normal AMR.

Parameter	Value
Height	848 pixels
Width	480 pixels
FOV	1.047
Frame rate	30 frames/sec
Noise type	Gaussian
Noise mean	0
Noise standard deviation	0.007

Table 6.1: RGB camera intrinsic parameters

(a) RGB image **(b)** Bounding box ground truth

Figure 6.2: The figures show the robot view of the simulated scenario. The left figure shows the RGB image from the camera sensor located on top of the AMR. The right figure shows an illustration of the bounding box ground truth of the RGB image.

6.1.2 Record and Replay

The simulation takes up a great amount of the resources of computers due to the physics and the rendering engines. To reduce the overhead of running the simulation alongside the experiments, the simulation is recorded to a ROS bag and replayed during the experiment runs. As mentioned in section 2.3, ROS provides the functionality to record the topics and allows the users to replay them later while preserving the same publishing rate and order of the messages. More importantly, the experiment is conducted on a real robotic system with limited resources. Such a system is not capable of running real-time simulation while maintaining the rest of the offloading framework. In real-world applications, AMRs only needs to maintain the driver of the camera and minimal software to be able to get the same image input. Therefore, in table 6.2, an experiment comparing the CPU usage of replaying a ROS bag and running the library for the Intel RealSense camera is presented. The experiment is conducted on a NUC machine equipped with an Intel(R) Core(TM) i3-8109U CPU. The results show that the two processes have comparable CPU usage on a robotic system. Furthermore, Gazebo slows down the simulation time compared to real-time when the system is strained. Therefore, to ensure a real-time simulation, specifying a replay rate of the ROS bag can guarantee the simulation time factor is within a reasonable range.

Process	CPU usage
ROS bag replay	14.12%
RealSense camera library	11.28%

Table 6.2: The table shows the CPU usage of the NUC machine for replaying the simulated scenario ROS bag and running the RealSense camera library. The results are the average of an experiment run of 58 seconds.

All topics during the simulation are recorded in the ROS bag, including the RGB images, the bounding box ground truth, the camera information, etc. The ROS bag consists of 1748 frames and the ROS bag is replayed at around 25 frames per second during the experiment. Topics for RGB images and the bounding box ground truth are replayed with best-effort reliability QoS setting to ensure the RGB images are replayed the same way it is recorded. Since the ROS topics are recorded at a different time than the experiment, the time stamps of the RGB image messages and the ground truth messages have to be overwritten by the receiving time of the offloading node.

6.2 Experimental Setup

In this section, the experimental setups for experiments in simulation and on a robotic system are specified. The baseline offloading strategies are first evaluated in simulation experiments to test their functionality and robustness. Then, the baseline strategies as well as the dynamic strategy are evaluated on an actual robotic system.

6.2.1 Simulation experiment setup

For experiments in simulation, the host machine is equipped with an Intel(R) Core(TM) i9-7900X CPU and an Nvidia GeForce GTX 1060 6GB GPU. The CPU has a total of 10 cores with 20 threads using Intel Hyper-threading technology. In order to simulate the AMR's and the edge computer in realistic conditions, the computation resources of the Docker containers are limited. The robot container is constrained to using only eight cores of the host machine, while the edge container is given ten cores. In addition, the edge computer has access to the GPU of the host machine and calculates the image inference on the GPU. Moreover, the robot container is also constrained to using only 4 GB of memory, while the memory of the edge container is not constrained and the host machine has access to 32 GB of memory, including the 4 GB memory assigned to the robot container.

The limitations of CPU and memory usage are realized by tools provided by Docker itself. According to the Docker documentation, the runtime memory limitation is realized by Memory Resource Controller provided by the Linux kernel. Furthermore, the Docker container can enforce hard memory limitation and soft memory limitation at runtime. The former does not allow the container to use any amount more than specified, while the latter only kicks in when certain conditions are met. the hard memory limitation is used on the robot container to simulate the hardware constraints. For CPU usage, the limitation is realized by the CFS scheduler, which is the Linux kernel CPU scheduler for normal Linux processes. This indicates that the limitation on CPU usage is achieved by limiting the accessible CPU cycles of the Docker container. The GPU of the host machine is exposed to the Docker containers using the Nvidia Container Toolkit.

In addition to resource limitation simulation, the network is also simulated to achieve realistic conditions for the robot and edge containers. The containers use the default Docker bridge network. Since the simulated scenario ROS bag is replayed in the robot container and the output data are also recorded in the robot container, the data flow between the robot container and the edge container only consists of offloaded images and detection processed by the edge container. To allow the data flow of around 25 frames per second of images with a resolution of 848 x 480 pixels, which is around 244 Mbps, the packet limit of in and out policies of the NetEm is set to 20000. The in and out delays are set to 50 ms for both policies. To investigate the influence of the network bandwidth, the experiments are carried out under two bandwidth conditions: with a 160 Mbps bandwidth constraint and no bandwidth constraints at all. The former aims to represent a limited network bandwidth that is unable to handle the data flow that is needed for a full offloaded execution, while the latter represents an unlimited network bandwidth.

For QoS settings, the offloading module uses a best-effort reliability policy to prevent the publisher from being blocked by the bag network connection caused by the bandwidth constraints. As mentioned in section 2.3, only Fast-DDS allows a true asynchronous publishing mode. Therefore, Fast-DDS is used as ROS middleware, and the publishing mode is set to asynchronous for the experiments, as described in section 2.3. The offloading module uses the "keep last" queuing settings and the queue size is set to five. Furthermore, the ROS bag

replay of the simulated scenario and the state monitors also use the best-effort reliability policy to simulate a camera sensor.

6.2.2 Robot experiment setup

Unlike the aforementioned simulation experiments, the robot experiments use two separate computers for the AMR's onboard system and the edge computer. For the robot, the onboard system uses a NUC equipped with an Intel(R) Core(TM) i3-8109U CPU, which is commonly used for AMRs. For the edge, a Linux desktop computer equipped with an Intel(R) Core(TM) i9-7900X CPU and an Nvidia GeForce GTX 1060 6GB GPU is used. On both computers, the offloading module and the perception modules have full access to the available resources. The ROS bag replay of the simulated scenario is carried out on the robot. The output data of the metrics are also recorded on the robot. The edge computer is solely responsible for the inference of the offloaded images and monitoring the edge system states and sending them back to the robot. The image inference is computed by the GPU on the edge computer with the help of PyTorch library, while the image inference on the robot is computed by its onboard CPU. As mentioned in section 5.2, the system states of the robot and the edge are measured with the "system status" package.

Two network interfaces are used to conduct the robot experiments. First, the experiment uses an Ethernet connection between the two computers. Similar to the simulation experiments, two network conditions are tested under the Ethernet connection. First, an Ethernet network of 1 Gbps bandwidth is used to simulate a perfect network connection between the robot and the edge. Then, the Ethernet network interface is constrained to 160 Mbps bandwidth with NetEm to simulate a constrained network connection. Finally, a Wi-Fi network interface is used to carry out the experiment. The Wi-Fi network has unstable network bandwidth from time to time as it is also used by other clients and therefore affects the experiment results drastically. This is to investigate different offloading strategies under dynamic network condition changes.

The robot experiment uses the same DDS and QoS settings as the simulation experiment. Under various QoS settings, this setup achieves the best performance of the object detection task. However, other QoS settings are also investigated during the experiments. In appendix A, the results of the experiment with a "reliable" reliability policy are presented.

6.3 Evaluation Framework

In order to understand the influence of different offloading strategies on the robot, an evaluation framework is introduced. More specifically, this section describes the metrics used to evaluate the results of the experiments and discusses the evaluation methods used to process the data.

6.3.1 Metrics

The evaluation metrics fall into two categories. The first category focuses mainly on the resources of the AMR. This includes CPU usage, power consumption, and bandwidth usage. These metrics represent the onboard resources for computation, energy, and network used by the AMR. For these metrics, the results for when the offloading module is idling are also subtracted from the baseline strategies to eliminate the influence of processes other than the

perception modules on the metrics. More specifically, while idling, the offloading framework still launches the offloading node and the perception nodes. It also replays and records the ROS bags. However, the offloading module is not sending any messages to the onboard perception module or the edge perception module, i.e., there are no object detection tasks being performed at all.

The second category describes the performance of the object detection task, including Average Precision (AP), Round-Trip Time (RTT), and the overall processed frame rate. For the average precision, only the human obstacles are considered in the evaluation. The detection from the perception nodes is compared with the bounding box ground truth from the simulation. The calculation of the AP from ROS messages is provided by the package "torchauto" as a part of the code base from AMSRL. For the execution latency, the two components, i.e., the network delay and the inference time, are evaluated separately for offloading and local computation. In addition, as the best-effort QoS reliability policy drops messages when the network is bad, to evaluate how the offloading framework is performing over the entire simulation, the overall processed frame rate evaluates how many frames of all frames are processed by either perception module.

6.3.2 Evaluation methods

For the comparison between the detection and the ground truth, the time stamps of the messages are crucial to the final result. This raises the question: what should be considered as simultaneity for the detection and the ground truth in task offloading? Two evaluation methods of the AP metric are presented in order to gain some insights into this question.

