The University Of Sheffield.

Classification and Management of Computational Resources of Robotic Swarms and the Overcoming of their Constraints

By:

Stefan Markus Trenkwalder

A thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy

The University of Sheffield
Faculty of Engineering
Department of Automatic Control and Systems Engineering

Sheffield, 24th March 2020

Abstract

Swarm robotics is a relatively new and multidisciplinary research field with many potential applications (e.g., collective exploration or precision agriculture). Nevertheless, it has not been able to transition from the academic environment to the real world. While there are many potential reasons, one reason is that many robots are designed to be relatively simple, which often results in reduced communication and computation capabilities. However, the investigation of such limitations has largely been overlooked.

This thesis looks into one such constraint, the computational constraint of swarm robots (i.e., swarm robotics platform). To achieve this, this work first proposes a computational index that quantifies computational resources. Based on the computational index, a quantitative study of 5273 devices shows that swarm robots provide fewer resources than many other robots or devices. In the next step, an operating system with a novel dual-execution model is proposed, and it has been shown that it outperforms the two other robotic system software. Moreover, results show that the choice of system software determines the computational overhead and, therefore, how many resources are available to robotic software. As communication can be a key aspect of a robot's behaviour, this work demonstrates the modelling, implementing, and studying of an optical communication system with a novel dynamic detector. Its detector improves the quality of service by orders of magnitude (i.e., makes the communication more reliable). In addition, this work investigates general communication properties, such as scalability or the effects of mobility, and provides recommendations for the use of such optical communication systems for swarm robotics. Finally, an approach is shown by which computational constraints of individual robots can be overcome by distributing data and processing across multiple robots.

Acknowledgements

I would like to express my gratitude to a number of people for their support and guidance. Namely, I thank my supervisors — Dr. Roderich Groß, Dr. Andreas Kolling, and Prof. Robert F. Harrison for their support and guidance. I wish to extend my special thanks to the Austrian Academy of Sciences (ÖAW) for awarding me the DOC fellowship and my fellowship supervisor Dr. Radu Prodan. The assistance provided by many members of the department of automatic control and systems engineering was greatly appreciated. In particular, I would like to thank Dr Iñaki Esnaola for his support and guidance even though I was not one of his PhD students.

I wish to express my gratitude to all the members of my research group and Sheffield Robotics for all of the friendships, advice, and discussions of ideas. To name only a few, I had a privilege to meet, work alongside with, and learn from Christina Georgiou, Fernando Perez-Diaz, Gabriel Kapellmann Zafra, Joao Vasco Marques, Melvin Gauci, Natalie Wood, Yuri Kaszubowski Lopes. My time at the University of Sheffield would not have been the same without them.

Finally, I want to express my sincerest thanks and appreciation to my parents Johann and Sabine for all the support and guidance. I would not have had the chance to reach this milestone without their many sacrifices for my sake. Lastly, I would like to thank Izabela Sylwia Stopinska for her sheer endless patience, support, and help to allow me to finalise this project.

Contents

1

Introduction

Contents

Since time immemorial, humankind has been developing tools/machines to increase productivity, efficiency or to reduce risks. With the industrial revolution, we learned to build powered machines that perform repetitive tasks. From that time forth, an increasing number of processes have been automated [Groover 2007; Chui et al. 2015; Degryse 2016]. Today's technology enables us to automate devices that interact with humans [Sheridan 2016; Kolling et al. 2016; Goodrich and Schultz 2007], vehicles [Okuda et al. 2014; Yuh 2000; Chao et al. 2010], agriculture [Edan et al. 2009; Grift et al. 2008], and more. These machines are, in many cases, called robots.

While the definition of robotics varies (e.g. [Cambridge University Press 2016] or [Oxford University Press 2016]), in this work, *robots* are mobile or stationary electromechanical devices that interact with their environment. This definition enables a large number of classifications [Siciliano and Khatib 2016], one of them being the operation environment, such as air [Valavanis and Vachtsevanos 2014; Gupte et al. 2012], space [Flores-Abad et al. 2014; Ellery 2000], underwater [Yuh et al. 2012; Williams et al. 2016], and earthbound [Diller et al. 2013]. A second classification relates to the size of the robots that can vary from aircraft-sized robots [Herrick 2000] down to micro-/nanorobots [Arab and Feng 2014]. Another classification refers to the number of robots, for instance, single-robot systems (e.g. surgical robots [Taylor et al. 2008]) or multi-robot systems [Iocchi et al. 2001].

One area of robotics in which this work is situated, *swarm robotics*, studies large numbers of relatively-simple robots, whose autonomous behaviour emerges through local interaction with the environment and other robots [Şahin 2005]. However, this definition is vague when considering the terms "*large numbers*", "*relatively simple*", and "*local interaction*". While Beni [2005] defines a swarm as too large to deal with a few-body problem ($> 10^2$) and too small to deal with statistical averages ($\ll 10^{23}$), Hamann [2018] avoids specifying swarm robotics over the number of robots (i.e., "*if robots implement a swarm behaviour then it should*

be considered a swarm [. . .]"). When it comes to the complexity of an individual robot, "relatively simple" should be understood in such a way that a single robot is either incapable or inefficient[1.0.I] concerning a given task. When looking at local interactions, these can be divided into interactions with the environment and with other robots (i.e., communication). Interactions with the environment are performed by sensors and actuators, and Şahin [2005] even declared actuation a key property of swarm robotics as it distinguishes swarm robotics from, for instance, sensor networks. While explicit[1.0.II] communication is not always used, Brambilla et al. [2013] describe communication as another key property of swarm robotics as it is essential to allow collaboration and cooperation between robots. Finally, swarm robotics has a further key property, *decentralisation*. While it is only indirectly written in the definition of swarm robotics, many within the swarm robotics community agree that swarm robotics systems should not access centralised control or global knowledge [Şahin 2005; Şahin and Winfield 2008; Brambilla et al. 2013; Hamann 2018]. Hamann [2018] goes as far as describing systems that use centralised communication systems (e.g., Bluetooth or Wifi) as not to be swarm robotics systems.

Swarm robotics has a wide range of potential applications [Şahin and Winfield 2008; Tan and yang Zheng 2013] including foraging [Acar et al. 2003], precision agriculture [Beni 2005; Ruckelshausen et al. 2009; Yaghoubi et al. 2013; Bangert et al. 2013], pattern formation [Bahceci et al. 2003], transportation [Dorigo et al. 2006; Chen et al. 2015], search and rescue [Kantor et al. 2003; Stormont 2005; Winfield et al. 2016], and collective exploration [Ducatelle et al. 2011*b*; Dorigo et al. 2013]. Swarm robotics' qualities of *robustness*[1.0.III], *flexibility*[1.0.IV] and *scalability*[1.0.V] open the door to its comprehensive implementation spanning a variety of disciplines [Şahin 2005; Brambilla et al. 2013; Hamann 2018]. In particular, applications that have proven pressing, such as plastic extraction from the sea [Rojas 2018] or space debris collection [Shan et al. 2016], could benefit from these properties. However, to the author's knowledge, swarm robotics is almost exclusively operated in academic environments and has not been able to bridge the gap[1.0.VI] between the academic and the real world.

While the lack of real-world systems can result from a wide range of reasons, Hamann [2010] for instance describes that swarm robotics' inter-disciplinary as potentially harmful to the progress: "*[. . .] it might imply the relevance of a vast number of fields making the research unclear and all efforts will inherently be incomplete*." Currently, this incompleteness manifests itself in such a way that the majority of swarm robotics research focuses on task-specific or emergent behaviour and tends to overlook other areas, such as systems research or digital communication. Another aspect of it is an interdependency between conducted research and available robots/hardware. While research advances can improve the capabilities of robots, it is also plausible that the robots that are available to the researchers influence which problems are investigated and which type of solutions are found. For instance, if an available robot has a relatively small amount of sensors and *computational resources*[1.0.VII], it is the author's firm belief that the research outcome will more likely be a minimalistic approach. Another hurdle could be a potential conflict between the dynamic, complex, and unpredictable nature of the real world and relatively-simple robots with their local interaction capabilities. However, the research into the capabilities and limitations of robots has been scant. While the author admits

[1.0.I] Inefficient, in this context, is task-specific and can be interpreted as either requiring too much time or producing inaccurate outcomes.

[1.0.II] Explicit communication describes an active state change of a medium or environment that can be observed by other robots. An active state change can be, for instance, the emission of light or radio signals. In other words, if the information is emitted and received, it is explicit communication.

[1.0.III] Robustness is the capability to operate despite the malfunction of parts of the system.

[1.0.IV] Flexibility is the capability of performing different tasks.

[1.0.V] Scalability is the capability to maintain or even improve performance when adding robots.

[1.0.VI] The technical or technological hurdles that require overcoming for a system to be deployed in the real world is often referred to as the reality gap.

[1.0.VII] Computational resources are memory and processing time.

that the issue at hand is complex, finding the answers to the questions of "how many computational resources are available on a typical platform?", "how are the resources managed?", "what are their limitations?", and "how can the limitations be reduced or even overcome?" can only benefit the understanding of why swarm robotics has not yet left its mark outside the academia.

1.1 Problem Statement

Arguably, one of the biggest challenges swarm robotics currently faces is bridging the gap between conceptual or academic work and real-world applications. As mentioned earlier, this is not for the lack of prospective uses, swarm robotics does have the potential to make an impact in an already technologically advanced world in real terms. However, the probable reasons behind this gap have not been investigated and the respective research is lacking.

The problem that this work tackles is to show how resources of the "*large numbers of relatively-simple*" robots can be used to overcome computational limitations of a single robot. As solutions to real-world problems can be complex, robots, particularly ones with few computational resources, often prove inadequate to solve the task at hand. Moreover, with the increased complexity of these solutions, fewer robots will provide sufficient computational resources to be of consequence. For instance, if a solution requires visual object recognition, a simple swarm robot, such as the e-puck [Mondada et al. 2009] that has the required sensors, would not be able to perform the given task as it cannot store a single image. However, with enough robots, it should be possible to provide sufficient resources to solve such a task.

In order to utilise the resources of groups of robots, additional problems need addressing.

(I) The first problem is to specify what "*relatively-simple*" means with regard to computational resources and what implication the given resources have on robots and their software. With the knowledge of how many resources are available, potential trends of the robot's usage and of the performed applications can be found. This could be used to identify potential limitations based on the robot's resources.

(II) The second problem is to identify how are computational resources managed and used on robots. Fundamentally, robots with more resources can potentially execute more complex software. However, when a more demanding system software[1.1.I] is used, fewer resources are available to the robot, which could impact the robot's behaviour. On the other hand, if system software is appropriately chosen, it can provide a wide range of features easing the development of robots.

(III) The third problem is to identify how communication is managed and used on robots. While communication is used often in swarm robotics (e.g., [Rubenstein et al. 2014; Garattoni and Birattari 2018]), there is little research on communication systems, their requirements, and their properties. However, when behaviours are based on communication and the communication system is not adequately investigated, communication errors can bias the outcome and, in the worst case, lead to wrong research conclusions. On the other hand, when chosen appropriately, communication systems can reduce the costs of communications (i.e., better quality of service and reduced transmission times).

By tackling these three issues, the foundations are laid to approach the main problem in which multiple robots collectively provide their local computational resources via a network to solve a common goal.

[1.1.I] System software is a software that provides a platform for other software to be executed.

1.2 Objectives

Based on the presented problem, the overall aim of this thesis is to demonstrate that the computational constraints of individual robots can be overcome by distributively utilising the resources of the group of robots. The specific objectives of this work are:

- To analyse the computational resources on swarm robots (i.e., swarm robotic platforms), how they can be classified, and how they impact the software that can be used.
- To investigate how resources are managed and made available to robotic behaviour software and to exploit that knowledge to develop a framework that reduces computational overhead.
- To investigate how communication is managed and used on swarm robots and to demonstrate how to increase the *quality of service*[1.2.I].
- To demonstrate a new approach to how a group of robots connected via a network can utilise and manage computational resources to solve a problem that a single robot could not.

