UCLouvain

École polytechnique de Louvain

Implementation of a robotic swarm platform based on the Balboa self-balancing robot

Towards cooperative target localization

Author: **Romain Englebert**
Supervisors: **Gianluca Bianchin, François Wielant**
Readers: **Gianluca Bianchin, François Wielant, Julien Hendrickx**
Academic year 2024–2025
Master [120] in Electro-mechanical Engineering

Acknowledgments

The completion of this master's thesis represents the culmination of months of effort, but it would not have been possible without the support, advice, and encouragement of many people whom I would like to sincerely thank.

First, I wish to express my deep gratitude to Prof. Gianluca Bianchin, who supervised this work and guided me throughout the project. His expertise, availability, and constant encouragement have been invaluable at every stage, from the early design choices to the final development guidelines. His attentive listening during our discussions and presentations, as well as his consistently positive attitude, have been a true source of motivation throughout this journey. I sincerely thank him for his trust and his insightful advice. It has been a real pleasure to carry out this project under his supervision.

I also wish to express my sincere thanks to François Wielant, whose technical support was equally essential to the success of this project. His practical and conceptual advice, his assistance in designing the mechanical integration, as well as his help in ordering and managing all the necessary hardware components, were invaluable throughout the project. His dedication, availability, and proactive involvement greatly facilitated the realization of this work, and I am deeply grateful for the support he offered throughout this adventure.

Additionally, I also want to thank Prof. Julien Hendrickx, member of my jury, for his time and the attention given to reviewing my work.

I would also like to thank Giuseppe Speciale for our collaboration. His work on distributed algorithms and the valuable explanations he shared with me greatly contributed to my ability to implement them on a real-world system.

Finally, I wish to warmly thank my parents for their support throughout my studies. Their encouragement, patience, and belief in me have been a constant source of strength and motivation, making this achievement possible.

Abstract

Swarm robotics draws inspiration from natural systems to implement decentralized cooperation. This thesis presents a fully embedded swarm communication platform that enables real-world execution of distributed iterative algorithms. Built on the self-balancing Balboa 32U4 robot, it integrates a Raspberry Pi Zero 2 for computation and a Decawave DWM1001 UWB (Ultra-Wideband) module for localization, offering a modular and energy-efficient architecture, capable of cooperative target localization.

A complete communication stack was developed on top of Bluetooth Classic, selected for its low energy consumption and native compatibility with Linux-based systems. Rather than relying on existing routing protocols or middleware, the stack was designed by integrating only the key features relevant to static mesh architectures, avoiding unnecessary overhead to ensure simplicity, efficiency, and full control. To support user interaction with the swarm, a dedicated deployment system was also developed, enabling streamlined and automated experiment configuration and execution. Additionally, the localization pipeline includes calibration and filtering models that significantly enhance measurement accuracy.

Validation through real-world experiments at the communication, localization, and application levels confirms the platform's versatility. Demonstrated applications include multi-consensus algorithms, an asynchronous stand-up behavior, and UWB-based cooperative target localization. These results position the platform as a promising, scalable tool for swarm robotics research.

Nomenclature

Abbreviations

RPi	Raspberry Pi
UWB	Ultra Wide Band
FOTA	Firmware Over the Air
IMU	Inertial Measurement Unit
IR	Infrared (sensor)
UAV	Unmanned Aerial Vehicle
ML	Machine Learning
FL	Federated Learning
OLSR	Optimized Link State Routing
DV	Distance Vector (routing protocol)
LS	Link State (routing algorithm)
DDS	Data Distribution Service
MQTT	Message Queuing Telemetry Transport (middleware)
RTPS	Real-Time Publish-Subscribe (middleware)
QoS	Quality of Service (middleware)
OSI	Open Systems Interconnection
RF	Radio Frequency
RSSI	Received Signal Strength Indicator
ToF	Time of Flight
ToA	Time of Arrival
TDoA	Time Difference of Arrival
FDoA	Frequency Difference of Arrival

RTT	Round Trip Time
NTP	Network Time Protocol
PTP	Precision Time Protocol
PANS	Protocol Adaptation Layer for Network Services
RAM	Random Access Memory
API	Application Programming Interface
TWR	Two-Way Ranging
RTLS	Real Time Localization System
MCU	Microcontroller Unit
PCB	Printed Circuit Board
NLoS	Non-Line-of-Sight
RMS	Root Mean Square
CPU	Central Processing Unit
BATMAN	Better Approach To Mobile Ad-hoc Networking
GATT	Generic Attribute Profile
BLE	Bluetooth Low Energy
TCP/UDP	Transmission Control Protocol / User Datagram Protocol
GPIO	General Purpose Input Output
P2P	Peer-to-peer
ISM	Industrial, Scientific, and Medical (radio band)
FHSS	Frequency-Hopping Spread Spectrum

Definitions

Multi-agents system	A system composed of multiple interacting autonomous agents capable of cooperation or coordination.
Balboa	A self-balancing mobile robot platform developed by Pololu, often used for control experiments.
Raspberry Pi	A low-cost, single-board computer commonly used for embedded systems, robotics, and educational purposes.
UART	Universal Asynchronous Receiver Transmitter; a simple asynchronous full-duplex serial communication protocol used for simple, low-speed (115 kHz) data exchange.

SPI	Serial Peripheral Interface; a master-slave synchronous full-duplex wired communication protocol used for high-speed communication between MCU and a few peripherals. It uses the lines MOSI (master output slave input), MISO, SCLK (clock), CS (chip select).
I²C	Inter-Integrated Circuit; a master-slave half-duplex wired synchronous communication protocol with included addressing, used to connect low-speed devices (max 400kHz). It uses the lines SDA (data) and SCL (clock).
Decawave DWM1001	A UWB (Ultra Wideband) module by Decawave, used for accurate indoor positioning and communication.
Broadcom BCM2835	The system-on-chip (SoC) used in Raspberry Pi models, integrating CPU, GPU, and RAM.
LiDAR	Light Detection and Ranging; a remote sensing method that uses laser light to measure distances for localization, maps reconstruction or SLAM (simultaneous localization and mapping) .
Sonar	Sound Navigation and Ranging; a technique that uses ultrasonic waves propagation to detect objects or measure distance (ToF).
Overlay	1 software layer built on top of an existing communication protocol to extend its functionalities without altering its underlying structure.
Overhead	Extra data and processing required to manage communication, such as headers, acknowledgments, and retransmissions, which do not carry the actual useful payload.
PHY	Physical Layer: Handles the transmission and reception of raw bit streams over a physical medium (e.g., radio, cable).
IP	Internet Protocol: Responsible for addressing and routing packets across networks (L3).
UDP	User Datagram Protocol: A connection-less, lightweight transport protocol with no guarantee of delivery (L4).
TCP	Transmission Control Protocol: A reliable, connection-oriented transport protocol that ensures ordered and error-checked delivery (L4).
TLS	Transport Layer Security: A cryptographic protocol that ensures secure communication over a network (L5).

Baseband	Responsible for the physical link setup, frequency hopping, packet framing, error correction, and timing. It handles the raw transmission of bits over the radio and manages low-level connection control (L1-2)
L2CAP	Sits above the baseband and provides logical channels for data transmission. It enables multiplexing multiple logical connections over a single physical link, handles packet segmentation and reassembly (L2).
RFCOMM	Transport protocol that emulates serial ports over the Bluetooth stack. It provides a reliable communication channel similar to TCP, and is commonly used to implement Bluetooth serial ports with Python (L4).
ZigBee	A low-power wireless mesh full stack protocol based on IEEE 802.15.4, often used in IoT networks.
Nordic nRF52	A family of low-power Bluetooth SoCs (System-on-Chips) designed for wireless applications.
J-Link	A debugging probe by SEGGER, used for flashing and debugging ARM-based MCUs.
Anchor	A fixed node in a localization system that knows its position and helps compute the position of mobile nodes.
Tag	A mobile node in a localization system whose position is to be estimated.
Zephyr RTOS	A small, scalable, open-source Real-Time Operating System designed for resource-constrained embedded systems.
BlueZ	The official Linux Bluetooth protocol stack, used to manage Bluetooth communications.
Sockets	Programming interfaces used to enable communication between devices over a network, using standard protocols.
ATMega 32U4	An 8-bit MCU by Atmel (now Microchip), used in many Arduino-compatible boards.

Contents

Chapter 1

Introduction

1.1 Context

Swarm robotics is a field that aims to design and deploy large groups of autonomous robots capable of cooperating and self-organizing to solve collective tasks. Inspired by natural swarms such as social insects, fish, or birds. These systems rely on simple local interactions that give rise to emergent collective behavior. This decentralized approach enhances robustness, fault tolerance, and flexibility in multi-robot systems. The field emerged in the early 2000's as an application of swarm intelligence, a broader study of self-organized behaviors used in optimization, telecommunications, and crowd simulation. Unlike traditional robotic systems, swarm robotics often requires a complete rethinking of robot design, particularly in perception, control, and localization. Despite its potential, industrial applications remain limited, as most real-world systems still rely on centralized control rather than fully decentralized swarm algorithms.

In this context, researchers in the broader field of multi-agent systems require user-friendly testbeds that act as flexible black-box platforms. To achieve this, the development of a robust communication system is essential to enable new experimental approaches. In addition, a system that provides modular communication modules, which can be assembled to create complex behaviors, would significantly improve flexibility. The more advanced and user-friendly the testbed, the faster researchers can test and validate novel algorithms in a controlled environment before deploying them in real-world scenarios. To navigate and explore their environment effectively, agents must be equipped with a diverse set of sensors that facilitate perception, decision-making, and interaction.

More specifically, a larger research goal led by Gianluca Bianchin is to develop a decentralized multi-agent system composed of multiple satellites and a central ground station that need to be localized. In this system, satellites can communicate directly with only a subset of other agents, forming an incomplete communication graph. Each satellite can measure its distance from the ground station and execute a decentralized self-localization algorithm. Based on that and a distributed optimization algorithm on top of a synchronous-messaging module, agents are able to ultimately estimate the position of the ground station.

This thesis addresses the design and implementation of a communication protocol tailored for a multi-agent system composed of Balboa 32U4 robots equipped with Raspberry Pi modules. The protocol is designed to be both efficient and adaptable, enabling the emergence of complex swarm behaviors without requiring structural modifications as system functionalities evolve. The communication stack will be built entirely upon Bluetooth Classic, encompassing all layers from low-level data transmission to mesh networking and high-level application protocols. To support decentralized navigation and coordination, each robot will be equipped with an onboard localization system capable of estimating its position and measuring relative distances to neighboring agents. These capabilities, when combined with trajectory planning, enable the implementation of spatial self-organization strategies. In order to streamline development, testing, and deployment, a suite of tools will be developed for managing swarm-wide operations such as firmware updates, program execution, and data collection. Finally, several swarm algorithms will be implemented and evaluated, including a distributed target localization system. Experimental demonstrations will highlight the platform's flexibility and its potential for real-time, decentralized multi-agent coordination.

The development of this communication platform raises multiple challenges. Integrating a Raspberry Pi onto a Balboa robot and ensuring effective communication between both components involves significant technical constraints, mainly due to hardware limitations and the master-slave nature of the architecture. In a multi-agent context, decentralization requires the design of a robust protocol capable of reproducing a wide range of natural decentralized behaviors, despite being potentially less efficient than centralized alternatives. Each communication technology presents specific trade-offs in terms of latency, architecture, compatibility, and energy consumption, making its selection a critical design choice. A low-level communication layer has been adopted as a foundation, with substantial effort dedicated to developing overlays that address the system's functional requirements. The software architecture must remain modular and easily deployable to facilitate integration and adaptation across diverse experimental scenarios. Accurate decentralized localization is another core requirement, yet achieving high precision under constrained computational resources remains particularly demanding. An onboard UWB module has been selected to meet most of the technical criteria, but its integration introduces further complexity due to the limited resources of the Balboa 32U4, necessitating additional post-processing to improve measurement quality. Maintaining the real-time balancing performance of the Balboa robot is also a key concern, as the addition of a shielded Raspberry Pi and the localization module alters the robot's center of gravity and inertia, affecting its dynamic stability.

1.2 Literature review

This section presents an overview of the current state of the art relevant to this work. It begins with a discussion of foundational concepts and developments in the field of multi-agent systems, with particular attention to existing swarm robotics platforms. The review then examines communication strategies within swarms, with a focus on synchronous mechanisms and routing algorithms, which form the basis for the protocols developed in this thesis. Finally, the section explores decentralized localization approaches applicable to resource-constrained mobile agents, providing context for the integration of onboard positioning systems.

1.2.1 Background on multi-agent systems

Cao et al. (1995) [1] provides a foundational analysis of cooperative mobile robotics, identifying key behaviors such as traffic control (multiple agents move within a common environment), box pushing (collaborating environment modification), and foraging (insect-inspired robots grabbing objects) as representative applications. It distinguishes centralized architectures, where a single entity manages the swarm, from decentralized ones, where agents make distributed decisions, often leading to scalable and robust emergent behaviors. The authors also emphasize the role of differentiation: while most systems are homogeneous for simplicity, heterogeneous systems enable more complex task distributions at the cost of increased coordination complexity. Communication structures are categorized into three types: environment-mediated interaction, sensing-based reaction, and explicit message communication, each introducing specific trade-offs in terms of robustness and complexity.

In addition to architectural considerations, the study highlights the importance of agent modeling, the ability of robots to predict or infer the states and actions of others, as a way to improve cooperation while reducing communication overhead. Resource conflicts, arising from competition for shared space or goals, require appropriate coordination mechanisms. The origins of cooperation are discussed in two main paradigms: eusocial behaviors, where simple agents collectively generate intelligent group behaviors, and cooperative strategies where agents actively collaborate to maximize individual utility. Multi-robot path planning, particularly the generation of non-intersecting trajectories and traffic control, remains a significant algorithmic challenge. Finally, due to technological constraints such as perception noise and hardware reliability and accuracy limitations, much of the early research in swarm robotics has been conducted in simulation environments.

Schranz et al. (2020) [2] is summarizing the swarm behaviors and existing systems for swarm research purpose. It details that swarm behaviors can be classified into four categories, each on them is detailed into several subcategories, as summarized in Table 1.1. Several other classification works have been proposed, such as Dudek et al. (2002) [3], based on communication, and Farinelli et al. (2004) [4], which focuses on coordination.

[2] also discusses industrial applications, which remain constrained by communication limitations, and the risk of real-world deployment. Most industrial systems that claim to use swarm robotics still operate under centralized control. The purpose of this work is to decentralize communication. Obviously, decentralized systems are not the most efficient. But it will enable us to study more real-world swarms as they are the most common, as detailed in the context.

Category	**Behaviors**
Spatial Organization	Aggregation
	Pattern Formation
	Self-Assembly
	Object Clustering and Assembly
Navigation	Collective Exploration
	Coordinated Motion
	Collective Transport
	Collective Localization
Decision Making	**Consensus**
	Task Allocation
	Collective Fault Detection
	Collective Perception
	Synchronization
	Group Size Regulation
Miscellaneous	Self-Healing
	Self-Reproduction
	Human-Swarm Interaction

Table 1.1: Classification of swarm behaviors. Highlighted behaviors are developed in this thesis: Collective localization – shared frame and data exchange to localize a target. Consensus – reaching a common decision. Collective perception – merging local data into a global view. Synchronization – aligning oscillator phases and frequencies.

The platform developed in this thesis provides a foundation for exploring advanced research topics in the field of multi-agent systems. As summarized by Dorigo et al. (2021) [5], current swarm robotics research is evolving along several key axes. These include hardware miniaturization, enabling the deployment of micro-robots in confined or hard-to-reach environments; heterogeneity, where agents possess different capabilities, roles, and behaviors; and hybrid control architectures, which aim to combine the robustness and scalability of decentralized systems with the global awareness of centralized ones. Additionally, machine learning, particularly through the use of neural networks, is increasingly employed to enhance swarm adaptability, moving beyond traditional evolutionary algorithms. Security also constitutes a growing concern, with research focused on safeguarding swarms against external attacks and ensuring safe task execution. Finally, human-swarm interaction (HSI) is an emerging area, investigating new forms of communication between humans and robot collectives through gestures, EEG signals, or augmented reality. Looking ahead, swarm systems hold promise across a wide range of applications, including space exploration through micro-rover swarms, surveillance of sensitive areas using coordinated drones, agricultural diagnostics for early disease detection, infrastructure monitoring and maintenance, and targeted drug delivery in medical contexts [5].

An interesting concept related to swarm theory is the stigmergy, Dorigo et al. (2021) [5] introduced it. It involves collaborating agents within a swarm, where each agent leaves a trace in the environment that stimulates other agents. It has been demonstrated that such robots are efficient for several tasks, including foraging, clustering (simulating the natural grouping of individuals or objects), and even biological dynamic replication, where robots simulate the behaviors of natural organisms such as aggregation of insects. One major work utilizing this concept is Swarm-bots, a multi-agent system capable of self-assembly to form cooperative structures. Swarmanoid extends Swarm-bots by incorporating heterogeneous agents, including flying, climbing, and ground-based robots.

1.2.2 Platforms for multi-agent systems

Several research platforms have been developed to study previously mentioned swarm behaviors and multi-agent topics. This section introduce the most recognized platforms as well as those related to this project.

Among terrestrial robots, Kilobots, developed by M. Rubenstein and R. Nagpal (2012) [6], is one of the most well-known. They were designed to validate collective algorithms like SDASH on very large swarms of up to 1024 robots. Each Kilobot measure about 3cm and uses a low-cost design based on an ATMega328 MCU, costing only $14 in parts and requiring five minutes to assemble, with an autonomy ranging from 3 to 24 hours. Communication is decentralized, using infrared signals reflected off the ground with a wide 60° emission angle, achieving data rates up to 30 kb/s at about 10 cm range (approximately six robot radii). A CSMA/CA mechanism is implemented to mitigate signal collisions within dense swarms. Kilobots feature a scalable deployment system allowing collective programming, powering on/off, and control through infrared flooding. For instance, reprogramming 25 robots can be achieved in 35 seconds. Locomotion relies on vibration motors, enabling movement at about 1 cm/s and rotation at 45°/s. However, this method prevents accurate odometry. Instead, relative localization is performed by estimating distances to neighbors through infrared signal intensity.

The Khepera IV robot, designed by K-Team and released in January 2015. It is designed for any indoor lab application, and was thoroughly reviewed by Soares et al. [7]. It features a Gumstix onboard computer coupled with a GS608 MCU and an extension bus for modular expansions. Its sensor suite includes 12 infrared sensors (effective from 4 to 12 cm), 5 ultrasonic sensors (covering 20 to 300 cm with a 92° field of view), an IMU with accelerometers and gyroscopes, two microphones, and a wide-angle color camera equipped with a distortion model. Actuation is provided by two DC motors for differential drive locomotion. The robot comes with the *libkhepera* software library, while the authors also developed the *KheperaToolbox* for enhanced data handling. It includes a module for parsing NMEA-formatted messages, a *Measurement* module for periodic data acquisition supporting various data sources, an *OdometryTrack* module for tracking motion, and an *OdometryGoto* module enabling waypoint-based navigation. Although Khepera IV robots offer strong onboard computation and sensing, they lack a native system for simultaneous mass programming. Deployment is done individually via SSH over Wi-Fi. Swarms typically involve 5 to 20 robots. While basic behaviors like way-point navigation and sensing are supported, complex swarm behaviors must be implemented by the user.

The e-Puck robot, originally developed for educational purposes in signal processing, automatic control, embedded programming, and distributed intelligent systems design, was commercialized in 2005 and presented by Mondada et al. [8] in 2009. Its successor, the e-Puck 2, available since 2018 at approximately €1250, features an STM32F4 MCU, eight infrared sensors, a time-of-flight (ToF) distance sensor, an IMU, a color camera, four microphones, and an additional infrared sensor for distance control. It offers around three hours of autonomy. The robot's user interface includes an in-circuit debugger connector, an infrared receiver for remote control, a classic RS232 serial interface, and a Bluetooth radio link enabling wireless programming and communication with a desktop computer via the BTcom protocol. Extension headers allow the addition of peripherals such as a rotating scanner with infrared triangulation (up to 40 cm range), a turret with three linear cameras for wide-field optical flow, and a Zigbee module for adjustable mesh communication. The software package provides a bootloader for Bluetooth-based programming, a low-level hardware control library, and a monitor application for interaction with a desktop computer.

Pickem et al. (2015) [9] present the GRITSBots, an inexpensive ($45) micro-robot designed for multi-robot research, offering between 30 minutes and five hours of autonomy. The robots can achieve linear speeds of up to 25 cm/s and rotational velocities up to 820 degrees per second. GRITSBots feature a wide range of capabilities, including a deployment platform, wireless programming, and collective control. Of particular interest are the automatic sensor calibration system using a dedicated machine, autonomous charging behavior where robots can navigate to the charging station. The robots have demonstrated behaviors such as rendezvous, formation control, and vehicle routing, tasks that require accurate local sensing and reliable locomotion. Although physically centralized, it is able to emulate decentralized behaviors such as the consensus. The use of an overhead camera for global localization limits the maximum size of the environment. Unlike other robots, GRITSBots do not use wheel encoders or classical odometry, instead, they rely on miniature stepper motors, with complex signal processing required to estimate motor velocities. Six infrared transmitters and receivers arranged every 60 degrees around the body, combined with an accelerometer and a gyroscope, allow basic slip detection and improve velocity and position estimation. An onboard RF transceiver provides efficient wireless communication at 2 Mbits/s, with significantly lower power consumption (16 mA) compared to Wi-Fi (250 mA), although local peer-to-peer communication via IR has not yet been implemented. For processing, GRITSBots use an ATmega168 microcontroller for low-level hardware control and an ATmega328 for higher-level tasks such as communication, path planning, and collective algorithms. Demonstrated capabilities include obstacle avoidance, autonomous charging, and collective consensus, illustrating the platform's suitability for multi-robot systems research.

The Robotarium is a remotely accessible swarm robotics testbed composed of GRITSBots, previously introduced, designed to allow users to quickly prototype and validate distributed control strategies on physical robots without the need to set up their own hardware. Wilson et al. published (2020) [10] and (2021) [11]. They present the system as a platform supporting research in control algorithms, path planning, task allocation, and behavior composition. The Robotarium offers a user-friendly interface. A simulation API, available in both MATLAB and Python, enables users to rapidly prototype distributed control algorithms and test their feasibility before deployment. Individuals access the Robotarium

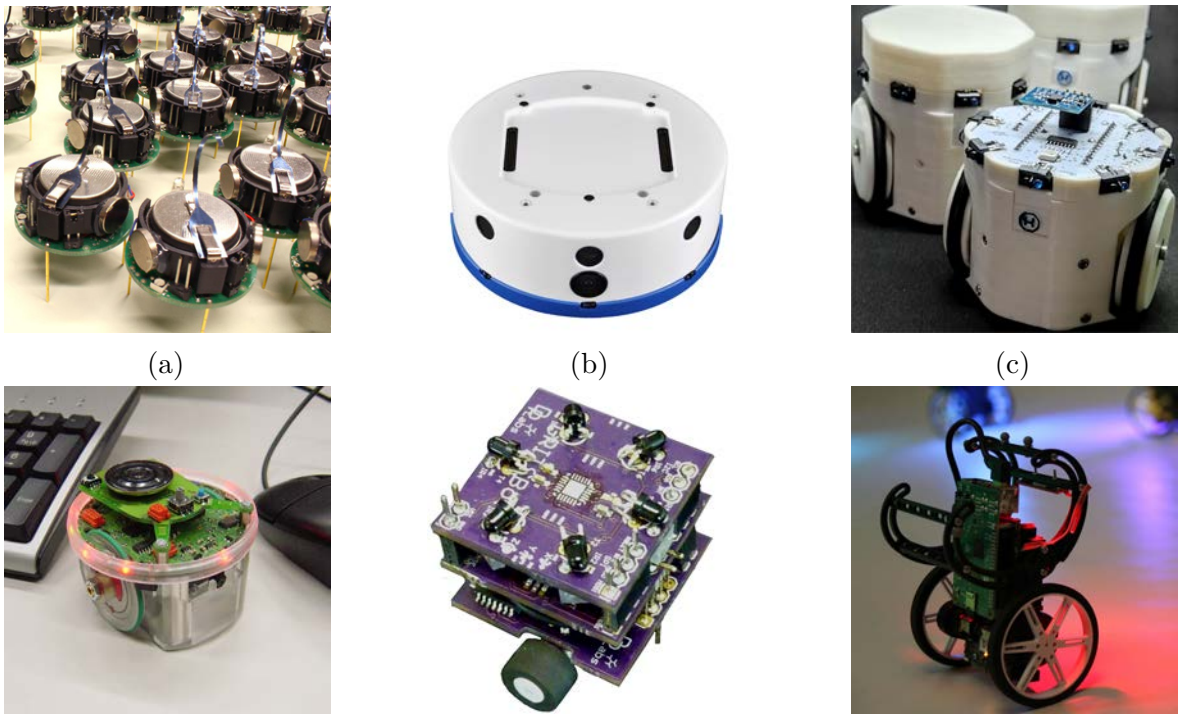
through a public web interface, following a workflow that moves from theory development to simulation, then script submission, code validation, physical deployment, and finally data and video feedback. Eight Vicon motion-capture cameras are mounted above the testbed to track each robot's position and orientation, using unique, non-symmetrical patterns of Vicon markers. This tracking system not only enables precise localization but also helps detect potentially harmful situations during experiments. Additionally, an ELP camera placed above the center of the testbed provides automatic video capture of experiments, while an Optoma EH200ST projector can project dynamic, time-varying environmental backgrounds onto the arena during user experiments.

Pickem et al. (2017) [12] presents the Robotarium for the first time. It emphasize the need for a safe and flexible swarm robotics platform, detailing the obstacle avoidance strategies and validating the system through real-world user experiments. A key aspect of the Robotarium architecture is its centralized control: while each robot runs a local velocity controller, the user-submitted code executes on a central server, which remotely provides velocity commands. This approach enhances robustness, simplifies data logging, enables formal safety guarantees, and facilitates automatic maintenance.

Rezeck et al. (2023) [13] presents Hero 2.0, a robot designed for swarm robotics research. It is built around an ESP8266 MCU, equipped with eight infrared sensors for obstacle detection, differential drive wheels reaching speeds of 25 cm/s, wheel encoders for odometry, an IMU to enhance motion estimation, a camera, and onboard displays. Hero 2.0 communicates using TCP/IP protocols and integrates with ROS for higher-level communication. Regarding localization, Hero 2.0 uses gray-coded patterns placed on the floor, allowing the robots to determine their position and orientation by decoding the patterns with onboard photodiodes. This system offers zero latency compared to traditional camera-based tracking systems and physical decentralization. However, for the same resolution, it costs approximately 700 USD. Programming and interaction rely on a FOTA system managed through ROS, operating in two distinct modes. Configuration mode: robots act as access points (AP mode) while the laptop acts as a client, allowing firmware updates and setup. Communication mode: a central laptop operates as an access point, with robots connecting as clients. Although physically centralized, this network topology simulates decentralized communication and enables real-time user interaction with the swarm. The behaviors demonstrated by Hero 2.0 cover several classical challenges in swarm robotics. Flocking behavior: Robots coordinate their motion to maintain a cohesive group, while avoiding collisions and aligning their velocities. Mapping task: The robots collaboratively explore an environment to build a map, typically by sharing local observations. Decentralized coverage: Robots autonomously spread out across a given area to maximize coverage without overlaps. Transportation tasks: Multiple robots coordinate to move an object that would be too large or heavy for a single agent. Lastly, [13] mentions the future addition of UWB sensors for indoor absolute localization as an improvement, which will be implemented in this project.

Barcis et al. (2019, 2020) [14, 15] proposed a unified mathematical model for synchronization and swarming, where each swarmalator coordinates both its internal phase and spatial position. While their model lies beyond the scope of this project, their choice of the Balboa 32U4 robot is relevant. Unlike limited platforms such as Kilobots, the Balboa, paired with a Raspberry Pi 3B+, offers sufficient computational power to run

complex tasks and ROS, with accurate state estimation via its onboard IMU and motor encoders. In this project, we use the Raspberry Pi Zero 2. Their implementation also highlights the visual appeal of the Balboa's self-balancing behavior in swarm scenarios. They modified the vendor-provided low-level controller (LLC), replacing the USB interface with a UART link to allow remote MCU reprogramming and custom protocols. In contrast, this project uses I²C to interface the Raspberry Pi and the LLC, freeing the USB port for debugging but requiring significant architectural changes and controller retuning. Barcis et al. employed ROS 2 and eProsima Fast RTPS middleware, with multi-hop communication via the Babel routing protocol. This thesis opts for a custom lightweight middleware and communication stack, prioritizing flexibility and low power consumption without ROS. Their indoor experiments relied on centralized OptiTrack-based localization, while a fully decentralized onboard UWB modules is adopted in this thesis. To streamline deployment, they used an Ansible-based system for remote program management. Here, a simpler, custom deployment solution will be developed for easier customization and use.



(a) (b) (c)

(d) (e) (f)

Figure 1.1: (a) Kilobots [16], (b) Khepera IV [7], (c) Hero 2.0 [13], (d) E-puck 2 [8] (e) GRITSBots [9], (f) Sandsbots [15].

Table 1.2: Summary of existing swarm robotic agents.