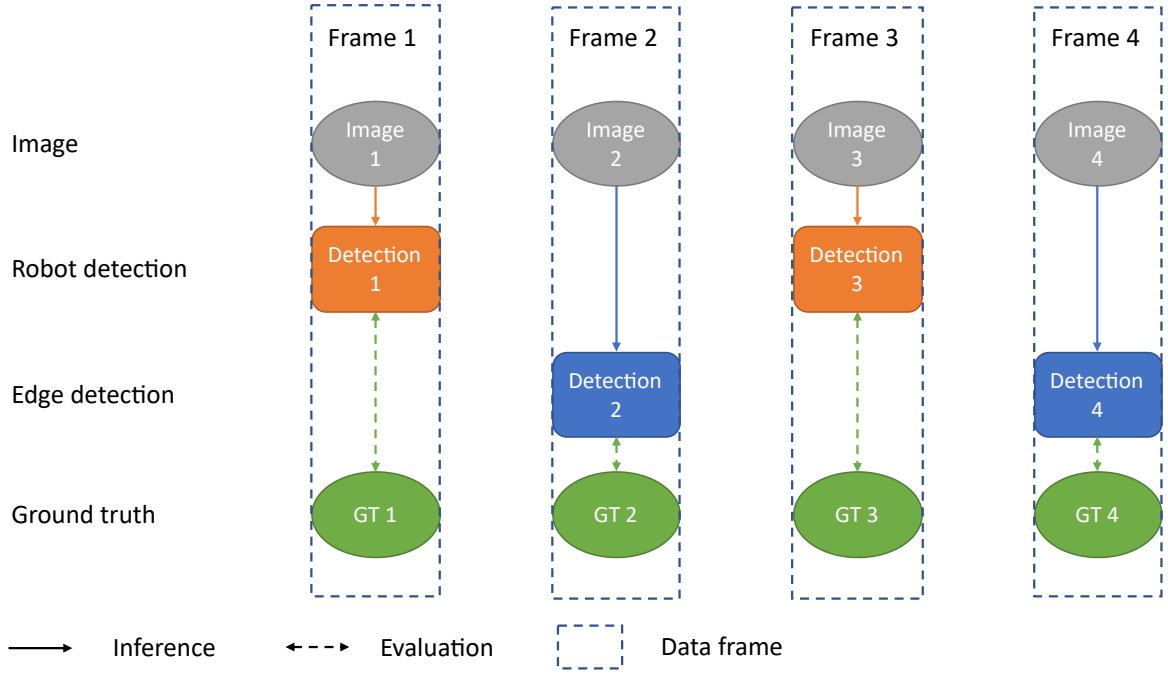

Figure 6.3: This figure shows the synchronous evaluation method. In the evaluation method, the detection is always compared with the ground truth of the original RGB image. The execution latency is not considered by this evaluation method.

In the synchronous evaluation, the detection is evaluated by being compared with the ground truth of the same time stamp. As illustrated in fig. 6.3, the image is either processed by the robot perception module or the edge perception module. The processed result and

the ground truth are recorded by the ROS bag. After the experiment runs, the recorded result and the ground truth with the nearest time stamps are grouped as one data frame by the message filter, provided by the ROS package "message_filters". The message filter groups the messages from different ROS topics. The approximate messages within a time threshold can be considered as the same data frame. The threshold is set to 0.05 seconds. The AP is calculated within the same data frame. For the entire experiment run, the mean and the standard deviation are calculated for all the data frames. This means the data frame is discarded if no detection or ground truth is present for the specific time stamp, i.e., the AP will not be affected if the robot onboard system or the edge computer is not capable of processing the image. The capability of processing the image is however reflected in the metric of the overall processed frame percentage as it counts how much percentage of all frames from the simulation is calculated.

The synchronous evaluation method does not take the execution latency into consideration, because the detection is compared against the ground truth with the same time stamp as the original image. The APs of the edge and the robot always remain the same and the final results are the interpolation of the two APs. However, this also does not represent the offloading scenario accurately. In real-world applications, the AMRs move continuously. The current situation of the AMR could be very different than the moment the image is taken by the camera if the processing takes too long and the detection is outdated. For example, if the AMR is turning at a speed of 60 degrees per second, the image will be completely different within two seconds. Therefore, the execution latency plays an important role in the accuracy of how well the AMRs perceives the environment.

With the aforementioned considerations, an asynchronous evaluation method that takes the execution latency into consideration is investigated, which is the evaluation method for AP metric that the thesis adopts. As illustrated in fig. 6.4, the detection is compared against the images with the time stamps when the AMRs actually receive the processed result. This is achieved by using the time stamps of the detection messages to the receiving time of the AMRs. Depending on how much the images have changed, the quality of the detection deteriorates with the execution latency, which is usually the case in real-world applications.

In the evaluation framework, since the pair bound between the ground truth and the detection no longer exists, the asynchronous evaluation method needs to match the ground truth with approximate detection. The data frames are created by a message filter by assigning the detection and ground truth with approximate timestamps into one data frame. This means that there will be ground truth data that are not matched with any detection. Evaluating the AP on the created data frames does not take this into consideration. Therefore, the AP metric is calculated by multiplying the AP on existing data frames and the overall processed frame rate. In this case, if a ground truth cannot be matched with any detection, the AP of that particular data frame is considered as 0.

6.4 Analysis

This section focuses on the analysis of the results from the experiments carried out in simulation and on a robotic system. First, the baseline offloading strategies are evaluated on the metrics defined in section 6.3. Then, the results of the dynamic offloading strategy are compared with the baseline strategies.

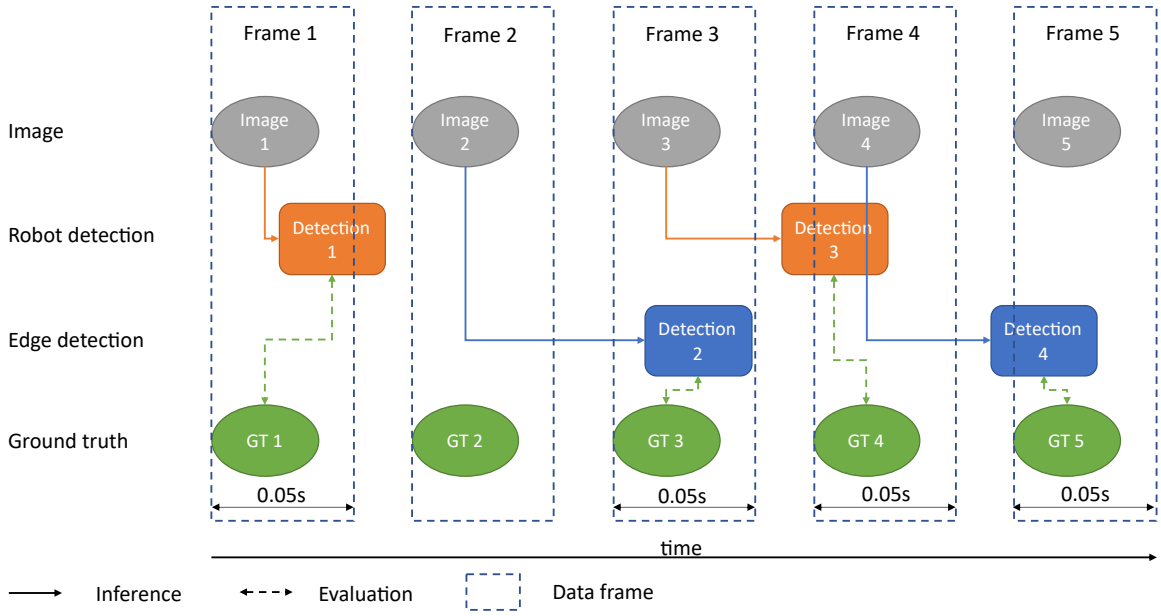

Figure 6.4: This figure shows the asynchronous evaluation method. A time parameter is added to the evaluation. The detection is compared with the RGB image of the time the AMR receives the detection output.

6.4.1 Evaluation of baseline strategies

When the network bandwidth is not constrained, the "edge only" strategy shows the best results in AP of the object detection task, as shown in fig. 6.5, while the "robot only" strategy has the worst results. As illustrated in fig. 6.7, the "edge only" strategy also processes the most frames in the experiments. The "robot only" strategy processes the least frames for about 50 percent of the total frames. However, the "edge only" strategy does not achieve the least RTT due to a slight increase in the network delay of the edge perception. Naturally, the "edge only" strategy uses the least CPU and the least power consumption of the robot's onboard system. The "edge only" strategy also uses the most bandwidth of the robot. More than 270 Mbps bandwidth is used for perception offloading.

The superiority of edge perception is mainly caused by two reasons. On one hand, the edge computer is using a more complex network, i.e., YOLOv5l, which provides more precise detection. On the other hand, the RTT of edge perception is lower than the robot perception thanks to its low inference time, as shown in fig. 6.6. In this case, the AMR is able to receive the detection from the edge faster than its onboard system. Moreover, the "edge only" strategy also processed the most frames in the experiments. This also makes the "edge only" strategy the safest among different strategies. With more detection frames, the AMR is able to adapt its behavior to the detection more quickly. Since the images are exclusively calculated on the edge computer, the "edge only" strategy uses less CPU and less energy from the onboard system but uses more resources from the edge computer. However, since the network is not constrained, the AMR is free to use as much network bandwidth as it desires. For image feed with 25 FPS at a resolution of 848 by 480 pixels in "bgr8" encoding, around 244 Mbps bandwidth is used for offloading. Such bandwidth will be a huge amount for wireless connection. For comparison, IEEE 802.11ac (Wi-Fi 5G) can support only up to 400 Mbps bandwidth with two 40 MHz channels [Int21]. Multiple access points on different frequencies can help increase the available bandwidth. However, if the number of the AMRs scales, the network cannot provide such bandwidth. Therefore, the "edge only" strategy is not suitable for industrial usage.