1.3 Preview of Contributions

The contributions of this thesis are:

- A metric to quantify the computational resources, namely computational index, and a classification of robots based on an in-depth study of the said indexes of 46 state-of-the-art robots and 5227 other computer systems. Under this new classification, a robot can be either non-computational, severely-constrained, weakly-constrained, or minimally-constrained. In addition, an analysis of software for robots demonstrated on which class of robots the respected software could be deployed, which indicates trends and possible limitations of robots.
- The design and implementation of a novel operating system for swarms of severely-constrained mobile robots, OpenSwarm. Its novelty is the proposed execution model within its hybrid kernel that allows thread-based preemptive scheduling for computationally-intensive tasks as well as event-based cooperative scheduling for swiftly-responsive tasks. Additionally, the software design enables the exchange of events via a network allowing distributed computation. It is open-source and lightweight with a small memory footprint of 2 kB of RAM and 10 kB of ROM.
- The systematic evaluation of OpenSwarm on physical robots is achieved by comparing it to other robotic system software and a *hardware-near*[1.3.I] implementation. In this study, the computational and memory overhead, as well as the *performances*[1.3.II], are compared showing that the choice of system software can effect executed behaviours, even when executing computationally minimalistic approaches.
- The first model describing the signal characteristics of the infra-red communication of the widely used robotic platform, e-puck. The model is validated against data from real robots. This model is also applicable to any device using the same components or the e-puck's successor, e-puck 2.

[1.2.I] The quality of service describes the overall performance of a network. It can be measured by several parameters, such as bit rate, package loss, or transmission delay.

[1.3.I] In this thesis, hardware-near describes the control of hardware functions directly through registers and flags.

[1.3.II] Performance, in this context, refers to the time an algorithm needs and the accuracy of the result.

- SwarmCom, a formally defined communication system for e-puck robots. Its key property is the dynamic detector that adapts to the ambient light during runtime and, more importantly, adapts its decision threshold to the incoming signal. This reduces bit-error probability by orders of magnitudes (i.e., more reliable communication). SwarmCom is evaluated against libIrcom [Gutiérrez et al. 2009*b*] in a static and dynamic environment, and it is demonstrated to outperform the latter.

- Studies on communication properties, such as mobility and scalability, show properties of short-range optical communication systems in general. Experiments demonstrate that mobility increases the average communication errors by magnitudes of power. Evaluations with up to 30 robots show that optical communication systems are scalable in particular for high-density[1.3.III] robot swarm. Overall, these insights apply not only to SwarmCom but also to any short-range and optical communication system.

- A proof-of-concept demonstrating that a group of severely-constrained robots can collectively utilise their resources, which enables the storing and processing of large data exceeding the capabilities of a single robot. It also shows that the proposed system, the first such system on severely-constrained robots, can virtualise sensors enabling the use of augmented environments and can integrate other infrastructure.

Note that these contributions are based on software and behaviours implemented and deployed on e-puck robots. As shown in Chapter 2, an e-puck robot is one of the most computationally-constrained robots. Unlike a Kilobot [Rubenstein et al. 2012] (an even more constrained robot), it provides capabilities that are commonly used in swarm robotics, such as differential drive, proximity sensors, accelerometers, gyroscope, microphones and a camera. The choice of sensors offered by an e-puck is in many cases available only on less-constrained platforms, such as marXbot [Bonani et al. 2010]. The e-puck has been chosen because it provides these common features while facing severe computational constraints. In addition, the e-puck is widely used in the swarm robotics community and, therefore, this work could benefit a wider audience.

1.4 Publications

This thesis represents the author's work and has lead to several original contributions to scientific knowledge. The presented work is built around the following peer-reviewed publications (two journal and one international conference papers):

> S. M. Trenkwalder (2019). 'Miniature Robots: Their Computational Resources, Classification, & Implications.' *IEEE Robot. Autom. Lett.*, 4(3):2722–2729
>
> S. M. Trenkwalder, Y. K. Lopes, A. Kolling, A. L. Christensen, R. Prodan, and R. Groß (2016). 'OpenSwarm: An event-driven embedded operating system for miniature robots.' In 'Proc. 2016 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2016),' (pp. 4483–4490). Piscataway, NJ: IEEE
>
> S. M. Trenkwalder, I. Esnaola, Y. K. Lopes, A. Kolling, and R. Groß (2019). 'SwarmCom: An Infra-Red-Based Mobile Ad-Hoc Network for Severely Constrained Robots.' *Auton. Robots*, (pp. 1–22). URL https://doi.org/10.1007/s10514-019-09873-0

The second publication was presented as a full paper at the international conference 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) in South Korea.

[1.3.III] High-density swarms describe groups of robots that can occur in large numbers on small space — for instance, 234 e-pucks could be placed within a square metre.

Chapter 2, 3, and 4 are based on and extend the material of the first, second, and third publication, respectively.

During the PhD project, the author has also contributed to the following projects that are not featured in this thesis.

> F. Perez-Diaz, S. M. Trenkwalder, R. Zillmer, and R. Groß (2016). 'Emergence and inhibition of synchronization in robot swarms.' In 'Proc. 2016 Int. Symp. Distrib. Auton. Robot. Syst. (DARS 2016),' Berlin, Germany: Springer
>
> Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß (2016). 'Supervisory control theory applied to swarm robotics.' *Swarm Intell.*, 10(1):65–97
>
> Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß (2017). 'Probabilistic Supervisory Control Theory (pSCT) Applied to Swarm Robotics.' In 'Proc. 16th Conf. Auton. Agents and Multi-Agent Syst. (AAMAS 2017),' (pp. 1395–1403). Richland, SC: IFAAMS

1.5 Thesis Outline

This thesis is structured in six chapters, including the current introductory chapter. The remaining work is organised as follows:

> Chapter 2 investigates the question "How many computational resources does a relatively-simple robot possess?" and "What limitations can be expected regarding software?". To provide an answer to these question, Section 2.1 proposes a quantification for computational resources and then applies it to 38 robots and 5227 other computation devices, which results in a computation classification of devices. In Section 2.2 and 2.3, robotic system software and tasks are investigated with regard to their computational requirements and which computational classes of robots have performed them. Section 2.4 summarises and draws concluding remarks.
>
> Based on the conclusions drawn in the previous chapter, Chapter 3 investigates the concept of embedded operating systems for severely-constrained robots. In particular, Section 3.1 reviews how existing operating systems manage computational resources. Based on that, Section 3.2 proposes a new operating system with a novel execution model. In Section 3.3, the system is evaluated and compared to other operating systems and robotic system software. Section 3.4 summarises and draws concluding remarks.
>
> As communication is often considered a key aspect in swarm robotics, Chapter 4 investigates communication systems and their properties on swarm robots. First, existing communication systems are reviewed in Section 4.1. Then, the structure of a swarm robotics network is defined in Section 4.2. Section 4.3 derives a model describing the signal properties, which is subsequently used to design an optical ad-hoc network in Section 4.4. In Section 4.5, the proposed communication system and its novel dynamic detector are evaluated in several configurations, including static and dynamic settings. After comparing the performance of the system to an existing communication system, the chapter is summarised and concluded in Section 4.6.
>
> Chapter 5 presents a proof-of-concept that combines the insights and the work of the previous chapters by proposing, in Section 5.1, to utilise the computational resources of multiple robots via a network to solve a task that a single robot could not solve individually. In Section 5.2, the system is evaluated with real robots storing and processing data distributively. Section 5.3 summarises and concludes the chapter.

Finally, in Conclusions, this work's research findings and their limitations are presented. In addition, the chapter provides suggestions for potential future work.

2

Robotic Systems

Contents

Robotic systems, or robots, are machines designed to interact with their environment [Siciliano and Khatib 2016]. The interaction is enabled by sensors and actuators as well as a computer system that manages and coordinates those action. As robots are designed to perform a variety of tasks in a range of environments, many different types of robots exist. Typical categorisations are based on their physical properties (e.g., actuators), operation environment (e.g., grounded or airborne), field of application (e.g., medical or educational) or the number of robots (e.g., single or multi-robot systems) as shown in [Dobra 2014; Siciliano and Khatib 2016; IEEE 2019]. While each of these categories describe a robot's interaction capabilities, the computational aspects, which are paramount to perform actions, are commonly overlooked.

The computation is commonly performed by *embedded systems* [Marwedel 2006]. These systems can operate from millions to multiple billions of Instructions Per Second (IPS) with a few kilobytes to tens of gigabytes of memory, respectively. For instance, an ATMega 328 can process $2.0 \cdot 10^7$ IPS and access 2 kB of primary memory (RAM) and an Intel i7-9700 can process $1.6 \cdot 10^{11}$ IPS and access up-to 128 GB of RAM. While high processing speeds enable complex calculations in short time, it is often traded for low-power (i.e., longer mobility and lower heat production), reduced size, weight, and costs. These properties are essential particularly to areas deploying many miniature robots — such as swarm robotics.

Bearing in mind this trade-off, the specification of the number of resources during the design of a robot is often based on experience and convictions instead of systematic selection. This stems from a lack of systematic comparison[2.0.I] of resources and a lack of knowledge of the requirements of future applications[2.0.II]. As a result, this chapter investigates:

- how to quantify resources of computer systems,

[2.0.I] In particular, it is difficult to compare systems providing more memory to the ones with more processing power.
[2.0.II] When trading off one resource against another, a system can be constrained to a smaller number of tasks that it can perform. For instance, image processing on an e-puck robot [Mondada et al. 2009] is only possible on a small subset of the image, as the requirements to process a complete frame exceed the resources of the robot.

Table 2.1: Classification of constrained IoT/WSN devices [Bormann et al. 2014].

Class	RAM	ROM
CCD 0	$\ll 10$ kB	$\ll 100$ kB
CCD 1	~ 10 kB	~ 100 kB
CCD 2	~ 50 kB	~ 250 kB

- how robots can be compared and classified,
- what robotic software can be deployed, what computational resources it requires, and what function it provides.

2.1 Computational Quantification & Classification

This work investigates devices based on a *random-access machine* (i.e., a Turing machine) with finite memory length as defined in [Robič 2015]. In other words, a device must contain at least one *microprocessor unit* (MPU) that executes instructions on registers and randomly accesses memory[2.1.I]. This definition covers the majority of devices including robots. However, systems such as quantum computers or biological systems are not considered as computation is performed differently and these systems are rarely used in robotics.

Table 2.2 shows 46 common robots and their resources. It is divided into three sections — non-miniature robots (grey), miniature robots (blue), and micro-/nanorobots (green). In this work, a robot is miniature if it is milli- or centimetre-sized as it is used in the literature (e.g., [Martel et al. 2001; Nardi and Holland 2007; Bonani et al. 2010]). Micro- and nanorobots are considered to be sub-millimetre sized.

While there is a lack of quantification and classifications standards in robotics, Bormann et al. [2014] provides this for the Internet-of-Things (IoT) and Wireless Sensor Networks (WSN). Bormann et al. [2014] quantifies the computational resources solely based on the available memory (RAM and ROM) and introduces three classes of constrained devices (CCD) — CCD 0, 1, and 2 — as shown in Table 2.1. However, when applying this to a list of robots shown in Table 2.2, it is evident that many robots exceed the limits of the classification. Therefore, the classification is insufficient for robotics.

When investigating the robots of Table 2.2, it can be seen that their computational resources vary from severely limited [e.g., a single 8-bit MicroController Unit (MCU)] to virtually unlimited for most software (e.g., two servers within one robot). However, it is currently not clear how to quantify computational power (e.g., how an increase in processing power compares to an increase in memory).

[2.1.I] Random-access memory (RAM) is memory where any element can be written to/read from by the processor independently of any previous access.

Table 2.2: Non-miniature[a] (grey), miniature[b] (blue), and nano-/micro-robots[c] (green), their computational resources, and classification. The table includes only robots for which data could be obtained from publications, datasheets, manuals, or project websites. Note that *Entert.*, *Educ.*, *Med.*, and *Reconf.* stand for entertainment, educational, medical, and reconfigurable, respectively.