Robot	**Architecture**	**Communication**	**Computer**	**Sensors**	**Deployment**
Kilobot [6] (2012)	Decentralized	IR (7cm)	Atmega328	Proximity, light	IR flooding FOTA
Khepera IV [7] (2015)	Both	Bluetooth	Gumstix and GS608	IR, ToF, mics, camera, IMU, odometers	Individual SSH FOTA
E-puck 2 [8] (2018)	Both	Wi-Fi, Bluetooth	STM32	Light, camera, ToF, mic, odometers, extensions (LiDAR, turret, Zigbee)	IR flooding FOTA
GRITSBots [9] (2015)	Centralized	RF	ATmega168 / 328	Optitrack, stepper motors, IR, IMU	Scalable FOTA
Sandsbots (Balboa) [15, 14] (2020)	Decentralized com., centralized loc.	Babel (WiFi Ad-Hoc) + DDS	RPi 3b+, ATMega32U4	IMU, magnetometer, encoders, and optitrack	Custom Ansible-based FOTA
Hero 2.0 [13] (2023)	Decentralized	Wi-Fi and IR	ESP8266	Odometry, IR, IMU, camera, displays, photodiodes for global loc.	ROS based FOTA
Balboa	Decentralized	Bluetooth based custom stack	RPi Zero 2, ATMega32U4	IMU, magnetometer, encoders, UWB, displays	SSH scalable FOTA

1.2.3 Mesh Wireless Communication

Mesh wireless networks enable distributed communication among nodes without the need for centralized infrastructure. These networks can adopt various topological models, each offering specific advantages depending on the application. In a 1:1 (point-to-point) configuration, two devices communicate exclusively, offering simplicity and robustness but limited scalability in multi-agent contexts. The 1:m (one-to-many) model allows a single node to broadcast data to multiple recipients, which is useful for command distribution. Conversely, the m:1 (many-to-one) model supports data aggregation from multiple nodes to a central unit, facilitating centralized control or logging. Finally, the m:m (many-to-many) architecture enables each node to communicate with several others, forming a flexible mesh structure well-suited for decentralized coordination in swarm systems.

For swarm communication, a m:m communication model is essential. There are two categories of multi-hop networks. The first are mobile ad-hoc networks (MANET) which are composed out of dynamic, fast changing topologies. The second are static network topologies like mesh networks. In the context of this project, static mesh topologies are preferred, and there is no requirement for dynamic routing. The 7-layer OSI model serves as a reference framework to structure communication protocols:

1. **Physical Layer**: responsible for the physical transmission of raw bits over a communication medium. It involves hardware elements like cables, switches, and connectors, as well as signal transmission (electromagnetic, light, or radio signals).

2. **Data Link Layer**: ensures reliable data transfer between devices on the same network. It handles error detection, framing of data packets, and provides addressing (such as MAC addresses). Technologies like Ethernet and WiFi work at this layer.

3. **Network Layer**: responsible for routing data between different networks. It handles logical addressing (like IP) and select the optimal path for data transmission.

4. **Transport Layer**: The transport layer provides end-to-end communication between devices. It ensures reliable data transfer, error correction, and flow control. TCP and UDP are the main protocols in this layer.

5. **Session Layer**: The session layer manages sessions between communication devices. It ensures that the session is established, maintained, and terminated correctly, providing synchronization and dialogue management during communication.

6. **Presentation Layer**: This layer is responsible for translating data between the application layer and transport layer. It can handle data encryption, compression, and transformation to ensure that data is in a format the application can process.

7. **Application Layer**: The application layer provides the interface and protocols that allow users and software applications to interact with the network. This layer includes protocols like HTTP, FTP, and DNS.

Table 1.3 summarizes the OSI model and some examples related to mesh communication technologies discussed in the next section, including this project.

Table 1.3: Examples of protocol stacks mapped to OSI layers

OSI Layer	**BATMAN, OLSR, Babel**	**DDS**	**MQTT**	**ZigBee**	**BLE Mesh**	**Balboa**
7. Application	—	DDS	MQTT	ZigBee App Profile	Mesh Model	Balware framework
6. Presentation	—	Type serialization	—	—	Foundation Model	Balware type serialization
5. Session	—	DDS session	TLS session	—	Access Layer	Balware session
4. Transport	—	UDP	TCP	—	Transport Layer	Balware overlays
3. Network	OLSR / Babel	—	—	ZigBee NWK	Network Layer	Balmesh
2. Data Link	BATMAN / IEEE 802.11	—	—	IEEE 802.15.4	BLE (L2CAP)	RFCOMM / L2CAP
1. Physical	IEEE 802.11	—	—	ZigBee PHY	BLE PHY	Bluetooth PHY

Mesh routing protocols:

Routing protocols at the network layer are responsible for dynamically establishing multi-hop paths between nodes in a decentralized manner and can be broadly classified into four main categories [17]. **Proactive** (table-driven) protocols, such as OLSR and Babel, maintain up-to-date routing information to all nodes at all times, reducing latency but potentially adding unnecessary overhead in dynamic or sparse networks. **Reactive** (on-demand) protocols like AODV and DSR create routes only when needed, minimizing idle overhead but introducing initial delays. **Hybrid protocols**, exemplified by Babel, combine both approaches by using proactive routing within local zones and reactive routing between them. Finally, **hierarchical** (cluster-based) protocols such as CBRP and ZRP divide the network into zones or clusters, improving scalability and routing efficiency through a multi-level decision structure. Here are some of the existing routing protocols and their classification:

- **ZigBee** is a full stack, developed in the early 2000's for low-power personal area networks. Its routing protocols is hierarchical, including a central coordinator, and based on AODV, a reactive routing protocol. Ramya et al. (2011) [18] is a study of this technology. Based on IEEE 802.15.4 for the physical and data link layers, it adds a network layer with its own mesh routing protocol and an application layer with profiles for home automation, metering, etc. ZigBee is popular in industrial and home IoT. It operates without IP, favoring low-energy, short-range communication with high reliability in dense mesh topologies. Its data rate is limited to 250 kbps and it requires dedicated hardware, but it has low latency and low power consumption.

- **B.A.T.M.A.N (Better Approach To Mobile Ad-hoc Networking)** is used and presented for the Starling1 project in Sanchez et al. (2021) [17] and in Davoli et al. (2019) [19]. Performances are analyzed in Liu et al. (2018) [20] and in Sailash et al. (2015) [21]. It is a decentralized proactive routing protocol introduced in the mid-2000's to replace OLSR with a lower latency and overhead and addressing the loop issue due to asymmetrical links. It runs over layer 2, in contrast with most other routing protocols that run on layer 3, and uses a periodic flooding - broadcasting packets known as OriGinator messages (OGM), corresponding to a 12 byte UDP payload (for a total packet size equal to 52 bytes, including IP and UDP headers) - to announce node presence for dynamic routing and link quality evaluation through the Transmission Quality (TQ) path metric. It uses Dijkstra algorithm to find shortest routes based on the weights of the links based on the TQ. BATMAN is highly adaptive and well-suited for mobile and unstable environments. It is not designed for fixed mesh topology.

- **Babel** is a modern hybrid distance-vector (DV), link-state (LS) routing protocol introduced in 2011. Chroboczek, J. (2011) [22] describes it thoroughly. Babel is designed for unstable networks by minimizing routing issues like loops and black-holes during reconvergence. After a mobility event, it usually stays loop-free and quickly restores connectivity, even if not optimally. This often requires no packet exchange. It then gradually converges to an optimal state using sequenced routes, inspired by DSDV.

- **Bluetooth Low Energy Mesh (BLE Mesh)**, standardized in 2017, is a complete communication stack tailored for low-power IoT devices. It offers limited bandwidth but ensures good scalability for small payloads, making it suitable for sensor networks. The protocol uses a decentralized model with managed flooding, TTL-based relaying, and message caching. Baert et al. (2018) [23] provide an overview and performance assessment, reporting an average round-trip time of approximately 25 ms per hop. BLE operates over 40 channels, with three dedicated to advertising and scanning, and 37 used in connected mode for data exchange. The mesh architecture includes provisioners (e.g., Raspberry Pi) that configure nodes, relays that selectively forward messages, friends that buffer data for low-power nodes, proxies for BLE compatibility, and standard nodes as participants in the network.

These routing technologies are summarized in Table 1.4.

Table 1.4: Comparison of routing protocols

Protocol	**Routing type**	**Latency (RTT)**	**Consumption**	**Hardware**	**Data rate**
BATMAN	Proactive (DV)	$\sim$ 20 ms (1 hop), 3-4 s (3-4 hops) @ 64 bytes [24]	Moderate	RPi	$\sim$ 45, 10 Mbps @ 2, 8 nodes [25]
OLSR	Proactive (LS)	$\sim$ 5-25 ms (1 hop), 2-3 s (3-4 hops) @ 64 bytes [24]	High	RPi	50, 6 Mbps @ 2, 8 nodes [25]
ZigBee	Hierarchical (DV)	$\sim$ 20 ms/hop @ 50 bytes [26]	Very Low	RF module	250 kbps @ 1 hop [18]
Babel	Hybrid (DV + LS)	$\sim$ 10-120 ms	Moderate	RPi	57, 7 Mbps @ 2, 8 nodes [25]
BLE Mesh	Managed Flooding	$\sim$ 25 ms / hop [23]	Low	nRF52	2 Mbps [23]
Balmesh	Predefined	$\sim$ 60 ms / hop @ 50 bytes	Low	RPi	25 Kbps @ 1 hop

Middlewares:

Middleware solutions operate above the transport layer and simplify the development of distributed applications by abstracting message exchange and synchronization.

- **DDS (Data Distribution Service)**, implemented by vendors like eProsima (Fast DDS), is a decentralized publish-subscribe middleware spanning OSI layers 5 to 7 [27, 28]. It uses the RTPS protocol over UDP to enable reliable, deterministic communication in IP networks. Nodes publish data on topics that others subscribe to. DDS offers advanced features such as Quality of Service (QoS) policies, automatic discovery, strongly typed messages via IDL, and topic persistence. QoS controls delivery parameters like reliability, latency, and durability, allowing late-joining subscribers to receive recent data. Since the 2010s, DDS has been widely adopted in real-time robotics, avionics, automotive systems, and platforms like ROS 2.

- **MQTT (Message Queuing Telemetry Transport)** is a lightweight application-layer protocol introduced in 1999 and widely adopted in IoT. Its architecture and performance are discussed in Thangavel et al. (2014) [29]. MQTT relies on a centralized broker model over TCP/IP and mainly operates at OSI layers 5 to 7. Unlike DDS, it is not peer-to-peer and requires an always-available broker to route messages between publishers and subscribers. While simpler and more energy-efficient than DDS, its reliability is ensured through three Quality of Service (QoS) levels. QoS 0 delivers messages at most once with no acknowledgment, QoS 1 ensures delivery at least once with confirmation, and QoS 2 guarantees exactly-once delivery using a four-step handshake. MQTT is widely used in smart home systems, telemetry, and mobile applications due to its minimal overhead and ease of implementation.

Table 1.5: Comparison of middleware solutions

Middleware	**Base layer**	**Functionalities**	**Architecture**	**Power consumption**
eProsima DDS	RTPS (5–7 OSI)	Pub-sub, discovery, QoS policies, asynchronous + synchronous messaging	Decentralized (but physically, depends on the routing)	High
MQTT	TCP (4–7 OSI)	Lightweight pub-sub, broker-based, event-driven, QoS levels	Centralized	Low
ZigBee stack	Full-stack	Routing, addressing, security, application objects	Decentralized (but coordinator)	Very low
Balware	Balmesh (4-7 OSI)	Flooding, multi-hop unicast, synchronous iterative algorithms, asynchronous periodic algorithms	Decentralized (but laptop configuration)	Very low

These middlewares are summarized in the Table 1.5. In summary, technologies like BLE Mesh and ZigBee include their own routing protocols and do not rely on IP, which makes them suitable for constrained devices but harder to integrate with IP-based middleware such as DDS. BATMAN, OLSR, and Babel provide flexible IP routing and can serve as the foundation for middleware like DDS or MQTT. Babel, in particular, offers a good compromise between adaptability and efficiency in dynamic wireless environments. Middleware protocols enable data exchange and algorithmic synchronization across nodes but differ greatly in architecture, functionalities, and suitability depending on the protocol sub-layers.

1.2.4 Focus on middlewares and routing technologies considered

The previous section shown different routing protocols and middlewares. It concludes that the custom protocol designed in this project incorporates specific middleware features, such as synchronous communication, and routing mechanisms like flooding and multi-hop unicast, this section provides a focus on the state of the art of these mechanisms. These components will be implemented at the middle layer of the architecture.

Routing mechanisms

Flooding is a fundamental routing technique in which a message is broadcast to all neighboring nodes, which in turn rebroadcast it to their own neighbors, continuing recursively until the message has propagated throughout the entire network. This approach is robust due to its inherent redundancy, ensuring high reliability even in the presence of node failures or unreliable links. However, this redundancy also introduces significant overhead, including unnecessary transmissions, packet collisions, and increased energy consumption, a phenomenon commonly referred to as the "broadcast storm problem."

To mitigate these drawbacks, numerous studies have proposed optimized flooding algorithms designed to reduce the number of transmissions while preserving full network coverage and reliability. Most of these techniques involve selecting a limited subset of nodes, known as relay nodes, to forward the messages. Determining the optimal set of relay nodes is an NP-hard problem. Therefore, various heuristics have been introduced to approximate optimal solutions. One notable example is the Multi-point Relaying (MPR) technique, which uses two-hop neighborhood information to elect neighbor nodes—typically those with high connectivity—as relays. This method is implemented in Bluetooth Mesh (BLE Mesh) networks [23]. Another approach is the tree-based flooding protocol proposed by Frank et al. (2008) [30], which constructs a lightweight, rooted tree spanning the entire network using a Breadth-First Search (BFS) algorithm. By restricting message forwarding to this structure, the protocol achieves efficient dissemination with minimal overhead. However, these methods fall outside the scope of this thesis.

There exists also multicast, a communication method where a message is delivered from one sender to multiple specific receivers simultaneously. In wireless sensor networks, multicast routing protocols aim to efficiently deliver messages to a group of nodes without flooding the entire network, optimizing bandwidth and energy usage. This mechanism will not be implemented in this thesis.

Multi-hop unicast is a routing strategy where a message is sent from a source to a specific destination through a sequence of intermediate nodes. Each node forwards the message to the next hop based on a routing table or predefined path. This approach is more efficient than flooding but requires maintaining routing information.

Synchronous messaging

Synchronous messaging is a communication mechanism in which nodes in a network send and receive messages simultaneously, often relying on precise time synchronization. This mechanism is particularly useful in wireless sensor networks for coordinated actions. For instance, it is included in eProsima RTPS [27]. Basic synchronization can be achieved either using hardware clocks or using clock-independent overhead. However, clock-synchronization methods are non-optimal because they are limited by clock drift and the needed time margin. Logical synchronization protocols, by contrast, use minimal timing by aligning nodes communication schedules through well thought message overhead, but this overhead increase a bit the power consumption.

The most efficient synchronous messaging algorithm ever designed is Glossy. It is presented in Ferrari et al. (2011) [31]. It is a low-latency flooding and network-wide time synchronization mechanism for wireless sensor networks. It operates at physical layer on specific radio hardware. It enables constructive interference by carefully timed transmissions, allowing all nodes to receive and forward packets nearly simultaneously. Glossy achieves high reliability and microsecond-level synchronization.

Chaos, builds on top of Glossy, presented in Landsiedel et al. (2013) [32], and introduces a data-centric approach. Glossy is used for the physical layer synchronous messaging while Chaos adds a user-defined merge operator, which allows users to freely program various merge operators, from simple aggregates to complex computations taking tens of thousands of clock cycles to execute This has very low latency and energy consumption for mesh communication. LWB (Low-power Wireless Bus), designed in 2012, is presented in Mager et al. (2019) [33]. It is another communication protocol built on top of Glossy. It provides a shared-bus abstraction, and hides the complexity of the underlying networks.

In Lim et al. (2013) [34], FlockLab is a hardware testbed that supports synchronized communication experiments in sensor networks. It integrates tightly with tools like Glossy and Chaos and allows for real-time monitoring and debugging of distributed protocols.

1.2.5 Localization of a swarm agent

In this section, we focus on non-collaborative localization strategies for individual swarm agents. We exclude cooperative schemes (where agents share and fuse each other's estimates). Localization technologies can be classified into architecture, environment and sensor type. For this project we will prefer decentralized absolute ($\Rightarrow$ exteroceptive) technologies in non-controlled environments. Even though proprioceptive sensors yield only relative localization (drift-prone odometry, IMU), they remain essential for later sensor fusion. For instance, Xu et al. (2022) [35] present a visual-inertial UWB state estimation system for aerial swarms and achieve sub-centimeter accuracy by combining two exteroceptive sensors with a single proprioceptive sensor.

- **Centralized (external infrastructure):** motion-capture systems such as OptiTrack.

- **Decentralized (on-board only):** the agent relies solely on its own sensors, further distinguished by:

  - **Controlled / non-controlled environment:** the ground could be patterned like a QR code such as a photodiode can detect its absolute position,

  - **Proprioceptive / exteroceptive sensing**: a sensor can measure its own properties such as speed (odometry) or acceleration (IMU) for instance.

  - **Absolute localization** (agent $\leftrightarrow$ environment) / **relative localization** (agent $\leftrightarrow$ agent). Note that absolute localization $\Rightarrow$ exteroceptive sensor and relative localization $\not\Rightarrow$ proprioceptive sensor (e.g., all exteroceptive between 2 agents give relative localization)

Existing localization technologies, the multi-agent platforms that employ them, and their classification are summarized in Table 1.6. Note that it is difficult to represent the classification on a 2D table, then the classes are ordered according to their importance: decentralization is more important than non-controlled environment.

Table 1.6: Classification of localization technologies associated with existing robot swarms.

Centralized	**Decentralized**		
	Controlled	**Non-controlled**	
		Proprioceptive	**Exteroceptive**
OptiTrack [15]	Ground patterns [13], GPS [15], camera	IMU [13, 7, 8], odometers [13, 7, 8], stepper motor [9]	Sonar (ToF) [8, 7], LiDAR, IR [6], RF (**This project**)

On-board exteroceptive ranging primitives

Ranging primitives refer to the conversion of signals into distance estimates, which are subsequently used for localization. Below are some of the most common technologies.

- **Received Signal Strength Indicator (RSSI):** This basic technique estimates distance by correlating received signal power with distance. However, it is highly sensitive to multi-path effects, which significantly limits its accuracy, typically to within a few meters. Sadowski et al. (2018) [36] investigated localization using RSSI-based range measurements with WiFi, BLE 4.0, ZigBee, and LoRaWAN. Their results indicate that the signal attenuation follows a logarithmic model: while tens of centimeters accuracy can be achieved at distances below 1 m, the error increases to several meters at greater ranges. Cao et al. (2021) [37] presents a BLE Mesh network-which is capable of exploiting the space, time, and frequency diversities in measurements- that is also used for RSSI-based localization.

- **Time-of-Flight (ToF)**: Two-Way Ranging (TWR) is a kind of ToF measurement. It estimates distance based on the total round-trip time (RTT) of a packet between two devices:

$$d = c \cdot (\texttt{RTT} - \texttt{Reply Time}) \tag{1.1}$$

  Where d is the distance, c is the speed of light, and Reply Time is the processing delay. ToF provides better accuracy than RSSI but typically requires additional hardware.

- **Time-of-Arrival (ToA)**: It is one of the most accurate technique available. It measures the absolute time at which a packet is received. Unlike ToF, ToA eliminates the uncertainty of the reply delay. However, it requires clock synchronization between devices with sub-microsecond precision. Clock synchronization is non-trivial and may involve protocols like NTP, PTP, or other synchronization strategies such as [38].

- **Angle-of-Arrival (AoA)**: Another advanced technology, it estimates the direction of an incoming signal. This requires a specific receiver with an array of antennas and complex signal processing. Girolami (2023) [39] presents this method thoroughly and show the specific hardware required.

- **Frequency-Difference-of-Arrival (FDoA)**: This Doppler effect based technique estimates range and relative motion by measuring Doppler shifts between received signals from moving sources. It's more commonly used in radar or satellite tracking, and is not suitable for short-range indoor localization in robotics due to hardware and processing complexity.

- **Vision (e.g., AprilTags)**: Computer vision techniques can be used to detect AprilTags, placed in the controlled environment, and deduce the robot position based on the tags locations. Pandey et al. developed such system for UAV, it is presented and validated in (2024) [40]. They got an accuracy of 60-90%. It highlights that AprilTags outperform LiDAR and Sonar in terms of setup simplicity, scalability, and cost-effectiveness. However, it is less efficient where environmental conditions vary widely or where tags could be occluded. It suggests to combine it with a LiDAR or a sonar for a more versatile and accurate solution.

Localization techniques

Absolute localization can be achieved based on range measurements, between a mobile agent and a set of fixed anchors. Below are some of the most common techniques.

- **Multilateration (ToF, RSSI):** Measure the distance d_i from the agent to each anchor i. In an N-dimensional space we need at least $N+1$ anchors (e.g. 3 anchors in 2D, 4 in 3D). Geometrically, each measurement defines a circle (2D) or a sphere (3D) centered at the anchor. the agent's position is found at their common intersection, which is often found using least squares. Cannizzaro et al. (2019) [41] explains multilateration in detail and compares the 2D localization accuracy with 3 and 4 beacons using BLE, showing that the difference is not significant, and that, using 4 beacons may even worsen the the accuracy according to the environment.

- **AoA & Range:** An AoA sensor at an anchor measures the bearing θ_i toward the agent, while a range measurement (RSSI, ToA or ToF) gives a distance d_i. In 2D, a single anchor with both θ and d suffices to pinpoint the location (intersection of a ray and a circle). In 3D, one anchor yields a circle of possible solutions, so at least two anchors—each providing (θ_i, d_i)—are required for a unique solution. In Taponecco et al. (2011) [42], a joint ToA/AoA estimator is proposed for UWB indoor localization under line-of-sight conditions, using an antenna array and simple demodulation circuitry. Results show good accuracy, with ranging errors around 10 cm and angular errors near 1°, depending on signal bandwidth.

- **Time-Difference-of-Arrival (TDoA).** Each anchor records the time t_i at which it receives a signal from the agent. The differences $\Delta t_{ij} = t_i - t_j$ define hyperbolic curves (2D) or surfaces (3D). At least 3 anchors (2 independent time-differences) are needed for 2D localization, 4 anchors in 3D. Kaun et al. (2012) [43] studied this method and compared it to ToA-based trilateration. It shows no major accuracy difference but unlike ToA, TDoA does not require the agent's clock to be synchronized with the anchors, only the anchors must share a common time base.

- **Fingerprinting:** This method relies on collecting ranging primitive (e.g., RSSI) at known reference locations to create a database or radio map. During position estimation, the agent compares its measurements to the database to estimate its position, often using k-nearest neighbors or other machine learning techniques. Fingerprinting is well-suited for complex indoor environments where propagation is affected by multi-path and obstacles. Unlike range-based methods, it does not require precise modeling of the signal, but its accuracy depends heavily on the density and quality of the reference data. In Cannizzaro et al. (2019) [41], results using trilateration were not great using multilateration due to RSSI low accuracy, then fingerprinting methods such as kNN, SVM and MLP were compared and reached an accuracy twice lower.

On-board localization systems for non-controlled environment

- **RF (with active anchors):**

  - **Wi-Fi:** The 802.11 b/g/n standard is embedded on the Raspberry Pi and supports RSSI-based multilateration. While Time-of-Flight (ToF) measurements are possible with 802.11mc (Round-Trip Time, RTT), this feature is not supported by standard Raspberry Pi hardware. Sadowski et al. (2018) [36] report a power consumption of approximately 200 mW for WiFi RSSI-based localization on a Raspberry Pi 3, which is relatively high, for a poor accuracy of a few meters. This result is mainly due to multi-path, NLoS conditions, and the short available bandwidth of WiFi.

  - **BLE:** Version 4.2 is support by the Raspberry Pi Zero 2, enabling RSSI-based localization and basic ToF multilateration. BLE 5.1 introduces more advanced localization capabilities, including Angle of Arrival (AoA), Time of Flight (ToF), Time of Arrival (ToA), and Time Difference of Arrival (TDoA), but requires newer hardware such as the nRF52840 SoC. Integration with the Raspberry

Pi would require an external USB dongle, applicable to both tags and anchors. BLE is significantly more energy-efficient than WiFi, with Sadowski et al. (2018) [36] reporting power consumption below 1 mW, although their measurements were performed on hardware different from that used in this thesis. BLE 4.2 RSSI-based multilateration offers accuracy comparable to WiFi systems, while more advanced techniques based on BLE 5.1 like ToF, ToA, and AoA can achieve localization precision up to 50 cm.

  - **UWB (Ultra-Wideband):** Its wide bandwidth allows the transmission of short pulses in the time domain, enhancing range measurements resolution and thus position accuracy up to 10 cm. Localization is typically achieved using TWR or ToA combined with multilateration or fingerprinting, or TDoA in more advanced setups. Some existing modules, like the Decawave DWM1001 (TWR with multilateration), are compatible with the Raspberry Pi and can operate as black boxes, abstracting the localization process from the host system. For instance, Poulose et al. (2020) [44] proposed a deep learning method based on convolutional neural networks (CNNS) and fingerprinting using ToA range measurements, aiming to mitigate the effects of multipath propagation and Non-Line-of-Sight conditions (NLoS) conditions.

- **Acoustic (with active anchors):**

  - **Sonar:** A low-power directional solution for ToF-based distance sensing. To achieve localization here is a potential solution. Time Division Multiple Access (TDMA) assigns distinct time slots to each active anchor to emit ultrasonic pulses sequentially, avoiding signal collisions. For this to work, anchors must be precisely time-synchronized to respect their slots. Robots listen for these pulses and identify the source by timing. Ultrasonic signals typically have a narrow directional cone (15°), so anchor placement must ensure sufficient coverage. NLoS conditions from obstacles or reflections can delay or block signals, reducing localization accuracy and reliability.

- **Optical (with passive anchors):**

  - **LiDAR:** Enables high-precision ToF sensing against reflective surfaces, at the cost of high power usage and computational complexity. Localization can then be derived with multilateration or fingerprinting.

  - **IR Rangefinder:** Also based on ToF, but with limited range (a few centimeters). Suitable for short-distance obstacle detection or fine localization.

These systems are summarized and characterized regarding the hardware of this project in Table 1.7.

Table 1.7: Comparative overview for possible non-controlled, on-board, absolute localization systems. The chosen setup for this project is highlighted.

	Wi-Fi	**BLE**		**UWB (DWM1001)**	**LiDAR**	**Sonar**
Range sensing	RSSI	RSSI	ToF, ToA, AoA	**TWR**, TDoA	ToF	ToF
Localization	Multilateration	Multilateration	Multilateration, TDoA, AoA+ToA	**Multilateration**, TDoA	Multilateration	Multilateration
Hardware	802.11 chip (RPi)	BLE 4.2 (RPi)	BLE 5.1 (USB Dongle)	**DWM1001**	LD19P	HC SR-04
Accuracy	∼20-50 @ 1, 1-5 m @ >3 m [36]	∼10-50 @ 1m, 1-3 m @ >3 m [36]	∼50 cm	**∼10 cm**	∼1 cm	>3 mm
Range	70 m [36]	60 m [36]	30 m	**100 m**	12 m	2-3 m
Price	-	-	< 10 €	**30 € / module**	100 €	∼ 1 €
Update rate	Very high [36]	High [36]	High [36]	**10 Hz**	10 Hz	10 Hz
Power requirements	∼ 200 mW [36]	<1mW [36]	<1mW [36]	**100 mW** [45]	900 mW	75 mW
Notes	Very simple but multi-path sensitive		Anchors also need to be BLE 5.1 compatible	**Hardware integration and interface need to be designed**	Orientation, line-of-sight, and surface dependent.	

1.3 Motivations and Objectives

The goal of this thesis is to develop a decentralized swarm robotic communication platform using Balboa 32U4 robots, augmented with embedded localization capability. Within the communication protocol, a main focus is put on the ability for the swarm to run distributed iterative algorithms based on synchronous-messaging.

The system must operate in a fully decentralized manner, including self-localization without relying on external infrastructure such as motion capture systems or GPS. Among several on-board systems for absolute localization in non-controlled environments, the Decawave DWM1001 UWB module was selected for its favorable trade-offs. It is currently one of the best option on the market, with an announced accuracy below 10 cm. Its software and hardware integration as well as an accurate calibration model will be designed in this thesis.

This work builds upon a lightweight and fully controllable communication backbone based on Bluetooth Classic, chosen for its low power consumption, native compatibility with the Raspberry Pi, and suitability for embedded swarm systems. On top of this physical layer, a full communication stack is developed, starting with a static mesh network layer. A key feature of the stack is its synchronous transport protocol, specifically designed to support distributed iterative algorithms with timing control, similar to systems like Glossy, but implemented in software on general-purpose Linux-based hardware.

In contrast to established routing protocols such as BATMAN or Babel, which are designed for dynamic IP networks and introduce significant overhead, our approach emphasizes simplicity and low energy consumption in static topologies. Unlike BLE Mesh, which lacks native support on Raspberry Pi and restricts access to lower communication layers, Bluetooth Classic offers greater flexibility for implementing custom protocols. Middleware solutions like eProsima (DDS) or MQTT, while powerful, often introduce unnecessary complexity and latency for static mesh networks, due to features such as QoS control that are not critical in tightly controlled swarm systems.