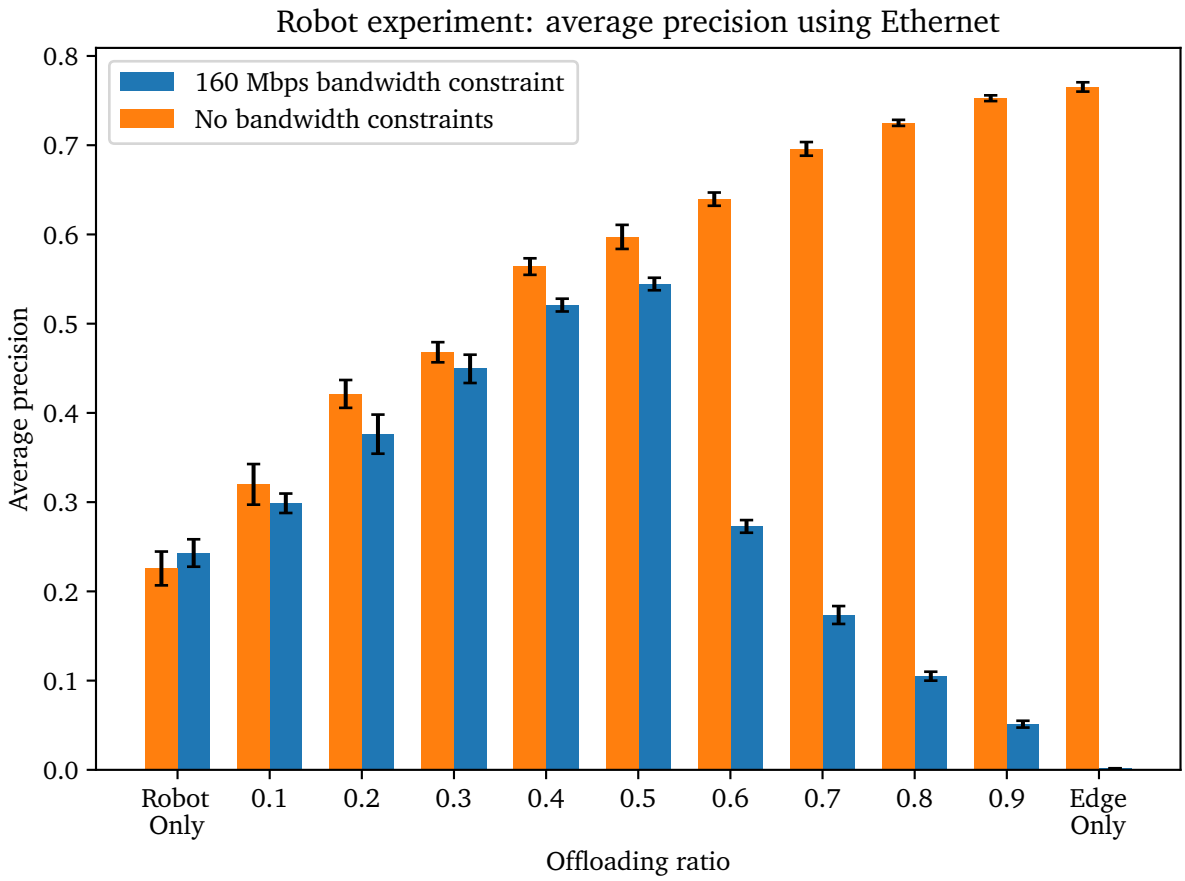

Figure 6.5: AP of different offloading ratios using Ethernet in robot experiment

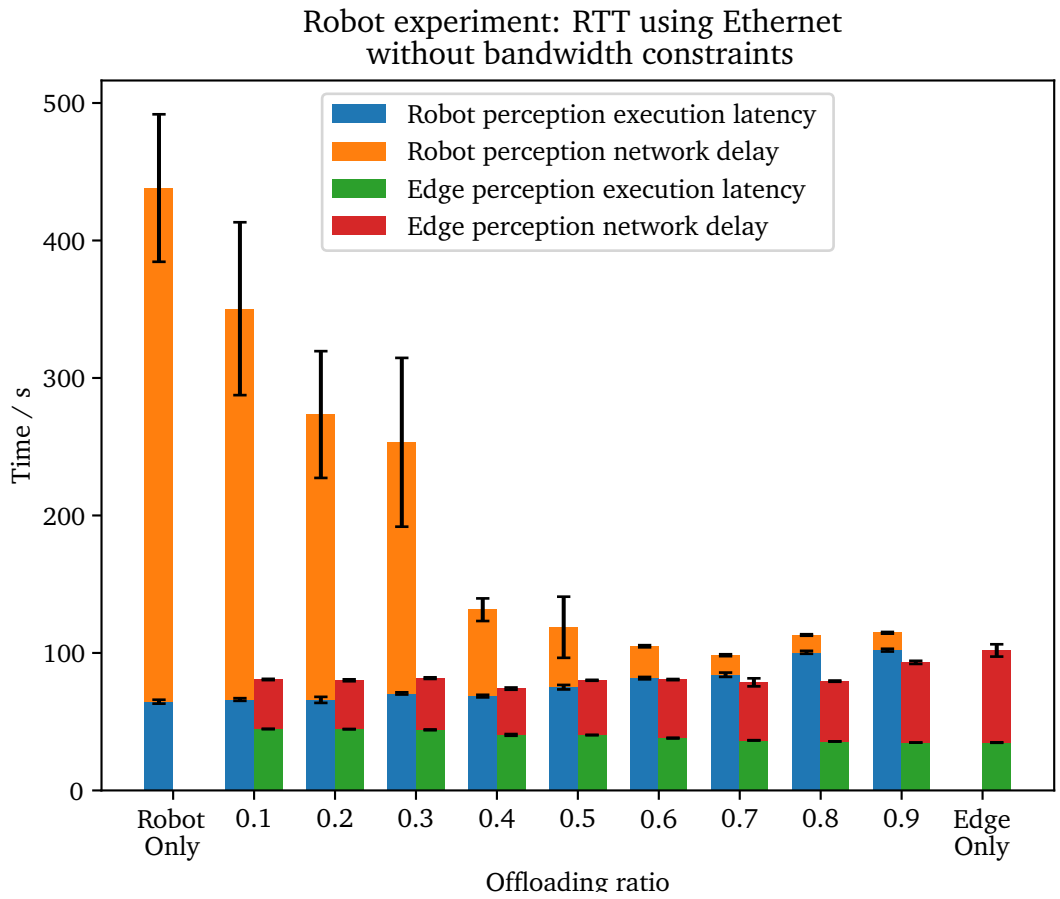

Figure 6.6: RTT of different offloading ratios using Ethernet without bandwidth constraints in robot experiment

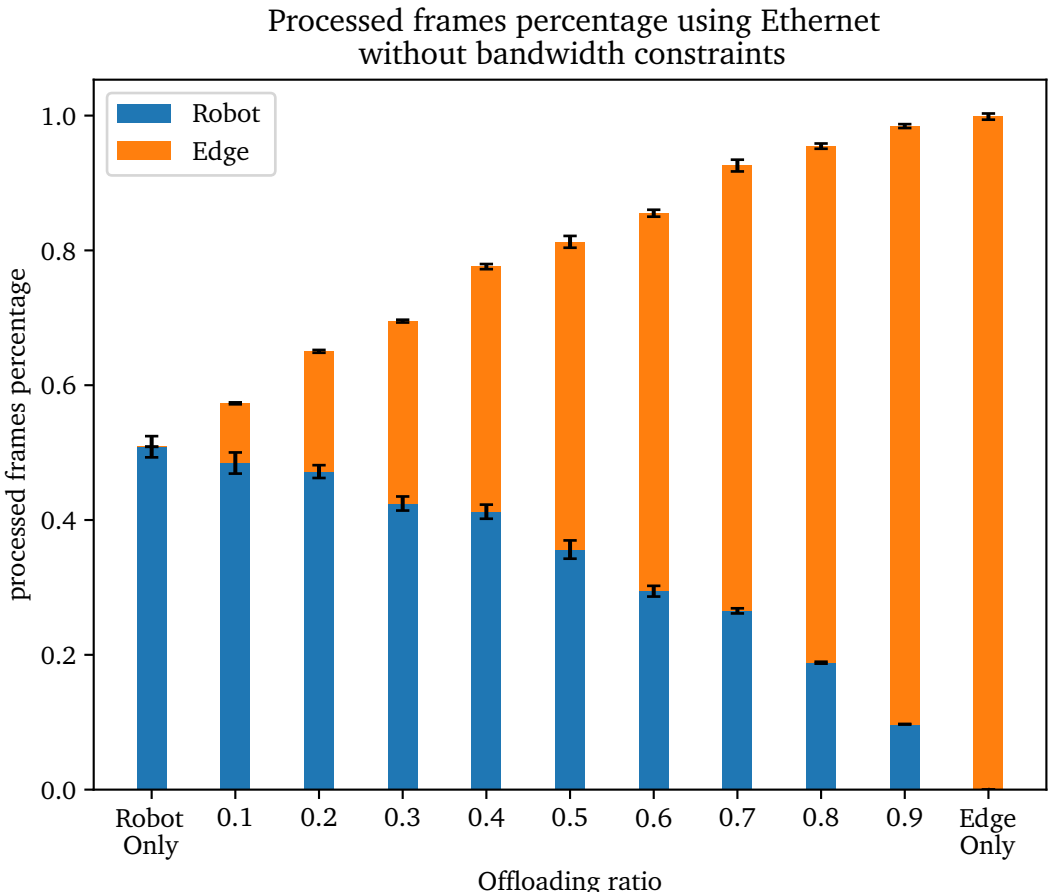

Figure 6.7: Processed frame percentage of different offloading ratios using Ethernet without bandwidth constraints in robot experiment