Robot	MPU/MCU	$\frac{\text{Cores}}{\text{MPU}}$	Arch.	Freq.	RAM	ROM	Application Environment	Group	Network	Group Size	CCD[d]	M_I P_I	C_I	Class
Baxter [Ju et al. 2014]	Intel i7	4	64 bit	3.4 GHz	4 GB	128 GB	Stationary Automation	Top-half Humanoid	Ethernet WiFi	Single	> 2	9.6 11.0	31.7	C_2
icub [Metta et al. 2008]	Intel Core Duo 2	2	64 bit	2.3 GHz	1 GB	-	Ground	Humanoid	Wired (CAN)	Single	> 2	9.0 10.3	29.7	C_2
Nao [Gouaillier et al. 2008]	Intel Atom	4	32 bit	1.91 GHz	4 GB	32 GB	Ground	Humanoid	Ethernet WiFi Bluetooth	Single Multi-Robot	> 2	9.6 10.3	30.1	C_2
Parrot AR.Drone 2.0 [Pestana et al. 2013]	ARM Cortex–A8	1	32 bit	1 GHz	1 GB	-	Air	Quadcopter	WiFi	Single Multi-Robot	> 2	9.0 9.3	27.6	C_2
PR2 [Willow Garage 2017]	2× Intel i7 Xeon	4	64 bit	3.4 GHz	4× 24 GB	2× 500 GB	Automation	Top-half Humanoid	Ethernet, WiFi Bluetooth	Single	> 2	11.0 11.4	33.8	C_2
Romeo [Gouaillier et al. 2008]	Intel Atom	4	32 bit	1.91 GHz	4 GB	10 GB	Ground	Wheeled Humanoid	Ethernet WiFi Bluetooth	Single	> 2	9.6 10.3	30.1	C_2
Roomba 980 [Tribelhorn and Dodds 2007]	ARM Cortex-M3	1	32 bit	72 MHz	16 MB	10 MB	Ground	Wheeled Vacuum	Ethernet WiFi	Single	> 2	7.2 8.0	23.1	C_2
Valkyrie [Radford et al. 2015]	3× Tegra3 ARM A9	4	64 bit	1.6 GHz	2 GB	-	Ground	Humanoid	Ethernet WiFi	Single	> 2	9.8 10.7	31.1	C_2
Youbot [Bischoff et al. 2011]	Intel Atom	2	32 bit	1.66 GHz	2 GB	32 GB	Ground	Wheeled Automation	Ethernet WiFi	Single Multi-Robot	> 2	9.3 9.9	29.2	C_2
AIBO ERS-7 [Fujita and Enteretainment 2000]	MIPS R7000	1	64 bit	576 MHz	64 MB	4 MB	Ground	Entert. Quad-Pedal	-	Single	> 2	7.8 8.8	25.3	C_2
AMiR [Arvin et al. 2009]	ATMega 168	1	8 bit	8 MHz	1 kB	16 kB	Ground	Wheeled Educ.	Infra-Red	Swarm	0	3.0 6.9	16.8	C_1
Cellulo [Özgür et al. 2017]	PIC32MZ	1	32 bit	200 MHz	512 kB	1 MB	Ground	Educ.	Bluetooth	Single Multi-Robot	> 2	5.7 8.3	22.3	C_1
Colias [Arvin et al. 2014]	ATMega 168 ATMega 644	1 1	8 bit 8 bit	20 MHz 20 MHz	1 kB 4 kB	16 kB 64 kB	Ground	Wheeled Educ.	Infra-Red	Multi-Robot	0	3.7 7.7	19.1	C_1
CrazyFly 2.0 [Giernacki et al. 2017]	STM32F405RG	1	32 bit	168 MHz	196 kB	1 MB	Air	Quatcopter	Bluetooth	Single Multi-Robot	> 2	5.3 8.2	21.7	C_1
Droplet [Farrow et al. 2014]	Xmega128A3U	1	8 bit	32 MHz	8 kB	128 kB	Ground	Wheeled Research	Infra-Red	Multi-Robot	1	3.9 7.5	18.9	C_1
e-puck [Mondada et al. 2009]	dsPic30	1	16 bit	7 MHz	8 kB	144 kB	Ground	Wheeled	Bluetooth Infra-Red	Swarm	1	3.9 6.9	17.7	C_1
Elmenreich's robot [Elmenreich et al. 2015]	ATmega328p	1	8 bit	8 MHz	2 kB	32 kB	Ground	Hexapedal	Infra-Red	Multi-Robot	0	3.3 6.9	17.7	C_1
Evo-bot [Escalera et al. 2016]	PIC24	1	16 bit	16 MHz	8 kB	128 kB	Floating	Reconf.	Wired (CAN)	Multi-Robot	1	3.9 7.2	18.3	C_1

Continued on the next page

Table 2.2: Continued

Robot	MPU/MCU	Cores / MPU	Arch.	Freq.	RAM	ROM	Application Environment	Group	Network	Group Size	CCD[d]	M_I / P_I	C_I	Class
GRITSBot [Pickem et al. 2015]	Atmega 328 Atmega 168	1 1	8 bit 8 bit	8 MHz 20 MHz	2 kB 1 kB	32 kB 16 kB	Ground	Wheeled	ANT Infra-Red	Swarm	0	3.5 7.4	18.4	C_1
GoPiGo [Pierson et al. 2017]	Rasberry Pi 3	4	64 bit	1.2 GHz	1 GB	8 GB	Ground	Educ. Wheeled	-	Single Multi-Robot	> 2	9.0 9.8	28.5	C_2
HyMod [Parrott et al. 2016]	ARM Cortex M4	1	32 bit	72 MHz	64 kB	256 kB	Ground	Modular Wheeled	Wired (CAN)	Multi-Robot	2	4.8 8.0	20.7	C_1
I-Swarm[e] [Seyfried et al. 2005]	Synopsys 8051	1	8 bit	12 MHz	2 kB	8 kB	Ground	Wheeled	Wired	Swarm	0	3.3 7.1	17.5	C_1
Jasmine[f]	ATMega168	1	8 bit	20 MHz	1 kB	16 kB	Ground	Wheeled	Infra-Red	Swarm	0	3.0 7.4	17.8	C_1
Khepera IV [Soares et al. 2016]	ARM Cortex-A8	1	32 bit	800 MHz	512 MB	4 GB	Ground	Wheeled	WiFi Bluetooth	Multi-Robot	> 2	8.7 9.2	27.1	C_2
Kilobot [Rubenstein et al. 2012]	ATMega328	1	8 bit	8 MHz	2 kB	32 kB	Ground	Mobile	Infra-Red	Swarm	0	3.3 6.9	17.2	C_1
Kobot [Turgut et al. 2008]	PXA255	1	32 bit	200 MHz	32 MB	32 MB	Ground	Mobile	Infra-Red	Swarm	>2	7.5 8.3	24.1	C_2
Lego Mindstorms NXT [Grega and Pilat 2008]	ATMEL AT91 ATMega48	1 1	32 bit 8 bit	48 MHz 8 MHz	64 kB 512 B	256 kB 4 kB	-	- Educ.	Bluetooth (USB)	Single	2 0	4.8 7.7	20.3	C_1
M-Block [Romanishin et al. 2015]	STM32F051 ARM Cortex-M0	1	32 bit	48 MHz	8 kB	64 kB	Ground	Reconf. Jumping	ANT Bluetooth	Multi-Robot	1	3.9 7.6	19.3	C_1
marXbot [Bonani et al. 2010]	i.MX31 (ARM 11)	1	32 bit	533 MHz	128 MB	-	Ground	Wheeled	WiFi Bluetooth	Multi-Robot	> 2	8.1 8.8	25.8	C_2
MHP [Doyle et al. 2016]	ARM Cortex M4	1	32 bit	72 MHz	64 kB	256 kB	Underwater	Modular	Infra-Red	Multi-Robot	2	4.8 8.0	20.7	C_1
MicroMVP [Yu et al. 2017]	ATMega328	1	8 bit	8 MHz	2 kB	32 kB	Ground	Wheeled	Infra-Red	Multi-Robot	0	3.3 6.9	17.2	C_1
Micro Quadrotor [Kushleyev et al. 2013]	ARM Cortex-M3	1	32 bit	72 MHz	16 MB	32 MB	Air	Quadcopter	ZigBee	Swarm	> 2	7.2 8.0	23.1	C_2
Mona [Arvin et al. 2018]	ATMega328	1	8 bit	8 MHz	2 kB	32 kB	Ground	Wheeled	Radio	Multi-Robot	0	3.3 6.9	17.2	C_1
Monsun II [Osterloh et al. 2012]	Blackfin BF537	1	16 bit	500 MHz	32 MB	40 MB	Underwater	UAV	Bluetooth (Wired)	Multi-Rrobot	> 2	7.5 8.7	24.9	C_2
mROBerTO [Kim et al. 2016]	Nordic nRF51422	4	32 bit	16 MHz	32 kB	256 MB	Ground	Wheeled Educ.	Bluetooth ANT	Multi-Robot	2	4.5 7.8	20.1	C_1
Pheeno [Wilson et al. 2016]	ARM Cortex-A7 ATmega328p	4 1	32 bit 8 bit	900 MHz 8 MHz	1 GHz 2 kB	- 32 kB	Ground	Wheeled Educ.	WiFi Bluetooth	Multi-Robot	> 2	9.0 9.5	28.1	C_2
r-one [McLurkin et al. 2013]	TI LM3S8962	1	32 bit	50 MHz	64 kB	256 kB	Ground	Wheeled	ZigBee	Multi-Robot Swarm	2	4.8 8.0	20.8	C_1
s-bot [Mondada et al. 2004]	Intel XScale	1	32 bit	400 MHz	64 MB	32 MB	Ground	wheeled	WiFi	Swarm	> 2	7.8 8.9	25.6	C_2

Continued on the next page

Table 2.2: Continued

Robot	MPU/MCU	$\frac{\text{Cores}}{\text{MPU}}$	Arch.	Freq.	RAM	ROM	Application Environment	Group	Network	Group Size	CCD[d]	M_I P_I	C_I	Class
Soft Robotic Fish [Marchese et al. 2014]	ATMega 644	1	8 bit	20 MHz	4 kB	64 kB	Underwater	Research Soft	ZigBee	Single	0	3.6 7.3	18.2	C_1
Thymio II [Riedo et al. 2013]	PIC24	1	16 bit	8 MHz	16 kB	128 kB	Ground	Wheeled	IEEE 802.15.4	Swarm	1	4.2 6.9	18.0	C_1
TurtleBot 3 (Burger) [Guizzo and Ackerman 2017]	Rasberry Pi 3 ARM Cortex-M7	4 1	64 bit 32 bit	1.2 GHz 216 MHz	1 GB 320 kB	- 1 MB	Ground	Wheeled Educ.	USB Ethernet	Single	>2	9.0 9.2	27.4	C_2
UltraSwarm [Nardi and Holland 2007]	Gumstix SBC ARM7	1	32 bit	40 MHz	2 GB	-	Air	Helicopter	Radio-based	Multi-Robot	> 2	9.3 7.6	24.4	C_2
Wanda [Kettler et al. 2010]	TI LM3S1960 Bluetechnix CM-BF561	1 2	32 bit 16 bit	50 MHz 600 MHz	64 kB 64 MB	256 kB 8 MB	Ground	Wheeled Educ.	Infra-Red ZigBee	Multi-Robot	> 2	7.8 9.1	26.0	C_2
WolfBot [Betthauser et al. 2014]	ARM Cortex-A8	1	32 bit	1 GHz	512 MB	4 GB	Ground	Wheeled Educ.	WiFi ZigBee	Multi-Robot	> 2	8.7 9.0	26.7	C_2
X4-MaG [Manecy et al. 2015]	dsPic33F ARM Cortex-A8	1 1	16 bit 32 bit	40 MHz 1.2 GHz	8 kB 512 MB	128 kB -	Air	Quadcopter	RS232 WiFi	Swarm	> 2	8.7 9.1	26.9	C_2
Capsule-type Microrobot [Lee et al. 2018]	-	0	-	0 Hz	0 B	0 B	Floating	Mobile Med.	-	-	< 0	0.0 0.0	0.0	$C0$

[a] A selection of common non-miniature robots has been taken from canonical robotic challenges of RoboCup and DARPA.