The final objective is to demonstrate the system's ability to support a real-world distributed application: cooperative target localization based on a gradient-tracking algorithm. By leveraging the previously introduced calibration model, the goal is to achieve high localization accuracy, showcasing the system's robustness under real conditions. This application furthermore serves as a proof of concept for the effectiveness of the proposed communication architecture.

1.4 Structure of the manuscript

This manuscript is organized into three main parts, each corresponding to a fundamental layer of the developed system:

- **Chapter 2: Embedded architecture of a swarm agent**: This part presents the hardware and software architecture of an individual swarm agent. Based on the Balboa 32U4 robot, it details the selection of the Raspberry Pi model and its integration via a communication interface with the robot. It also discusses design decisions and constraints related to the embedded localization system, including post-processing techniques developed to significantly enhance accuracy. Finally, the power consumption and autonomy of the agent are analyzed.

- **Chapter 3: Multi-agent communication**: This section focuses on the development of a modular and RPi-compatible mesh communication protocol built on Bluetooth Classic. It introduces a low-level socket-based infrastructure that transforms 1:1 Bluetooth links into a bidirectional mesh architecture supporting multiple processes. A middle-level communication layer implements modular overlays, including flooding, multi-hop unicast, asynchronous and synchronous messaging, with a particular focus on the latter. Finally, the performance of the designed communication stack is analyzed.

- **Chapter 4: Applications**: The final part highlights the relevance and flexibility of the proposed architecture through the implementation of several distributed applications on the highest layer of the stack. These include state-based behaviors such as collective stand-up, iterative consensus, multi-process LED synchronization, and a distributed target localization algorithm using UWB modules. Each application is examined in terms of implementation details, communication logic, and performance results.

Chapter 2

Embedded architecture of a swarm agent

2.1 Introduction

The Balboa 32U4 is a self-balancing robot developed by the company Pololu. It includes differential wheels, encoders, an IMU, magnetometer, and an ATMega32U4 MCU. Originally designed for teaching control theory, it offers expansion features such as level shifters, GPIO extenders, etc. This robot serves as the basis for each agent in our swarm. A detailed model and advanced control strategies were presented by Aurélien Soenen in his master's thesis [46].

Figure 2.1: Swarm agent based on Balboa. a Raspberry Pi Zero 2W has been added, a Decawave DWM1001 on the right and a OLED screen on the left. The wiring is custom made.

Each Balboa robot is coupled with a Raspberry Pi, forming an autonomous swarm agent. The Balboa is responsible for low-level operations such as motor control, LED signaling, button handling, and sensor access. In contrast, the Raspberry Pi handles high-level tasks including swarm communication, algorithm execution, and data logging. This chapter presents the integration of these components and evaluates their communication performance.

To enable collective behaviors—such as exploration, coordinated navigation, and localization, each agent must perform self-localization without relying on centralized infrastructure. The selected solution, based on the Decawave DWM1001 UWB module, is detailed, along with the post-processing techniques applied to enhance its performance.

The chapter concludes with a power consumption analysis, assessing the energy usage of individual components and estimating the system's autonomy based on empirical measurements.

2.2 Raspberry Pi

This section provides a detailed evaluation of the available Raspberry Pi models in terms of their suitability for the intended application. It further examines various wired communication bus for establishing an interface between the Raspberry Pi and the Balboa, highlighting the respective advantages and limitations of each approach. The subsequent parts present the hardware and software integration, along with a performance assessment.

2.2.1 Overview of the Raspberry Pi models

First, several Raspberry Pi models have been evaluated for integration into the swarm. The goal was to find the best trade-off between processing power, energy consumption, wireless capabilities, and weight. Table 2.1 compares the most relevant models considered.

Table 2.1: Comparison between available Raspberry Pi models

Criterion	**RPi 2**	**RPi 4**	**RPi 5**	**RPi Zero 2 W**
Bluetooth	USB dongle	5.0	5.0	4.2
WiFi	USB dongle	802.11ac	802.11ac	802.11n
Weight	45g	46g	48g	11g
Consumption	3.5W	6W (idle), 13W	7W (idle), 15W	1.5W
Price	€35	€40 (1GB)	€60 (2GB)	€20
RAM	1GB	1/2/4/8GB	2/4/8/16GB	512MB
Connectivity	4× USB 2.0, HDMI, audio jack	2× USB 3.0, 2× USB 2.0, 2× micro HDMI	2× USB 3.0, 2× USB 2.0, micro HDMI, RTC	1× micro USB (OTG), mini HDMI

The Raspberry Pi 2 offers low power consumption and sufficient computational resources for basic applications. However, it lacks onboard WiFi and Bluetooth, requiring external USB dongles for connectivity. This setup increases both cost and complexity, making it less suitable for a compact and efficient swarm setup.

The Raspberry Pi 4 and 5 provide significantly greater processing power and memory options, making them suitable for more demanding applications, such as image processing or AI-based tasks. They support modern wireless standards (Bluetooth 5.0 and WiFi 802.11ac), which improves connection stability and range compared to older models such as 802.11n. The Raspberry Pi 5 includes a RTC (real-time clock) usable with an external battery, it could be useful for some distributed applications requiring synchronized clocks where no internet connection is available. However, their higher power consumption and weight can negatively impact the balance and autonomy of the robot. A heavier board shifts the center of gravity, making self-balancing more difficult, although still manageable with appropriate control tuning.

The Raspberry Pi Zero 2 W offers a highly favorable compromise. Its compact size and low weight are ideal for balancing applications, and while its computational power is more limited, it remains sufficient for distributed algorithms such as those used in this project. Although it only supports Bluetooth 4.2 and WiFi 802.11n, these versions are adequate for the communication protocols implemented in this project, which require only moderate data rates and have limited range demands. The reduced number of connectors is not an issue here, as all required I/O is managed via GPIO in this project.

In conclusion, the final swarm configuration is composed of 4 Raspberry Pi Zero 2 W units to prioritize energy efficiency and mechanical simplicity, and 2 Raspberry Pi 4 (2GB) units for prototyping and testing more computationally intensive applications in the future.

2.2.2 Overview of the agent-level communication

Various options for the wired communication bus between the Raspberry Pi and the Balboa 32U4 are reviewed and compared, considering the constraints imposed by the available hardware.

UART

UART (Universal Asynchronous Receiver-Transmitter) is a full-duplex serial communication protocol used to transmit and receive data between two electronic devices. It operates asynchronously, meaning there is no shared clock signal between the transmitter and receiver, with each side synchronizing its data at its own rate. The maximum baud rate of the ATMega32U4 is approximately 115 Kbps.

Either the native UART or the USB port can be used. But using a USB-UART connection would prevent using the Balboa's USB port for other purposes, such as serial debugging or operating with the motors disabled. In addition, since the Raspberry Pi is shield-mounted on the Balboa, its TX/RX pins are physically inaccessible. We could still solder wires directly onto the Raspberry Pi or design a custom GPIO extender PCB with pin headers to place between the Balboa and the Raspberry Pi, but this solution is not ideal. The Balboa operates at 5V, while the Raspberry Pi uses 3.3V. Therefore, a level shifter is needed

between the Balboa's TX and the Raspberry Pi's RX. There is a general-purpose level shifter available on the Balboa that we could use. However, we will need this level shifter later on for the Decawave module. Note that it is the chosen solution of Sandbots[15], which is also based on Balboa 32U4.

SPI

SPI (Serial Peripheral Interface) is a full-duplex synchronous serial communication protocol used to transfer data between a master device and one or more peripheral devices. It operates with a shared clock signal, allowing for faster data transfer, with a throughput around 10 Mbps, compared to asynchronous protocols like UART.

The unavailability of the Raspberry Pi pins remains constraining. Moreover, the SPI bus requires four wires: MOSI, MISO, CLK, and CS. Among them, three need a level shifter (MOSI, CS, CLK). We would need external level shifters, which is impractical.

I²C

I²C (Inter-Integrated Circuit) is a synchronous serial communication protocol used to connect multiple devices using only two wires: one for the clock signal (SCL) and one for data (SDA). It allows for communication between a master device and multiple peripheral devices, with each device having a unique address.

This is the communication method recommended by Pololu, and a level shifter is already included on the Balboa between the ATMega32U4 and the Raspberry Pi. Therefore, no additional hardware is needed. However, the IMU required for balancing is also connected to this I²C bus. Initially, the Balboa acts as its master. This configuration needs to be revised, as I²C does not support multiple masters. The best solution would be to move the balancing control loop to the Raspberry Pi instead of the Balboa. In this case, the Raspberry Pi would directly read the IMU data, and send/request data to/from the Balboa.

Table 2.2: Comparison between wired buses for Raspberry Pi - Balboa interface.

Criterion	**I²C**	**SPI**	**UART**
Rate	100 Kbps	10 Mbps	115 Kbps
Hardware integration	Easy	Complex	Medium
Software development	Difficult	Easy	Easy
Connection	SDA/SCL	MOSI/MISO/CS	USB or TX/RX
Architecture	1 master	1+ master	P2P
Other advantages	Level-shifter included	Fast, full-duplex	Asynchronous
Other drawbacks	Shared bus with IMU, then software refactoring needed	USB and general purpose level shifter unavailable, Raspberry Pi GPIO are inaccessible	

The main characteristics of each wired communication bus are summarized in Table 2.2. Among the available options, I²C emerges as the most suitable solution for interfacing the Raspberry Pi with the Balboa. It is the only bus that does not require extra hardware and offers a data rate sufficient for most foreseeable applications, as discussed in Section 2.2.5. However, its integration entails substantial software refactoring, as previously mentioned.

2.2.3 Hardware

The I²C wiring is embedded on the Balboa, as illustrated in Figure 2.2. A dual-channel level shifter separates the I²C bus into two voltage domains: 3.3 V and 5 V. This is necessary because the Raspberry Pi and the IMU operate at 3.3 V, while the Balboa's ATmega MCU and the OLED display require 5 V. The components related to the Decawave module are discussed in Section 2.3.

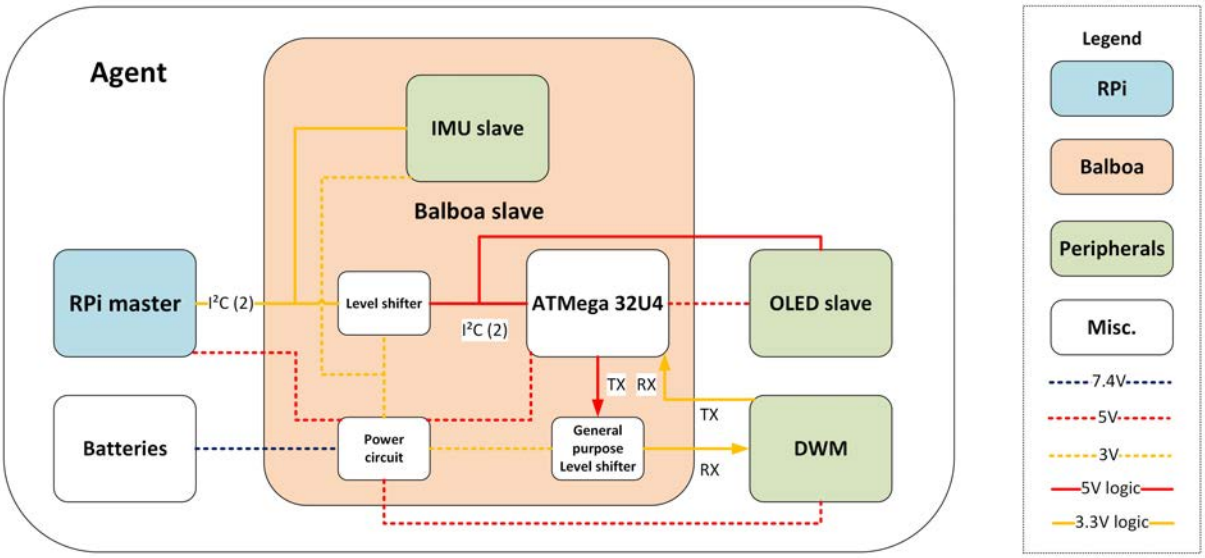


Figure 2.2: Agent hardware architecture. The Raspberry Pi acts as the I²C master and controls three slaves: the IMU, the OLED, and the Balboa. The DWM1001 is connected to the Balboa via UART, only one level shifter is required (and available), since the Balboa can read 3.3 V signals. Numbers in parenthesis indicate the number of wires used. All components are powered by the main power circuit, except for the OLED display. This diagram is not exhaustive.

2.2.4 Implementation

The software architecture has been refactored and extended beyond the standard Balboa library to enable communication with the Raspberry Pi. Originally, the Balboa managed the IMU directly and executed the balancing control loop onboard. All peripheral interfaces on the Raspberry Pi, as well as their interconnections, are summarized in Figure 2.3. The Decawave-related components will be detailed in Section 2.3.

In the new architecture, the Raspberry Pi acts as the I²C master and can request data from, or send commands to, its slaves. A custom data structure has been implemented on the Balboa to handle the variables exchanged via the I²C bus. Since the Balboa is no longer the IMU master, it cannot directly retrieve IMU measurements to perform balancing.

Consequently, the control loop has been ported from the Balboa to the Raspberry Pi. The Raspberry Pi now communicates with the IMU via an I²C interface and performs the control computations.

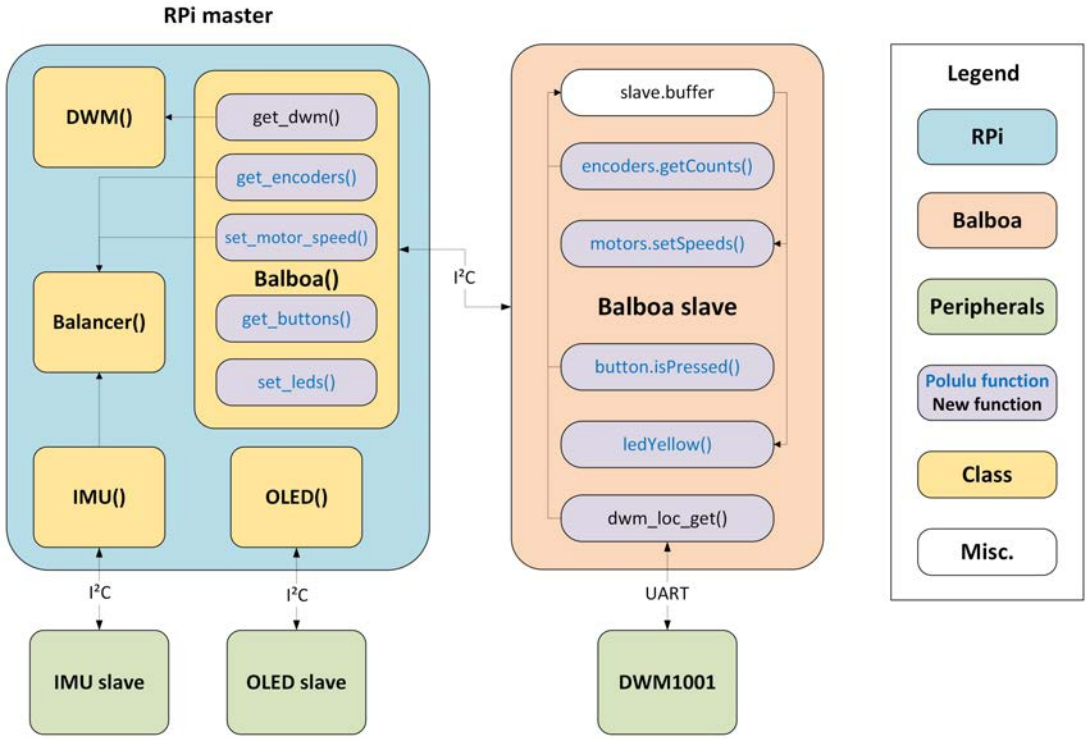


Figure 2.3: I²C communication within an agent. The Raspberry Pi is the I²C master of the IMU, OLED, and Balboa. The Balboa runs a loop that updates `slave.buffer` with Decawave measurements, encoder readings, and button states. It also sets motor outputs and LED states based on values written to this buffer. The Decawave module is interfaced via UART and its measurements are post-processed in the `DWM()` function on the Raspberry Pi. The balancing control loop is implemented in `Balancer()`, while `IMU()` and `OLED()` are I²C interfaces.

The Broadcom BCM2835 MCU of the Raspberry Pi has a known hardware bug affecting its I²C implementation: improper handling of clock stretching can lead to communication failures or data corruption with some slave devices. This issue prevents the use of standard Arduino I²C libraries on the Raspberry Pi. A detailed explanation of the bug is available in [47]. To mitigate this, the company Pololu provides a custom I²C library [48] that dynamically reduces the data rate when necessary. While this results in slower communication, it significantly improves reliability. The actual data rates achieved will be discussed in the next section.

2.2.5 Performance analysis

Benchmarking the agent's communication shows that, with the I²C bus set to 100 kHz, the Raspberry Pi achieves read and write speeds of approximately 21 kbit/s and 53 kbit/s, respectively. These values are lower than the theoretical value because of the discussed clock stretching issue.

The balancing controller, originally running on the Balboa, has been ported to the Raspberry Pi and operates at a loop frequency of 100 Hz. At each iteration, the Raspberry Pi performs the following transactions:

- Reads encoder values from the Balboa: 4 bytes
- Reads IMU measurements directly from the IMU: 12 bytes
- Sends motor speed commands to the Balboa: 4 bytes

This results in a total communication load of:

$$\text{Read: } 100 \cdot 8 \cdot (4 + 12) = 12.8\,\text{kbit/s}, \quad \text{Write: } 100 \cdot 8 \cdot 4 = 3.2\,\text{kbit/s}$$

These values remain well below the observed maximum throughput, leaving ample room for additional, less frequent data exchanges, such as LED states, button inputs, or Decawave data.

The company Pololu also recommends keeping the I²C buffer size under 64 bytes. In this project, the total buffer size is 28 bytes, broken down as follows:

- Motor speeds: 2×4 bytes
- Encoder values: 2×4 bytes
- LEDs: 3×1 byte
- Buttons: 3×1 byte
- Position: 2×2 bytes
- Distance: 2 bytes

In conclusion, both the data throughput (12.8 kbit/s read, 3.2 kbit/s write) and the buffer size (28 bytes) meet the constraints imposed by the 100 kHz I²C bus when using the `PololuRPiSlave` library.

2.3 Localization System

Equipping each agent with an embedded self-localization system enables a wide range of swarm behaviors, particularly those related to navigation, coordination, and spatial organization. For the purposes of this project, the system must be capable of measuring inter-agent distances ranging from 0.1 m to 4 m, with as high accuracy as possible. Ideally, the positioning error should remain below 10 cm. Additionally, since the system must be integrated onto the Balboa robot, the sensing technology must be omnidirectional and mechanically integrable without compromising the robot's balancing stability.

In Chapter 1, localization systems has been categorized as centralized, proprioceptive, or environment-based, see Table 1.6. In our context, we require an on-board, decentralized, and absolute localization system, which implies an exteroceptive approach suitable for operation in a non-controlled environment. Among the feasible techniques, range estimation can be achieved through WiFi RSSI, BLE (RSSI, ToF, or ToA), UWB (ToF or ToA), sonar, or LiDAR, summarized in Table 1.7. However, both LiDAR and sonar are unsuitable in this case: they are directional, difficult to integrate mechanically, and may interfere with the robot's ability to balance properly due to their size and mounting requirements.

Considering these constraints, we adopt the Decawave UWB module as the localization solution, owing to its superior accuracy among the remaining candidates. Nevertheless, its integration introduces several challenges, particularly in terms of hardware and software interfaces design, that will be addressed in the following sections. Furthermore, in order to improve the reliability and accuracy of the distance measurements, a post-processing step including calibration and filtering will be designed and implemented.

2.3.1 Decawave DWM1001

The Decawave DWM1001 is presented in a more detailed way in this section, in order to get a better evaluation of its integration process. The Decawave DWM1001-DEV is composed of an UWB antenna, a microprocessor (Nordic nRF52832) and a J-Link allowing us to program the microprocessor. We can either use a custom firmware or the one provided by Decawave, which is called PANS API. It contains all needed function in order to setup (with a mobile application) and use the localization system. It is possible to communicate with the embedded MCU by using UART bus, SPI bus or BLE.

Figure 2.4: Decawave DWM1001 as an anchor. The height of the support corresponds to the height of the tag on the Balboa.

Each module can be configured either as a tag (see Figure 2.1) or an anchor (see Figure 2.4). The position of the tags are to be determined based on their measured distance with the anchors that are located at known position. This distance is measured either using Two-Way Ranging (TWR, a form of ToF) or ToA. Based on that, the firmware from the Decawave MCU performs repectively multilateration or TDoA to compute the position of tags. However, only the TWR-multilateration is implemented in the PANS API.

The integration of the module is not trivial due to the fact that the Raspberry Pi pins are not available and the Balboa operates at 5V while the Decawave uses 3.3V logic. Moreover, the communication interface between the agent and the Decawave need to be designed, based on the PANS API. The whole design have been thought in detail and reveals to be possible without any external hardware. It will be explained thoroughly in the next few sections.

2.3.2 Hardware

The connection options between the Decawave module and the other components of the agent, along with their limitations are presented in this section. On the Balboa robot, the Raspberry Pi is mounted in a shielded position with no access to its GPIO pins. As such, connecting the Decawave module directly to the Pi would require either soldering onto the board or designing a custom GPIO extender between the Pi and the Balboa, both solutions being impractical and undesirable. To avoid external hardware and risky modifications, the most viable solution is to connect the Decawave module directly to the Balboa's MCU, which has accessible extension headers. Note that the Decawave operates at 3.3 V logic, while the Balboa uses 5 V. To protect the Decawave, a level shifter are required on signals from the Balboa to the Decawave. Fortunately, the Balboa can read 3.3 V signals, a single channel level shifter is enough.

This project uses the PANS API, which allows communication with the Decawave module via UART, SPI, or BLE. Since BLE is already used for swarm communication, using it for the Decawave could cause conflicts. A wired interface is therefore preferred, either SPI or UART. SPI requires three level-shifted signals (SCLK, MOSI, and CS), whereas UART only needs level shifting on the Balboa's TX line. Conveniently, the Balboa includes a bi-directional, single-channel level shifter, making UART the simplest and most hardware-efficient option. The final hardware configuration is shown in Figure 2.2.

2.3.3 Implementation

The implementation of the Decawave interface, enabling communication between the Raspberry Pi and the Decawave module, is presented in this section. As outlined in the previous hardware discussion, two interfaces are required.

First, the I²C interface between the Balboa and the Raspberry Pi, implemented in the DWM() class (see Figure 2.5). Its integration within the system is also depicted in Figure 2.3. The existing I²C interface described in Section 2.2.3 has been extended to include Decawave measurements. The Raspberry Pi retrieves these measurements via I²C and performs post-processing (detailed in the next section) within the DWM() class.

Second, the UART interface on the Balboa is implemented through the function shown in Figure 2.5, and detailed in Algorithm 1. The UART-based PANS API offered by Decawave can operate in two modes: shell mode and generic mode. The former allows for human-readable command input via a terminal, whereas the latter uses binary TLV (Type-Length-Value) messages, offering greater efficiency and better integration into embedded systems. For this project, the generic mode is used.

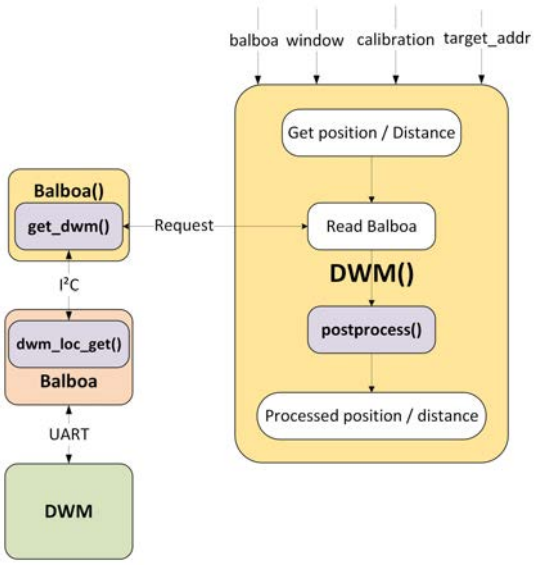


Figure 2.5: Block diagram of both interfaces involved in Decawave communication. The Balboa queries the Decawave module via UART at 10 Hz using the interface developed on the Balboa. The Raspberry Pi then reads the result over I²C via the Balboa() interface, before applying post-processing in DWM() class.

The operating principle is straightforward: a TLV-formatted binary request corresponding to a specific PANS API command is sent over UART from the Balboa to the Decawave module. The Decawave responds with a binary message containing the requested data.

Example 1 (Tag node):

TLV request	
Type	**Length**
0x0C	0x00

TLV response										
Type	**Length**	**Value**	**Type**	**Length**	**Value**				**Type**	**Length**	**Value**
		err_code			**position**, 13 bytes						1 byte, number of distances encoded in the value
					4-byte x	4-byte y	4-byte z	1-byte quality factor			
0x40	0x01	0x00	0x41	0x0D	0x4b 0x0a 0x00 0x00 0x1f 0x04 0x00 0x00 0x9c 0x0e 0x00 0x00 0x64				0x49	0x51	0x04

TLV response (residue of the frame from previous table)								
Value								
2 bytes UWB address	4-byte distance	1-byte distance quality factor	position in standard 13 byte format	...	2 bytes UWB address	4-byte distance	1-byte distance quality factor	position in standard 13 byte format
position and distance AN1				AN2, AN3	position and distance AN4			

Figure 2.6: Excerpt from the Decawave API guide [49] showing the use of dwm_loc_get().

The TLV format begins with a one-byte type field, followed by a one-byte length field, and then a variable-length value field (0 to 253 bytes), as specified by the length field. This format is shown in Figure 2.6, which is an excerpt from the Decawave API documentation [49].

Following an in-depth review of the Decawave UART API documentation [50], the dwm_loc_get() command was selected, as it provides both the current tag's position and the distances to all visible anchors. Algorithm 1 outlines its implementation on the Balboa: the function sends the appropriate binary request and parses the response, extracting only the tag's position and the distance to the anchor matching the target UWB address.

Algorithm 1 `dwmLocGet()`: Retrieve position and distance data from DWM1001

Require: Serial connection to the DWM1001
Ensure: Position and distance data stored in `slave.buffer`

```
Send command 0x0C/00 over serial
Read incoming bytes from serial into rx_data
data_cnt ← 0
if rx_data[data_cnt] = 0x40 then
    error ← rx_data
    data_cnt ← data_cnt +3
end if
if rx_data[data_cnt] = 0x41 then
    data_cnt ← data_cnt +2
    (x, y, z, qf) ← rx_data
    slave.buffer ← (x, y, z, qf)
    data_cnt ← data_cnt +13
end if
if rx_data[data_cnt] = 0x49 then
    (length, anchor_number) ← rx_data
    data_cnt ← data_cnt +3
    for i ← 0 to anchor_number do
        if length > 0 then
            (uwb_addr, d, qf) ← rx_data
            slave.buffer ← (d, qf)
            data_cnt ← data_cnt +4
            (x, y, z, qf) ← rx_data
        end if
        data_cnt ← data_cnt +13
    end for
end if
```

2.3.4 Error characterization, filtering, and calibration

The measurements performed by the Decawave are not limitless. In this section, the accuracy and precision of the position estimation are analyzed and post-processing techniques such as calibration and filtering are designed in order to improve accuracy and precision of the measurements.

Accuracy is defined as the ability to provide average measurement nearby the actual value, it is limited by systematic bias. Here are the main biases that affects the accuracy of the measurements:

- Anchor position accuracy: anchor position measurement is limited.
- Non-Line-of-Sight conditions (NLoS): fixed obstacles may introduce a bias.
- Orientation: the beam may be omnidirectional, it is not perfectly isotropic. Measurements vary with the orientation of the antenna.

The precision corresponds to the variability of several measurements in identical conditions. Here are the main biases that affects the precision of the position estimation of the tags:

- Reply time accuracy: TWR algorithm need an estimation of the reply time of the receiver, but it has a small variability.
- Hardware bias: device-to-device bias due to delays, connections, etc.
- Random noise on range measurements: due to interferences, thermal noise, etc.
- Non-Line-of-Sight conditions (NLoS): Fixed obstacles may introduce a systematic bias. But the obstacle presence may not be systematic, in this case the precision is affected.

The precision could be improved by using filtering techniques that smooth the measurements, and reject outliers, while the accuracy could be improved with a calibration that remove the systematic biases by designing a sensor model. Ideally, we would process the distances on which the multilateration is based, consequently the position accuracy would be improved as well. However, we do no have access inside the PANS API, then we distinguish the position error, and distance error, both will be post-processed differently.