On the other hand, the "robot only" strategy delivers the worst results in the object detection task. This is mainly due to limited onboard resources. The robot perception module is strained when more images are calculated locally on the AMR's onboard system. As shown in fig. 6.12, the robot perception has the highest RTT with the "robot only" strategy. The RTT of robot perception is broken down into different components. The inference time of the robot perception stays the same. However, the network delay is increased. This indicates that the robot perception module has too many images waiting for processing and the images start to queue, causing the network latency of robot perception to increase. The network latency of robot perception stabilizes around the offloading ratio of 0.6. This infers that the AMR's onboard system, i.e., the NUC, is only able to process less than 40 percent of the offloaded images, i.e., around 12 FPS. This corresponds to the 99.9 ms RTT of the robot perception at the offloading ratio of 0.6. Naturally, the "robot only" strategy also uses the most CPU and energy, since it computes all images onboard, as shown in fig. 6.8 and fig. 6.9. Moreover, the "robot only" strategy still uses 0.4 MB/s network bandwidth because the state monitors need to send system state data from the edge computer to the robot's onboard system, such as messages for network delay.

As illustrated in fig. 6.5, the increase of the AP of the object detection task slows down between offloading ratios 0.5 and 0.8. This indicates that both the robot perception and the edge perception are working under their optimal workload. After this point, the edge perception becomes more dominant for the performance. The performance continues to improve because the edge is able to computer more frames and it causes the AP to increase.

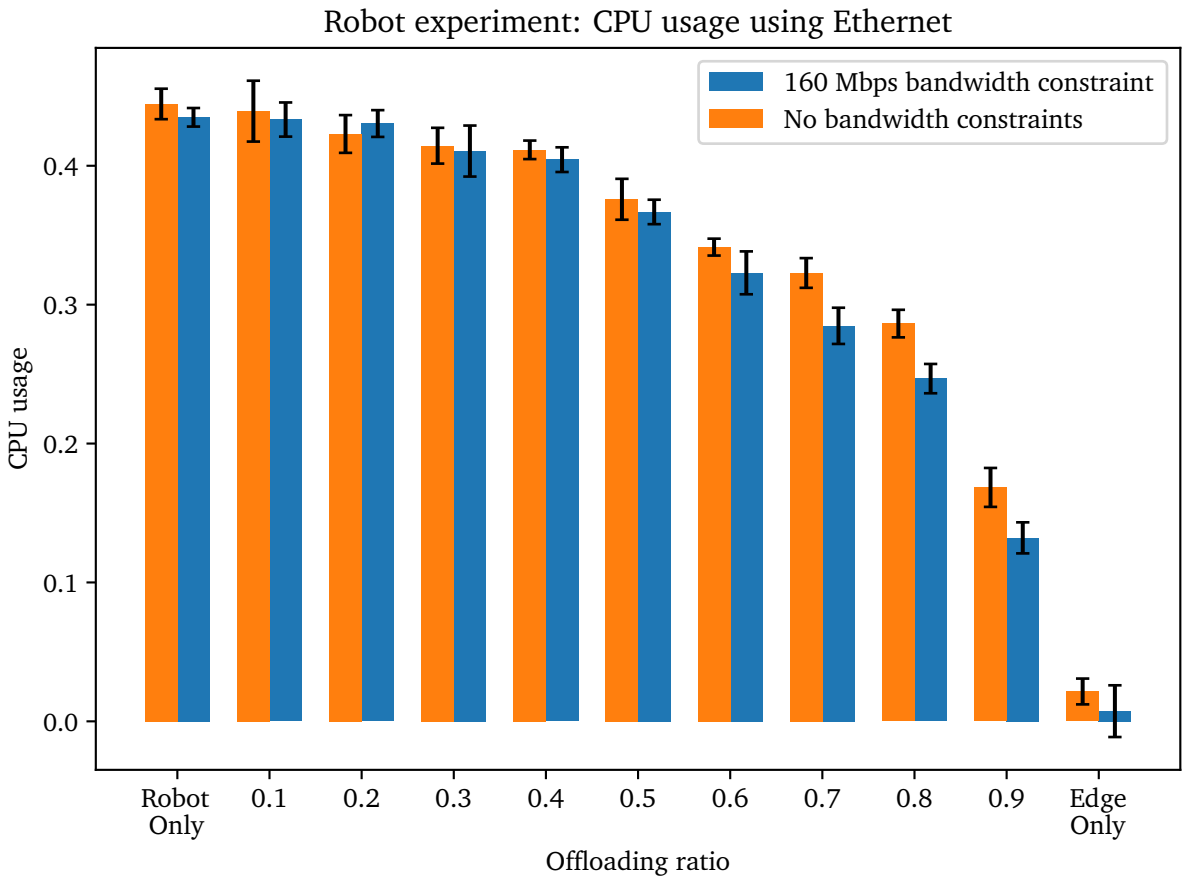

Figure 6.8: Robot CPU usage of different offloading ratios using Ethernet in robot experiment

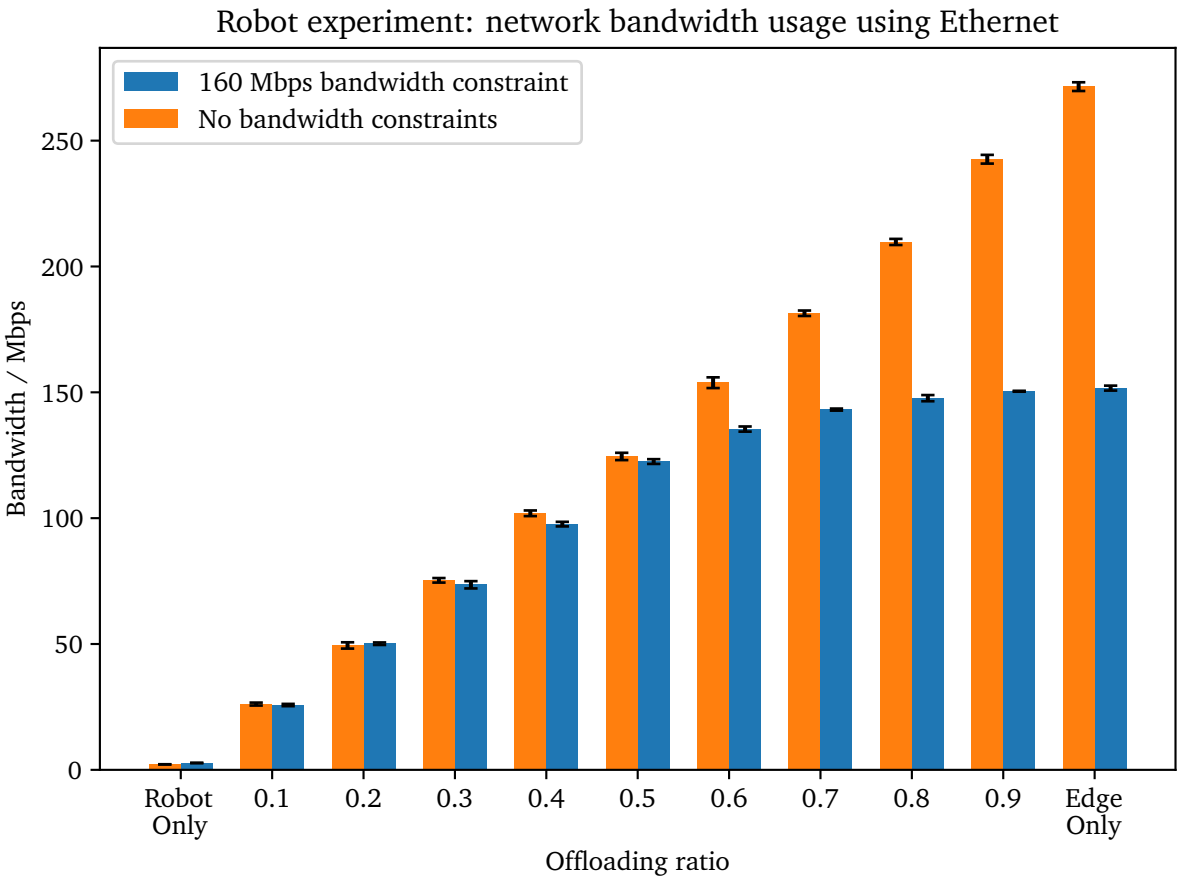

Figure 6.9: Robot CPU energy consumption of different offloading ratios using Ethernet in robot experiment