[b] This selection of miniature robots have been obtained from various sources — including [Siciliano and Khatib 2016; Hamann 2018; Wikimedia Foundation 2019*a*; Moubarak and Ben-Tzvi 2012]. A robot has been added if it is miniature and details of its computational capabilities could be obtained from publications, datasheets, manuals, or project websites. If the robot has not been used in publication within the last 10 years (e.g., Alice) or technical specification is not available (e.g., Anki Vector, Cubelets, Dash, Root, and Sphero), the robot has not been included.

[c] Only one nanorobot has been added as nanorobots solely reacts on environmental changes through physical or chemical mechanisms and lacks a computational component that can be programmed; hence, any of these robots provides the same amount of computational resources (none).

[d] The class of constrained devices proposed in [Bormann et al. 2014] is abbreviated with CCD.

[e] All details were taken from the I-Swarm project homepage (http://www.i-swarm.org/MainPage/Robots/R_Description1.htm).

[f] All details were taken from the Jasmine project homepage (http://www.swarmrobot.org/GeneralDesign.html).

2.1.1 Computational Index

As any computer system can be described by processing power (i.e., instructions per second) and memory (i.e., number and length of elements that can be accessed), two indexes are proposed — a memory index, M_I, and a processing index, P_I:

$$M_I = \log\left(1 + m\right), \tag{2.1}$$

$$P_I = \log\left(1 + \sum_i \underbrace{n_i\, f_i\, e_i}_{\text{IPS per MPU}}\right), \tag{2.2}$$

where m is the available primary memory[2.1.II] in bytes and n_i, f_i, and e_i[2.1.III] are the number of cores, the operation frequency, and the average amount of instructions per clock cycle of an MPU i. Note that both M_I and P_I are chosen logarithmically to linearise the exponential growth of computational resources over time (Moore's Law) and to better describe the magnitudes of power difference between high-end microprocessors and low-end microcontrollers.

While a single index quantifies the magnitude of either processing power or memory, it often is not sufficient to classify a system for two reasons. First, single resources can be increased without increasing the system's capabilities. For instance, e-pucks do not have enough memory to store a single image of its on-board camera and increasing processing power does not enable a robot to load and process that frame. Second, processing power and memory are not independent as implementations of an algorithm can prioritise memory consumption or execution time while performing the same action. As a result, the relation between processing power and memory needs to be found.

In cryptoanalysis, the relation between processing time and memory is proposed by [Hellman 1980] in form of a time-memory trade-off. It is used to calculate a general *one-way function*[2.1.IV] inverter. Hellman [1980] describes that a function can be sped-up by pre-calculating lookup tables (increasing memory consumption) or its memory can be reduced by recalculating values. Due to the complexity of the inverters, this method applies to functions with complexity of NP or less — even outside of cryptoanalysis. The trade-off is formulated by

$$m\, t^2 = k = \textit{const.}, \tag{2.3}$$

where m and t are the respective memory and processing time for a given implementation of a given algorithm. The algorithm-specific constant, k, is a simplification as it depends on multiple parameters (see [Hellman 1980] for more details).

Assuming an algorithm is implemented to use all of the available memory, $m = m_{max}$, the processing time is a minimum, $t = t_{min}$. When the processing power, p', is changed to p, (2.3) becomes

$$m_{max}\, (t'_{min})^2 = m_{max} \left(t_{min}\, \frac{p}{p'}\right)^2 = k', \tag{2.4}$$

$$m_{max}\, p^2 = \frac{(p')^2\, k'}{t_{min}^2} = \textit{const.}, \tag{2.5}$$

[2.1.II] Note that m is the size of the primary memory (i.e., RAM) as the processor has direct random access as described by a random-access machine. As data from the secondary (e.g., FLASH, hard drives) and tertiary memory (e.g., SD Cards, cloud storage) is first transferred to the primary memory before it can be accessed, the primary memory is the defining factor and, therefore, is used.

[2.1.III] Note that e_i is not available or documented for every platform. This value can vary from high-end microprocessors providing larger values (e.g., Intel i7 7500: $e_i = 9.1$, AMD Ryzen 7 1800x: $e_i = 10.6$) to low-end microcontrollers approaching 1 (e.g., ATmega128: $e_i = 1.1$, DSPic30F: $e_i = 1.05$). In case e_i is not available, it is set to 1 (worst-case).

[2.1.IV] A one-way function and its inversion are functions with a maximum of polynomial (P) and a minimum of nondeterministic polynomial (NP) complexity, respectively.

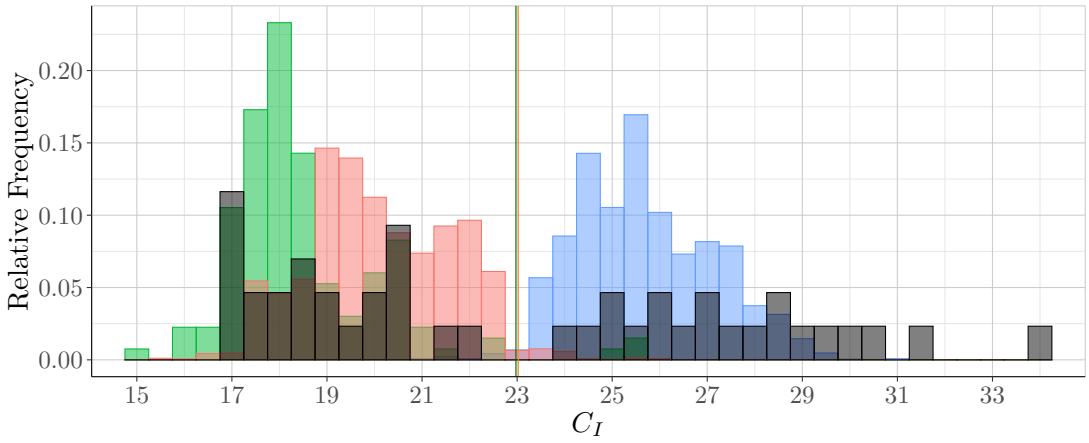


Figure 2.1: Histogram of the computational index values of 5264 devices including robots from Table 2.2 (black) as well as sensor network nodes (green), microcontrollers (red), and embedded computer systems (blue) from Appendix C.

where the available memory and the square of the processing power are constant for the given algorithm and processing time, t_{min}. In other words, any system satisfying (2.5) can process an algorithm satisfying (2.3) within t_{min}; therefore, the systems are equally computationally powerful. By combining (2.1), (2.2), and the logarithm of (2.5), this work proposes the computational index,

$$C_I = M_I + 2\,P_I, \tag{2.6}$$

which is applied to Table 2.2.

2.1.2 Computational Classification

When applying (2.6) to a wide range of other computer systems[2.1.V], it can be seen in Figure 2.1 that the values of C_I populate two regions. The lower region ($C_I < 23$) is mostly populated by microcontroller unit (MCU)-driven devices and the higher region ($C_I > 23$) is mostly populated by embedded computer systems. As shown in Figure 2.1 (black), the robots of Table 2.2 populate both regions separated by a gap at $C_I = 23$.

Due to the existence of these two regions, robots are grouped into two sets referred to as severely-constrained ($C_I \leq 23$) and weakly-constrained robots ($C_I > 23$). The threshold of 23 is estimated based on the data set of Figure 2.1 indicating the middle of the gap between both sets. Robots that do not compute (e.g., [Fusco et al. 2014; García-López et al. 2017; Lee et al. 2018]) are referred to as non-computational robots. When robots accessing external infrastructure and the combined computational resources exceed the capabilities of any individual computer system by magnitudes (e.g., [Terrissa et al. 2015; Wan et al. 2016]), the system is referred to as minimally-constrained robot. To simplify the referencing, let class C_0, C_1, C_2, and C_∞ refer to non-computational, severely-constrained, weakly-constrained, and minimally-constrained robots, respectively. Note that this classification is shown in Table 2.2.

2.1.3 Discussion

The presented computational indices quantify computational resources and enable them to be compared. They can also be used to classify the entire spectrum of robots reaching from non-computational to minimally-constrained robots.

[2.1.V] In total, 5227 devices are used including sensor network nodes, microcontrollers, and embedded systems, which are listed in Appendix C.

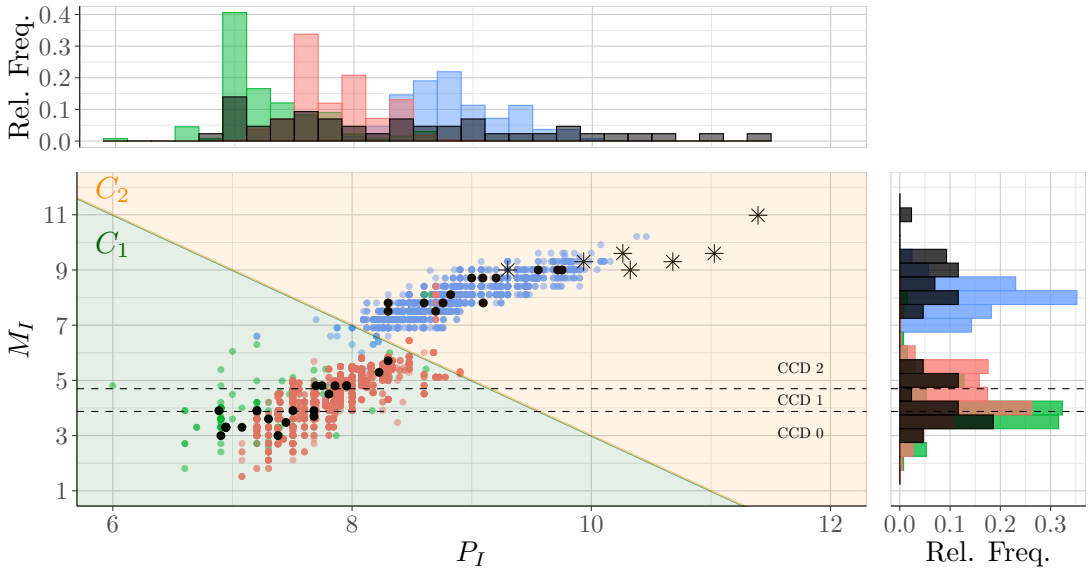


Figure 2.2: Memory (M_I) and processing indices (P_I) of 5273 devices including robots of Table 2.2 (black) as well as sensor network nodes (green), microcontrollers (red), and embedded computer systems (blue) from Appendix C. On the top and the right are histograms of P_I and M_I, respectively. Note that two horizontal lines indicate the border between classes proposed by [Bormann et al. 2014]. Stars indicate non-miniature robots of Table 2.2.

The scatter plot of Figure 2.2 shows two distinct sets of devices. While the two sets are less distinguishable when considering the processing power [see Figure 2.2 (top)], the distribution of available memory shows a clearly distinguishable gap between both sets [see Figure 2.2 (right)]. This can be explained by the fact that integrated memory on a processor die (i.e., MCU) is technologically constrained regarding size and circuit complexity. On the other hand, different components for memory and processor allow the use of cheap and large memory with high operation frequency; hence, the gap. While separate memory and processor components are beneficial regarding computational power, integrated systems commonly use less power, are smaller, and cheaper.

When investigating the computational resources of robots, Figure 2.2 (black) reveals that miniature robots (dots) tend to provide less resources than non-miniature robots (stars). While miniature robots can be within C_1 or C_2, the majority (61 %) are severely-constrained. Interestingly, C_1 robots are almost exclusively deployed in research environments. This potentially stems from technical hurdles, such as few computational resources. Recent work, such as [Jones et al. 2018], supports this argument as it extends the computational resources of miniature robots to overcome specifically the reality gap.

It could be argued that computational resources of future systems would increase in line with Moore's law, which would shift robots into C_2. While this might shift the threshold between C_1 and C_2, it can be expected that the gap between single-chip (i.e., MCU-based) systems and systems with discrete components remains due to the technological differences stated before. Furthermore, robots — such as [Wood et al. 2013] — rely on further miniaturisation, which itself favours MCUs due to their size, weight, and power consumption.