Filtering

Filtering techniques aim to improve precision through a two-step process:

1. **Outlier removal**
   Some measurements may be heavily biased. These are detected and removed using the Interquartile Range (IQR) method. A value is considered an outlier if it lies outside the following bounds:

   $$\text{IQR} = \text{Q3} - \text{Q1}, \quad \text{LB} = \text{Q1} - 1.5 \cdot \text{IQR}, \quad \text{UB} = \text{Q3} + 1.5 \cdot \text{IQR}, \tag{2.1}$$

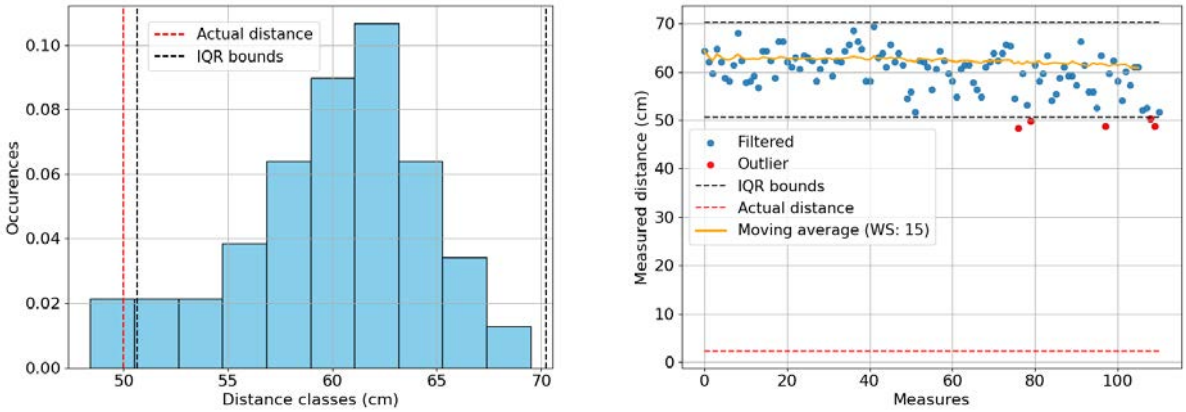
   where Q1 and Q3 are the 25th and 75th percentiles of the dataset, respectively. LB and HB designate the lower bound and the upper bound, respectively.

2. **Moving average**
   To smooth out rapid variations and to keep distance estimation as close to the measurement's mean as possible, a moving average is applied. The filtered value at each step, $\bar{m}_t$, is the average of the previous N measurements, m_{t-i}, where N is the window size.

   $$\bar{m}_t = \frac{1}{N} \sum_{i=0}^{N-1} m_{t-i}. \tag{2.2}$$

High-frequency variations can cause issues. For instance, in a trajectory planning algorithm, position ripple may cause the robot to oscillate. In a localization algorithm, it can delay convergence or even prevent it entirely. However, more the signal is smoothed, more the setting time increases, slowing the dynamic of the system.

Figure 2.7 illustrates the effect of distance filtering. Outliers are removed, and the ripple is effectively reduced. Regarding position filtering, the distance between each measurement and the centroid of the sample is computed and the IQR technique is applied as well. Since the measurements are ideally clustered near the centroid, only an upper bound is applied to retain those that lie within a certain radius, effectively defining a circular area of acceptance.



(a) (b)

Figure 2.7: (a) Distribution of raw distance measurements. (b) Illustration of moving average on raw distance measurements. Window size is 15 and actual distance is 50 cm. Measurements have been performed at three different positions.

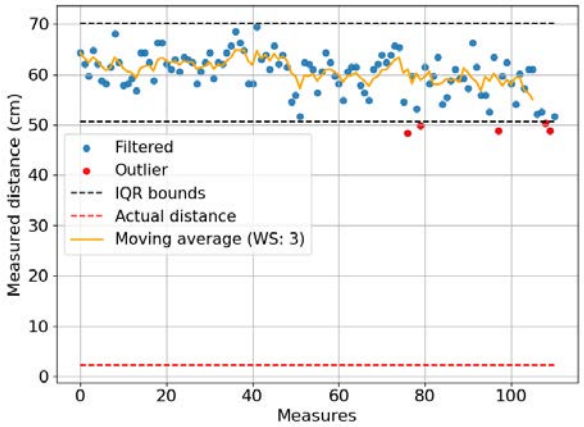


Figure 2.8: Window size reduced to 3 to better follow moving agents.

If the agents are moving, a smaller window size may be needed to reduce position estimation latency. However, this increases sensitivity to ripple. Figure 2.8 shows results with a reduced window size of 3 for a static agent.

Calibration setup

Calibration consists in comparing the estimated positions or distances to their corresponding ground truth values over a range of tag placements. The sensor's transfer function or the predictor is derived using linear regression. For optimal accuracy, calibration should be repeated whenever the anchors are repositioned, in order to cancel the accuracy loss due to it.

To achieve this, a precise setup was arranged on a 2×2 m table, with three anchors and one target fixed at known positions. Markers were distributed on the ground to indicate the locations where accurate calibration measurements should be taken, as shown in Figure 2.9. For each distance or position, the Decawave tag was rotated by 90° between sequences of approximately ten measurements per orientation.

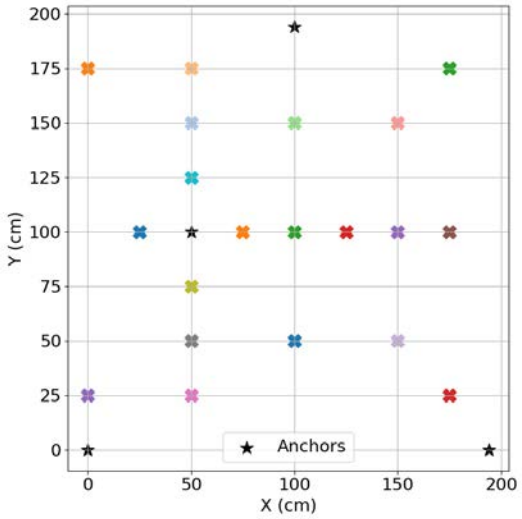


Figure 2.9: Position of the agent for measurements. A video demonstration is available here, showcasing the measurement process. A specific program, available on the project's GitHub, is controlled via the Balboa's buttons and starts and stops measurements between each position and robot orientation.

Further improvements could be made by developing a specific calibration model for each agent, in order to account for device-to-device variations. Additionally, a more suitable model could be chosen to better reflect the system's inherent non-linearity. Incorporating accelerometer and gyroscope data to factor in the agent's orientation within the model is another promising avenue for enhancing estimation accuracy.

Distance calibration

Distance calibration is performed using a simple one-dimensional linear regression. The sensor's transfer function is estimated using the least squares method applied to the measurements and their ground truth. The residual sensor model error after calibration is defined as the difference between the predicted measure from the sensor model and the filtered measurements at a given actual distance. These are shown in Figure 2.10, and the corresponding equations are provided below.

Let N the number of distinct distances tested, m the measured distance, $\hat{m}$ its model-based estimation, d the actual distance, and $\hat{d}$ the estimated true distance. The uncalibrated sensor model is simply:

$$\hat{m} = d. \tag{2.3}$$

The calibrated model, obtained from linear regression, becomes:

$$\hat{m} = 1.056d + 6.299. \tag{2.4}$$

With the residual model error being:

$$e_r = \hat{m} - m. \tag{2.5}$$

Finally, the filtered residual model error is denoted $\bar{e}_r$

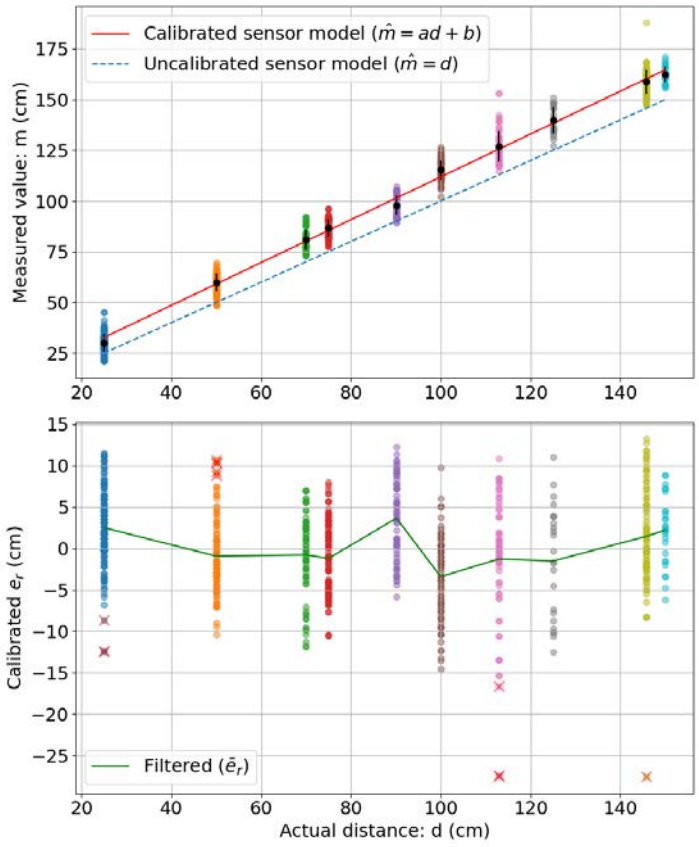


Figure 2.10: Top: Colored dots are raw measurements; black dots are filtered values. The red line represents the calibrated sensor model $\hat{m} = 1.056d + 6.299$, while the blue line corresponds to the uncalibrated one $\hat{m} = d$. Bottom: Dots represents the sensor model residual error e_r for each measurement. Outliers are highlighted with a red cross. The green curve represents $\bar{e}_r$.

Figure 2.10 shows a good fit between the linear regression sensor model and the measurements, confirming its suitability. Next, we quantify the model's quality through error metrics. The mean model error mostly reflects systematic bias and can be significantly reduced through calibration:

$$\bar{e}_{r,av} = \frac{1}{N} \sum_{i=0}^{N} \bar{e}_r(d_i). \tag{2.6}$$

In contrast, the root-mean-square (RMS) error of the sensor model gives an indication of the average spatial deviation, regardless of distance. $\bar{e_r}$, the filtered residual model error (see 2.5), is centered in order to capture only deviation effect.

$$\bar{e}_{r,\text{RMS}} = \sqrt{\sum_{i=0}^{N} (\bar{e_r}(d_i) - \bar{e}_{r,av})^2}. \tag{2.7}$$

This metric reflects the overall model accuracy. Although it could be further reduced using more complex (nonlinear) models, this might result in over-fitting. In a linear model, the reduction in $\bar{e}_r$ corresponds to an offset correction, while the decrease in $\bar{e}_{r,\text{RMS}}$ reflects an improved slope adjustment that better captures the trend of the data.

The error metrics before and after calibration are summarized in Table 2.4. The calibrated model's statistics correspond to $e_r = 1.056d + 6.299 - m$, i.e., the green line in Figure 2.10, while the uncalibrated one corresponds to $e_r = d - xm$. On the training set, $\bar{e_r}$ is nearly canceled and $\bar{e}_{r,\text{RMS}}$ fall from 3.02 down to 2.12 cm.

Table 2.3: RMS and mean error of the sensor model before and after calibration

Metric	**Uncalibrated Model**	**Calibrated Model**
$\bar{e}_{r,\text{RMS}}$	3.02 cm	2.12 cm
$\bar{e}_{r,av}$	11.51 cm	0.072 cm

Finally, we can estimate the true distance from a measurement by inverting the sensor model:

$$\hat{d} = 0.946m - 5.96, \tag{2.8}$$

and the associated estimation error is defined as:

$$e_e = \hat{d} - d. \tag{2.9}$$

.

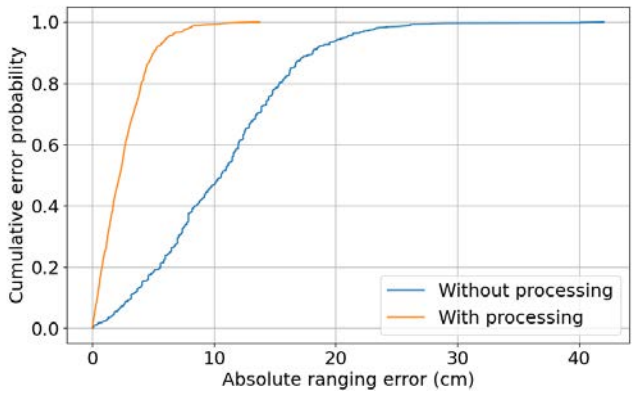


Figure 2.11: Cumulative error distribution for the distance estimator.

To evaluate the accuracy improvement thanks to processing, Figure 2.11 shows the cumulative distribution of the distance estimation error with and without calibration. It indicates that 90% of the estimates fall within 4 cm of the true distance for the calibrated model, while this value reach 19 cm without calibration, a reduction of 79% in error.

Position Calibration

Position calibration is also performed using linear regression, applied to the same set of measurements used for distance calibration, but this time focusing on positional data. While distance calibration initially derives a sensor model, position calibration is solely used to obtain a position predictor. However, by inverting Equation 2.10, a sensor model could theoretically also be derived from this predictor. Figure 2.12 illustrates the raw measurements at each training position, the filtered data, the resulting calibrated positions, and the ground truth.

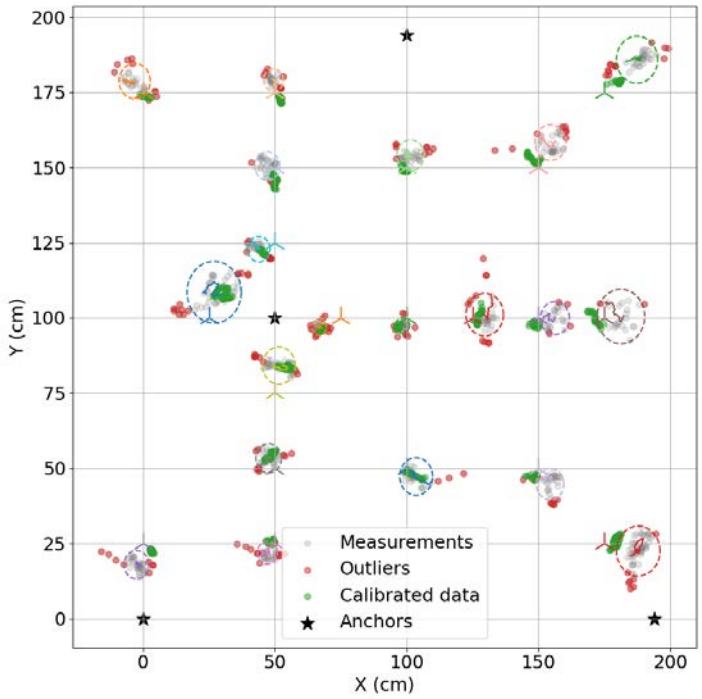


Figure 2.12: Position calibration on a 2D map. Raw measurements are shown in grey. A dashed circle is drawn at their centroid, with its radius corresponding to the IQR bound. Ground truth positions are indicated by three-pointed stars. Outliers are marked in red, and calibrated positions in green.

Let p denote the actual position, $\hat{p}$ its estimated value, and m the measured position. The 2D linear regression model obtained by calibration for position estimation is:

$$\hat{p} = A \cdot m + b = \begin{bmatrix} 0.935 & 0.003 \\ -0.011 & 0.945 \end{bmatrix} \cdot m + \begin{bmatrix} 4.624 \\ 5.086 \end{bmatrix}. \tag{2.10}$$

The estimation error is defined as:

$$e_e = \|\hat{p} - p\| \tag{2.11}$$

The estimation error after calibration, denoted $e_{e,\text{cal}}$, is computed using the estimated position $\hat{p}$ from the linear model. The uncalibrated error, $e_{e,\text{raw}}$, corresponds to the case where $\hat{p} = x$. Figure 2.13 shows both the calibrated error and the ratio $\frac{e_{e,\text{cal}}}{e_{e,\text{raw}}}$, which represents the error gain. A ratio close to 0 indicates significant improvement through calibration, while a value near 1 means little to no improvement, and values above 1 indicate degradation.

We observe that the residual error after calibration ranges from 1 to 9 cm, with higher values near the anchor located at coordinates (50, 100), this anchor will later serve as a target for localization. The increased error in this region might stem from the non-standard anchor configuration used. While anchors are often arranged in a rectangular pattern, our system is expected to operate in arbitrary configurations to support localization anywhere in the area.

The heatmap of $\frac{e_{e,\text{cal}}}{e_{e,\text{raw}}}$ shows that calibration reduces the error down to 40% across most of the map. However, two local regions exhibit an error increase of up to 140%. These variations are attributed to the system's inherently non-linear nature, compounded by other non-linearities such as multi-path effects, measurement noise, and occasional NLoS conditions, despite careful mitigation during data collection. While these could be addressed with more sophisticated non-linear models, the calibrated error remains sufficiently low for our requirements, and more complex models risk overfitting, especially given the limited measurement precision.

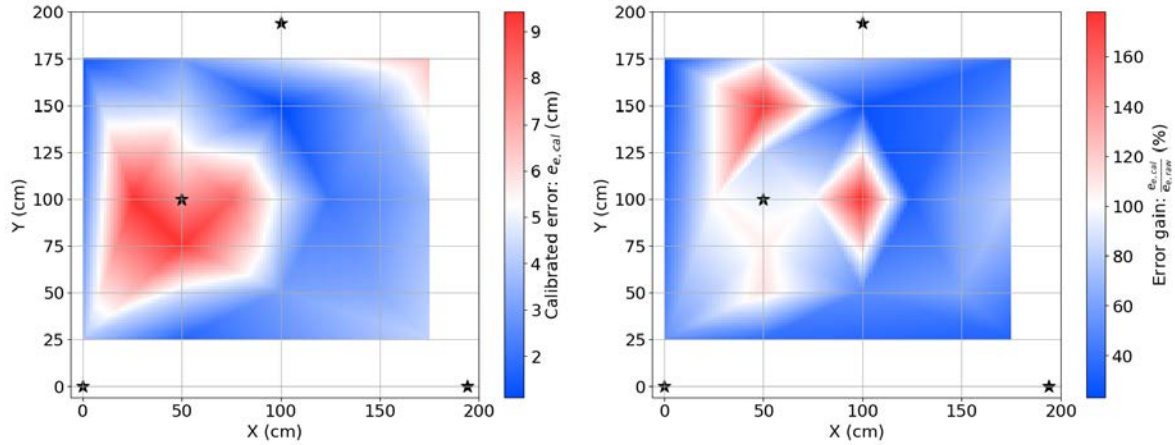


Figure 2.13: Residual error after calibration ($e_{e,\text{cal}}$), and ratio of calibrated to uncalibrated error ($\frac{e_{e,\text{cal}}}{e_{e,\text{raw}}}$), representing the error gain.

The definitions of RMS and mean from Equations 2.7 and 2.6 are naturally extended to two dimensions. The overall average spatial error over the full map is estimated using the RMS of $e_{e,\text{cal}}$, with the mean value also computed. The corresponding values are reported

in Table 2.4. We observe that the spatial averaged error of the calibrated estimator fell down to 0 and its RMS value is reduced by 35 %, highlighting the good fit of the model.

Table 2.4: RMS and mean position error before and after calibration

Metric	**Uncalibrated Model**	**Calibrated Model**
$e_{\bar{\text{RMS}}}$	5.17 cm	3.51 cm
$\bar{e}_{r,av}$	1.31 cm	0 cm

Finally, the global calibration effect is assessed with the cumulative distribution of the position error, shown in Figure 2.14. We observe that 90% of the measurements originally exhibit an error below 15 cm, which improves to under 8 cm after calibration, a reduction of 47%.

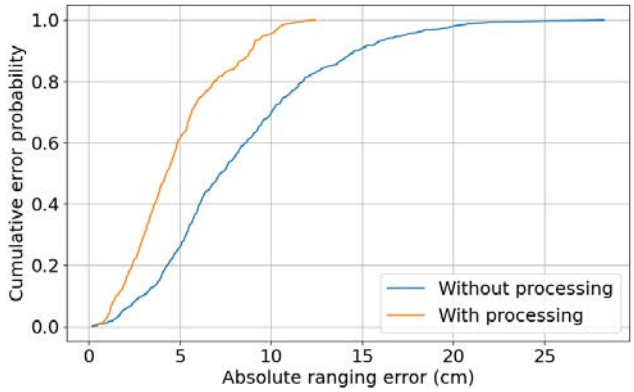


Figure 2.14: Cumulative distribution of the position estimation error.

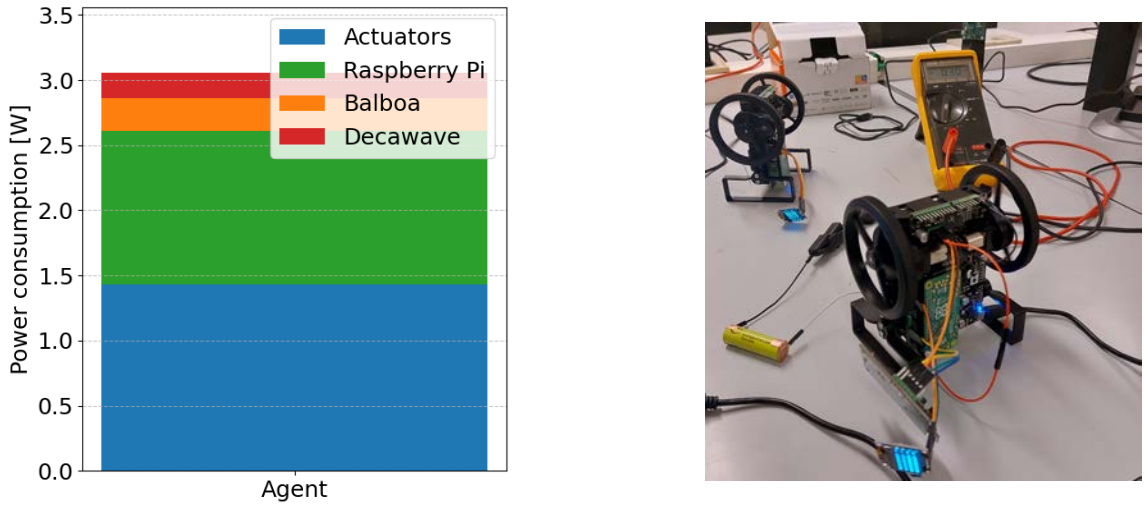
2.4 Power Consumption of an Agent

The agent is composed of several hardware components that each contribute to the total power consumption. To estimate the system's autonomy, current measurements were taken at the battery output. The measurements were taken using a multimeter in series with the batteries, with one cell placed outside the enclosure as shown in Figure 2.15b.

Table 2.5: Average measured current from the batteries when they were at 7.54V. Each column add an additional component, keeping the previous.

	Balboa	Raspberry Pi	Decawave	Actuators
Current	33 mA	140-190 mA	230 mA	300 - 420 mA
Power	249 mW	1.05 - 1.43 W	1.73 W	2.26 - 3.16 W

To understand how consumption is distributed across components, we combined theoretical values from datasheets with experimental measurements. Measurements were performed in various configurations, with each column of Table 2.5 adding one component to the previous setup. The voltage during testing was 7.54 V, the resulting consumption per block is illustrated on Figure 2.15a.



(a) (b)

Figure 2.15: Power consumption analysis. (a) Consumption stack sorted. (b) Current measurement setup.

- **Balboa (ATMega32U4 + onboard components)**: According to its datasheet [51], the ATMega32U4 consumes typically 10–15 mA at 5 V (50–75 mW). The rest of the measured power (approx. 200 mW) is attributed to the Balboa's supporting components (regulators, sensors, etc.).

- **Raspberry Pi Zero 2 W**: Literature reports a typical current draw of 120 mA (600 mW) with Wi-Fi enabled and HDMI disabled [52]. Experimentally, we observe an increase from 249 mW to 1.05–1.43 W when the Pi is active and running consensus algorithms via I²C. This corresponds to a net power draw of 750–1180 mW.

- **Decawave DWM1001**: Its average power consumption is 100 mW, peaking at 250 mW during reception [45]. When added to the system running localization at 10 Hz, the total power rises to 1.73 W, suggesting a Decawave contribution of 200 mW.

- **Actuators (Motors)**: Power draw from motors varies with speed, torque, and activity. Our setup does not allow measurement during balancing, so values were taken with the robot running unloaded at different speeds. The observed increase (1.43 W) is a rough estimation, as real conditions include lower speed but higher torque and IMU polling. For accurate profiling, embedded power monitoring would be required.

2.4.1 Autonomy Estimation

Assuming a 6-cell AA NiMH battery pack (1.2 V nominal per cell, 2400 mAh capacity):

$$\text{Total energy} = 7.2\ \text{V} \times 2.4\ \text{Ah} = 17.28\ \text{Wh},$$

$$\text{Estimated autonomy} = \frac{17.28\ \text{Wh}}{3.16\ \text{W}} \approx 5.46\ \text{hours}.$$

This is an optimistic upper bound. In real scenarios, losses from voltage regulators and fluctuating loads (e.g., CPU/motor peaks) reduce autonomy. A more realistic value lies between **4.5 and 5 hours** under typical operating conditions. The uncertainty stems mainly from imprecise motor power characterization.

2.5 Summary

This chapter described the upgraded architecture of the swarm agent. The Raspberry Pi Zero 2 was selected for its compact size, energy efficiency, and sufficient computational power. Although limited to legacy Bluetooth and Wi-Fi, these are adequate for the intended applications.

The Raspberry Pi communicates with the Balboa via the I²C bus, chosen for its no extra hardware requirement and sufficient throughput. It now also handles real-time balancing, having taken over the IMU acquisition and control loop previously managed by the Balboa.

For localization, the Decawave DWM1001 UWB module was integrated, offering sub-10 cm accuracy. Due to GPIO limitations, it is interfaced with the Balboa via UART using the PANS API, and computed positions are forwarded to the Raspberry Pi over I²C. Calibration and filtering significantly improved accuracy: 90% of distance errors decreased by 79% (to max. 4 cm), and 90% of position errors by 47% (to max. 8 cm).

Finally, power consumption measurements show an average draw of 3 W during operation, leading to an estimated autonomy of 4.5–5 hours.

Chapter 3

Multi-agent communication

3.1 Introduction

Establishing reliable and efficient wireless communication within a swarm of mobile agents requires the design of a robust and well-adapted communication system. This chapter first compares possible solutions involving technologies explored in the literature review. Then, the chosen architecture and its implementation are detailed, followed by a performance analysis.

The swarm initially consists of six agents, with the option to scale up. Each agent communicates with a limited set of neighbors, forming a mesh topology where nodes represent agents and edges represent communication links. As agents are battery-powered, low consumption is needed, low latency is preferred to increase convergence rate while running iterative algorithms. For the Balboa's balancing ability, solution without additional hardware are preferred. Since the goal is to run distributed algorithms rather than exchange large data files, messages are typically small, just a few dozen of bytes

A core requirement in this context is decentralization: no central entity should manage communication. While this increases complexity and may reduce overall efficiency, it enables bio-inspired behaviors. Such systems are easy to simulate, but challenging to deploy in real-world conditions due to constraints like latency and synchronization.

Our approach begins with a low-level protocol, Bluetooth Classic, that offers compatibility, flexibility, and low energy consumption. This protocol is extended to support mesh and multi-process communication. On top of this, routing mechanisms such as flooding and unicast, and transport algorithms such as (a)synchronous messaging enable the construction of complex algorithms.

Finally, efficient swarm deployment is essential. But some communication protocols make this task harder than others. Firmware updates, agent-specific program, and data retrieval would otherwise be quite time-consuming. To address this, all agents connect to a central computer running dedicated deployment tools, then they are free to run without this computer.

3.2 Overview

In the literature review, existing routing protocols (e.g., BATMAN, Babel), some middleware (e.g., eProsima Fast RTPS (DDS)) and some full stack (e.g., BLE Mesh, ZigBee) communication protocols have been detailed, see Tables 1.3, 1.4, and 1.5. In this section, solutions based on these technologies as well as on other lower-level protocols (e.g., IR, Classic Bluetooth, WiFi, BLE), encountered in other multi-agents systems, are compared regarding this project's requirements.

3.2.1 Infrared (IR)

Infrared is not considered a suitable candidate for this project due to its very short range, very small throughput, sensitivity to ambient light, and strict line-of-sight requirements. However, its main advantage is its lightness and simplicity. That is why it is often used in swarms with at least hundreds of agents such as the well known Kilobot [6], but it is obviously not adapted with a ten agent swarm spread on several meters.

3.2.2 Zigbee

We have shown in the literature review that Zigbee offers very good efficiency, with low power consumption and latency. But there are three reasons that make it unsuitable for this project. First, it uses additional hardware, even though it is small, the balancing capability and the mobility of the robot may be negatively affected. Second, it is a full stack that do no include synchronous messaging, we could implement it on the top but it would be better to design it closer to the hardware. Finally, we have no control on the mesh topology, and the overhead due to the routing mechanism use energy for something we don't need.

3.2.3 WiFi

WiFi offers high bandwidth and long communication range, making it a solid candidate for data-intensive distributed systems. However, traditional router-based architectures introduce a central coordinator, which contradicts the decentralization objective of this project.

Ad-Hoc WiFi provides a decentralized communication model in which each Raspberry Pi broadcasts its own access point, forming a star-like topology. Although it can serve as a low-level base for a fully custom stack, this mode requires substantial configuration and development efforts. Notably, internet access must be bridged through a specific node, and deployment tools relying on centralized connectivity become incompatible, unless agents temporarily switch to a router-based mode (e.g., via physical interaction like a button press) to enable provisioning via a central device.

Due to these constraints, adopting an existing mesh routing protocol along with a middleware appears to be a more efficient solution, at the cost of no control on the mesh topology. Among the available mesh protocols, the literature review identifies BATMAN (Better Approach To Mobile Adhoc Networking) as one of the most effective options. It operates at Layer 2 (data link), creating a virtual interface `bat0` for mesh communication. IP addressing (Layer 3) can be handled either manually or dynamically using DHCP. On

top of this, synchronous communication mechanisms can be built using standard transport protocols (TCP/UDP) through Python sockets. Access to the routing table from Python is possible using system calls to the `batctl` utility. A middleware such as eProsima Fast DDS could alternatively be used to implement high-level communication features (Layers 5 to 7), including Quality of Service (QoS), message synchronization, and publish/subscribe semantics.