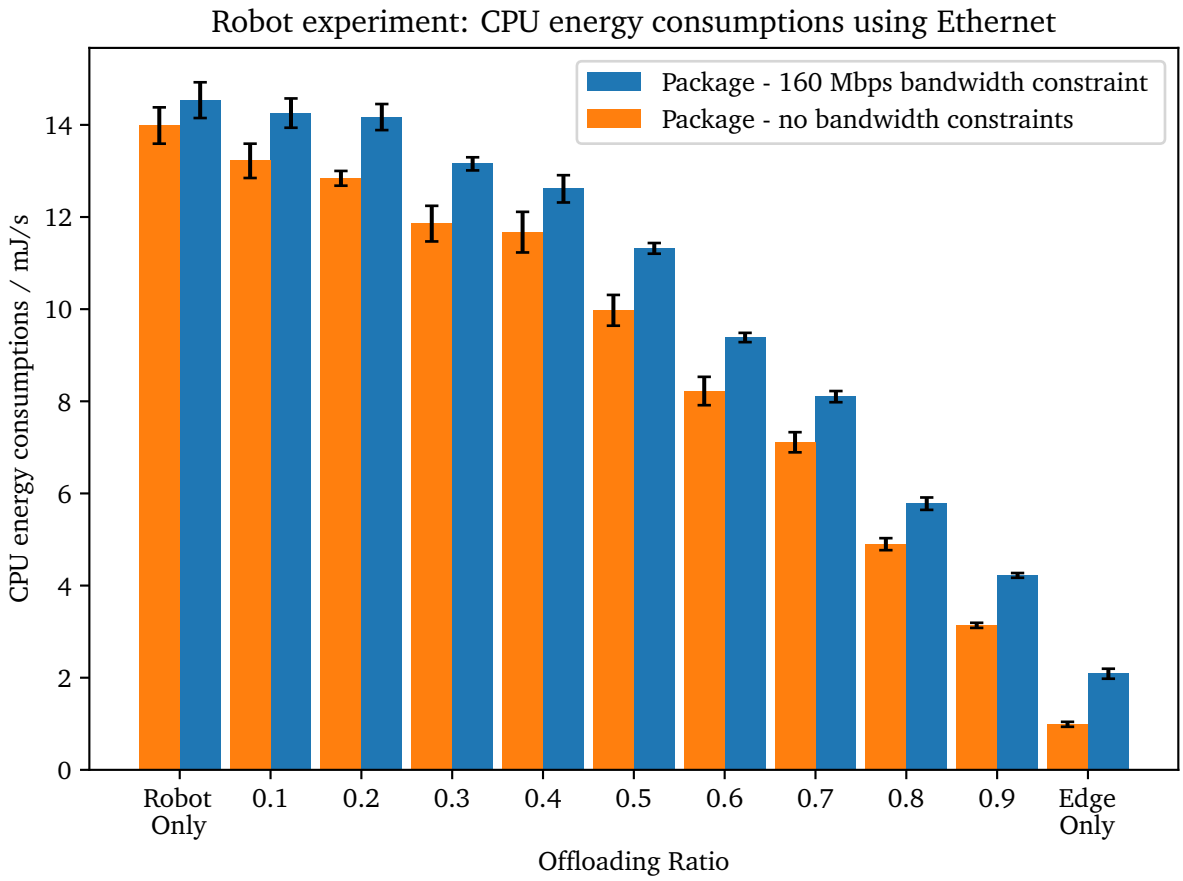

Figure 6.10: Network bandwidth usage of different offloading ratios using Ethernet in robot experiment

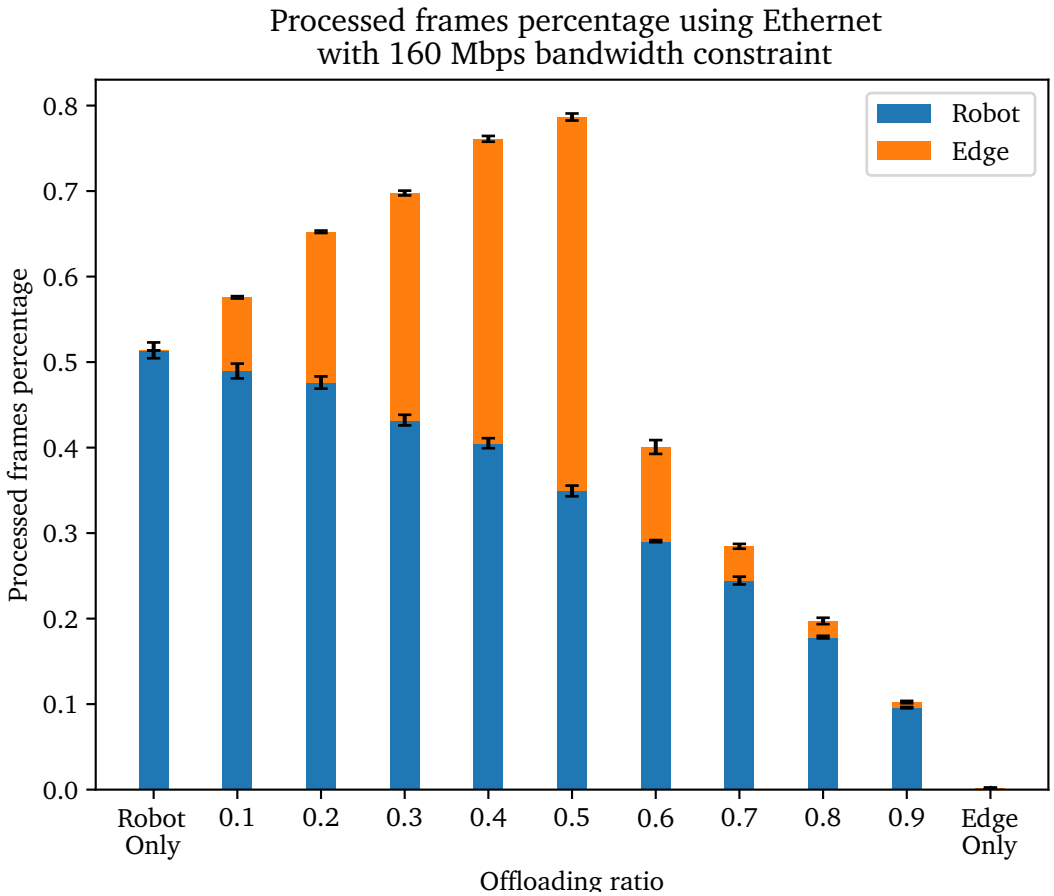

Figure 6.11: Processed frame percentage of different offloading ratios using Ethernet with 160 Mbps bandwidth constraint in robot experiment

When the network is constrained, the "edge only" strategy shows the worst result in AP, as shown in fig. 6.5. The "edge only" strategy also processed the least frames among different offloading strategies, as shown in fig. 6.11. The "edge only" strategy still uses the least CPU and energy of the AMR's onboard system. The bandwidth usage of the "edge only" strategy drops to 150 Mbps. The best performance occurs around the offloading ratio of 0.5. It also processes the most frames. Offloading 50 percent of the perception task also reduces CPU usage by 23 percent and reduces CPU power consumption by 16 percent, as shown in fig. 6.8 and fig. 6.9. It also uses only 45 percent of the network bandwidth compared to the "edge only" strategy without bandwidth constraints, as shown in fig. 6.10.

Since the network is not capable of transmitting the amount of data the strategy needs, the messages start to queue, and the best-effort reliability policy allows the middleware to drop many messages. This can be visualized and fig. 6.11. The processed frame percentage starts to drop drastically after the offloading ratio exceeds 0.5. This also corresponds to the network bandwidth in fig. 6.10. The network bandwidth is at its capacity after the offloading ratio exceeds 0.5. Furthermore, with the offloaded images hogging the network bandwidth, it is unlikely for the AMRs to transmit other data over the network, which can be essential for the safety of the AMRs. This makes the "edge only" strategy the worst strategy under constrained network conditions.