When comparing the proposed classification with [Bormann et al. 2014], both classifications are compatible as any class of [Bormann et al. 2014] is classified as C_1. Consequently, Bormann et al. [2014] can be seen as a sub-classification. However, Bormann et al. [2014] use ambiguous[2.1.VI] ranges, which makes the classification difficult to apply. When applying

[2.1.VI] It is not clear in which class, for instance, a PIC18F67K40 with 3.5 kB off RAM and 128 kB of ROM falls.

the Bormann's classification to 5273 devices (particularly C_1 devices) as shown in Figure 2.2, the benefits of the classification is questionable as the devices are distributed across all classes without showing any denser areas or gaps between them. Furthermore, it is based solely on memory, which is not always sufficient as described previously. In comparison, the proposed classification's ranges are based on empirical data.

2.2 Robotic System Software

After investigating what and how many computational resources are provided by robots, let us examine which software is deployed on robots. Fundamentally, software can be divided into two categories — *application software* and *system software*. Application software is software that performs a user-defined behaviour or solves a given task. System software, on the other hand, is any software that controls and manages the system without implementing a specific behaviour or application. As a result, system software provides a platform to implement applications while reducing complexity as well as increasing deployability and resuability. Examples are firmware, device drivers, operating systems, and domain-specific language compiler/interpreter.

In this section, a series of robotic system software is discussed and its key-aspects identified. Finally, the computational requirements of system software are analysed.

2.2.1 Cloud-Enabled System Software

One solution to compensate for computational constraints is to outsource the required resources to an external infrastructure — a *cloud*. A cloud (or cloud computing environment) is an infrastructure composed of a pool of computer resources that provide services to other systems. A could typically contains computational resources magnitudes higher than an individual computer system. Currently, there are two approaches to combine cloud computing and robotics — *Robots-as-a-Service* (RaaS) [Chen et al. 2010; Terrissa et al. 2015] and *cloud robotics* [Goldberg and Kehoe 2013; Kehoe et al. 2015].

RaaS describes cloud applications that use robots, where a robot is a single resource (unit) based on a service-oriented architecture[2.2.1] [Chen et al. 2010]. For instance, in the Robot Cloud Center (RCC) [Du et al. 2011] — a RaaS implementation —, tasks are selected by a user and the software transparently selects a robot and accesses its capabilities remotely.

While RaaS solutions focus on applications utilising robots, in cloud robotics, robots utilise the cloud to store data or perform demanding computations [Wan et al. 2016]. Cloud robotic systems can be divided into systems that provide information (e.g., RoboEarth) and that process information (e.g., Rapyuta). RoboEarth is a repository where networked robots store/share information and collaborate [Waibel et al. 2011]. It can be seen as a knowledge database for different platforms. In contrast, Rapyuta remotely processes tasks in one or more environments of high-performance clusters [Mohanarajah et al. 2015]. Furthermore, it allows access to RoboEarth, ROS packages as well as other robotic middleware (see Section 2.2.2).

Fundamentally, both approaches are a special case of robotics where additional infrastructure is needed (i.e., clouds). As the system software on both a robot and on the cloud needs to be considered as robotic system software, determining the computational requirements for a single robot would be insufficient. Therefore, the computational classification of the entire system is a minimally-constrained robotic system (i.e., C_∞) as it exceeds multifold the computational resources of an individual computer system.

While the robots can be designed more cost-efficiently thanks to their reduced computational requirements, continuous access to the cloud is paramount. Therefore, the system's per-

[2.2.1] Service-oriented architecture is a design paradigm in which resources are available as services [Erl 2012]. Similar to a client-server architecture, the resource can be operated by components or applications.

formance depends on the quality of service of the network, which is a significant restriction. In swarm robotics, such a reliable network connection is often not feasible for two reasons. Firstly, swarm robots are often equipped with simplistic communication methods that cause long latencies, low throughput, and relatively weak error-correcting capabilities (see Chapter 4 for more details). This limits the transmittable data and the quality of the data. Secondly, these systems are designed for a single or small groups of robots. It is not clear if these systems can scale to large swarms, in particular when considering the network to the cloud. High-density swarms would face complications as hundreds or thousands of robots would need a reliable communication within a small space at the same time.

2.2.2 Robotic Middleware

When reliable communication to the external infrastructure is not available, local processing is often a more practical approach. On the majority of modern computer systems, operating systems are used to improve the development and execution of software. These systems provide limited interaction options with other devices (except communication) or their environment. As one of the main purposes of robotics is to interact with its environment, robotic *middleware* can be used to facilitate these robotic aspects.

A middleware is any system software that is executed between applications and an operating systems. It provides features to measure, act, and execute behavioural algorithms on a generic operating system. In this section, common robotic middleware[2.2.II] — Miro, ORoCoS, Player, ROS — are discussed and their computational requirements analysed. Note these systems were selected as they are widely used for RoboCup, in real-time & safety-critical environments, for research, and in industry, respectively.

2.2.2.1 Miro

Miro[2.2.III] is an open-source middleware aiming to improve the software development process of mobile robots [Utz et al. 2002]. Miro provides three layers:

- Miro Device Layer provides an abstraction for any platform-specific hardware (e.g., sensors and actuators),
- Miro Service Layer allows access to actuators and sensors through Common Object Request Broker Architecture (CORBA)[2.2.IV], and
- Miro Class Framework provides common features and functions (e.g., localisation or path planning routines).

This layered structure and its object-oriented design make Miro adaptable and extendable [Kruger et al. 2006]. Furthermore, with the use of CORBA, distributed control is possible through its cross-platform and inter-process interoperability. However, Miro does not provide real-time properties.

2.2.2.2 ORoCoS

The Open Robot Control Software[2.2.V] (ORoCoS) is an open-source, object-oriented, real-time control software for industrial robots and machines [Bruyninckx et al. 2003]. ORoCoS provides four elements:

- Real-Time Toolkit provides real-time functions and features,
- Kinematics and Dynamic Library allows the calculation of kinematic chains,

[2.2.II] A comprehensive list of robotic middlewares can be found in [Namoshe et al. 2008; Mohamed et al. 2008; 2009; Elkady and Sobh 2012].
[2.2.III] For further details: https://sourceforge.net/projects/miro-middleware.berlios/ (Miro).
[2.2.IV] CORBA is an interface description language designed for seamless communication in heterogeneous systems. This allows the usage of the interface in different programming languages and across different platforms.
[2.2.V] For further details: http://www.orocos.org/ (ORoCoS).

- Bayesian Filtering Library implements efficiently filtering methods (e.g., Kalman Filters [Grewal and Andrews 2014] and Particle Filters [Ristic 2015]), and
- ORoCoS Component Library provides common features (similar to Miro Class Framework).

Target platforms of ORoCoS are stationary industrial robots. They focus on fixed sequences of high-precision steps. However, in many mobile robot applications, dealing with unknown environments, overcoming their limitation, and mobility takes priority over high-precision actions.

2.2.2.3 Player

Player[2.2.VI] is a middleware that has widely been used in academia [Collett et al. 2005]. The open-source software runs on POSIX[2.2.VII] operating systems. Player uses a client-server architecture[2.2.VIII] to link a server executed on the robot (server) to the control software executed on a computer (client). Each part of the server (e.g., each sensor or actuator) is controlled by a thread, which is executed by the operating system.

A reason for Player's popularity in academia is its integration with Stage (a simulation tool), which allows swift simulation results. However, the need for remote control of robots is a significant drawback outside academia.

2.2.2.4 ROS

The Robot Operating System[2.2.IX] (ROS) is an open-source, distributed framework with the largest community in research and industry [Quigley et al. 2009]. Despite its name, ROS is not an operating system, as defined in [Tanenbaum 2009; Stallings 2014], as it is executed on top of an operating system.

ROS is a collection of tools and function components, called *nodes*, connected through a *publish-subscribe architecture*[2.2.X]. A single node commonly provides a single function — such as control software, hardware abstraction, or complex algorithms (e.g., SLAM). When a message (in this case, an XML-based RPC[2.2.XI]) is published, any subscribed node receives the data — even when executed distributively. This enforces clear interfaces, which improves the reusability and modularity of the code.

Despite its distributed design, ROS uses a central control component, called ROS Master, to coordinate messages and subscriptions. Research has shown that single nodes can be executed on severely-constrained devices [Bouchier 2013]. However, ROS requires an active ROS Master, which requires an Ubuntu Linux, to perform. The drawback of requiring a central unit (i.e., a single point of failure) motivated the development of ROS 2[2.2.XII]. The ROS Master functionality is replaced by a distributed middleware, Data Distribution Service (DDS), bypassing the need for a central unit. However, this has increased the computational and communication overhead considerably.

[2.2.VI] For further details: http://playerstage.sourceforge.net/ (Player).
[2.2.VII] The Portable Operating System Interface (POSIX) is a standardised interface between applications and some operating systems — commonly UNIX-based platforms.
[2.2.VIII] A client-sever architecture describes a system where commonly a single server provides access to resources to multiple clients which are connected via a network.
[2.2.IX] For further details: http://ros.org/ (ROS).
[2.2.X] A publish-subscribe architecture describes a system that allows any components to subscribe (i.e., allow receiving of data) and publish (i.e., transmitting of data) to a common data channel.
[2.2.XI] A remote procedure call (RPC) is a message that, when received, causes the execution of a function.
[2.2.XII] ROS 2 is in the early stages of development and has been first released in 2018 at https://github.com/ros2/ros2.

Table 2.3: Computational requirements for common generic operating systems.

Operating System	C_I
Linux[a]	26.5
macOS[b]	28.6
Windows[c]	25.6

[a]The requirements for Linux are based on the embedded version, Ubuntu Mate 18.04, as the current version of ROS (Melodic Morenia) requires it.

[b]The requirements for Apple's macOS are based on macOS Mojave 10.14.

[c]The requirements for Microsoft Windows are based on the embedded version, Windows 10 IoT.

2.2.2.5 Discussion

A robotic middleware uses the computational capabilities of an operating system and extends them to access the robot's capabilities. A middleware aims to decrease developmental efforts while increasing portability and maintainability of the software.

While each robotic middleware provides different merits, they share specific properties.

- Each middleware provides a layer (Miro and ORoCoS), a thread (Player), or units/nodes (ROS) to abstract robotic hardware — including actuators, communication, and sensors. This allows software to interact with the robot's environment transparently.
- Each system provides functions for modular software development. This improves the reuseability (i.e., one module can be used in multiple applications) and maintainability (i.e., malfunctions can be located at their respected self-contained module). Furthermore, Miro, OroCoS, and ROS provide a clear interface between modules — CORBA (Miro and ORoCoS) or RPC (ROS). This improves adaptability (i.e., modules can be exchanged easier) and allows the development with various programming languages.
- Each system provides a large set of commonly used algorithms and features to reduce development efforts.

As a result, it can be concluded that successful robotic middleware should provide hardware abstraction, modular software development, interface definitions, and a set of common features.

From a computational point of view, robots must be capable of executing both the operating system and the middleware. As current middlewares do not provide minimal computational requirements, the operating system's minimum requirements are considered the middleware's lower bound. The minimal requirements for common generic operating systems are shown in Table 2.3. Due to these high requirements, many miniature robots cannot utilise such system software.

2.2.3 Robotic Languages & Virtual Machines

Recently, an increasing number of research investigates a different form of system software — *Domain Specific Languages* (DSL) and their *virtual machines* (VMs) [Kosar et al. 2008]. A DSL combines commonly needed functions of a specific domain (in this case, robotics) into a language that often improves development time and reduces efforts. The code is written in the respected language, compiled, and executed or directly interpreted on a robot's VM.

In this section, the DSLs — URBI, Buzz, ASEBA, and supervisory control — and their VMs are discussed as they have been deployed and used on miniature robots.

2.2.3.1 URBI

The Universal Robotic Body Interface (URBI)[2.2.XIII] is an software attempting to provide a programming standard to program or control robots [Baillie 2004]. Similar to Player, URBI uses a client-server architecture, where the robot executes a URBI's server (i.e., VM). The client software (i.e., behaviour) can be executed remotely or locally, which increases the computational overhead. While the memory footprint is not publicly available, URBI is currently designed to run on APERIOS[2.2.XIV] that is designed for the Aibo robot; therefore, the memory footprint must be below $64\,\mathrm{MB}$ of RAM and $4\,\mathrm{MB}$ of ROM. As URBI is executed on top of APERIOS, it is also considered a robotic middleware.