Importantly, BATMAN-based networks are generally easier to deploy than basic Ad-Hoc setups, as any computer can join the mesh as a node without specific configuration. A deployment tool, as described in Section 3.5 and Appendix C, can be implemented using SSH over the mesh network leveraging BATMAN IP addresses.

The following table 3.1 summarizes the imagined WiFi-based communication architecture and provides a comparison between the custom protocol stack and an equivalent implementation using eProsima DDS:

Table 3.1: Architecture of the synchronous messaging system built on top of BATMAN-adv, and optional mapping with eProsima DDS

Layer	**Custom middleware**	**eProsima DDS**
L7: Application	Custom framework for distributed iterative algorithms	DDS application logic
L6: Presentation	Serialization handled in Python (e.g., struct)	DDS serialization
L5: Session / Messaging	Synchronous messaging layer with retries, timeouts, routing-table access via `batctl`	Publish/Subscribe middleware with QoS policies, discovery, reliability
L4: Transport	Python sockets using UDP	
L3: Network	Static or dynamic IP addresses over `bat0`	
L2: Data Link	BATMAN-adv	
L1: Physical	Wi-Fi	

In conclusion, while WiFi brings substantial capabilities in terms of bandwidth, range, and mesh support through BATMAN, the not necessary routing algorithm that implies additional overhead, the limited control over the mesh topology and the relatively high energy consumption limit a bit its relevance for small, energy-efficient swarm with predefined topology. But it remains suitable for this project.

3.2.4 Bluetooth Low Energy (BLE)

While BLE Mesh is supported on dedicated MCU (such as Nordic's nRF52 family, often running Zephyr RTOS), it is not natively supported on Raspberry Pi using standard Linux Bluetooth stacks (e.g. BlueZ). Therefore, implementing BLE mesh on a swarm of Raspberry Pi would require each Raspberry Pi to be coupled with a separate BLE Mesh-capable MCU and a UART or SPI bridge, adding significant integration complexity and energy consumption.
For these reasons, although BLE Mesh offers excellent decentralized communication, it is currently impractical to use it directly on Raspberry Pi-based platforms without external hardware and its low-energy advantage is not longer applicable due to additional hardware requirement.

Bluetooth Low Energy (BLE) is a low-power wireless technology particularly suited for short-range communication and energy-constrained devices. Unlike Wi-Fi, it offers a low throughput and range but enables power-efficient and lightweight communication, which makes it attractive for swarm robotics or embedded sensing tasks. BLE relies on Generic Attribute Profile (GATT) at L5-7, which defines how devices expose and interact with structured data via services and characteristics.

However, GATT introduces several limitations when attempting to design a flexible, symmetric, and scalable communication system. It enforces a strict client/server architecture, where only the server (peripheral) can expose data and the client (central) can initiate transactions. This asymmetry prevents the design of a true peer-to-peer or multi-hop communication model, making it ill-suited for creating a decentralized mesh topology (BLE Mesh is based at the top of BLE's L2 so it does not use GATT but its own protocol). For these reasons, the BLE-based GATT stack is not retained for this project.

3.2.5 Bluetooth

Bluetooth Classic enables point-to-point (1:1) communication and supports multiple simultaneous connections, up to seven in theory. It was not designed for swarms with too many connections, but it could work with a swarm composed of thousand of agents with less than 7 connections each. Beside of that, it requires prior pairing and a connection delay applies at the start of a program.

The transport layer relies on RFCOMM protocol, which emulate a bidirectional serial port. To use it, Bluetooth sockets, available in Python, enable direct communication via a socket API, allowing each agent to act as both server and client using multi-threading. This makes it possible to create a physically decentralized mesh with predefined topology. The low-level nature of this protocol, along with its high compatibility and low energy consumption, provides great flexibility. However, this comes at the cost of increased development complexity. Indeed, communication layer from L3 to L7 should be developed as Bluetooth Classic is not initially made for mesh communication.

A key advantage of Bluetooth is its coexistence with WiFi, which remains available for internet connection without gateway, and deployment tasks such as individual access, data collection or firmware updates. Combined with its reliability and wide compatibility, this makes Bluetooth a suitable option for integration into the swarm platform.

Bluetooth Classic was chosen for communication due to its native peer-to-peer architecture, compatibility, and low overhead. A socket-based full stack was developed from scratch, avoiding heavy routing and middlewares protocols with their unnecessary features such as dynamic routing or QoS. This lightweight foundation provides decentralized control over message flow and enables the implementation of synchronous messaging patterns required by distributed iterative algorithms. It also allows for custom static routing schemes, giving full control over the mesh topology.

3.3 Communication architecture

The communication stack implemented in this project is structured in layers, each providing increasing levels of abstraction. The purpose is not to standardize it with the OSI model, but to make it flexible and efficient.

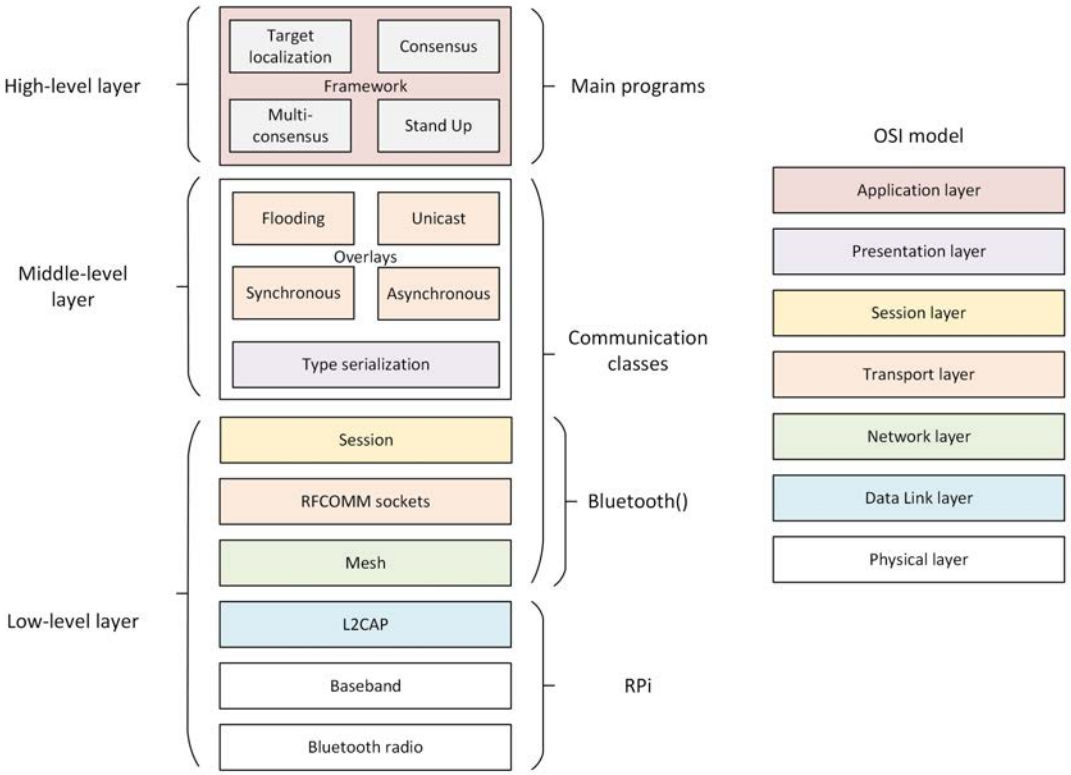


Figure 3.1: Protocol stack with OSI model correspondence.

3.3.1 Low-level layer

The lower-level layer of the system relies on Bluetooth Classic. A serial communication channel is formed by the layers from physical radio layer to the RFCOMM Python sockets.

Bluetooth radio and baseband

The Bluetooth radio layer (L1) handles the wireless transmission over the 2.4 GHz ISM band using frequency hopping spread spectrum (FHSS), providing robustness against interference. The baseband manages device discovery, connection establishment, and low-level packet framing.

L2CAP

L2CAP (Logical Link Control and Adaptation Protocol) sits above the baseband and provides multiplexing of data streams, segmentation and reassembly (default: 1021 bytes / packet on RPi). While L2CAP is capable of handling multiple logical channels, in this project a single channel per connection is used. In order to use simultaneous processes anyway, a multi-process feature has been designed on L3-6 although it may be less efficient than L2 multi-channel. This is described in Section 3.4.2

Mesh

Bluetooth Classic does not natively support mesh networking. Thus, a static mesh topology has been designed at a higher abstraction level (L3) by manually managing connections between nodes using several RFCOMM sockets. Each agent establishes and maintains connections with its neighbors according to a predefined adjacency matrix.

RFCOMM sockets

On top of Bluetooth Classic, Python RFCOMM sockets are used to form bidirectional links within the mesh. Each connection allows simultaneous sending and receiving of messages between two nodes.

Session

Within the `Bluetooth()` class, a mechanism that establish all predefined links within the mesh is called at the each start of program. It blocks the program while all link have not yet been established. This mechanism is detailed in Section 3.4.1.

3.3.2 Middle-level layer

The low-level layer operates with binary data, while the middle level layer uses interpretable data thanks to a type serialization mechanism. Several communication overlays have been implemented to support various coordination and routing mechanisms.

Type serialization

This mechanism is the link between the low-level layer and the middle-level layer. It translates binary data into overlay-specific interpretable data structures. It is described thoroughly in Section 3.4.2.

Overlays

Although these overlays handle transport-layer functions (such as delivery and synchronization), they are implemented above the session and presentation layers. This departs from strict OSI layering but brings significant benefits in terms of flexibility and efficiency. By working directly on interpretable data structures, overlays are easier to design, adapt, and debug. In this setup, a single Bluetooth instance is shared across all overlays. If they followed the OSI model strictly, being placed below the session layer, each overlay would have to be instantiated before the connection, leading to unnecessary memory use and higher energy consumption.

This design allows each overlay to be used as a modular black box, improving modularity without the cost of strict OSI compliance. The overlays are listed below and detailed in Sections 3.4.3 to 3.4.6. They are briefly introduced hereafter:

- **Synchronous**: where agents are able to run distributed iterative algorithms. To do so, several synchronization waiting loops using message overhead have been designed.
- **Asynchronous**: where agents react to incoming periodic messages in applications such that the timing and message order is not important.
- **Flooding**: a broadcast-based dissemination protocol with acknowledgment back-propagation.
- **Multi-hop unicast**: which enables message forwarding across multiple agents to reach a distant destination through the shortest path.

3.3.3 High-level layer: Framework

At the top of the stack lie the user coding interface to interact with the communication stack with the highest abstraction level. Some distributed algorithms and applications have been implemented on it, they are detailed in Chapter 4. They are briefly introduced hereafter:

- **Consensus**: Agents agreeing on a state, based on an iterative algorithm.
- **Multi-consensus**: Double consensus. In this case, they are corresponding to the phase and frequency of the blinking LEDs of each agent. This is for highlighting the developed multi-processing feature.
- **Target localization**: Cooperative estimation of a target's position using a distributed iterative algorithm based on gradient descent. Each agent contributes by measuring its own position and estimating its distance to the target through UWB multilateration modules.
- **Stand Up**: Enables asynchronous collective behavior. Each agent stands up as soon as at least one of its neighbors has already done so.

3.4 Implementation

The whole implementation of the low and the middle-level communication layer are presented in this section. An overview of all communication-related classes, their relationships and their applications is illustrated in the block diagram shown in Figure 3.2.

The software architecture is divided into three main categories: communication, applications and peripheral management. Communication-related classes will be presented in this section, applications will be analyzed in detail in the next Chapter, and peripheral management implementation was presented in Chapter 2.

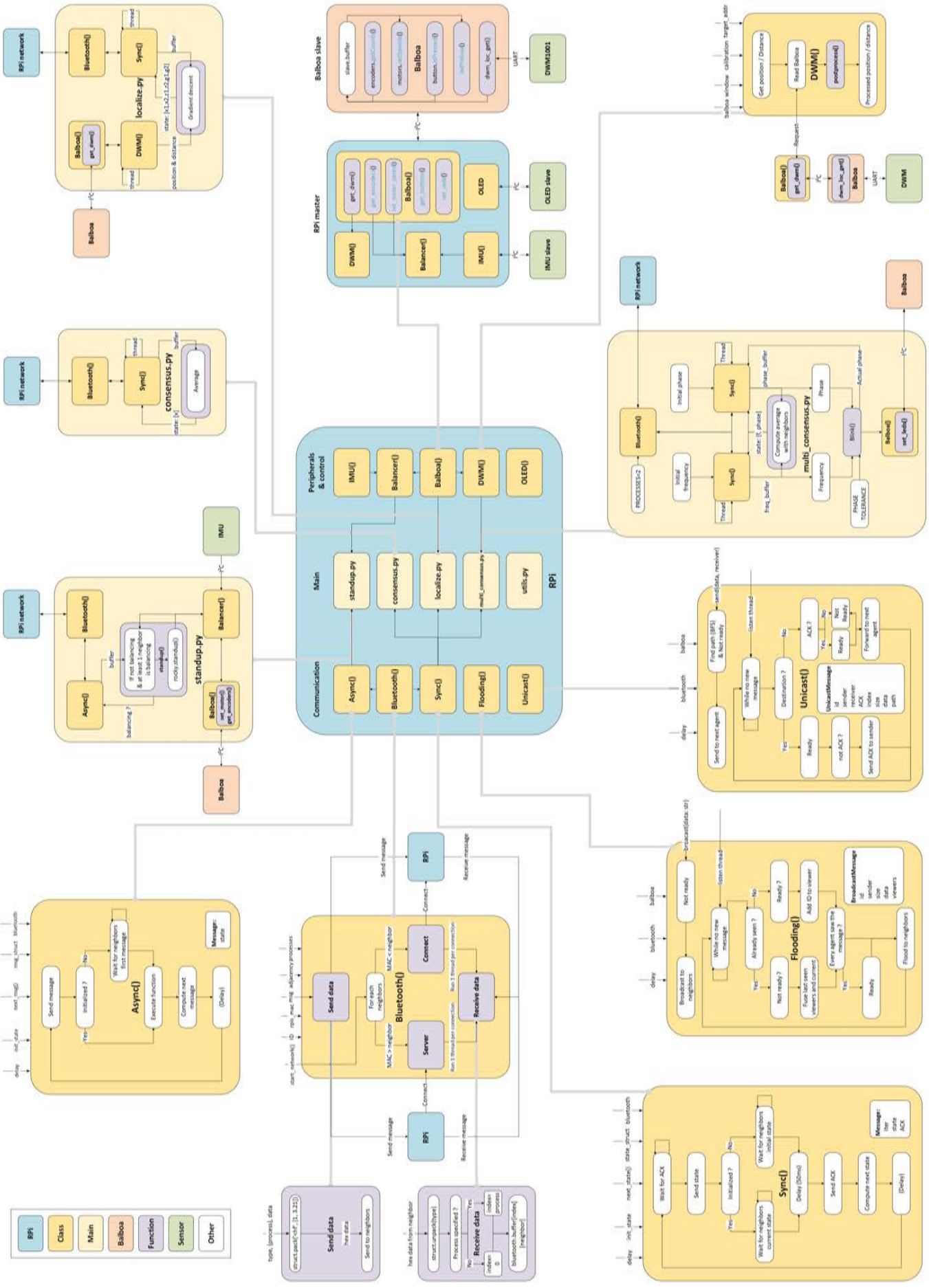


Figure 3.2: Block diagram of the multi-agent communication system. The whole source code is available in this project's GitHub.

3.4.1 Mesh architecture and session management

This section presents the `Bluetooth()` class, which spans from the *mesh* layer (L3) to the *session* layer (L5). Its structural design is depicted in Figure 3.3.

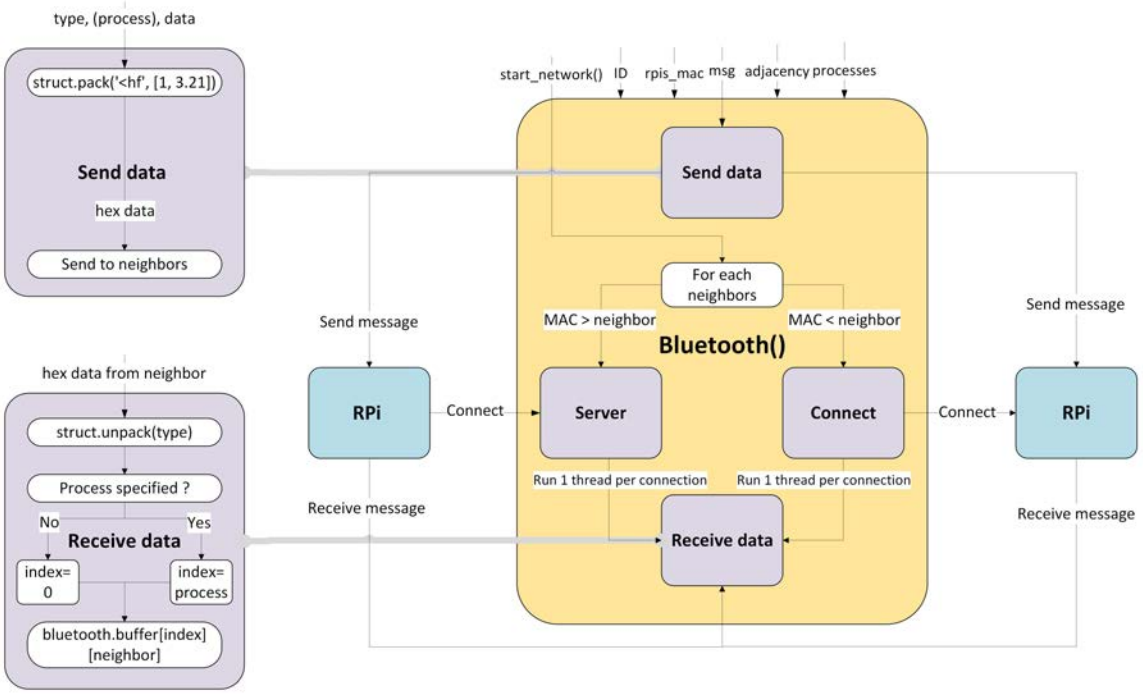
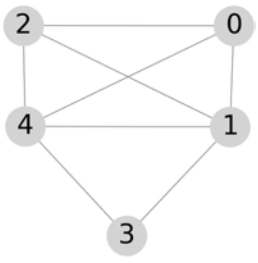


Figure 3.3: Bluetooth class schema block: Connection and communication system. The start_network() function establishes all connections, as detailed in 3.5. Once a message is received, it is processed and stored in Receive data. Message sent are serialized in Send data using the specified data structure.

The mesh is initialized according to a predefined topology represented by a symmetric binary adjacency matrix, where $A_{ij} = 1$ indicates a link between nodes i and j. Each node corresponds to an agent, and each link represents a connection in the communication graph, as illustrated in Figure 3.4.



	0	1	2	3	4
0	1	1	1	0	1
1	1	1	1	1	1
2	1	1	1	0	1
3	0	1	0	1	1
4	1	1	1	1	0

Figure 3.4: Example of a graph and the corresponding adjacency matrix.

Each device is identified by both its MAC address and a corresponding assigned ID. A single instance of the `Bluetooth()` class can serve multiple use cases simultaneously by specifying it in the message, thanks to its developed support for multi-processing. The protocol involves two main stages: connection and communication.

Connection

Before communication can take place, each agent must establish a connection with each of its neighbors. A design constraint here is that simultaneous connection attempts between two devices will fail. To address this, a mechanism ensures that only one side of each link initiates the connection. Specifically, the device with the lower MAC address always initiates the connection. This approach leverages the only shared and unique information between all nodes: their MAC addresses. This mechanism is represented in Figures 3.3 and 3.5.

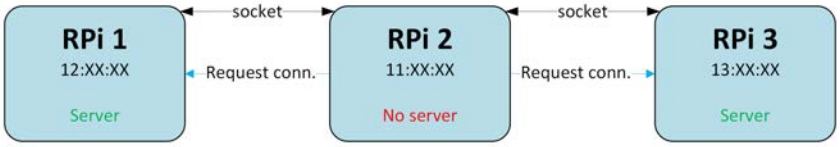


Figure 3.5: Schema block of connection step with the single side connection attempt mechanism.

Communication

Once a connection is established, each agent spawns a dedicated thread to listen in server mode for each connected neighbor. While a traditional publish/subscribe mechanism typically operates at Layer 7, in this project the messaging do not need that flexibility: each agent communicate with its fixed neighbors. This has been implemented directly at Layer 3, remaining closer to the hardware. The implemented system uses a mechanism in which incoming messages are stored in buffers associated with the sender's ID and the designated buffer ID. This structure is illustrated in Figure 3.6.

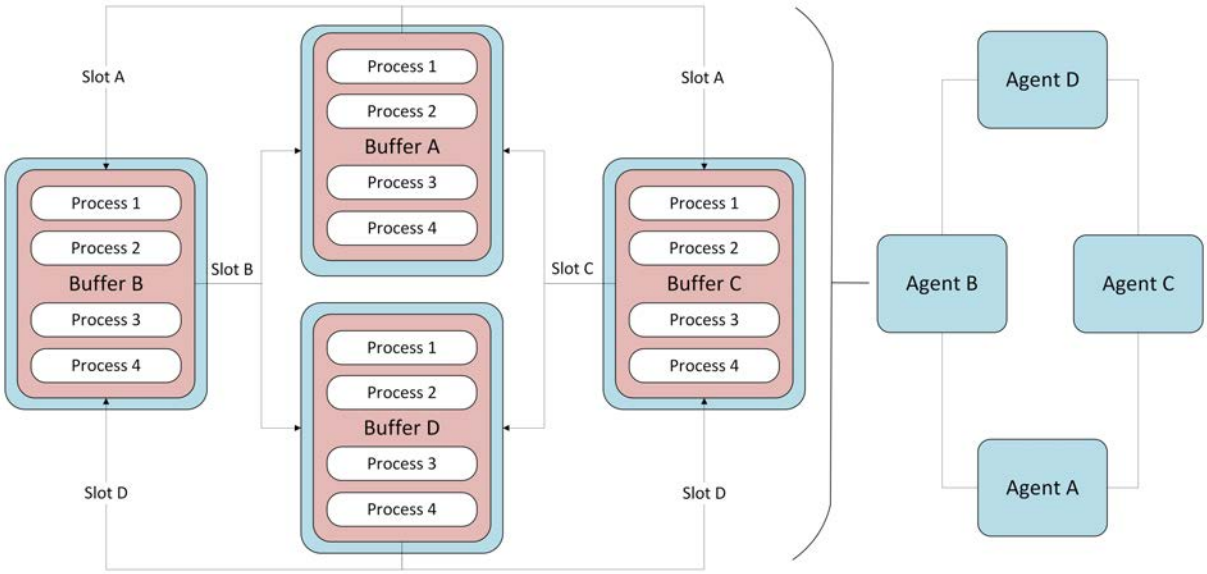


Figure 3.6: Bidirectional multi-process communication mechanism. Each agent publish data in its buffer at the corresponding process slot, which will be sent to the neighbors in the buffer's slot corresponding to the sender ID.

Session management

The program waits until the agent is connected to all its neighbors before allowing any message exchange. To ensure no message is lost at the start of an algorithm with periodic exchanges, an additional synchronization mechanism is used: the algorithm only begins once a first message has been received from each neighbor. This guarantees that all agents operate their first iteration simultaneously in a distributed architecture, even if some take longer to become fully ready. This is observable on Figure 3.11: agents are not ready at the same time but they all begin their first iteration together.

3.4.2 Data serialization

This section presents the designed message data structure. The binary low-level format has been chosen to be generic, offering flexibility for upper-layer processing. A dedicated serialization mechanism bridges the low-level and middle-level layers. Data serialization mechanism is responsible for extracting, decoding, and organizing the content from raw binary messages into interpretable data structures. The middle-level data structure is optimized for compactness to minimize overhead during transmission. An overview of the complete structure is shown in Figure 3.7.

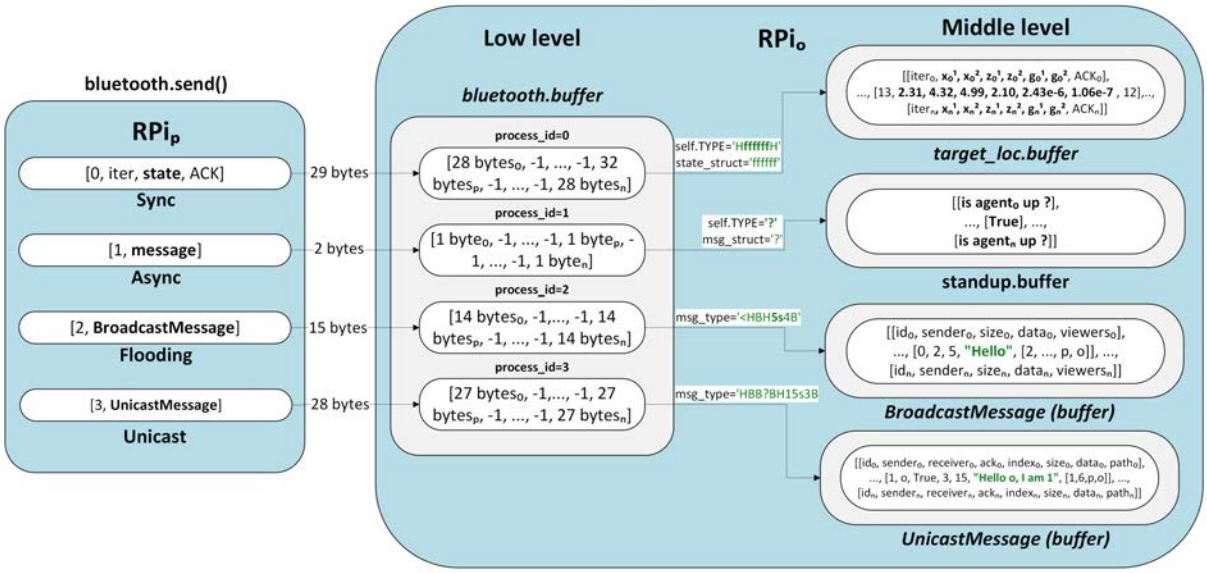


Figure 3.7: Data structure used in the multi-agent communication system. In this example, four processes are encoded into each message to help the receiver identify them. The network consists of n agents. Here, RPI_o has three neighbors: 0, p, and n.

The low-level reception buffer is structured as a two-dimensional array of size: `[number of processes] × [number of agents]`. When a message is received, it is stored in its raw byte form in the corresponding slot based on both the process ID (encoded in the first byte of the message) and the sender's ID. Once this indexing is done, the process ID is no longer stored in the buffer, as it has already served its purpose for sorting. Since the process is known, the data type and structure can be determined, allowing proper decoding. The data serialization is based on the python `struct` library [53].

- **Async** only includes the agent state. It is detailed in Section 3.4.3.
- **Sync** includes two additional flags: `iter` and `ACK`, two unsigned integers that help maintain optimal temporal synchronization during iterative algorithms. Their function is detailed in Section 3.4.4.
- **Routing overlays** Routing-based overlays rely on more extensive data structures with multiple flags. These are grouped into a Python `NamedTuple` for clarity and structure. Details of their implementation and purpose are discussed in Sections 3.4.5 and 3.4.6.

3.4.3 Asynchronous Communication

This messaging protocol is designed for simple, event-independent communication between agents. At each iteration, it reads the most recently received state from each neighbor, computes a new state using a user-defined function, and finally sends its updated state. This structure is suitable for algorithms that do not rely on strict message ordering or precise timing. A typical example is the Stand Up application, which will be presented in details in the next chapter. An overview of the class behavior is shown in Figure 3.8.

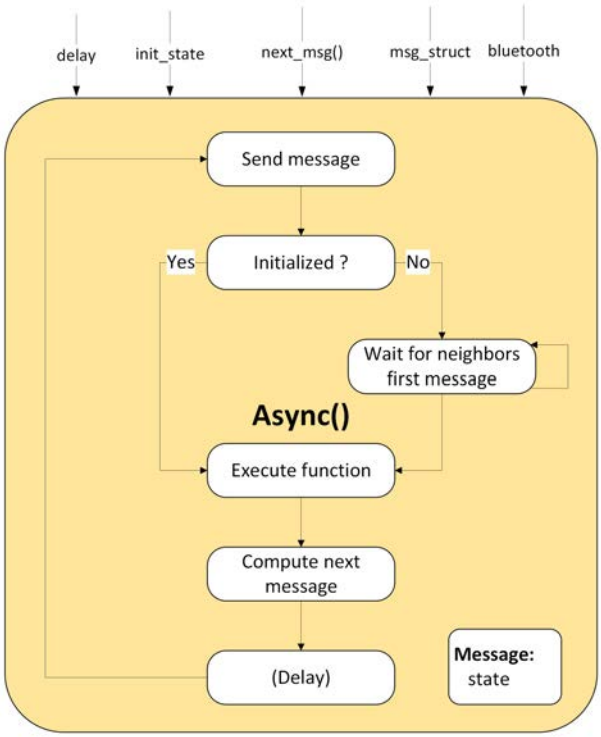


Figure 3.8: Simplified structure of the asynchronous communication class. The user must specify the arguments on top. `next_msg()` is a function executed at each iteration, it must also return the next state. We can use any state structure by specifying it in `msg_struct`.