On the other hand, the "robot only" strategy delivers similar results under the two network conditions. Since the "robot only" strategy computes the object detection locally on the onboard system, the performance should be independent of the network conditions. How-

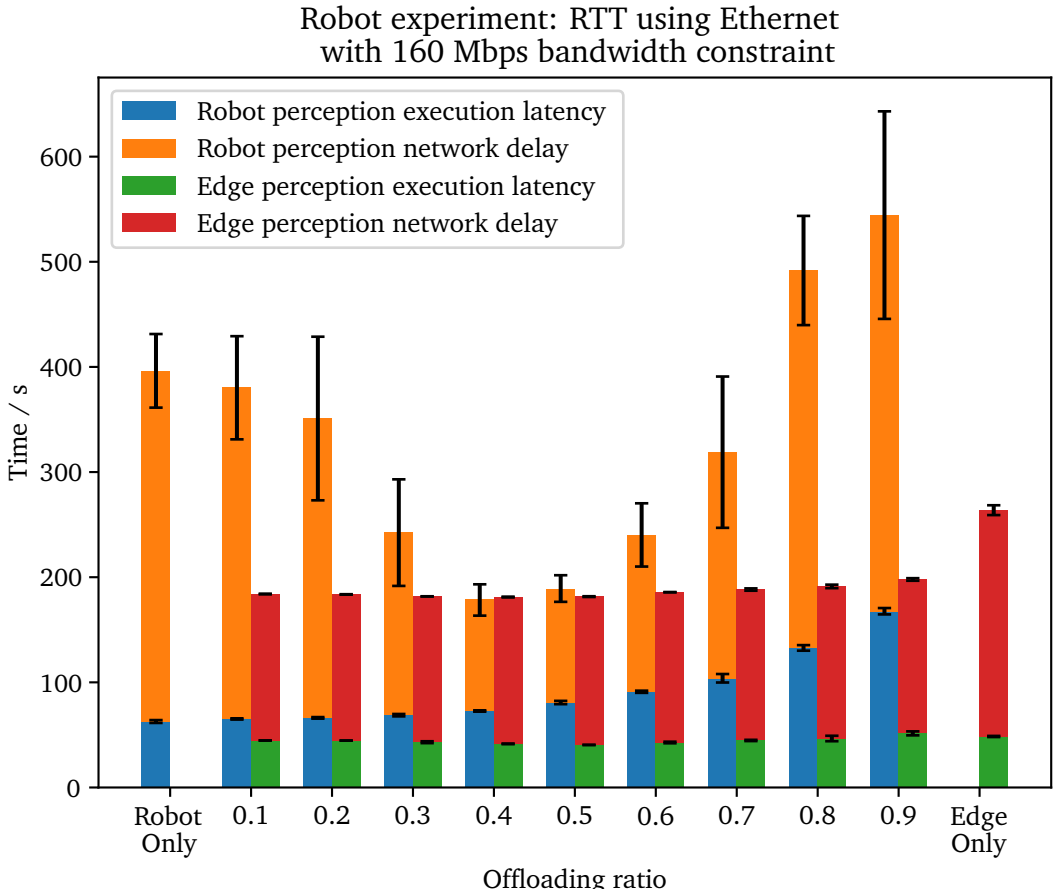

Figure 6.12: RTT of different offloading ratios using Ethernet with 160 Mbps bandwidth constraint in robot experiment

ever, it is worth noticing that the network delay increases when the network is occupied with offloaded images. It is suspected that the NetEm layer affects the ROS middleware and thus causes an increased latency in transmitting the data between the offloading module and the robot perception module.

At this offloading ratio of 0.5, the network bandwidth has not exceeded its limits and the robot perception is able to process the given workload. Therefore, the RTTs of both perception modules are low and the AMR is able to get the detection in time. Moreover, offloading at this ratio also allows the robot to use fewer onboard resources. Therefore, under constrained network conditions, offloading a portion of the images is the best strategy. However, the offloading ratio is dependent on the available network bandwidth and the computation capabilities of the AMR's onboard system.

6.4.2 Evaluation of dynamic offloading strategy

From the evaluation of the baseline strategies, the RTT is an important metric to improve the performance of the perception task. Therefore, the RTT is used as a criterion for making offloading decisions for a dynamic offloading strategy, which is introduced in section 4.2. The dynamic offloading strategy is evaluated under the same conditions as the evaluation carried out in the previous section. Three network conditions are used to evaluate the dynamic offloading strategy: Ethernet with 160 Mbps bandwidth constraint, Ethernet without bandwidth constraints, and Wi-Fi network.

The comparison conducted with an Ethernet connection with constrained bandwidth is shown in table 6.3. The dynamic offloading strategy is compared with the baseline strategies. In the baseline strategies, the "edge only" strategy, the "robot only" strategy, and the strategy that achieves the best AP among different offloading ratios are selected to compare with the dynamic offloading strategy. When the network bandwidth is constrained, the dynamic offloading does not provide good results compared to the strategy that offloads with an offloading ratio of 0.5. However, it still manages to compute 45.47 percent of the frames, most of which are computed on the robot.

Strategies	Robot only	Edge only	Best among ratios	Dynamic offloading
AP	24.30% ± 1.54%	0.16% ± 0.01%	**54.44% ± 0.70%**	23.21% ± 8.67%
Robot perception RTT / ms	729.83 ± 70.02	/	**297.63 ± 25.35**	731.50 ± 130.64
Edge perception RTT / ms	/	479 ± 9.36	**322.59 ± 0.72**	1195.19 ± 99.52
Processed frames percentage	51.37% ± 0.93%	0.22% ± 0	**78.66% ± 0.75%**	45.47% ± 15.74%
CPU usage	43.49% ± 0.67%	**0.73% ± 1.86%**	36.68% ± 0.88%	35.32% ± 13.64%
Power consumption / mJ/s	14.54 ± 0.39	**2.09 ± 0.11**	11.32 ± 0.12	11.91 ± 3.69
Bandwidth usage / Mbps	**2.73 ± 0.08**	151.67 ± 0.98	97.64 ± 0.91	38.18 ± 42.60

Table 6.3: Metrics of dynamic offloading strategy compared with baseline strategies using Ethernet with 160 Mbps bandwidth constraint

The reason for the poor results of the dynamic offloading strategy under constrained network bandwidth is that even though the AMR can hardly get any detection back from the edge perception node, there are still a small number of detection received by the edge perception. The inference time and the network delay of them are used to compute the RTT of the edge perception. However, the values of RTT are relatively low thanks to the best-effort strategy used for offloading. Therefore, the offloading module prefers the edge perception over the robot perception. This causes the results of the dynamic offloading strategy to deteriorate under constrained network bandwidth.

The comparison conducted with an unconstrained Ethernet connection is shown in table 6.4. Since the "edge only" strategy outperforms all strategies with different offloading ratios in terms of AP, only the "edge only" strategy is included in the table. There, the dynamic offloading strategy offers similar metrics to the "edge only" strategy. The dynamic offloading strategy achieves the best AP among all offloading strategies and achieves the lowest RTT of robot perception compared to the "robot only" and the "edge only" strategy. However, it is worth noticing that the dynamic offloading strategy does not offload all frames to the edge computer. It still keeps a small portion of the frames computed locally with the robot's onboard system.

Strategies	Robot only	Edge only	dynamic offloading
AP	22.57% ± 1.89%	76.53% ± 0.52%	**77.02% ± 0.46%**
Robot perception RTT / ms	811.75 ± 107.29	/	**544.50 ± 115.42**
Edge perception RTT / ms	/	**168.82 ± 8.96**	177.27 ± 13.04
Processed frames percentage	50.87% ± 1.57%	**99.85% ± 0.45%**	99.84% ± 0.15%
CPU usage	44.45% ± 1.10%	**2.15% ± 0.93%**	3.36% ± 1.22%
Power consumption / mJ/s	13.98 ± 0.39	**0.98 ± 0.05**	1.20 ± 0.07
Bandwidth usage / Mbps	**2.18 ± 0.04**	271.46 ± 1.73	272.84 ± 1.85

Table 6.4: Metrics of dynamic offloading strategy compared with baseline strategies using Ethernet without bandwidth constraints

Finally, the dynamic offloading strategy is evaluated on a Wi-Fi network. The dynamic offloading strategy provides decent results under the Wi-Fi network. It manages to achieve 45.81 percent of AP and processes over 80 percent of the overall frames. Compared to the strategy that offloads with a ratio of 0.5, the dynamic offloading strategy uses less onboard CPU and less onboard power. However, it uses more network bandwidth on average and the RTTs of the robot perception and the edge perception is comparably higher.

Strategies	Robot only	Edge only	Best among ratios	dynamic offloading
AP	25.17% ± 2.25%	32.20% ± 14.73%	45.70% ± 8.07%	**45.81% ± 22.34%**
Robot perception RTT / ms	653.70 ± 130.30	/	**634.23 ± 394.16**	992.06 ± 226.89
Edge perception RTT / ms	/	606.02 ± 226.21	**417.96 ± 214.35**	607.47 ± 410.55
Processed frames percentage	49.01% ± 1.31%	71.60% ± 26.38%	75.28% ± 2.90%	**80.31% ± 34.85%**
CPU usage	43.55% ± 1.29%	**3.42% ± 2.29%**	38.89% ± 2.82%	8.86% ± 7.70%
Power consumption / mJ/s	13.37 ± 0.42	**0.58 ± 0.25**	11.17 ± 0.85	2.03 ± 2.44
Bandwidth usage / Mbps	**1.86 ± 0.11**	112.21 ± 72.77	91.49 ± 1.83	203.79 ± 93.75

Table 6.5: Metrics of dynamic offloading strategy compared with baseline strategies using Wi-Fi

It is worth noticing that the metrics have high standard deviation under the Wi-Fi network. The reason is that the network condition is inconsistent among different experiment runs. However, such fluctuation is intended to investigate the performance of the dynamic offloading strategy under dynamic network conditions. The difference between the dynamic offloading strategy and the ratio strategy is caused by two factors. First, the ratio offloading strategy achieves lower RTT of the edge and the robot perception. Second, the ratio strategy also processes more frames than the dynamic offloading strategy and thus achieves better results in AP. Moreover, with the low standard deviation of the ratio strategy, better results could be caused by a transient maximum of the network bandwidth during the experiment.