The behaviour is implemented via an event-oriented script language that is interpreted by the URBI VM. The interpretation requires a parsing and lexical analysis during runtime, which causes a considerable computational overhead. As a result, the developers recommend implementing computational-intensive or platform-specific algorithms in other languages (e.g., C++) to improve performance [Baillie 2004].

2.2.3.2 Buzz

Buzz[2.2.XV] is a DSL executed on an VM [Pinciroli et al. 2015]. It is designed to enable the modelling of both a single robot and an entire heterogeneous group of robots. Buzz uses a stack-based VM that executes *bytecode*[2.2.XVI] (i.e., compiled script). The Buzz VM has a small memory footprint and has been implemented on the Khepera IV (i.e., a weakly-constrained robot). A limited version BittyBuzz [Beltrame and Dentinger 2019] has been implemented on a severely constrained robot (Kilobot) consuming $2\,\mathrm{kB}$ of RAM and $32\,\mathrm{kB}$ of ROM.

The language is designed as an extension language to widen the functionality of systems, such as ROS or ORoCoS. The script is compiled to a bytecode, which reduces the program size and the execution time on its VM. When executed, the VM iteratively obtains sensor/communication data, processes the bytecode, and finally applies data to actuators/communication devices. This sequentialisation ensures data and value integrity; however, it can also increase response times and delays.

2.2.3.3 ASEBA

ASEBA[2.2.XVII] is a DSL executed on an VM designed for robots with multiple MCUs connected via a fieldbus (e.g., [Bonani et al. 2010]) [Magnenat et al. 2011]. Each MCU executes an ASEBA VM and processes data locally reducing the load of the fieldbus. As a result, the volume of data and the latency between perception and action is reduced. Similar to Buzz, ASEBA uses also a stack-based VM executing bytecode. It is implemented on Thymio and e-puck, where the VM consumes $4\,\mathrm{kB}$ of RAM and $10\,\mathrm{kB}$ of ROM on the e-puck.

ASEBA uses an event-based script language that is compiled to a bytecode and uploaded to the robot. Similar to Buzz, the ASEBA VM sequentially senses, executes the script and actuates, which can increase latency and delays.

2.2.3.4 Supervisory Control

In contrast to the previous DSLs, *supervisory control* is a framework attempting to produce verifiable and testable behaviours [Lopes et al. 2016]. Based on the supervisory control theory [Ramadge and Wonham 1987], the automata are created and tested to guarantee properties

[2.2.XIII] For further details: http://sourceforge.net/projects/urbi/ (URBI).

[2.2.XIV] APERIOS is a proprietary real-time operating system developed by Sony for the Aibo robot [Fujita and Enteretainment 2000].

[2.2.XV] For further details: http://the.swarming.buzz/ (Buzz).

[2.2.XVI] A bytecode is a compact instruction set for a software interpreter — in most cases, a virtual machine.

[2.2.XVII] For further details: https://aseba.wikidot.com/ (ASEBA).

such as *deadlock*[2.2.XVIII] freedom. Automata are then compiled and processed on a robot's VM. This VM has been implemented on the e-puck and Kilobot and has a memory footprint of at minimum 13 kB of ROM (Kilobot).

Fundamentally, a behaviour is defined through automata composed of states connected by events (i.e., formal language). The automata is compiled to a finite-state machine (i.e., supervisor) that can be executed robot's VM. The behaviour is then achieved by a sequence of events that advance the state machine.

2.2.3.5 Discussion

A domain-specific language aims to ease the development efforts by providing a set of commonly used functions and notation for a specific domain of application. Commonly, a DSL introduces a high degree of abstraction, which allows behaviours to be implemented in a few lines of code, short time, and improved quality [Prechelt 2000; Kosar et al. 2008; MacConnell 1993]. However, a higher-level of abstraction is often traded for less-optimised run-time performance [Van Deursen et al. 2000].

The latter systems operate interpreters of source code (URBI), bytecodes (ASEBA and Buzz), or state information (supervisory control). In comparison to source code, interpreting bytecode reduces considerably the overhead, which allows a smaller memory footprint and faster execution. However, interpreted code — except the calling of *directly-implemented*[2.2.XIX] functions — is executed often with lower *execution efficiency*[2.2.XX] as shown in [Romer et al. 1996; Prechelt 2000; Ertl and Gregg 2003]. For example, ASEBA provides an execution efficiency of $\frac{1}{70}$ (averaged) as reported in [Magnenat et al. 2011].

While a language provides frequently-used features of its domain, it cannot be assumed that the DSL provides all required features for any possible scenario. Therefore, each language supports generic programming capabilities — such as branching, arithmetic, and logical operations — despite being often the least execution-efficient [Romer et al. 1996; Ertl and Gregg 2003]. Consequently, computationally-intensive tasks are often required to be implemented directly on the target platform and linked to the remaining code of the DSL. In other words, a DSL can be used with high-execution efficiency, if it is used to link direct implementations at a high level. In the light of the complexity of real-world tasks, it can be expected that behaviour needs to be implemented in multiple languages — the computational-intensive parts in a compiled language and the overall algorithm in the DSL. To avoid this, further research could be conducted to move from VMs to compilers allowing the generation of hardware-optimised code while providing a high level of abstraction.

When considering the computational requirements, the VMs of ASEBA, BittyBuzz, and supervisory control can be deployed on severely-constrained robots with 17.7, 17.2, and 17.2 (i.e., C_1 robots), respectively. In comparison, URBI have only been implemented on robots with an operating system and computational index of 25.3 (i.e., C_2 robots).

2.2.4 Comparison

When comparing the different robotic system software, the shared properties are:

1. abstraction of hardware (i.e., actuators, sensors, and communication) allowing fast high-level development, and
2. modular design capabilities allowing better adaptability and maintainability.

Many systems across all categories provide interfaces between modules — in particular, CORBA (Miro, ORoCoS), RPC (ROS), events (ASEBA, Urbi, supervisory control). This allows better

[2.2.XVIII] A deadlock is a situation where different components of a system indefinitely block each other from executing.

[2.2.XIX] In this work, direct implementation describes code that is compiled to hardware instructions.

[2.2.XX] In this work, execution efficiency describes how many operations are needed to execute a single DSL instruction. Note that this does not apply to supervisory control.

Table 2.4: Common robotic system software and their class of guaranteed deployability (CGD).

Task	CGD
(I) Cloud-enabled System Software	
Robot Cloud Center [Du et al. 2011]	C_∞
Rapyuta [Mohanarajah et al. 2015]	C_∞
RoboEarth [Waibel et al. 2011]	C_∞
(II) Robotic Middleware	
Miro [Utz et al. 2002]	C_2
ORoCoS [Bruyninckx et al. 2003]	C_2
Player [Collett et al. 2005]	C_2
ROS [Quigley et al. 2009]	C_2
(III) Robotic Languages & Virtual Machines	
ASEBA [Magnenat et al. 2011]	C_1
Buzz [Pinciroli et al. 2015]	C_1
Urbi [Baillie 2004]	C_2
Supervisory Control [Lopes et al. 2016]	C_1

reusability and sometimes distribution across a network.

To compare the computational requirements of robotic system software, this work introduces a Class of Guaranteed Deployability (CGD). The CGD is determined as follows: First, an extensive literature search for each system software was conducted. Then, all platforms on which the software could be deployed were identified and their C_I calculated. Finally, the CGD is the class of the platform with the smallest C_I. Consequently, the CGD indicates that the software can be performed by any robot of that class. Note that it is possible that the respected software could be performed by a system with lower C_I; however, no evidence was found that would support such a claim.

When comparing the CGD of robotic system software as shown in Table 2.4, it can be seen that each type of system software focuses on a different group of robots. Interestingly, robotic system software for more computational powerful robots — robotic middleware and cloud-enabled systems — provide a large set of features and libraries that are used by their community. In particular, RoboEarth is designed as a large repository for models, libraries, and features. In contrast, system software for severely-constrained robots often does not provide a comprehensive set of features. This could stem from multiple reasons:

- Weakly-constrained robots are more frequent, and system software for those robots tend to have larger communities (e.g., ROS).
- Weakly-constrained robots provide magnitudes higher computational power than severely-constrained robots. Therefore, tools and libraries can be implemented more generically and less hardware-optimised without significantly effecting the robot's behaviour.
- System software of severely-constrained robots often use large segments of platform-optimised code for more efficient execution and smaller memory footprint. As a result, this code is not portable to other platform without considerable reimplementation efforts.

Finally, the majority of system software can only be deployed on weakly-constrained robots, and only three systems were deployed on severely-constrained devices (ASEBA, Bitty-Buzz and supervisory control). This is likely to originate from the additional challenges, such as finding an appropriate trade-off between abstraction and performance. For instance, ASEBA provides a high degree of abstraction but poor execution efficiency in comparison to the presented supervisory control framework which has shown to be executed with high execution efficiency (see [Lopes et al. 2016]) but is modelled on a lower level of abstraction and is more

difficult to implement. While the author believes that performance should be prioritised, in particular on severely-constrained robots, abstraction simplifies the development, reduces the lines of code and, therefore, increases the software robustness.

As the majority of behaviours are still implemented directly (i.e., no abstraction), additional engineering and research efforts are required to improve software engineering methods on severely-constrained robots. As an alternative to the existing systems, this work presents another type of robotic system software — embedded operating systems (see Chapter 3 for more details).

2.3 Robotic Tasks

Robotics has many potential applications. Most notable are autonomous cars, industrial applications, warehouse automation, and search & rescue. Organisations such as the RoboCup[2.3.I] Foundation and DARPA[2.3.II] promote robotics research under real-world conditions. They provide real-world challenges that are commonly designed for individual robots or small groups of robots. Most robots used in those challenges are non-miniature and provide large amounts of computational resources (i.e., C_2).

To investigate the impact of computational constrains of miniature robots, the tasks in which such robots are used need exploring. Many miniature robots are deployed in reconfigurable robotics and swarm robotics. While most tasks in reconfigurable robotics investigate how to create and operate assembled robots of specific shapes, swarm robotics offers a larger variety of tasks aiming to solve problems in potential future applications. Consequently, this work focuses on swarm robotics tasks.

Overall, swarm robotics research often solve rudimentary tasks as they can be challenging due to: control of a large number of robots, lack of central control, and no access to global state information. These tasks are grouped into single-task and multi-task behaviours. A single-task behaviour describes cases that solve a single basic behaviour — containing (I) spatially-organising behaviours, (II) navigation behaviours, (III) collective decision making, and (IV) miscellaneous collective behaviours — and (V) multi-task behaviours combining multiple single-tasks [Şahin 2005; Navarro and Matía 2012; Brambilla et al. 2013; Bayındır 2016; Hamann 2018].

Table 2.5 shows that a single-task behaviour, (I)–(VI), can be performed by severely-constrained robots. The only exception is collective exploration/mapping. No evidence has been found where $C_I \leq 25.6$ robots were utilised; therefore, it is likely that collective exploration/mapping is difficult to implement or not feasible on severely-constrained robots. Similarly, when the complexity of a task increases (i.e., multi-task behaviours), these tasks are mostly performed by more powerful (i.e., weakly-constrained) robots suggesting that these tasks are difficult to implement or not feasible on severely-constrained robots.

Foraging, for example, is a class of tasks that can combine exploration/mapping/localisation, path-planning, task-allocation, and decision-making [Winfield 2009; Hoff III 2011; Sakthivelmurugan et al. 2018]. Note that there exists a version of foraging that uses C_1 robots [Mayet et al. 2010; Reina et al. 2017]. In these cases, the robots use the environment (pheromone tracks) to implicitly path-plan/navigate. This is a severe limitation as it requires an environment and the capabilities to allow the deployment/detection of pheromones. Research not limited in such ways consistently uses C_2 robots.

Similar to foraging, surveillance, as well as search and rescue, are used in swarm robotics [Schwager et al. 2011; Couceiro 2016]. However, both cases are exclusively performed by C_2 robots. In most cases, these tasks require SLAM, which by itself has only been deployed on C_2 robots ($C_I \geq 23.2$ [Steux and Hamzaoui 2010]).