A mechanism ensures that the algorithm only begins once a first message has been received from all neighbors and all connections are established. Next, a loop updates the state. Figure 3.9 shows the resulting iteration evolution, agents are allowed to wait longer than other and they do not wait for other agents at each iteration.

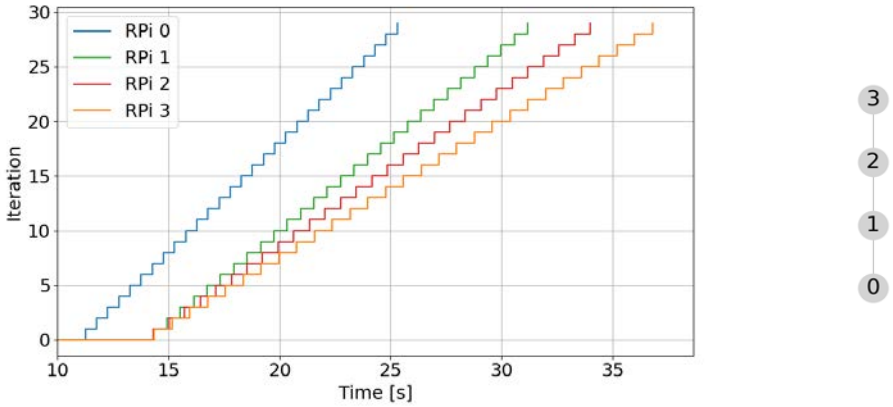


Figure 3.9: Iteration progress over time for four Raspberry Pi devices executing an asynchronous algorithm. Agent 0 started earlier, despite the loop waiting for each agent to receive its first message. This behavior happens sometimes and is expected. Agent 0 began execution as soon as it has received a message from its only neighbor, agent 1. The start time of agents 1, 2, and 3 corresponds to the moment when agent 2 sent its first message to agents 1 and 3. Agent 3 sent its first message before agent 2.

3.4.4 Synchronous communication

This complex messaging protocol is designed to support distributed iterative algorithms. Unlike asynchronous exchanges, synchronous communication ensures that agents proceed in lockstep, making it well-suited for algorithms requiring strict message ordering and iteration control. A simplified overview of the class behavior is shown in Figure 3.10.

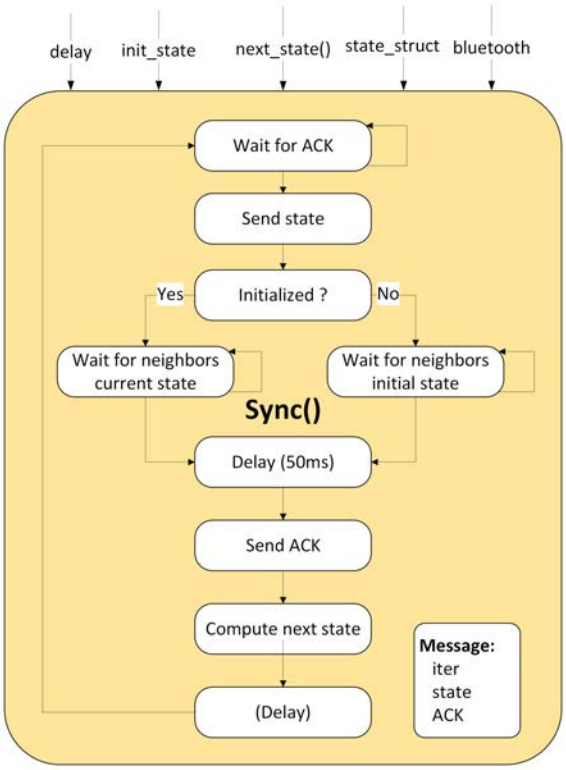


Figure 3.10: Synchronous class

The synchronous overlay shares a similar structure with the asynchronous one, but introduces two additional synchronization loops, made possible by the use of two extra flags in the exchanged messages. This overlay is said to be pseudo-optimal as it acts as soon as it can thanks to the three synchronization waiting loops, and the only delay that cannot be replaced by a loop was carefully chosen.

- Wait for neighbors current states (`iter`): The next state cannot be computed until the current states of all neighbors are received. Each message includes the iteration, allowing the system to verify that messages correspond to the expected iteration.

- Wait for acknowledgment (`ACK`): Before an agent can send its next state, it must ensure that all neighbors have read the last one. This is achieved through an acknowledgment mechanism, where agents send the iteration of the last read message.

This acknowledgment mechanism introduces a latency penalty, as a minimal delay of 50 ms is required after each ACK to avoid sending the next message too quickly, which would risk disrupting the Bluetooth connection. This delay corresponds to the half maximum RTT for same-size packets margin for security (see Section 3.6.2).

Experimental results are shown in Figure 3.11. A device can only transmit its next state after receiving all acknowledgments, and no subsequent state transition can occur before the corresponding acknowledgment has been received. Beside of that, we observe on the results that the states are successfully synchronized and the time for 30 iterations in that configuration is approximately 9 seconds, giving approximately 3 iteration per second.

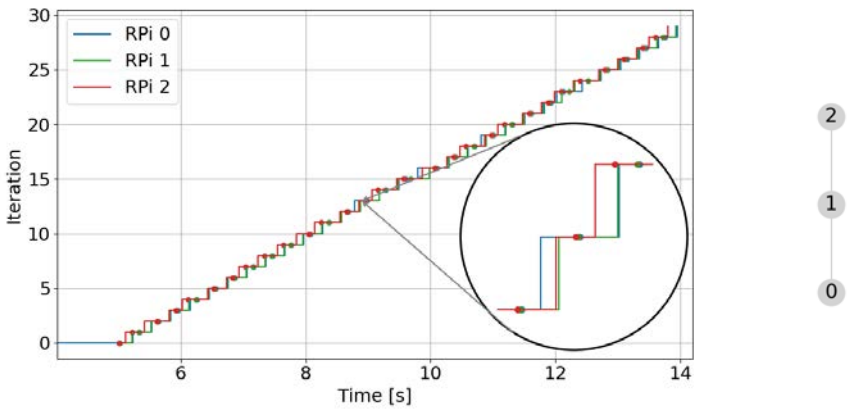


Figure 3.11: Current iteration states over time for three Raspberry Pi devices executing a synchronous-based algorithm. Solid lines represent the evolution of each agent's state, while dots indicate the moments when all neighbors have acknowledged the current state.

3.4.5 Flooding

Flooding is a simple and robust broadcast algorithm. A sender broadcasts a message, including a list of viewers, to all its neighbors. Each receiver checks whether the message is new or the viewer list has changed. If so, it forwards the message. This process continues until all nodes have received the message and the viewer list is complete. An overview of the class is shown in Figure 3.12.

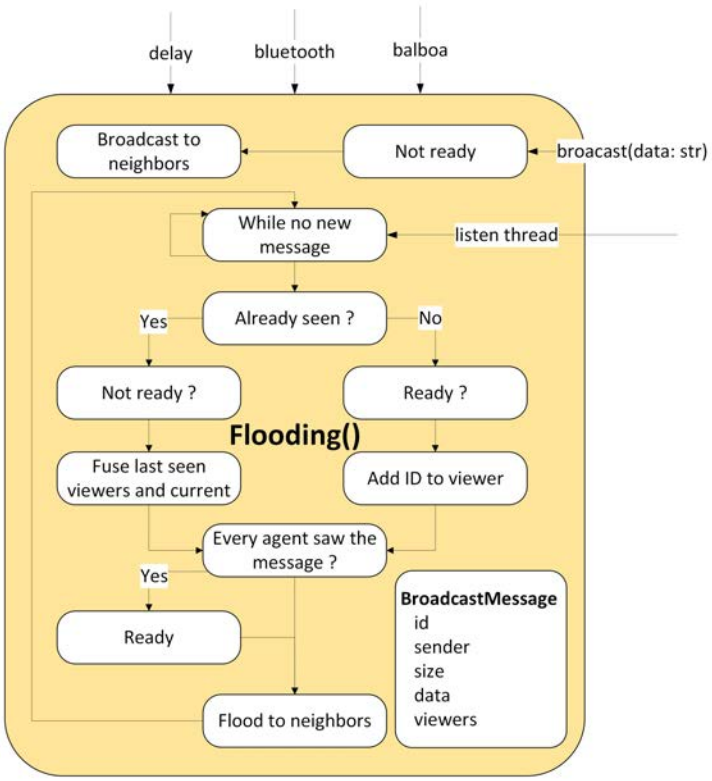


Figure 3.12: Flooding class block diagram, including message structure.

Upon receiving a new message (identified by its ID), an agent becomes busy and waits until all other agents have seen the message before returning to the ready state. Experimental results are shown in Figures 3.13 and 3.14. We observe that it takes between 1.5 and 2 seconds to flood a linear network of 6 agents, and that many redundant messages are sent, which is common in flooding algorithms as discussed in the literature review.

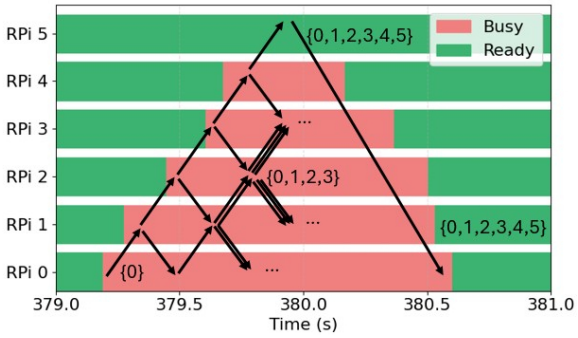


Figure 3.13: Flooding initiated by RPi 0. Each message is forwarded to all neighbors until the viewer set is complete. Next, agents return to the ready state after one final broadcast. A video demonstration is available here

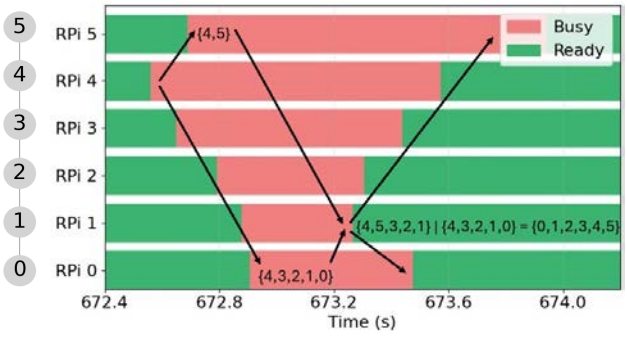


Figure 3.14: Flooding initiated by RPi 4. RPi 1 merges the viewer sets and is the first to become ready. Other agents follow either through further merging or via RPi 1's retransmission.

3.4.6 Multi-hop Unicast

This overlay implements a routing mechanism to send a message from a sender to a single, specific receiver by following the shortest path in the mesh network. The sender determines this path using a Breadth-First Search (BFS) algorithm. A simplified overview of the class behavior is shown in Figure 3.15.

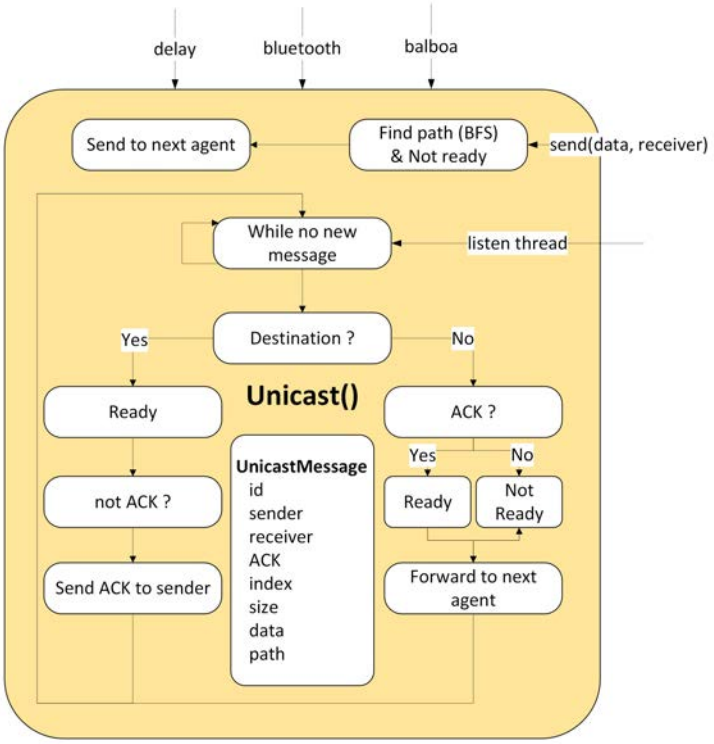


Figure 3.15: Multi-Hop Unicast class schema block, including the structure of the messages

Algorithm 2 Breadth-First Search (BFS) for shortest path

Require: Graph $G = (V, E)$, start node s, goal node g
Ensure: Shortest path from s to g, if one exists
  Initialize queue with $(s, [s])$
  Initialize visited set as empty
  while queue is not empty **do**
    Pop $(current, path)$ from queue
    if $current = g$ **then**
      return *path*
    end if
    Add *current* to visited
    for all neighbors of *current* in G **do**
      if neighbor not in visited **then**
        Append $(neighbor, path + [neighbor])$ to queue
      end if
    end for
  end while
  return No path found

BFS is well-suited for this application because it finds the shortest path in terms of number of hops in an unweighted graph. Unlike Depth-First Search (DFS), which explores paths as deeply as possible before backtracking, BFS guarantees the shortest path if one exists. In more advanced mesh routing protocols like BATMAN, link quality is used as a metric to determine optimal paths. In that case, we can use algorithms such as Dijkstra, which allow weighting the nodes. The pseudo code of BFS algorithm is shown on Algorithm 2.

Experimental results are shown in Figure 3.16. Each intermediate node along the route forwards the message to the next hop and becomes temporarily busy, waiting for an acknowledgment. Once the message reaches the destination, an acknowledgment is sent back to the sender, traversing the same path in reverse. Each node switches back to a ready state upon receiving this acknowledgment, allowing new messages to be processed. Unicast is obviously faster than flooding and a minimal number of messages are sent. For a sending, sender and receiver send 1 message and intermediate nodes send only 2 messages.

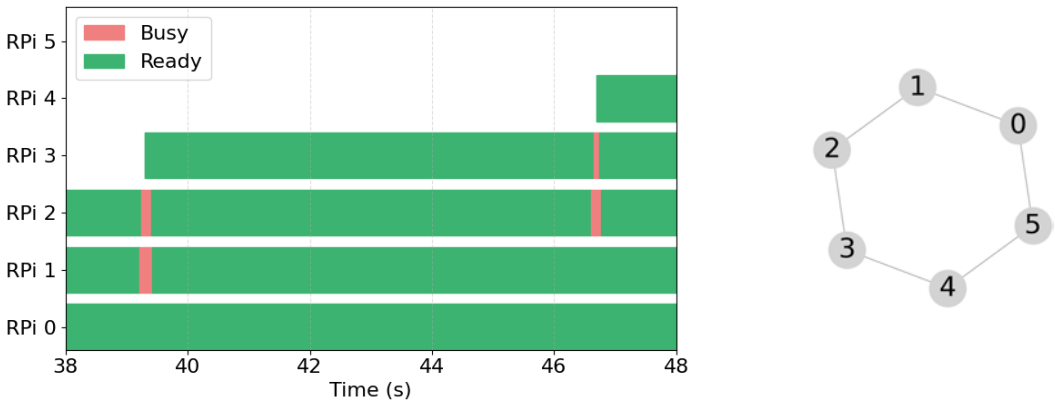


Figure 3.16: First, RPi 1 send a message to RPi 3. The graphs circular, the message can either go through 1-0-5-4-3 or through 1-2-3. RPi 1 choose the shorter way with BFS. Secondly, RPi 2 send a message to RPi 4. A video demonstration is available here.

3.5 Deployment

An important aspect that has already been introduced is the deployment of the swarm. Without dedicated tools, interacting with a group of agents can quickly become tedious. To address this, several scripts have been developed to automate deployment tasks, tailored specifically to the architecture and configuration of this project. The choice of Bluetooth for communication has been complemented by WiFi connectivity, which allows all agents to connect to a common access point (AP) and thus to be accessed collectively from a central laptop. This dual-network approach enables efficient remote management of the swarm through a centralized server.

As illustrated in Figure 3.17, a wireless AP can be configured on a mobile phone, a laptop, or a dedicated router. Once the AP is available, each Raspberry Pi automatically connects to it and becomes accessible through SSH from any device on the same network. A collection of deployment scripts have been developed. They run on the laptop and enable interaction with the swarm through the following actions:

- **Firmware update**: Synchronizes the entire project directory on selected Raspberry Pi devices with the local version on the laptop.
- **Program execution**: Launches specific Python programs on the selected Raspberry Pis, with the ability to pass individual arguments based on the RPi ID. Each RPi runs the code within a pre-configured Python virtual environment that includes all required libraries, including those for hardware interaction (e.g., CircuitPython).
- **Data retrieval**: During program execution, each RPi logs data locally. Afterward, the laptop can retrieve and centralize all files from selected agents.
- **General command execution**: Enables the user to send any terminal command to a selected subset of RPis.

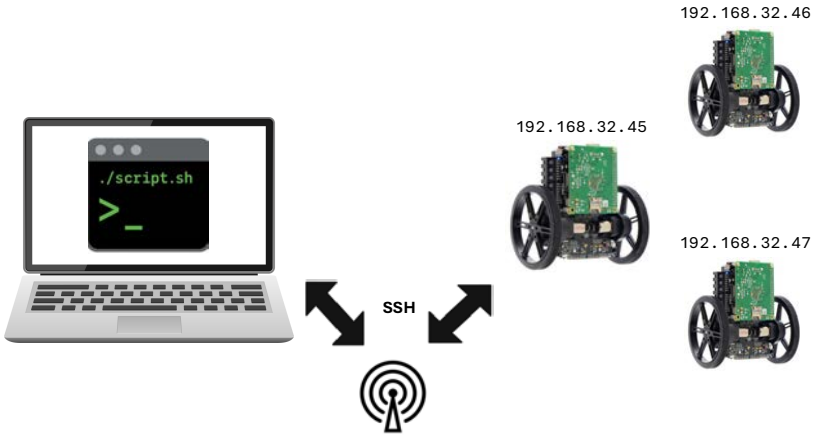


Figure 3.17: Deployment setup for interacting with the whole swarm automatically. The source code is available on this project's GitHub.

3.6 Performance analysis of the low-level layer

3.6.1 Maximum data size

While Bluetooth Classic handles low-level message fragmentation and reassembly, practical limitations arise from the message handling in the transport layer (L3). On Raspberry Pi, the baseband layer supports packets up to 1021 bytes by default, defined by the MTU (Maximum Transmission Unit). If a message exceeds this size, the L2CAP layer fragments and reassembles it automatically. However, the RFCOMM socket on top of L2CAP imposes another limit: `recv()` operations are capped at 1024 bytes. When sending larger messages, they are split into multiple chunks, and the receiver must manually reassemble them in Python.

Future improvements could involve implementing custom fragmentation and reassembly to support payloads larger than 1024 bytes. This would offer greater flexibility but would also add complexity, overhead, and latency, as fragmented messages are sent sequentially over a single channel.

3.6.2 Latency and Throughput

The latency of a single message is evaluated by sending a timestamp to one agent, which then sends it back to the sender. The RTT (Round-Trip Time) is measured based on the time at which the sender receives its message back. This measurement is therefore independent of clock differences between agents. The source code of the measurement program is available on this project's GitHub.

In Bluetooth Classic, as long as the packet size remains below the baseband MTU (1021 bytes), no fragmentation occurs. Consequently, latency is expected to be relatively independent of packet size within this limit. For larger messages, reassembly above the RFCOMM layer is not automatically handled, making such measurements unreliable. However, if reassembly were implemented, we would expect latency to increase proportionally with the number of fragments.

Assuming single-packet transmissions, the throughput $T(s)$ can be estimated as:

$$T(s) = \frac{s}{\frac{\texttt{RTT(s)} - \texttt{Reply Time}}{2}} \approx \frac{2 \cdot s}{\texttt{RTT(s)}} \quad [\text{bytes/s}], \tag{3.1}$$

where T is the measured latency for a packet of size s, and `Reply Time` is the time taken by the receiver to send back the message. We will neglect `Reply Time` as it is only a few ms. Due to the overhead of the RFCOMM socket layer, the goodput (i.e., the amount of useful application-level data) is approximately 70% of T.

Figure 3.19 shows the measured latency and the corresponding throughput. We observe that latency is dominated by a fixed component and only slightly affected by packet size (as long as no fragmentation occurs). The maximum throughput is achieved when the packet size approaches the MTU.

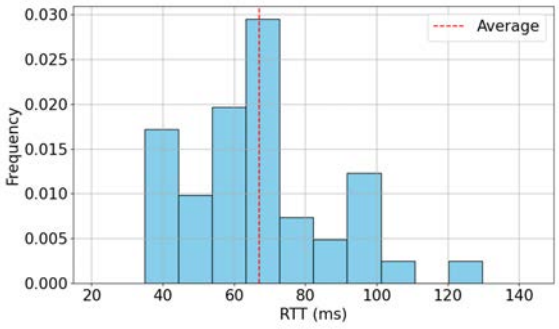


Figure 3.18: Latency (RTT) measurement histogram, there are 43 RTT measurements of 200 bytes packets. The mean RTT is 67 ms and the standard deviation is 21 ms.

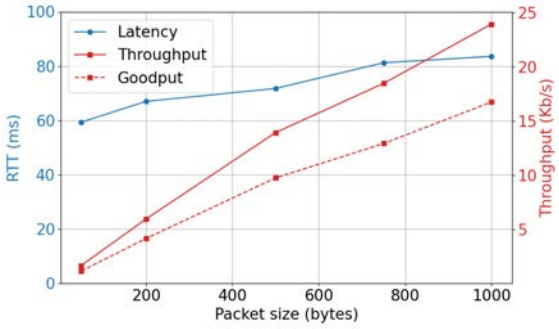


Figure 3.19: Latency measurement for several packet sizes (size < MTU), along with the resulting throughput and goodput.

The actual goodput G at the application level depends on the overlay used at the middleware layer, due to protocol-specific overhead. Let *size* denote the useful payload (in bytes), and N the number of agents in the mesh. Based on the payload size given in Figure 3.7, the estimated goodput for each overlay is given by:

- **Asynchronous**: $G \approx 0.7 \cdot \frac{size}{size+1} \cdot T$,
- **Synchronous**: $G \approx 0.7 \cdot \frac{size}{size+5} \cdot T$,
- **Flooding**: $G \approx 0.7 \cdot \frac{size}{size+6+N} \cdot T$,
- **Unicast**: $G \approx 0.7 \cdot \frac{size}{size+9+N} \cdot T$.

These formulas reflect how the relative size of control overhead impacts the efficiency of message delivery, depending on the chosen communication overlay. As expected, overlays involving routing (such as Flooding and Unicast) introduce higher overhead, reducing the effective bandwidth available.

3.7 Summary

Several communication technologies (Wi-Fi, IR, Zigbee, Bluetooth, BLE, etc.), along with their respective routing protocols (e.g., BATMAN, Babel) and middleware solutions (e.g., eProsima, MQTT), were reviewed in the literature. In this chapter, the selection of Bluetooth Classic was justified by comparing it with an alternative WiFi-based design. Combined with lightweight RFCOMM sockets, this solution offers a compelling trade-off between energy efficiency, system flexibility, and compatibility with the Raspberry Pi Zero 2. While this choice significantly increased the development complexity, it enabled full control over the architecture, from mesh topology management to application-layer integration.

To support synchronous distributed algorithms such as gradient descent, a dedicated messaging overlay was developed. It enables low-latency, fully decentralized iterations between agents. Additional overlays were implemented to support broader communication needs: the flooding overlay ensures reliable message broadcasting through acknowledgments, while the multi-hop unicast overlay enables targeted routing via a BFS algorithm. These mechanisms have been thoroughly tested and are now available as modular black-box components, ready to be reused in various application scenarios.

While the system imposes some constraints, such as a limit of seven direct Bluetooth connections per agent and the need for a predefined mesh topology, it offers full control over network structure and message flow. Although dynamic reconfiguration is not currently supported, it could be achieved through the flooding overlay if needed.

Ultimately, this communication stack lays a robust and energy-efficient foundation for a wide range of distributed applications, and proves that lightweight, custom-designed protocols can rival more complex solutions when carefully tailored to the system's constraints and goals.

Chapter 4

Applications

4.1 Introduction

The developed stack is effective and can be used over several applications. Four applications are presented in this chapter and their results are analyzed. The final application about the synchronous messaging-based target localization system make use of the UWB localization module with the designed post-processing techniques. It also make use of the controller, and the I²C communication interface between the robot and the Raspberry Pi. The source code for all simulations and experiments is available in this project's GitHub, and several applications are demonstrated in the accompanying video recordings, available at this link. As a reminder, the following classes are involved in these applications:

- `Bluetooth()` handles the low-level communication. It manages the sockets that define the mesh structure and is responsible for sending and receiving binary messages between agents.

- `Async()` manages a type of middle-level messaging logic. It decodes the received binary messages from `Bluetooth()`, based on the application-specific data structure, and computes the new state using the user-defined function.

- `Sync()` manages a type of middle-level messaging logic. It decodes the received binary messages from `Bluetooth()`, based on the application-specific data structure, and is capable of executing synchronous iterative algorithms on the distributed system.

- `Balancer()` implements the control logic required for the Balboa robot to balance autonomously.

- `Balboa()` acts as an I²C interface between the Raspberry Pi and the robot. It sends commands to the Balboa and request information.

- `DWM()` is an interface for the Decawave. It asks the Balboa for its measurements, and it applies some post-processing techniques on it.

4.2 Asynchronous

This section presents an application based on the asynchronous messaging. Asynchronous communication has been detailed in the previous chapter, it allows agents to update their states based on the latest data received from neighbors, without requiring strict synchronization. Each asynchronous message contains:

- Process ID: 1 byte,
- Current state: size depends on the application.

4.2.1 Stand-up

This application instructs each robot to stand up if at least one of its neighbors is already balancing. The behavior propagates through the network like a line of dominoes.

Implementation

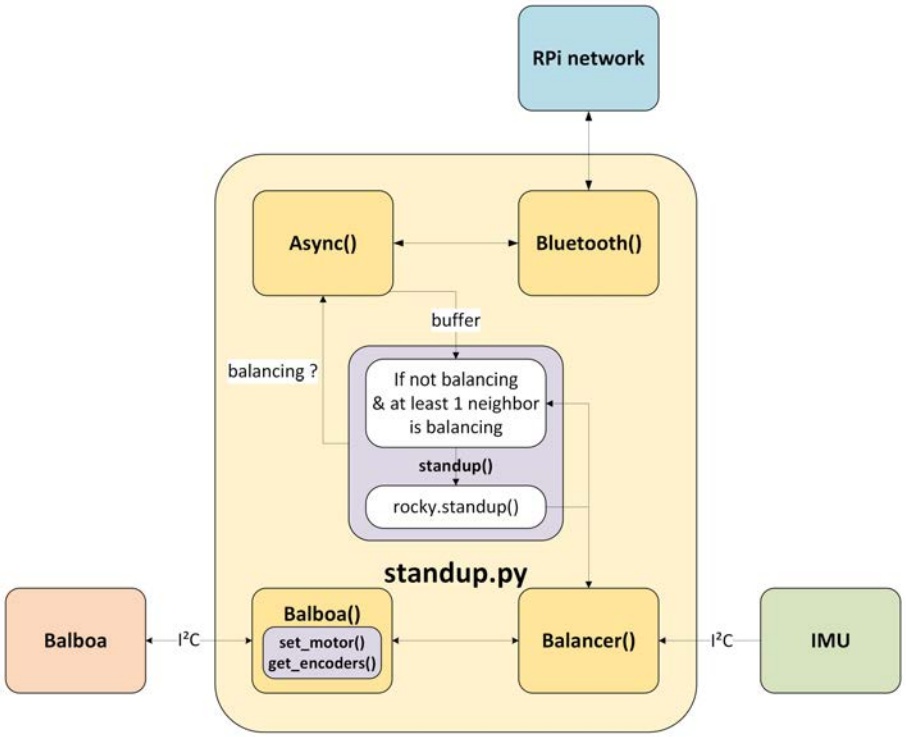


Figure 4.1: Stand-up application. The `Async()` messaging layer updates its mid-level buffer based on the low-level buffer of the `Bluetooth()` interface, via the serialization layer. The `Balancer()` controller commands the Balboa to stand up through the `Balboa()` interface, based on the application-specific `standup()` function.

A simplified overview of the implementation is shown in Figure 4.1. The `standup()` function determines whether the robot should stand up, based on its internal state and the received states of its neighbors. In this application, the state consists of a single boolean (1 byte) indicating whether the robot is currently balancing. The total message size is therefore 2 bytes.

As introduced in Section 3.6.2, the goodput of the asynchronous overlay can be estimated as:

$$G = 0.7 \cdot \frac{1}{2} \cdot T = 0.35 \cdot T,$$

where T is the total throughput. This means that approximately 35% of the available bandwidth is effectively used to transmit useful data in this application.

Results

Figure 4.2 shows the evolution of the balancing state over time. Once one Raspberry Pi stands up, the others follow after a fixed delay. The delay may vary slightly depending on when each agent receives the information and its current step in the program.