Chapter 7

Conclusion and Future Work

7.1 Summary and Discussion

In order to investigate the influence of different offloading strategies on the important metrics of the AMRs, this thesis designs and implements an offloading framework for perception offloading using ROS. The robotic perception task is chosen to use YOLOv5 algorithms with an image stream with a frame rate of around 25 FPS and a resolution of 848 by 480 pixels. The bandwidth usage of an entire computation offloading can amount to 244 Mbps. To balance the computation resources of the two systems and the execution latency of the object detection task. Two models with different sizes are chosen respectively for the AMR's onboard system and the edge computer.

Afterward, this thesis implements the baseline offloading strategies and conducted experiments in simulation. In order to carry out the experiment, a simulated scenario of a factory warehouse is implemented for AMRs in Gazebo. Furthermore, different experimental setups are created for experiments in simulation and with actual robotic systems. In addition, this thesis designs and implements an asynchronous evaluation method by considering the actual robotic application of perception offloading. After identifying the RTT as an important factor for the performance of the object detection task as well as the AMR's onboard system, this thesis implements a dynamic offloading strategy by using RTT as an offloading decision-making criterion with the goal of improving the performance of the object detection task and reducing the execution latency as well as the usage of the AMR's onboard resources. Finally, this thesis conducts an experiment with an actual robotic system and an edge computer to evaluate the dynamic offloading strategy against the baseline strategies. The experiment is conducted with Ethernet and Wi-Fi connection.

The experiment results show that both "edge only" and "robot on" strategies are not ideal for computing time-sensitive tasks like object detection. Moreover, the results show that the offloading ratio is dependent on the available onboard resources and the network condition for a partially offloading strategy. From the robot experiments, the dynamic offloading strategy shows the ability to adapt to dynamic network changes and improves the metrics compared to baseline strategies. More specifically, the dynamic offloading strategy improves the average precision of the object detection on human class compared to "robot only" and "edge only" strategies, while drastically reducing the CPU usage and CPU power consumption compared to "robot only" strategy.

7.2 Limitations

The pre-trained YOLOv5 object detection models impose some limitations on the evaluation of the experiment results. Since the pre-trained models are trained with COCO dataset, only the person class can be accurately detected. Other obstacles, such as shelves, boxes, and pallets, are not considered in the evaluation of the average precision of the object detection task. However, they still contribute to the complexity of the simulation scenario by providing false positive detections and occlusions, as discussed in section 6.1.

Another limitation results from the Gazebo simulation. Due to technical issues, the bounding box camera sensor in Gazebo cannot generate accurate bounding box ground truth for moving actors in Gazebo. However, to the author's knowledge, dynamic human obstacles can only be implemented as actors in Gazebo. Therefore, the human obstacles are static in the simulated scenario. Luckily, the AMR is implemented dynamically in the simulated scenario. In the view of the camera, the human obstacles are still static. However, this still imposes some limitations on the simulated scenario since the animation of the human obstacles is disabled.

7.3 Future Work

Foremost, the offloading pipeline is implemented in an open-loop manner, i.e., the processed results are not used for downstream applications. In future works, the object detection results can be used for obstacle avoidance and local path planning to increase the AMR's safety. This will allow the offloading strategy to modify the behavior of the AMR and create more meaningful metrics for offloading strategies, e.g., safety metrics. The experiment can measure the times when the AMRs collides with an obstacle.

Additionally, this thesis only considers the scenario where one AMR offloads to one edge computer to reduce the complexity in evaluation. For future works, the number of the AMRs can also be increased while there is still only one edge computer. Within the work of this thesis, the bandwidth constraints are achieved by limiting the available bandwidth between the AMR and the edge computer. However, in real-world applications, the AMRs will also compete over the available bandwidth of the edge computer. In that case, the dynamic offloading strategy will also have to consider such competition between AMRs. For example, the game-theory approaches, described in section 3.2, can be considered for reconciling the resource competition among multiple AMRs. Furthermore, other metrics can be considered for multi-robot offloading, such as the joint efficiency that measures the overall energy consumption and the network bandwidth usage.

Appendix A

Results for Robot Experiment with Reliable QoS Reliability Policy

The experiment with the actual robotic system is also carried out in reliable QoS reliability policy. The ROS middleware still uses Fast-DDS. The Ethernet connection is used and the bandwidth is constrained to 160 Mbps by NetEm.

The results are shown in table A.1. In general, the performance of the object detection task is lower compared to using the best effort QoS reliability policy. The best offloading ratio occurs at 0.2. Since the reliable policy requires a confirmation of the receipt of the message, both robot and edge RTTs take longer. The "edge only" strategy cannot process any frames. Therefore, it does not have any measurement in terms of object detection performance. The dynamic offloading strategy still manages to adapt to these network changes and outperforms the "robot only" strategy by 1.5% in AP.

Strategies	Robot only	Edge only	Best among ratios	dynamic offloading
AP	18.08% ± 0.99%	0	**31.01**% ± **1.24**%	19.47% ± 1.54%
Robot perception RTT / ms	1634.72 ± 64.30	/	1781.34 ± 142.89	**1594.10 ± 109.72**
Edge perception RTT / ms	/	/	**322.47 ± 0.21**	2517.29 ± 257.00
Processed frames percentage	59.87% ± 2.45%	0	**68.88**% ± **0.27**%	61.92% ± 4.07%
CPU usage	48.69% ± 0.85%	**6.35**% ± **0.80**%	49.24% ± 0.65%	48.00% ± 0.81%
Power consumption / mJ/s	17.17 ± 0.73	**2.14 ± 0.09**	16.21 ± 0.05	16.94 ± 0.89
Bandwidth usage / Mbps	**3.22 ± 0.14**	151.73 ± 0.92	59.50 ± 0.77	12.79 ± 0.97

Table A.1: Metrics of dynamic offloading strategy compared with baseline strategies using Ethernet with 160 Mbps bandwidth constraint and reliable QoS reliability policy

Bibliography

[OMG] (OMG), O. M. G. *Real-time Publish-Subscribe (RTPS)*. URL: https://www.omgwiki.org/ddsf/doku.php?id=ddsf:public:guidebook:06_append:glossary:r:rtps (visited on 05/03/2023).

[Bax+22] Baxi, A., Eisen, M., Sudhakaran, S., Oboril, F., Murthy, G. S., Mageshkumar, V. S., Paulitsch, M., and Huang, M. "Towards Factory-Scale Edge Robotic Systems: Challenges and Research Directions". In: *IEEE Internet of Things Magazine* 5.3 (2022), pp. 26–31. ISSN: 2576-3180. DOI: 10.1109/IOTM.001.2200056.

[BWL20] Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. M. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV].

[Bon+12] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: Association for Computing Machinery, 2012, pp. 13–16. ISBN: 9781450315197. DOI: 10.1145/2342509.2342513.

[Cha+22] Chaâri, R., Cheikhrouhou, O., Koubâa, A., Youssef, H., and Gia, T. N. "Dynamic computation offloading for ground and flying robots: Taxonomy, state of art, and future directions". In: *Computer Science Review* 45 (Aug. 2022), p. 100488. DOI: 10.1016/j.cosrev.2022.100488.

[CLD15] Chen, M.-H., Liang, B., and Dong, M. "A semidefinite relaxation approach to mobile cloud offloading with computing access point". In: (June 2015). DOI: 10.1109/spawc.2015.7227025.

[Che+16] Chen, X., Jiao, L., Li, W., and Fu, X. "Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing". In: *IEEE/ACM Transactions on Networking* 24.5 (Oct. 2016), pp. 2795–2808. DOI: 10.1109/tnet.2015.2487344.

[Chi+19] Chinchali, S., Sharma, A., Harrison, J., Elhafsi, A., Kang, D., Pergament, E., Cidon, E., Katti, S., and Pavone, M. "Network Offloading Policies for Cloud Robotics: A Learning-Based Approach". In: (June 2019). DOI: 10.15607/rss.2019.xv.063.

[Cor21] Corporation, I. *The Edge Infrastructure Handbook*. Intel Corporation. 2021. URL: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/intel-edge-handbook-2021-final.pdf (visited on 05/04/2023).

[Dem+19] Demidovskij, A., Gorbachev, Y., Fedorov, M., Slavutin, I., Tugarev, A., Fatekhov, M., and Tarkan, Y. *OpenVINO Deep Learning Workbench: Comprehensive Analysis and Tuning of Neural Networks Inference*. 2019. DOI: 10.1109/iccvw.2019.00104.

[Fu+19] Fu, M., Sun, S., Ni, K., and Hou, X. "Mobile Robot Object Recognition in The Internet of Things based on Fog Computing". In: *2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. 2019, pp. 1838–1842. DOI: 10.1109/APSIPAASC47483.2019.9023187.

[GL18] Guo, H. and Liu, J. "Collaborative Computation Offloading for Multiaccess Edge Computing Over Fiber–Wireless Networks". In: *IEEE Transactions on Vehicular Technology* 67.5 (May 2018), pp. 4514–4526. DOI: 10.1109/tvt.2018.2790421.