[2.3.I] For more details see https://www.robocup.org/ (RobotCup).

[2.3.II] For more details see https://www.darpa.mil/program/darpa-robotics-challenge (DARPA Challenge).

Table 2.5: Common research tasks[a] in swarm robotics and their class of guaranteed deployability (CGD). The selection of tasks is based on [Bayındır 2016]. Larger selection of tasks and their detailed description can be found in [Brambilla et al. 2013; Bayındır 2016; Hamann 2018]. Note that the CGD is based on publications in their respected area.

Task	CR
(I) Spatially-Organising Behaviours	
Aggregation[b] [Gauci et al. 2014*b*]	C_1
Pattern Formation[c] [Rubenstein et al. 2014]	C_1
Object Clustering[d] [Gauci et al. 2014*a*]	C_1
(II) Navigation Behaviours	
Collective Exploration/Mapping[e][Ducatelle et al. 2011*a*]	C_2
Collective Movement[f] [Shirazi and Jin 2017]	C_1
Collective Transport[g] [Chen et al. 2015]	C_1
(III) Collective-Decision Making	
Consensus Achievement[h] [Trianni et al. 2016]	C_1
Task Allocation[i] [Li et al. 2017]	C_1
(IV) Miscellaneous Collective Behaviours	
Collective Fault Detection[j] [Tarapore et al. 2017]	C_1
Human-Swarm Interaction[k] [Kapellmann-Zafra 2017]	C_1
(V) Complex Multi-Task Behaviours	
Foraging[l] [Reina et al. 2017]	C_1/C_2
Search and Rescue[m] [Couceiro 2016]	C_2
Surveillance[n] [Schwager et al. 2011]	C_2

[a] A comprehensive list of swarm robotics tasks can be found in [Hamann 2018].
[b] In aggregation, a group of robots should move to one location.
[c] In pattern formation, a group of robots should position themselves in a defined pattern.
[d] In object clustering, a group of robots moves objects to one location.
[e] In collective exploration/mapping, a group of robots searches for an object/area or maps an environment.
[f] In collective movement, a group of robots follows a trajectory without changing their relative spatial formation.
[g] In collective transport, a group of robots transports a single or multiple objects to a target location.
[h] In consensus achievement, a group of robots chooses an option over alternatives.
[i] In task allocation, a group of robots decides which robot performs a specific task.
[j] In collective fault detection, a group of robots tries to detect wrongly behaving or faulty robots.
[k] In human-swarm interaction, one or more humans try to operate or cooperate with a group of robots.
[l] In foraging, a group of robots explores an environment, finds one or more sources, collects units from sources, and delivers them to a nest.
[m] In search and rescue, a group of robots explores an environment, finds one or more victims, and either reports their location or attempts to transport it to a designated area.
[n] In surveillance, a group of robots explores an environment, finds/follows a point or object of interest, collects and reports the data.

In summary, it has been demonstrated that severely-constrained robots can perform rudimentary tasks. However, due to the lack of severely-constrained robots performing multi-task behaviours unless in specialised/simplified environments, it is likely that the increased complexity presents a significant hurdle or might even be not feasible on these robots. As a result, computational constrains are a considerable hurdle.

2.4 Discussion

This chapter examined an often-overlooked aspect of robots: the computational resources of robots and requirements of software. It demonstrated how the proposed computational indices can quantify and classify individual resources or a system as a whole. The computational index, C_I, enables the systematic comparison of any computer systems as a whole based on

their resources. By calculating the indices of 46 state-of-the-art robots and 5227 computer systems, a classification was introduced grouping the entire spectrum of robotics into non-computational, severely-constrained, weakly-constrained, and minimally-constrained systems.

Based on the presented data, it was shown that the majority of miniature robots are severely-constrained and that the miniaturisation tends to decrease the amount of available computational resources. As a large proportion of severely-constrained robots only performs single-task behaviours, many of these systems are unlikely to be deployed in the real world. Consequently, more research efforts are required to enable such robots to perform more complex tasks in a more complex environment. Note that early research efforts have been undertaken to design low-computation control — for instance, computation-free[2.4.1] control [Gauci et al. 2014*b*]. However, in this research, the low-computation control is possible because the complexity of the detection and classification of objects was considerably reduced through a simplified environment and setup. Even though this is a first step towards low-computation approaches, many technical challenges are yet to be overcome to deploy such a system in a real-world environment.

Alternative solutions to overcoming computational limitations could include (I) outsourcing computation to external infrastructures (i.e., clouds), (II) reducing computation by designing additional hardware/physical mechanisms, or (III) distributing computation efforts across multiple robots.

(I) As described in Section 2.2.2.5, outsourcing is only suitable for a small number of robots and is not feasible for larger swarm robotics systems unless further research on communication systems for severely-constrained robots is conducted. In particular, scalability, throughput, and reliability are a concern in these systems.

(II) Alternatively, outsourcing computation to physical or electrical mechanisms (e.g., similar to the Braitenberg vehicle) can be efficient and in many cases miniaturised (e.g., [Richards 2016]). However, each design is specific to the environment and tasks; therefore, when the environment or task changes, it is likely that it would need to be redesigned. Overall, it lacks the flexibility of software.

(III) Finally, the distributing and sharing of resources between robots is a more flexible and general approach that can be replicated in a multitude of problems. However, it presents multiple challenges — management of local resources; reliable and scalable communication; and new design methods of behaviours.

In summary, this chapter investigated how simple are “relatively-simple” robots with regard to computational resources. It showed that many swarm robots are often severely-constrained and likely limited in terms of what system software or behavioural software they can use. As a result, there are two recommendations this chapter has to offer. Firstly, more research is required on system software with a focus on execution efficiency, as the author believes that severely-constrained robots should not be further constrained. Secondly, further research on how to overcome computational resources is required to enable systems as described in (III).

[2.4.1] Note that Gauci et al. [2014*b*] uses computation-free synonymously to “without arithmetic operations" as the controller is a fully-connected Mearly automata, hence, computing.

3

OpenSwarm: Processing on Severely-Constrained Robots

Contents

Before the computational resources of multiple robots can be combined, each robot must be capable of managing its local resources. As described in the previous section, this should be conducted with a high execution-efficiency due to the severe constraints on many *swarm robots* (i.e., swarm robotics platforms). The most execution-efficient way to implement software is to do so directly on the desired target platform. However, each robot, even a simple one, contains circuits with at least hundreds of components and designing software would require an in-depth understanding of them. Furthermore, the operation of *peripherals*[3.0.I] differs considerably across manufacturers and architectures, and each of the peripherals requires manual configuration. As a result, this makes the development of behavioural software complex and error-prone [Kemerer 1995].

To reduce complexity, system software can be used. It provides reusable often-required features and functions. However, as shown in the previous chapter, robotic system software often uses additional layers (e.g., VMs) that execute software. Each additional layer tends to increase the *computational overhead*[3.0.II]. In contrast, system software called *operating systems* can execute software directly on the hardware.

An operating system controls the execution of software and access to resources (e.g., execution time, memory, and access to I/O[3.0.III] devices) [Tanenbaum 2009; Stallings 2014]. The

[3.0.I] Peripherals are parts of a system that are connected to a computer via a bus or to input/output ports of its processing unit.

[3.0.II] The computational overhead of software is the additional code that is executed, increasing memory consumption or processing time.

[3.0.III] Input and output (I/O) devices are parts of an integrated circuit or peripheral circuit that allow microcontroller units to interact with other devices or the environment. Common I/O devices are sensors, actuators, and commu-

two fundamental functions of any operating are defined in this work as:

- **Hardware abstraction** provides functionality while hiding transparently the details of the implementation.
- **Resource management** controls access to the resources, prevents *data corruption*[3.0.IV], and ensures *fair*[3.0.V] use of resources. If access is restricted, it should prevent *deadlocks* or *livelocks*[3.0.VI].

This chapter first presents existing operating systems and discusses their benefits and limitations. Then, the design, implementation, and study of OpenSwarm, an operating system for severely-constrained robots, is presented and discussed. The implementation of OpenSwarm is tested regarding memory as well as processing overhead and is compared to other system software. Finally, the impact on a swarm robotic task when using OpenSwarm is investigated and compared to other robotic system software.

3.1 Existing Operating Systems

Depending on the field of application, an operating system's requirement can vary in function and operation. The most relevant systems for this work are (I) generic operating systems as they are feature-rich, (II) smart-card operating systems as they are the smallest, and (III) sensor network operating systems as they are similarly-constrained as swarm robots.

3.1.1 Generic Operating Systems

The most common operating systems are generic operating systems — also referred to as general-purpose operating systems. These systems are designed for personal computers with often tens of gigabytes of primary memory and terabytes of secondary memory. With these amounts of resources, generic operating systems are function- and features-rich.

Generic operating systems are commonly used on robots that execute robotic middleware such as ROS (for more details, see Section 2.2.2). Current versions of generic operating systems require computational resources of $C_I = 26.5$ (Linux), 28.6 (macOS), and 27.0 (Windows). As a result, they are not suitable for severely-constrained robots ($C_I < 23$).

It is worth noting that versions of Linux and Windows with reduced size exist, and are usually referred to as embedded versions. Windows Embedded IoT and Debian Linux, for instance, can be deployed on systems with a computational index of 25.9 (Windows) and 24.4 (Debian Linux) [Microsoft 2017; Mauerer 2017]. However, these systems are still unsuitable for severely-constrained robots ($C_I < 23$). Furthermore, these embedded adaptations of generic operating systems are often less predictable and less efficient and, therefore, less favourable in robotics than the alternatives presented below [Stallings 2014].

3.1.2 Smart-Card Operating Systems

On the other side of the spectrum of computational requirements, smart-card operating systems are operating systems for the most computationally-constrained devices — smart cards.

---

nication.

[3.0.IV] Data corruption is a situation where the value of data is undefined due to malfunction or race conditions (i.e., the outcome or correctness of data depends on the timing and sequence of software).

[3.0.V] Fairness ensures that a resource is used by multiple components or software equally often.

[3.0.VI] A livelock and deadlock are situations in which one or multiple programs cannot progress. In a deadlock, these programs are blocked and wait for a non-occurring state. In a livelock, they continue to execute but cannot proceed as the same sequences are executed repeatedly.

Smart cards are SoC[3.1.I] containing 8 to 32-bit MCUs with cryptographic coprocessors[3.1.II] mounted inside plastic credit-card-sized cards. The systems contain between 256 B and 16 kB of RAM as well as 4 kB and 400 kB ROM. It can be used to communicate wirelessly (i.e. RFID[3.1.III]/NFC[3.1.IV]) or directly via connections on the card [Rankl and Effing 2004]. When in contact with a reading device, the smart card is powered and initiates the execution of its function. Their main applications are security, encryption, and authentication.

Smart card operating systems, such as JavaCard and MULTOS, provides a set of fast cryptographic functions while being executed on severely-constrained hardware [Deville et al. 2003]. JavaCard [Chen 2000] provides a reduced JVM that enables the execution of small applications, called applets. While it can hold multiple applets, only one is executed at a time and, therefore, it does not provide *concurrency*[3.1.V]. MULTOS [Deville et al. 2003] is an open high-security *multiprogramming*[3.1.VI] operating system which enables dynamic loading, updating, and deleting of programs.

Overall, smart-card operating systems can be executed on severely-constrained robots due to their small design. However, they have several drawbacks. As the execution is only started when the card is powered, the system often executes a single function. As the smart-cards operating systems perform exclusively cryptographic algorithms and data access, they lack any hardware abstraction except its primary communication ports. In contrast, robots often need to execute multiple tasks (i.e., sensing, control, and communication) at the same time with frequent I/O interactions.

3.1.3 Sensor Network Operating System

Another area that deploys severely-constrained devices is sensor networks — in particular, wireless sensor networks. Sensor networks are widely used in automobile and aviation industry, where a base-station obtains measurements from a large number of sensor devices, called nodes. Nodes are commonly severely-constrained devices, as shown in Appendix C.2. Due to their computational constraints and their frequent use of sensors, these systems have many similarities to swarm robotics.