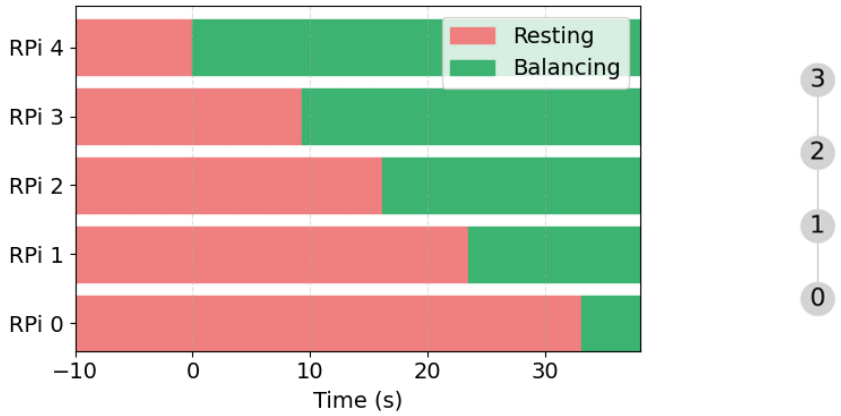


Figure 4.2: State evolution during the stand-up application. RPi 4 is manually commanded to stand up. A delay of 6 seconds is set between each iteration. Video demonstrations of the system are available at this link.

4.3 Synchronous

Synchronous communication enforces step-by-step coordination among agents through strict timing and message order, making it essential for distributed algorithms requiring temporally aligned updates. As a reminder, the data structure of sent messages for the synchronous overlay is the following:

- Process ID: unsigned char (1 byte),
- Current iteration: unsigned short (2 bytes),
- Current state: size depending on application,
- Acknowledgment: unsigned short (2 bytes).

4.3.1 Consensus

This first application allows a group of agents to reach agreement over a shared variable using iterative averaging with their neighbors, inspired by Reza et al. (2007) [54]. Each agent starts with a local value and updates it at each step using the values received from its neighbors. The process converges to a value close to the average of all initial values, and exactly equal to it if the graph is fully connected. The update rule is defined as follows:

$$x_i^{t+1} = \frac{1}{|\mathcal{N}_i| + 1} \sum_{j \in \mathcal{N}_i \cup \{i\}} x_j^t, \tag{4.1}$$

where x_i^t is the state of agent i at iteration t, and $\mathcal{N}_i$ is the set of neighbors of agent i.

Implementation

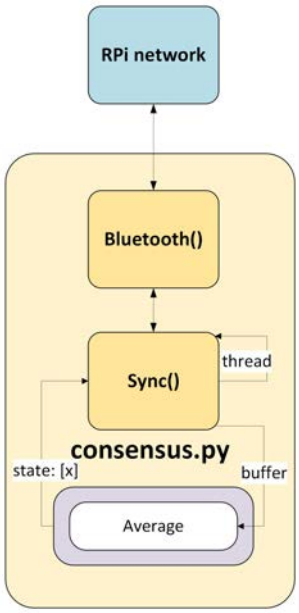


Figure 4.3: Block diagram of the consensus application. The `Sync()` block updates the middle-level buffer based on the low-level buffer of the `Bluetooth()` interface using the serialization layer. Based on this data, the next state is computed by the `Average` block implementing formula 4.1.

The state in this application is represented as a single `float` variable (4 bytes). Thus, the size of each message is 9 bytes. As introduced in Section 3.6.2, the goodput of the asynchronous messaging can be estimated by:

$$G = 0.7 \cdot \frac{4}{9} \cdot T = 0.44 \cdot T,$$

where T is the total throughput. In other words, approximately 44% of the available throughput is effectively used to transmit useful data.

Results

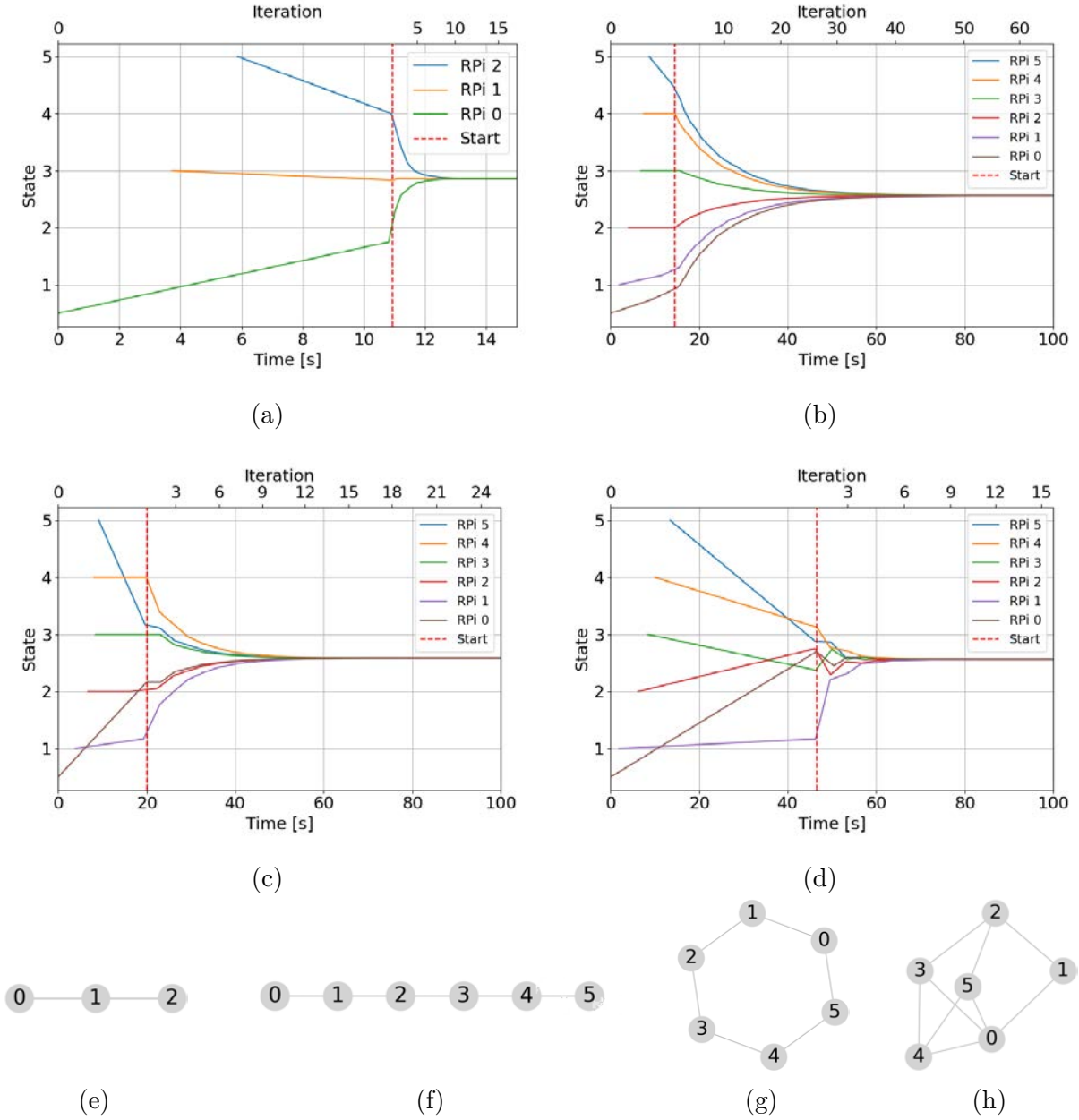


Figure 4.4: State evolution in the consensus application. The mesh-to-graph correspondence is as follows: (a)-(e), (b)-(f), (c)-(g), and (d)-(h). Initial values are 0.5, 1, 2, 3, 4, and 5. A video demonstration is available at this link, showcasing the next application that uses two consensus algorithms to synchronize LEDs in both phase and frequency.

Figure 4.4 shows the convergence of the algorithm across four different topologies. Agents start at different times because programs are launched sequentially, starting with agent 0. Each agent first establishes connections with its neighbors, then sends its initial state, and finally waits to receive the initial state of all its neighbors. An agent starts iterating only once all these steps are completed. The start time shown in the plots corresponds to the latest agent becoming ready.

The results show that the start time increases with the number of connections per agent. The average time per iteration also increases. However, the number of iterations before convergence decreases. In configurations 4.4b, 4.4c, and 4.4d, the longer connection time and iteration duration compensate for the reduced number of iterations, leading to an overall convergence time of about one minute in all cases. The exact values are presented in Table 4.1.

Table 4.1: Experimental results of the consensus application.

	(a)	**(b)**	**(c)**	**(d)**
Start time [s]	10.9	14.5	20.0	46.6
Avg. iteration time [s]	0.28	1.31	3.26	3.55
Consensus value	2.85	2.5625	2.5833	2.5625
Iterations before convergence	10	50	12	6
Convergence time [s]	13	70	60	70

4.3.2 Multi-consensus for LED synchronization

This second synchronization-based application demonstrates how all agents synchronize their LED blinking in both frequency and phase, using two concurrent consensus processes. It serves as a demonstration of the underlying low-level multi-process architecture.

Implementation

Each consensus process maintains a float (4 bytes) representing either the frequency or the phase. The total size of a packet is then 9 bytes, as for consensus application. However, twice as many packets are exchanged per iteration. The theoretical goodput will be the same as for consensus application. The simplified implementation is shown on Figure 4.5, and the `Blink()` function is detailed in Algorithm 3.

For this application, we define: Δt the average iteration duration, θ the phase consensus state, f the frequency consensus state, θ' the agent own phase computed based on its clock (clock are synchronized through NTP), f' the actual blinking frequency, and θ_{tol} the phase tolerance used to trigger the LED blinking at the appropriate moment.

The Blink() function triggers a blinking cycle whenever θ' falls within the tolerance window. Each period is slightly shortened to let the phase evolve. At each blink, the phase θ is updated with θ', while also adjusted by the consensus. For convergence, the phase consensus rate should be two to three times higher than the blinking frequency. Delays are maximized to reduce CPU usage and free room for concurrent threads.

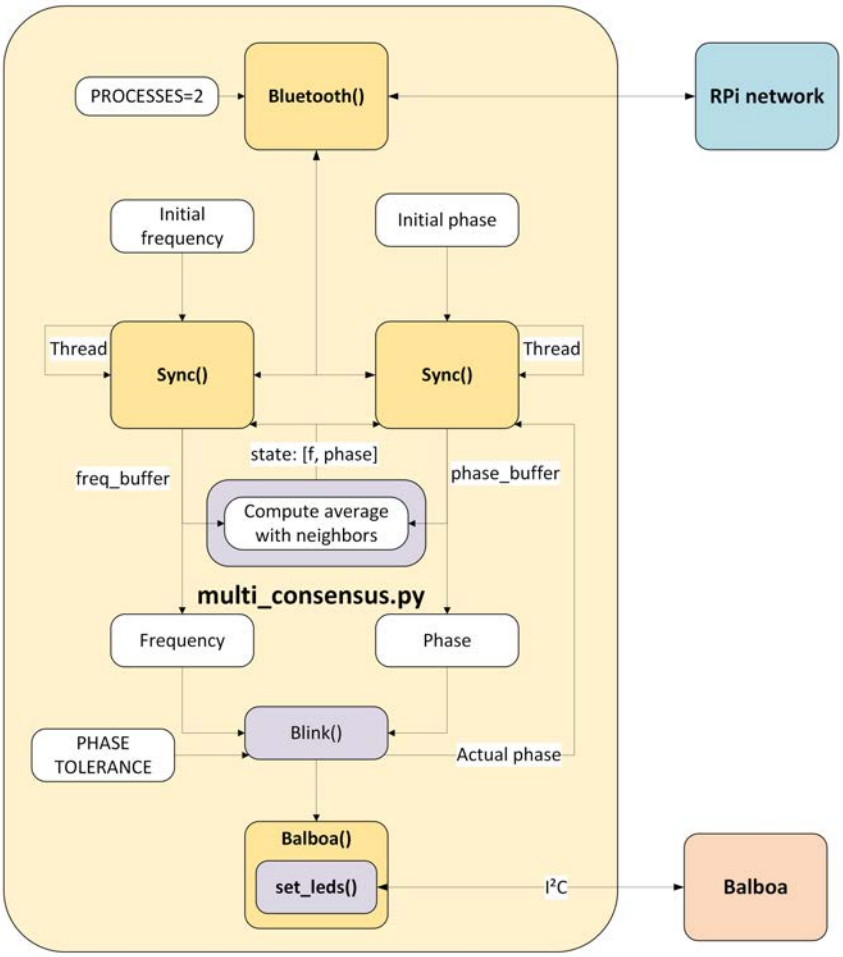


Figure 4.5: Phase and frequency synchronization application. Two consensus are running simultaneously on two different process slots of the `Bluetooth()` interface. Both resulting states are used to temporize the LED blinking, detailed in Algorithm 3.

Algorithm 3 `Blink()` function

while True **do**
  Compute phase tolerance: $\theta_{\text{tol}} \leftarrow \frac{1}{16 \cdot f}$
  Compute current phase: $\theta' \leftarrow t \bmod \left(\frac{1}{f}\right)$
  if $\theta' \in [\theta - \theta_{\text{tol}},\ \theta + \theta_{\text{tol}}]$ **then**
    Update phase consensus: $\theta \leftarrow \theta'$
    Turn LED off
    Wait $\frac{1}{2f}$
    Turn LED on
    Wait $\frac{1}{3f}$
  end if
  Wait θ_{tol}
end while

Results

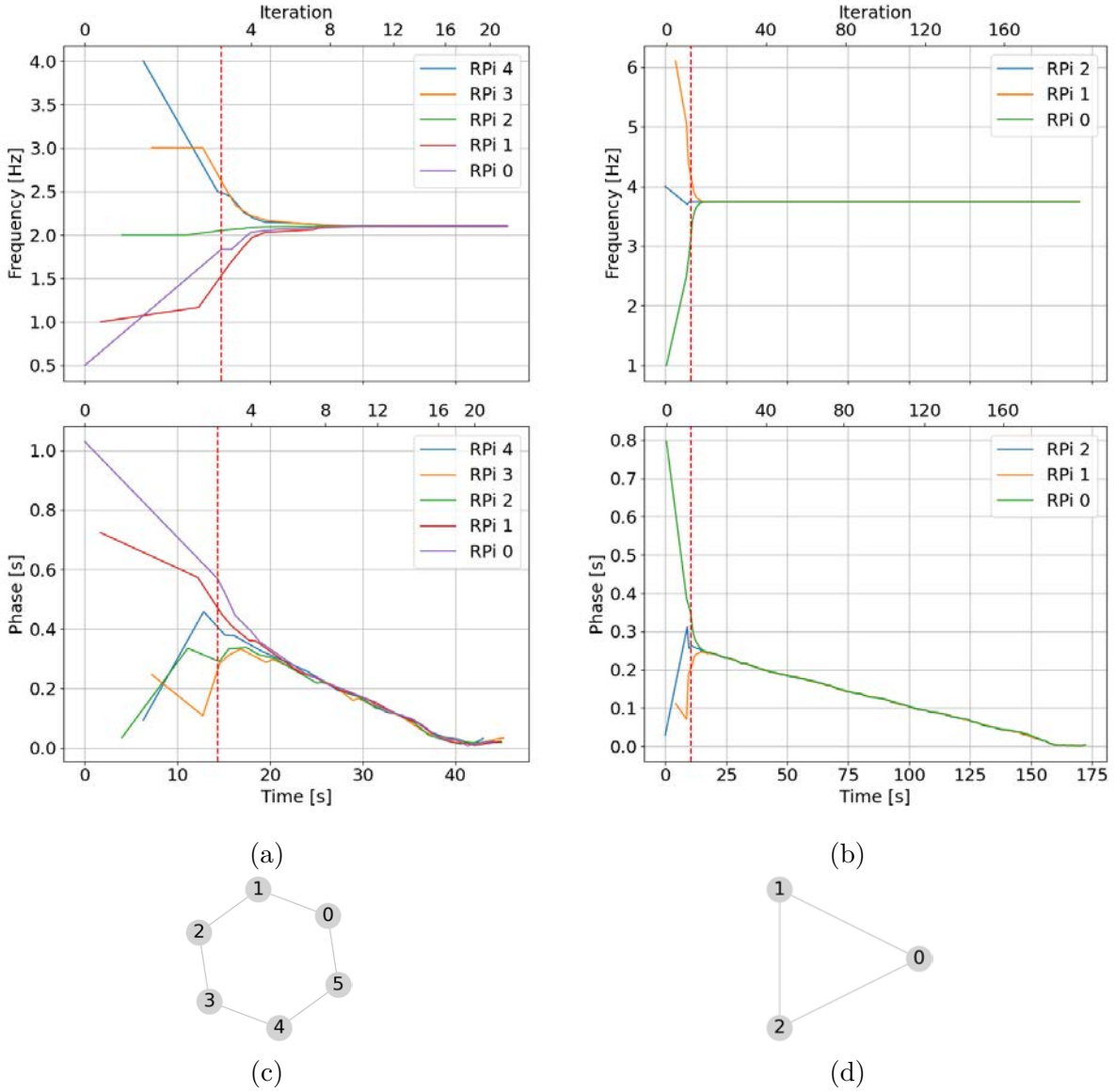


Figure 4.6: Experimental results of the LED's synchronization application. (a) Uses mesh (c) and $\theta_{\text{tol}} = \frac{1}{16 \cdot f} = 0.03$ s. (b) Uses mesh (d) and $\theta_{\text{tol}} = \frac{1}{64 \cdot f} = 0.004$ s. A video demonstration is available at this link.

Figure 4.6 presents the experimental results for two different values of θ_{tol}. We observe that as soon as the frequencies becomes closely aligned, the phases also synchronize. A phase drift is observed between initial convergence and final stabilization to zero, which is induced by the use of θ_{tol}. As blinking is consistently triggered near the lower bound of the tolerance interval, the effective blinking period becomes slightly shorter. Such tolerance is necessary to avoid missing any blinking period, as the execution rate of the blinking loop is limited. This drift does not prevent phase alignment but slightly affects the actual blinking frequency. Let $\Delta\theta$ be the phase drift over one iteration. Given that $\Delta\theta < \theta_{tol}$, we have:

$$f' = \frac{1}{\frac{1}{f} + \Delta\theta} < \frac{f}{1 + f\theta_{tol}}.$$

The stabilization at zero is due to the modulo operator used in phase computation (see Algorithm 3), which keeps $\theta' \in [0, \frac{1}{f}]$. This acts as a negative feedback loop. When f' tends to be slightly higher than f at $\theta = 0$ (due to the phase tolerance), θ' wraps to $\frac{1}{f}$ instead of decreasing below zero. The agent must then wait until it reaches $\theta' = 0$ again, as its phase temporarily falls out of the tolerance range (centered in $\theta = 0$). This introduces a delay and effectively pulls the phase back into alignment, this time near the upper bound of the tolerance.

Let us analyze the results in Figure 4.6a, where $\Delta t = 1.56$ s and $f = 2.1$ Hz. We estimate:

$$\Delta\theta \approx \frac{\theta(t = 30) - \theta(t = 25)}{30 - 25} \cdot \Delta t = -0.022\,\text{s}.$$

It gives $f' = 2.2$ Hz, a difference of 0.1 Hz. After 40 seconds, we have exactly the right frequency $f' = f$ thanks to phase stabilization.
We can also decrease the tolerance, as shown on Figure 4.6b. This reduction in θ_{tol} also reduces $\Delta\theta$, but increases the time required for phase stabilization to zero. We have $f = 3.74$, $\Delta\theta = -0.001$, $f' = 3.76$, giving a slight frequency difference of 0.02 Hz. However, the convergence to zero time is as high as 160 s. If the tolerance is decreased too much, the algorithm will start missing some blinking periods, depending also on the frequency.

4.3.3 Target localization

This final application leverages embedded UWB localization modules and the synchronous messaging to estimate the position of a target in a distributed fashion. Each agent measures both its own position and its distance to the target using its UWB module. Through a distributed gradient descent algorithm, all agents collaboratively estimate the target's position.

Algorithm

The gradient tracking algorithm aims to minimize a cost function, it is explored in Nedic et al. (2016) [55]. In this case, the cost function to be minimized corresponds to the difference between the measured distance and the estimated distance between an agent and the target:

$$f_i^t(x_i^t) = (\phi_i^t)^2 - \|x_i^t - p_i^t\|^2, \tag{4.2}$$

where ϕ_i^t is the measured distance at iteration t between agent i and the target (using UWB), x_i^t is the estimated target position by agent i, and p_i^t is agent i's estimated own position obtained via UWB multilateration. We compute the gradient of the cost function analytically:

$$g_i^t(x_i^t) = \nabla f_i^t(x_i^t) = 4 \cdot ((\phi_i^t)^2 - \|x_i^t - p_i^t\|^2) \cdot (p_i^t - x_i^t). \tag{4.3}$$

However, due to the distributed nature of the system, this gradient cannot be used directly, as agents need access to neighbors' states and gradients, which are only available at iteration $t + 1$. To address this, we estimate the gradient at the next iteration. Note that although the gradients are computed using the current position estimates from iteration $t+1$, they rely on measurements from iteration t, which introduces a minor latency. In this static target scenario, however, $\phi_i^t \approx \phi_i^{t+1}$ and $p_i^t \approx p_i^{t+1}$, making this latency negligible.

The resulting iterative update rule used is the following:

$$x_i^{t+1} = \sum_{j \in \mathcal{N}_i} w_{ij} x_j^t - z_i^t - \gamma g_i(x_i^t), \tag{4.4}$$

$$z_i^{t+1} = \sum_{j \in \mathcal{N}_i} w_{ij} z_j^t - \gamma g_i(x_i^t) + \gamma \sum_{j \in \mathcal{N}_i} w_{ij} g_j(x_j^t), \tag{4.5}$$

$$g_i^{t+1}(x_i^{t+1}) = 4 \cdot ((\phi_i^t)^2 - \|x_i^{t+1} - p_i^t\|^2) \cdot (p_i^t - x_i^{t+1}). \tag{4.6}$$

Where, z_i^t is a local variable updated at each iteration, w_{ij} is the row-normalized weight from the adjacency matrix, and $\mathcal{N}_i$ is the set of agent i's neighbors. As the system is 2D, the vectors x_i^t, z_i^t, g_i^t, and p_i^t are all of size 2. The adapted distributed version of the algorithm is given in Algorithm 4.

Algorithm 4 Gradient descent algorithm at iteration t on agent i

Require: Neighbors' states x_j^t, z_j^t, gradients $g_j(x_j^t)$ for $j \in \mathcal{N}_i$, adjacency matrix $\mathcal{W}_G$, step-size γ

for all $j \in \mathcal{N}_i$ **do**

$x_i^{t+1} \leftarrow x_i^{t+1} + w_{ij} \cdot x_j^t$

$z_i^{t+1} \leftarrow z_i^{t+1} + w_{ij} \cdot (z_j^t + \gamma \cdot g_j(x_j^t))$

end for

$x_i^{t+1} \leftarrow x_i^{t+1} - z_i^t - \gamma \cdot g_i(x_i^t)$

$z_i^{t+1} \leftarrow z_i^{t+1} - \gamma \cdot g_i(x_i^t)$

$g_i^{t+1} \leftarrow 4 \cdot \left((\phi_i^t)^2 - \|x_i^{t+1} - p_i^t\|^2\right) \cdot (p_i^t - x_i^{t+1})$

Implementation

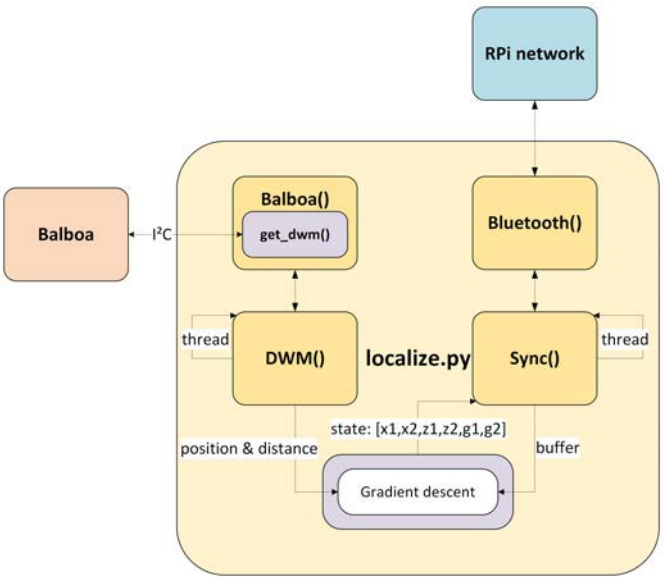


Figure 4.7: Distributed target localization system block diagram. One thread reads Decawave measurements on the Balboa up to 10 Hz, while another manages synchronous messaging using gradient descent as the iterative algorithm.

The system state consists of the vectors x_i^{t+1}, z_i^{t+1}, and g_i^{t+1}, each represented as 2D vectors of two floats, resulting in a total of 24 bytes per state update. The complete data packet size is 29 bytes. As introduced in Section 3.6.2, the goodput G of the synchronous overlay can be approximated by:

$$G = 0.7 \cdot \frac{24}{29} \cdot T \approx 0.58 \cdot T,$$

where T is the total throughput. This implies that roughly 58% of the total throughput is effectively utilized to transmit meaningful data in this application.

Configuration

The anchor configuration used here is identical to the setup described for Decawave calibration in Chapter 1, except that one anchor is now designated as the target to be localized. This setup is illustrated in Figure 4.8.

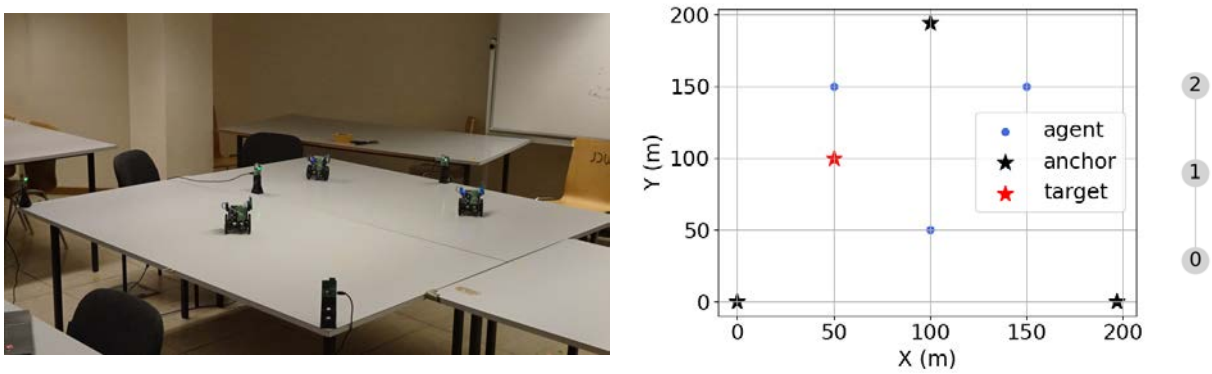
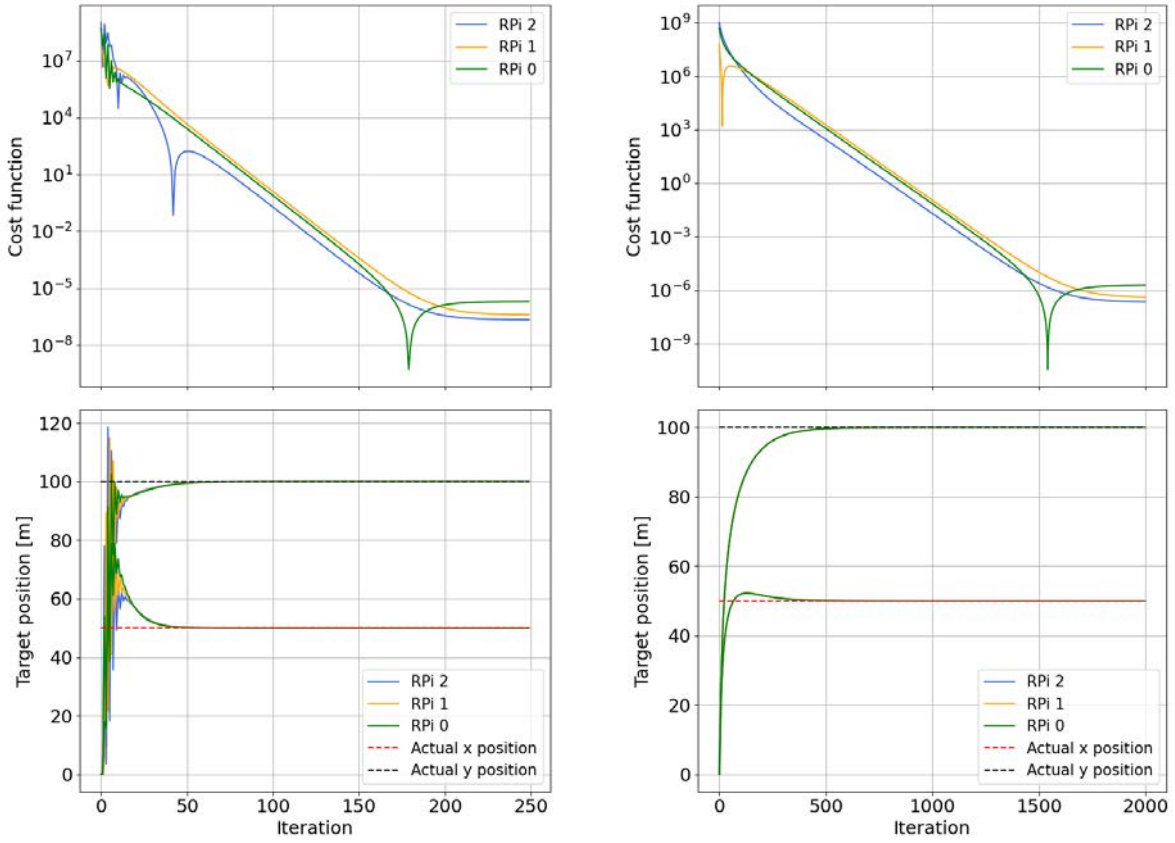


Figure 4.8: Target localization application configuration. A video demonstration is available at this link, showcasing the robots balancing while performing target localization.