[He+17] He, K., Gkioxari, G., Dollár, P., and Girshick, R. "Mask R-CNN". In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988. DOI: 10.1109/ICCV.2017.322.

[He+16] He, K., Zhang, X., Ren, S., and Sun, J. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[Hon+19] Hong, Z., Huang, H., Guo, S., Chen, W., and Zheng, Z. "QoS-Aware Cooperative Computation Offloading for Robot Swarms in Cloud Robotics". In: *IEEE Transactions on Vehicular Technology* 68.4 (Apr. 2019), pp. 4027–4041. ISSN: 0018-9545. DOI: 10.1109/tvt.2019.2901761.

[Hua+19] Huang, L., Feng, X., Zhang, C., Qian, L., and Wu, Y. "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing". In: *Digital Communications and Networks* 5.1 (2019). Artificial Intelligence for Future Wireless Communications and Networking, pp. 10–17. ISSN: 2352-8648. DOI: 10.1016/j.dcan.2018.10.003. URL: https://www.sciencedirect.com/science/article/pii/S2352864818301469.

[Hua+22] Huang, P., Zeng, L., Chen, X., Huang, L., Zhou, Z., and Yu, S. "Edge Robotics: Edge-Computing-Accelerated Multirobot Simultaneous Localization and Mapping". In: *IEEE Internet of Things Journal* 9.15 (Aug. 2022), pp. 14087–14102. ISSN: 2327-4662. DOI: 10.1109/jiot.2022.3146461.

[Int21] Intel Corporation. *Different Wi-Fi Protocols and Data Rates*. 2021. URL: https://www.intel.com/content/www/us/en/support/articles/000005725/wireless/legacy-intel-wireless-products.html (visited on 10/28/2021).

[Joc+22] Jocher, G., Chaurasia, A., Stoken, A., Borovec, J., NanoCode012, Kwon, Y., Michael, K., TaoXie, Fang, J., imyhxy, Lorna, Yifu, Z., Wong, C., V, A., Montes, D., Wang, Z., Fati, C., Nadar, J., Laughing, UnglvKitDe, Sonck, V., tkianai, yxNONG, Skalski, P., Hogan, A., Nair, D., Strobel, M., and Jain, M. *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*. Version v7.0. Nov. 2022. DOI: 10.5281/zenodo.7347926. URL: https://doi.org/10.5281/zenodo.7347926.

[Kek+18] Kekki, S., Featherstone, W., Fang, Y., Kuure, P., Li, A., Ranjan, A., Purkayastha, D., Jiangping, F., Frydman, D., Verin, G., Wen, K.-W., KwihoonKim, Arora, R., Odgers, A., Contreras, L. M., and Scarpina, S. *MEC in 5G networks*. Tech. rep. ETSI, June 2018.

[Lin+19] Lin, L., Liao, X., Jin, H., and Li, P. "Computation Offloading Toward Edge Computing". In: *Proceedings of the IEEE* 107.8 (Aug. 2019), pp. 1584–1607. DOI: 10.1109/jproc.2019.2922285.

[Lin+14] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. "Microsoft COCO: Common Objects in Context". In: (2014), pp. 740–755. ISSN: 0302-9743. DOI: 10.1007/978-3-319-10602-1_48.

[Liu+16] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., and Berg, A. C. "SSD: Single Shot MultiBox Detector". In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. DOI: 10.1007/978-3-319-46448-0_2.

[Lu+20] Lu, H., Gu, C., Luo, F., Ding, W., and Liu, X. "Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning". In: *Future Generation Computer Systems* 102 (Jan. 2020), pp. 847–861. DOI: 10.1016/j.future.2019.07.019.

[Mac+20] Macenski, S., Martín, F., White, R., and Clavero, J. G. "The Marathon 2: A Navigation System". In: (2020). DOI: 10.48550/ARXIV.2003.00368.

[Mac+22] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (May 2022). DOI: 10.1126/scirobotics.abm6074.

[Mar+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[Nin+19] Ning, Z., Dong, P., Kong, X., and Xia, F. "A Cooperative Partial Computation Offloading Scheme for Mobile Edge Computing Enabled Internet of Things". In: *IEEE Internet of Things Journal* 6.3 (June 2019), pp. 4804–4814. DOI: 10.1109/jiot.2018.2868616.

[Ope20] Open Robotics. *SDFormat Documentation*. 2020. URL: http://sdformat.org/ (visited on 05/03/2023).

[Ope] Open Robotics. *Gazebo Simulation*. URL: https://staging.gazebosim.org (visited on 05/03/2023).

[Pas+19] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[PM21] Penmetcha, M. and Min, B.-C. "A Deep Reinforcement Learning-Based Dynamic Computational Offloading Method for Cloud Robotics". In: *IEEE Access* 9 (2021), pp. 60265–60279. DOI: 10.1109/access.2021.3073902.

[Pha+18] Pham, Q.-V., Leanh, T., Tran, N. H., Park, B. J., and Hong, C. S. "Decentralized Computation Offloading and Resource Allocation for Mobile-Edge Computing: A Matching Game Approach". In: *IEEE Access* 6 (2018), pp. 75868–75885. DOI: 10.1109/access.2018.2882800.

[Qui+09] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. Y. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[Red+16] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. "You Only Look Once: Unified, Real-Time Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 779–788. DOI: 10.1109/cvpr.2016.91.

[Ren+15] Ren, S., He, K., Girshick, R., and Sun, J. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems*. Ed. by Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

[ROS21] ROS Galactic. *ROS Galactic Documentation*. 2021. URL: https://docs.ros.org/en/galactic/index.html (visited on 05/03/2023).

[Rug21] Ruggeri, F. "Safety-Oriented Task Offloading for Human-Robot Collaboration: A Learning-Based Approach". MA thesis. KTH, 2021. URL: https://kth.diva-portal.org/smash/record.jsf?dswid=3553&faces-redirect=true&language=en&searchType=LIST_LATEST&query=&af=[]&aq=[[]]&aq2=[[]]&aqe=[]&pid=diva2:1624928&noOfRows=50&sortOrder=author_sort_asc&sortOrder2=title_sort_asc&onlyFullText=false&sf=all.

[RLS18] Ruiz-del-Solar, J., Loncomilla, P., and Soto, N. *A Survey on Deep Learning Methods for Robot Vision*. 2018. arXiv: 1803.10862 [cs.CV].

[Sae+21] Saeik, F., Avgeris, M., Spatharakis, D., Santi, N., Dechouniotis, D., Violos, J., Leivadeas, A., Athanasopoulos, N., Mitton, N., and Papavassiliou, S. "Task offloading in Edge and Cloud Computing: A survey on mathematical, artificial intelligence and control theory solutions". In: *Computer Networks* 195 (Aug. 2021), p. 108177. DOI: 10.1016/j.comnet.2021.108177.

[SSV21] Saleh, K., Szénási, S., and Vámossy, Z. "Occlusion Handling in Generic Object Detection: A Review". In: *2021 IEEE 19th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*. 2021, pp. 000477–000484. DOI: 10.1109/SAMI50585.2021.9378657.

[Sat+23] Satka, Z., Ashjaei, M., Fotouhi, H., Daneshtalab, M., Sjödin, M., and Mubeen, S. "A comprehensive systematic review of integration of time sensitive networking and 5G communication". In: *Journal of Systems Architecture* 138 (2023), p. 102852. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2023.102852. URL: https://www.sciencedirect.com/science/article/pii/S1383762123000310.

[Sat+09] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82.

[SZ15] Simonyan, K. and Zisserman, A. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[Sos+22] Sossalla, P., Rischke, J., Nguyen, G. T., and Fitzek, F. H. P. "Offloading Robot Control with 5G". In: (Jan. 2022). DOI: 10.1109/ccnc49033.2022.9700709.

[Tan+] Tanwani, A. K., Mor, N., Kubiatowicz, J., Gonzalez, J. E., and Goldberg, K. "A Fog Robotics Approach to Deep Robot Learning: Application to Object Recognition and Grasp Planning in Surface Decluttering". In: *Proceedings - IEEE International Conference on Robotics and Automation* (). URL: https://par.nsf.gov/biblio/10111294.

[The] The Linux Documentation Project. *tc-netem(8) - Linux man page*. URL: https://man7.org/linux/man-pages/man8/tc-netem.8.html (visited on 05/03/2023).

[Wik] Wikipedia. *Gazebo simulator*. URL: https://en.wikipedia.org/wiki/Gazebo_simulator (visited on 05/03/2023).

[Xie+21] Xie, Y., Guo, Y., Chen, Y., and Mi, Z. "Real-Time Instance Segmentation for Low-Cost Mobile Robot Systems Based on Computation Offloading". In: (Oct. 2021). DOI: 10.1109/ccci52664.2021.9583186.

[XYL20] Xu, F., Yang, W., and Li, H. "Computation offloading algorithm for cloud robot based on improved game theory". In: *Computers and Electrical Engineering* 87 (Aug. 2020), p. 106764. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2020.106764.

[Zha+15] Zhao, Y., Zhou, S., Zhao, T., and Niu, Z. "Energy-efficient task offloading for multiuser mobile cloud computing". In: (Nov. 2015). DOI: 10.1109/iccchina.2015.7448613.