Simulations

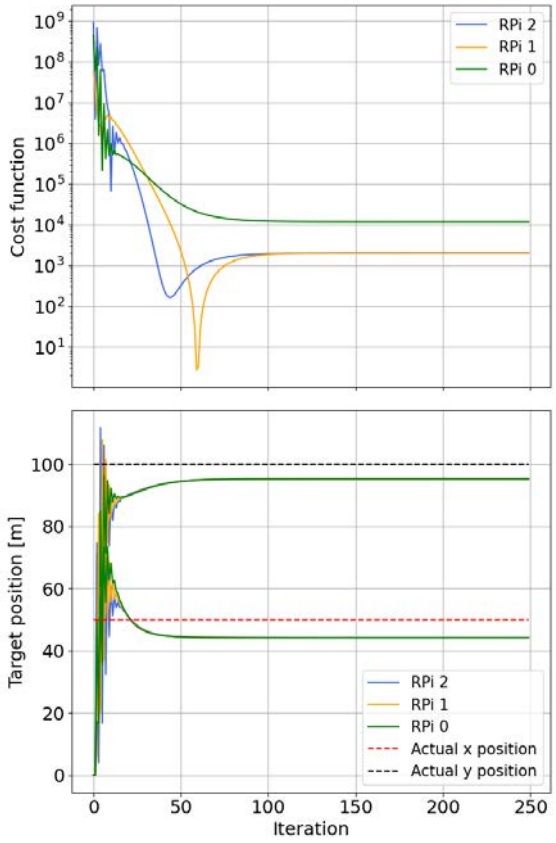
Simulations played a crucial role in tuning the distributed system parameters and validating its correctness. The step size γ was analyzed as a trade-off between convergence speed and estimation smoothness. For the tested configuration, convergence was achieved with γ up to 7×10^{-6}, found by trial and error. Two cases, $\gamma = 4 \times 10^{-6}$ and $\gamma = 5 \times 10^{-7}$, are shown in Figure 4.9. The cost function decays exponentially and stabilizes at a similar final accuracy regardless of γ, but smaller steps significantly increase convergence time while larger steps cause more oscillations.

Measurement errors were modeled by adding a 5% random error to agent positions and distances (accuracy bias), and a 2% periodic noise every 5 iterations (precision bias), shown in Figures 4.9c and 4.9d. Accuracy bias raises the cost function and reduces localization precision, causing earlier convergence. Precision bias induces ripples in both cost and position estimates. Although noise levels do not exactly replicate Decawave errors, their magnitude is representative to illustrate qualitative effects for comparison with experimental data.

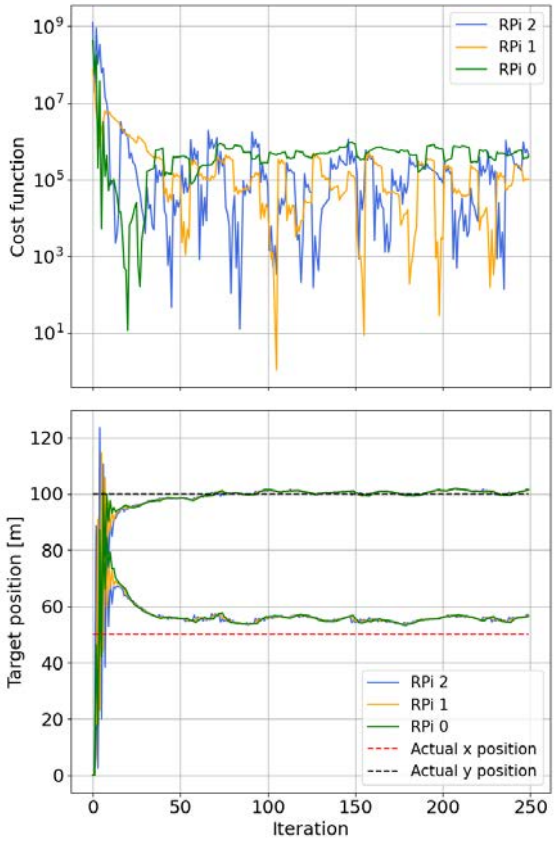


(a)

(b)



(c)



(d)

Figure 4.9: Simulation results of the target localization system. (a), (c), (d) use $\gamma = 4 \cdot 10^{-6}$, (b) uses $5 \cdot 10^{-7}$. (c) includes a 5% accuracy bias; (d) adds a 2% precision bias.

Results

The target localization algorithm was implemented and tested on the actual distributed system using the UWB localization hardware and post-processing model developed in Chapter 1. Experimental results validate the system's performance. Figure 4.10 and Table 4.2 present the position estimations, cost function evolution, and detailed statistics with and without post-processing, for two step sizes γ.

In 4.10a, the results without calibration resemble the simulated noisy system shown in Figure 4.9d. Experiment 4.10b uses identical parameters with post-processing applied, and 4.10c presents results with post-processing and a smaller time step. In cases 4.10b and 4.10c, accurate calibration substantially reduced the accuracy bias, and filtering smoothed the position estimates. The results closely match the simulations shown in Figures 4.9a and 4.9b.

In Figure 4.10d, the evolution of the target position estimation is shown for the worst and best cases, without calibration and with the parameters from Figure 4.10c, respectively. Without calibration, agents positions are biased and fluctuate more over time, highlighting the effectiveness of the calibration method developed in Chapter 2. Furthermore, as seen in Figures 4.10a, 4.10b, and 4.10c, the estimation is significantly smoother in the calibrated case, with strong agreement among agents at each time step, and the final target estimation is more variable and biased without calibration.

Table 4.2: Experimental results for the target localization application corresponding to Figure 4.10

	(a)	**(b)**	**(c)**
Convergence time	8 s	8 s	1 min
Convergence iterations	50	50	400
Average x-position error after convergence	-4.4 cm	2.44 cm	1.71 cm
Average y-position error after convergence	0.78 cm	-0.66 cm	0.01 cm
Average position error after convergence	4.46 cm	2.52 cm	1.71 cm
Maximum absolute position error	9.18 cm	4.45 cm	2.13 cm
Average cost function after convergence	979,362	94,068	85,798

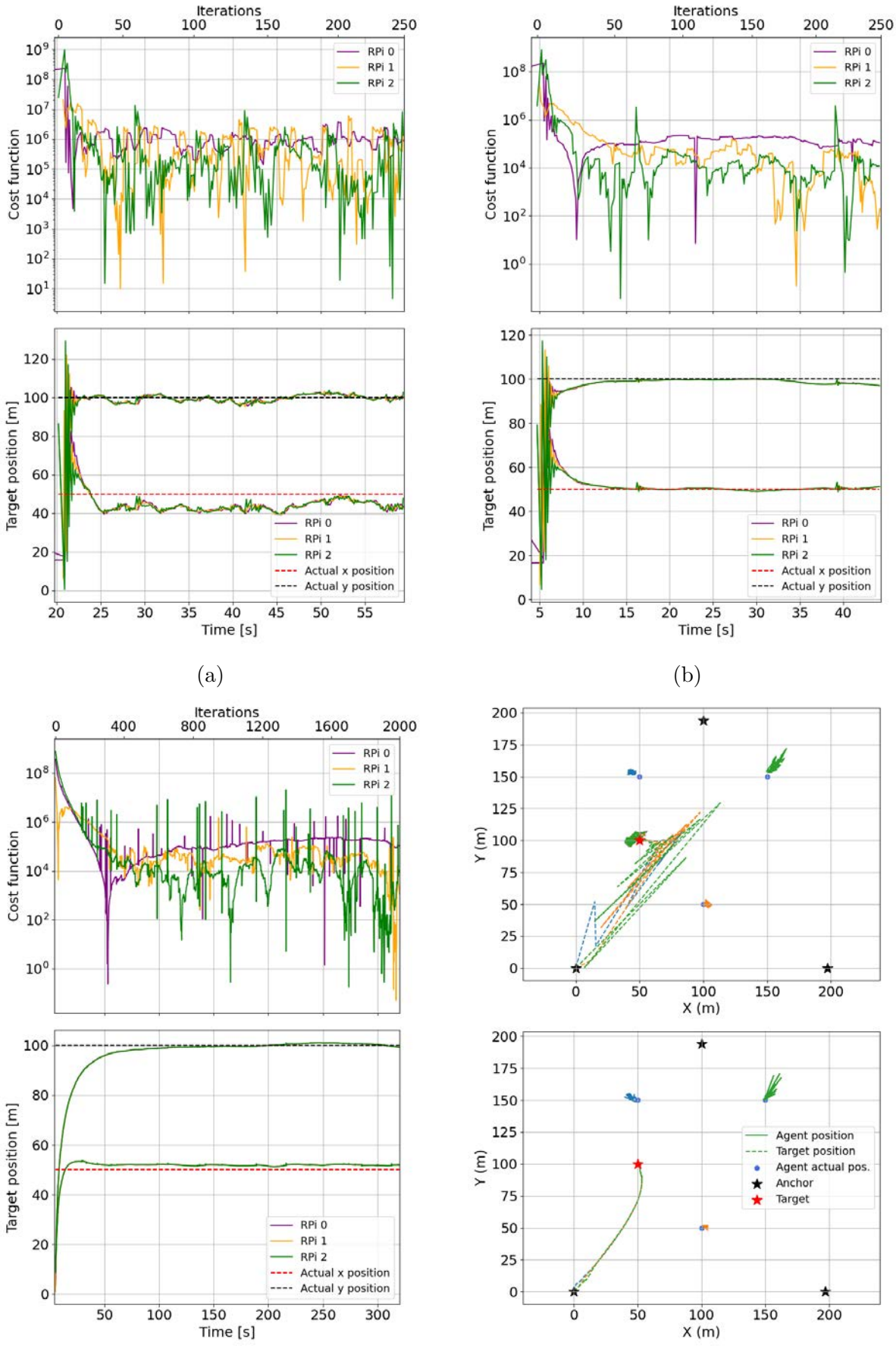


Figure 4.10: Experimental results for the target localization system: (a) Without post-processing, $\gamma = 4 \times 10^{-6}$. (b) With post-processing, $\gamma = 4 \times 10^{-6}$. (c) With post-processing and window size 100, $\gamma = 5 \times 10^{-7}$. (d) X-Y graph with results from (a) and (c).

Chapter 5

Conclusions and perspectives

This work led to a fully embedded swarm platform featuring a streamlined communication stack (25 Kb/s) and improved post-processed localization with sub-10 cm accuracy, all while maintaining low energy consumption and up to 5 hours of autonomy, culminating in a target localization system that achieves outstanding accuracy below 2 cm.

Nevertheless, some limitations remain. The reliance on Bluetooth Classic, despite its low consumption and compatibility with Raspberry Pi, imposes a maximum of 7 simultaneous connections. The absence of message reconstruction above RFCOMM layer limits the packets size to 1021 bytes. Moreover, while localization is effective for absolute positioning at 10 Hz, it does not yet leverage all available onboard sensor data for optimal accuracy.

Several directions could be explored to further improve the platform. First, localization could be significantly improved by fusing UWB data with wheel encoder and IMU measurements using techniques like Kalman filtering. Integrating trajectory planning capabilities would enable behaviors such as coordinated navigation and collaborative obstacle avoidance. Another direction is to combine this with control: in a way to use the communication and localization to inform the navigation of the robots.

At the network level, adding packet segmentation would allow the transmission of larger payloads across the mesh. Beside of that, optional dynamic mesh adaptation via a periodic flooding algorithm could be considered to increase autonomy, although it introduces overhead and complexity typically handled at lower layers. Another quite tricky option would be to run BLE Mesh directly on the Decawave's nRF52 chip to improve energy efficiency and latency. This approach, however, would require reprogramming the Decawave and managing potential communication bottlenecks on the Raspberry Pi interface. Since higher data rates might be necessary, SPI could replace UART, which would in turn require additional hardware such as level shifters.

Overall, this work lays the foundation for a modular and extensible swarm robotics platform, opening up diverse avenues for future research in decentralized multi-agent systems. Beyond the technical achievements, this project demonstrates how careful integration of hardware, communication protocols, and localization strategies can enable robust collective behavior without relying on centralized control. Such an approach reflects

the growing importance of distributed intelligence in robotics, particularly for applications in dynamic, resource-constrained, or infrastructure-free environments. As a future engineer, this project has also strengthened my ability to approach complex systems, from low-level implementation to high-level design decisions, while anticipating the needs of scalability, flexibility, and long-term maintainability. It is my hope that this work will help support advanced practical applications in the field of swarm robotics.

Appendices

Appendix A

Setup of the localization system

Detailed documentation and source code are available in the archive titled "DWM1001C Software and Documentation Pack", accessible in [50]. This archive will be referenced in the following bullet points. As the system includes many functionalities and can be complex to configure, the key steps relevant to this project are summarized below:

1. The first step is to flash the firmware containing the PANS API, which can be found in the archive within the "on-board" package.

2. The easiest way to configure each device is to use the RTLS (Real Time Localization System) mobile app, available at the same link [50] under the name "Android Application (APK file)".

3. The anchors should preferably be placed in a rectangular or equilateral triangle configuration, depending on their number. Their positions can be estimated using the auto-positioning function in the RTLS app.

4. The tags need to be attached to the Balboa robots, making use of the level shifter and the ATMega32U4 TX/RX.

5. In order for the Raspberry Pi to access the Decawave measurements, the Balboa must first communicate with its own tag. I developed a UART interface based on the PANS API, implementing only the functions needed for this project. The Balboa then shares the data with the Raspberry Pi via the I^2C bus upon request.

Appendix B

Configuration of the agent

Detailed instructions regarding the technical setup are available on the project GitHub here.

To summarize, the main configuration steps prior to using the communication system are as follows:

1. Set up a WiFi hotspot (no internet connection required).
2. Flash the Raspberry Pi with Raspberry Pi OS and configure it with the access point and SSH credentials.
3. Connect to the Raspberry Pi using predefined SSH credentials from a computer connected to the access point.
4. Enable I²C, UART, and VNC on each Raspberry Pi.
5. Configure I²C and UART on each Raspberry Pi.
6. Install and configure the virtual Circuit Python environment for OLED use, install all libraries from this project in this virtual environment
7. Pair the Raspberry Pis with each other via Bluetooth.
8. Connect the Raspberry Pi and the Decawave module to the Balboa robot.
9. Use deployment scripts for update firmware, run a program or collect data from all swarm agents.

Appendix C

Multi-agent deployment scripts

C.1 Deploy the software

The `deploy.sh` script allows transferring files quickly and easily to multiple Raspberry Pis on the same LAN. It is a useful alternative to tools like `Filezilla`, especially when dealing with many RPis. The script uses `scp` for secure file transfer.

```
./deploy.sh <IP1> <IP2> ...  <IPn>
```

Configure the `LOCAL_DIR`, `REMOTE_DIR`, `USER`, and `SSH_KEY` variables in the script according to the setup.

C.2 Run a program

The `run.sh` script allows you to execute the same Python program on multiple Raspberry Pis via `ssh`. You can specify the Python file to run and the arguments for each RPi. The first argument should be the RPi's ID, which is automatically assigned based on the IP order.

```
./run.sh <python_program> <IP1>:<param1>,<param2>,...
```

Configure the `USER` and `SSH_KEY` variables in the script according to the setup.

C.3 Run a command

The `command.sh` script sends the same command to multiple Raspberry Pis simultaneously. It is useful for tasks such as restarting or shutting down all RPis in a network.

```
./command.sh 'command' <IP1> <IP2> ...  <IPn>
```

Configure the `USER` and `SSH_KEY` variables in the script according to the setup.

C.4 Fetch data

The `fetch_csv.sh` script allows you to retrieve files from multiple Raspberry Pis on the same LAN network. It is helpful for quickly gathering data, such as logs or sensor data, from several devices.

Usage:

```
./fetch_csv.sh file_name <IP1> <IP2> ...  <IPn>
```

Configure the `DEST_DIR`, `REMOTE_CSV_PATH`, `USER`, and `SSH_KEY` variables in the script according to the setup.

Resources for reproducibility

The full source code of the swarm agent, a tutorial for reproducing the setup and using the developed deployment tools, the performance measurement scripts, the simulation code, as well as all plots, their source code, and the associated data are available in the following repository:
https://github.com/trebelge0/Balboa_network.git

Video Demonstrations

Video recordings of the applications, calibration procedures, and other relevant processes are available at this link.

Declaration of generative AI tools

During the writing of this thesis, ChatGPT was used as a language assistant to improve the clarity, structure, and overall quality of the text. All technical content, designs, and results remain the sole work of the author.

References

[1] Y.U. Cao et al. “Cooperative mobile robotics: antecedents and directions”. In: *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*. Vol. 1. 1995, 226–234 vol.1. DOI: 10.1109/IROS.1995.525801.

[2] Schranz M., Umlauft M., Sende M., Elmenreich W. “Swarm Robotic Behaviors and Current Applications”. In: *Front Robot AI* 7-36 (2020). DOI: doi:10.3389/frobt.2020.00036.

[3] Gregory Dudek et al. “A taxonomy for multi-agent robotics”. In: *Auton. Robots* 3 (Dec. 1996), pp. 375–397. DOI: 10.1007/BF00240651.

[4] A. Farinelli, L. Iocchi, and D. Nardi. “Multirobot systems: a classification focused on coordination”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34.5 (2004), pp. 2015–2028. DOI: 10.1109/TSMCB.2004.832155.

[5] Marco Dorigo, Guy Theraulaz, and Vito Trianni. “Swarm Robotics: Past, Present, and Future [Point of View]”. In: *Proceedings of the IEEE* 109.7 (2021), pp. 1152–1165. DOI: 10.1109/JPROC.2021.3072740.

[6] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. “Kilobot: A low cost scalable robot system for collective behaviors”. In: *2012 IEEE International Conference on Robotics and Automation*. 2012, pp. 3293–3298. DOI: 10.1109/ICRA.2012.6224638.

[7] Jorge Soares, Iñaki Navarro, and A. Martinoli. “The Khepera IV Mobile Robot: Performance Evaluation, Sensory Data and Software Toolbox”. In: Dec. 2015. ISBN: 978-3-319-27145-3. DOI: 10.1007/978-3-319-27146-0_59.

[8] Paulo Gonçalves et al. “The e-puck, a Robot Designed for Education in Engineering”. In: *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions* 1 (Jan. 2009).

[9] Daniel Pickem, Myron Lee, and Magnus Egerstedt. “The GRITSBot in its natural habitat - A multi-robot testbed”. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4062–4067. DOI: 10.1109/ICRA.2015.7139767.

[10] Sean Wilson et al. “The Robotarium: Globally Impactful Opportunities, Challenges, and Lessons Learned in Remote-Access, Distributed Control of Multirobot Systems”. In: *IEEE Control Systems Magazine* 40.1 (2020), pp. 26–44. DOI: 10.1109/MCS.2019.2949973.

[11] Sean Wilson et al. “The Robotarium: Automation of a Remotely Accessible, Multi-Robot Testbed”. In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 2922–2929. DOI: 10.1109/LRA.2021.3062796.

[12] Daniel Pickem et al. “The Robotarium: A remotely accessible swarm robotics research testbed”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017, pp. 1699–1706. DOI: 10.1109/ICRA.2017.7989200.

[13] Correa M.F.S. et al Rezeck P. Azpurua H. “HeRo 2.0: a low-cost robot for swarm robotics research”. In: *Auton Robot 47* (2023), pp. 879–903. DOI: https://doi.org/10.1007/s10514-023-10100-0.

[14] Agata Barciś, Michał Barciś, and Christian Bettstetter. “Robots that Sync and Swarm: A Proof of Concept in ROS 2”. In: *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*. 2019, pp. 98–104. DOI: 10.1109/MRS.2019.8901095.

[15] Agata Barciś and Christian Bettstetter. “Sandsbots: Robots That Sync and Swarm”. In: *IEEE Access* 8 (2020), pp. 218752–218764. DOI: 10.1109/ACCESS.2020.3041393.

[16] Caroline Perry. *A self-organizing thousand-robot swarm*. https://seas.harvard.edu/news/2014/08/self-organizing-thousand-robot-swarm. Accessed: 2025-05-26.

[17] Mahmoud Tarek et al. “Attitude and Orbit Control Algorithms for Swarm Satellites Used in earth Observation”. PhD thesis. Feb. 2021. DOI: 10.13140/RG.2.2.31969.43362.

[18] Muthu Chellappa, Shanmugaraj Madasamy, and R. Prabakaran. “Study on ZigBee technology”. In: Apr. 2011, pp. 297–301. ISBN: 978-1-4244-8678-6. DOI: 10.1109/ICECTECH.2011.5942102.

[19] Luca Davoli et al. “Design and experimental performance analysis of a B.A.T.M.A.N.-based double Wi-Fi interface mesh network”. In: *Future Generation Computer Systems* 92 (Feb. 2018). DOI: 10.1016/j.future.2018.02.015.

[20] Ligang Liu et al. “Performance Evaluation of BATMAN-Adv Wireless Mesh Network Routing Algorithms”. In: June 2018, pp. 122–127. DOI: 10.1109/CSCloud/EdgeCom.2018.00030.

[21] Sailash Moirangthem and Viswanath Talasila. “A practical evaluation for routing performance of BATMAN-ADV and HWMN in a Wireless Mesh Network test-bed”. In: Dec. 2015, pp. 1–6. DOI: 10.1109/SMARTSENS.2015.7873617.

[22] J. Chroboczek. *RFC 6126: The Babel Routing Protocol*. USA, 2011.

[23] Mathias Baert et al. “The Bluetooth Mesh Standard: An Overview and Experimental Evaluation”. In: *Sensors* 18.8 (2018). ISSN: 1424-8220. DOI: 10.3390/s18082409. URL: https://www.mdpi.com/1424-8220/18/8/2409.

[24] Elis Kulla et al. “Performance comparison of OLSR and BATMAN routing protocols by a MANET testbed in stairs environment”. In: *Computers Mathematics with Applications* 63 (Jan. 2012), pp. 339–349. DOI: 10.1016/j.camwa.2011.07.035.

[25] Dana Turlykozhayeva et al. “Experimental Performance Comparison of Proactive Routing Protocols in Wireless Mesh Network Using Raspberry Pi 4”. In: *Telecom* 5 (Oct. 2024), pp. 1008–1020. DOI: 10.3390/telecom5040051.

[26] Silicon labs. *N1138: Zigbee Mesh Network Performance*. https://www.silabs.com/documents/login/application-notes/an1138-zigbee-mesh-network-performance.pdf.

[27] Sumit Paul, Danh Lephuoc, and Manfred Hauswirth. *Performance Evaluation of ROS2-DDS middleware implementations facilitating Cooperative Driving in Autonomous Vehicle*. Dec. 2024. DOI: 10.48550/arXiv.2412.07485.

[28] Giil Kwon et al. “Development of Real-Time Data Publish and Subscribe System Based on Fast RTPS for Image Data Transmission”. In: 2018. URL: https://api.semanticscholar.org/CorpusID:209411737.

[29] Dinesh Thangavel et al. “Performance evaluation of MQTT and CoAP via a common middleware”. In: *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. 2014, pp. 1–6. DOI: 10.1109/ISSNIP.2014.6827678.

[30] Raphael Frank, Thomas Scherer, and Thomas Engel. “Tree Based Flooding Protocol for Multi-hop Wireless Networks”. In: *2008 Third International Conference on Broadband Communications, Information Technology Biomedical Applications*. 2008, pp. 318–323. DOI: 10.1109/BROADCOM.2008.8.

[31] Federico Ferrari et al. “Efficient network flooding and time synchronization with Glossy”. In: *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*. 2011, pp. 73–84.

[32] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. “Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale”. In: Nov. 2013. DOI: 10.1145/2517351.2517358.

[33] Fabian Mager et al. “Competition: Low-Power Wireless Bus Baseline”. In: Feb. 2019.

[34] Roman Lim et al. “FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems”. In: *2013 ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 2013, pp. 153–165. DOI: 10.1145/2461381.2461402.

[35] Hao Xu et al. “Omni-Swarm: A Decentralized Omnidirectional Visual–Inertial–UWB State Estimation System for Aerial Swarms”. In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3374–3394. DOI: 10.1109/TRO.2022.3182503.

[36] Sebastian Sadowski and Petros Spachos. “RSSI-Based Indoor Localization With the Internet of Things”. In: *IEEE Access* 6 (2018), pp. 30149–30161. DOI: 10.1109/ACCESS.2018.2843325.

[37] Yuan Cao, Harsha Kandula, and Xinrong Li. “Measurement and Analysis of RSS Using Bluetooth Mesh Network for Localization Applications”. In: *Network* 1 (Dec. 2021), pp. 315–334. DOI: 10.3390/network1030018.

[38] Luca Schenato and Giovanni Gamba. “A distributed consensus protocol for clock synchronization in wireless sensor network”. In: *2007 46th IEEE Conference on Decision and Control*. 2007, pp. 2289–2294. DOI: 10.1109/CDC.2007.4434671.

[39] Michele Girolami et al. “A Bluetooth 5.1 Dataset Based on Angle of Arrival and RSS for Indoor Localization”. In: *IEEE Access* 11 (2023), pp. 81763–81776. DOI: 10.1109/ACCESS.2023.3301126.

[40] Shruti Pandey et al. “AprilTag-Based Self-Localization for Drones in Indoor Environments”. In: *SSRN Electronic Journal* (Jan. 2024). DOI: 10.2139/ssrn.4815151.

[41] Davide Cannizzaro et al. “A Comparison Analysis of BLE-Based Algorithms for Localization in Industrial Environments”. In: *Electronics* 9 (Dec. 2019), p. 44. DOI: 10.3390/electronics9010044.

[42] L. Taponecco, A.A. D’Amico, and U. Mengali. “Joint TOA and AOA Estimation for UWB Localization Applications”. In: *IEEE Transactions on Wireless Communications* 10.7 (2011), pp. 2207–2217. DOI: 10.1109/TWC.2011.042211.100966.

[43] Regina Kaune. “Accuracy studies for TDOA and TOA localization”. In: *2012 15th International Conference on Information Fusion*. 2012, pp. 408–415.

[44] Alwin Poulose and Dong Han. "UWB Indoor Localization Using Deep Learning LSTM Networks". In: *Applied Sciences* 10 (Sept. 2020), p. 6290. DOI: 10.3390/app10186290.

[45] Tommaso Polonelli, Simon Schläpfer, and Michele Magno. "Performance Comparison between Decawave DW1000 and DW3000 in low-power double side ranging applications". In: *2022 IEEE Sensors Applications Symposium (SAS)*. 2022, pp. 1–6. DOI: 10.1109/SAS54819.2022.9881375.

[46] Aurélien Soenen. "From Model-Based to Data-Driven Control: Applications to Self-Balancing Robots". Promoter: Gianluca Bianchin. Master's thesis. Belgium: École Polytechnique de Louvain, Université Catholique de Louvain, 2024. URL: http://hdl.handle.net/2078.1/thesis:46172.

[47] Advamation mechatronics. *Raspberry Pi I2C clock-stretching bug*. http://www.advamation.com/knowhow/raspberrypi/rpi-i2c-bug.html. Accessed: 2025-04-07. 2013.

[48] Pololu. *pololu-rpi-slave-arduino-library*. https://github.com/pololu/pololu-rpi-slave-arduino-library. Accessed: 2025-04-07. 2018.

[49] Qorvo. *DWM1001 FIRMWARE APPLICATION PROGRAMMING INTERFACE (API) GUIDE*. https://forum.qorvo.com/uploads/default/original/1X/dcac22d1c0feaf8238f68d11515ad55dbef1b963.pdf. Accessed: 2025-04-24. 2019.

[50] Inpixon. *UWB Localization: Time Difference of Arrival vs Two-Way Ranging*. https://www.qorvo.com/products/p/MDEK1001. Accessed: 2025-04-08. 2020.

[51] Microchip. *ATmega16U4/ATmega32U4 datasheet*. https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7766-8-bit-AVR-ATmega16U4-32U4_Datasheet.pdf. Accessed: 2025-04-10.

[52] Raspberry Pi Dramble. *UWB Localization: Time Difference of Arrival vs Two-Way Ranging*. https://pidramble.com/wiki/benchmarks/power-consumption. Accessed: 2025-04-10.

[53] Python. *struct — Interpret bytes as packed binary data*. https://docs.python.org/3/library/struct.html. Accessed: 2025-04-21.

[54] Reza Olfati-Saber, J. Alex Fax, and Richard M. Murray. "Consensus and Cooperation in Networked Multi-Agent Systems". In: *Proceedings of the IEEE* 95.1 (2007), pp. 215–233. DOI: 10.1109/JPROC.2006.887293.

[55] Angelia Nedic, Alex Olshevsky, and Wei Shi. "Achieving Geometric Convergence for Distributed Optimization Over Time-Varying Graphs". In: *SIAM Journal on Optimization* 27 (July 2016). DOI: 10.1137/16M1084316.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**