Sensor networks have a wide range of application (e.g., maintaining function of aircrafts or monitoring the movement of glaciers) [Akyildiz et al. 2002; Chong and Kumar 2003]. This results in a large variety of operating systems [Farooq and Kunz 2011], where the most common are TinyOS[3.1.VII], Contiki[3.1.VIII], Mantis OS[3.1.IX], Nano-RK[3.1.X], NuttX[3.1.XI], and LiteOS[3.1.XII].

In Table 3.1, these systems are compared based on memory consumption, used program-

[3.1.I] A System-on-a-chip (SoC) is a device that contains processing units, memory, and all peripheral circuit integrated in one chip.
[3.1.II] A cryptographic coprocessor is an integrated circuit that provides fast and low-power instructions to perform cryptographic procedures.
[3.1.III] Radio-Frequency IDentification (RFID) is a method that uses short-range radio communication to power and communicates with an integrated circuit to identify the device.
[3.1.IV] Near-Field Communication (NFC) is a high-frequency communication method. It is used for close-range communication on, for instance, phones and contactless debit cards.
[3.1.V] A system is concurrent when its components can be executed parallelly or sequentially in any order without affecting the outcome.
[3.1.VI] In contrast to uniprogramming, multiprogramming allows concurrent execution of multiple programs, which has been shown to increase the computational throughput [Stallings 2014].
[3.1.VII] For further details see http://www.tinyos.net (TinyOS)
[3.1.VIII] For further details see http://www.contiki-os.org/ (Contiki)
[3.1.IX] For further details see http://mantisos.org/ (Mantis OS)
[3.1.X] For further details see http://www.nanork.org/ (Nano-RK)
[3.1.XI] For further details see https://nuttx.org/ (NuttX)
[3.1.XII] For further details see http://lanterns.eecs.utk.edu/software/liteos/ (LiteOS)

Table 3.1: Sensor network operating systems. Values[a] are taken from [Gay et al. 2005; Dunkels et al. 2004; Bhatti et al. 2005; Eswaran et al. 2005; Cao et al. 2008; Nutt 2016].

Name	RAM	ROM	Language	Events	Threads
Contiki [Dunkels et al. 2004]	2 KB	32 kB	C	yes	one
LiteOS [Cao et al. 2008]	1 KB	6 kB	LiteC++	no	yes
Mantis OS [Bhatti et al. 2005]	500 B[b]	14 kB	C	no	yes
Nano-RK [Eswaran et al. 2005]	2 KB	16 kB	C	no	yes
NuttX [Nutt 2016]	2 KB	18 kB	C	no	yes
TinyOS [Levis et al. 2005]	500 B[b]	32 kB	nesC	yes	yes[c]

[a] Note that these values can vary depending on the hardware architecture (e.g., 8-bit architectures provide smaller footprints than 64-bit architecture).
[b] The memory requirements are based solely on the core element of the system. A deployed system would require additional functions increasing its memory requirements.
[c] Threads are executed in a thread-library and is not part of TinyOS' design. The consequences of this are discussed in Section 3.2.2.

ming language, and if the system provides event-based[3.1.XIII] or multi-threading[3.1.XIV] mechanisms. These operating systems can be divided into two groups. Contiki and TinyOS build one group where the system uses event-driven execution. LiteOS, Mantis OS, Nano-RK, and NuttX build the other group where software is executed as one or more threads similar to Linux. Note that a more detailed analysis of these systems (e.g., architecture, execution model, memory management, etc.) are conducted later in their respected sections.

Overall, each system is designed to fit into computationally severely-constrained devices. The operating systems are designed for two purposes — infrequent measuring and communicating without any actuation. Consequently, these systems maintain long idle/sleeping times between the measurements. As resources — such as execution time and memory — are, in most cases, sufficiently available, software is sequentially executed to completion. In contrast, multiple sensors and actuators are used frequently and continuously in robotics. Run-to-completion could increase reaction times as it delays other functions and might impact the robot's behaviour.

3.1.4 Discussion

Fundamentally, there are three related operating system types: generic, smart-card, and sensor-network operation systems. Generic operating systems provide many features but are unsuitable for severely-constrained robots. In contrast, smart-card operating systems are suitable; however, they lack ways to interact with their environment (i.e., through I/O devices) which is paramount in robotics. Sensor-network operating systems, on the other hand, allow the interaction with their environment while being deployed on devices constrained similarly to swarm robots. However, they are designed for long idle times and short sequences of measuring and communicating. The lack of actuation abstraction and device utilisation makes these systems difficult to apply to many robotic applications.

Based on the latter overview, it can be concluded that an operating system for severely-constrained robots would need similar memory requirements as smart-card or sensor network operating systems. The system should allow short response times (i.e., for timely execution) and enable the execution of multiple tasks. It should provide environments for algorithms with long (similar to generic operating systems) or short runtime. Furthermore, it should provide

[3.1.XIII] An event is a signal that a state of the system or a component has changed (e.g., a new camera frame was obtained). See Section 3.2.2.1.
[3.1.XIV] Multi-threading is a form of multiprogramming where smaller parts of a software (i.e., thread) can be executed concurrently. See Section 3.2.2.2.

transparent interfaces to sensors and actuators to simplify their use. An in-depth discussion of the properties above and their design is presented in the following section.

3.2 OpenSwarm

This section proposes an embedded operating system for severely-constrained robots, called OpenSwarm. The system has been designed to combine the aspects of robotic system-software and operating systems. Its key design properties are

- **small memory footprint** to allow OpenSwarm to be deployed on severely-constrained robots,
- **high execution efficiency** to weaken the limitation of computational constraints and to allow more complex behaviours,
- **concurrency** to allow the execution of multiple pieces of software and to further facilitate complex behaviours,
- **hardware abstraction** to allow a high-level and transparent use of the robots' capabilities,
- **modular design capabilities** to allow good adaptability and maintainability, and
- **clear interfaces between modules** to further improve adaptability and maintainability.

In the following section, the kernel of OpenSwarm is designed to provide the given properties.

3.2.1 Architecture

The architecture of an operating system defines the structure of the kernel[3.2.I], its memory requirements, and how software is executed. There are different architecture types:

- *monolithic* kernels combine all core functions into a single program [Stallings 2014]. This architecture is compact and provides high performance[3.2.II]. As a result, it is often used on embedded operating systems — for instance, JavaCard, MULTOS, Contiki, Nano-RK, TinyOS. However, this architecture is commonly difficult to maintain and extend, and single faults in any subroutine could cause a crash of the entire system.
- *layered* kernels are structured into layers, where a lower layer provides functions (e.g., process[3.2.III] allocation and switching) to a higher layer. This aims to improve maintainability. It is used in systems such as Mantis OS. However, many functions can be highly entangled, which prevents distinct layers. In addition, the additional interfaces between the layers render the system slower and larger than a monolithic system.
- *modular* kernels are similar to the layered architectures. However, instead of layers, the functionality is encapsulated into modules and loaded during runtime. This modularising creates a computational and memory overhead but also improves maintainability and extendability. This architecture is used on LiteOS, Nuttx, and Linux.

OpenSwarm uses a modularised monolithic kernel, as illustrated in Figure 3.1, to provide high performance and small memory footprint [Stallings 2014]. To counteract the drawback of a monolithic design (i.e., reduced maintainability and extendability), each functionality of the system is a self-contained module. The kernel is modularised in four elements:

- the *event module* controls the information flow within the system,
- the *process module* creates, organises, schedules, and terminates executed programs (i.e., processes),
- the *memory module* dynamically allocates, organises, and deallocates memory, and

[3.2.I] A kernel is the central component of an operating system providing its core functionality.
[3.2.II] If not stated otherwise, the performance of a software is a measure of how efficiently it is executed. In other words, if the software consumes less memory or is processed within shorter time, the performance is higher.
[3.2.III] A process is an executed instance of a program (i.e., a set of instruction on data that can be executed).

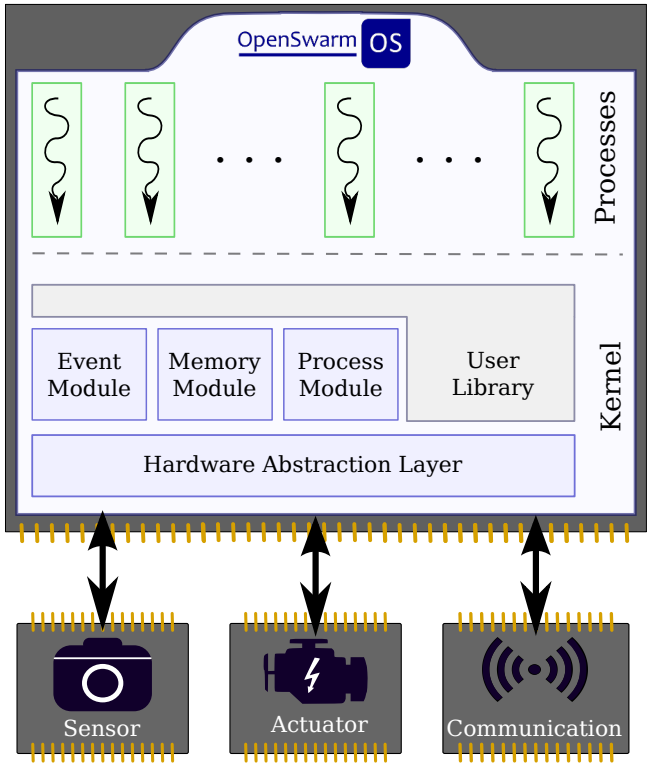


Figure 3.1: OpenSwarm's architecture. Blue boxes are the three core modules of the kernel. Green boxes symbolise threads and dark grey boxes indicate hardware.

- the *hardware abstraction layer (HAL)* interfaces the hardware with the rest of the system.

Overall, user processes (green boxes) are executed on top of the kernel, which also interfaces the hardware (dark grey boxes) with the system. Note that the event and process module both execute code and build together a processing model that is presented in the next section.

3.2.2 Execution Model

The execution model defines how behaviour can be implemented and executed. A robot might execute short behaviours (e.g., obstacle avoidance) or long computational-intensive algorithms (e.g., path planning). Ideally, a robotic operating system should offer both fast response times and efficient[3.2.IV] execution.

In robotic system software (e.g., ROS, Miro, and ASEBA), event-based development is common as well as in systems such as Contiki and TinyOS. Overall, the execution with events has a small computational overhead (i.e., fast response times) to execute short sequences of code. However, when longer sequences of code are executed or multiple events occur, mechanisms need to be found to either interrupt and resume or to queue their execution, which delays response times.

As an alternative to events, process-based execution offers long periods of efficiency. Generic operating systems (e.g., Linux and Windows) and embedded operating system based on Unix or Linux architectures (e.g., LiteOS and NuttX) use this as the primary execution model. As loading and deleting processes takes considerate time and memory, it is inefficient for short processes (i.e., many responsive behaviours).

Both methods — events and processes — can be useful in scenarios such as avoiding

[3.2.IV] Execution efficiency is a measure that compares the execution time and memory consumption. In short, higher efficiency results in lower computational requirements while performing the same action. In other words, it is better performing.

obstacles while simultaneously mapping an environment. Consequently, OpenSwarm provides a dual-execution model where both events and processes are used to execute code. For each type of execution, a corresponding module exists.

3.2.2.1 Event Module

The event module controls the occurrence and processing of *events*, where an event indicates a significant state change within the system or its components. An event can be triggered by hardware (e.g., a new camera frame was obtained) or by software (e.g., a camera frame was processed).

Existing Event Mechanisms

When considering system software for severely-constrained devices (i.e., Contiki and TinyOS), events are created exclusively by hardware interrupts. The respected interrupt routines trigger processing of an assigned function preempting the current execution. After completion, it continues with the previous execution. In contrast to robotics, events are assumed to be infrequent and, therefore, event execution is sequential and cannot be nested.

On systems similar to ROS and Miro, events are created and managed by software. It regulates the access and can prevent race conditions[3.2.V]. In many cases, it is designed as a publish/subscribe mechanism which enables, for example in ROS, execution of software locally and distributively (i.e., across multiple devices). In contrast to hardware interrupts, software events can trigger the execution of multiple pieces of software — even in parallel. However, software events tend to increase the computational overhead.

OpenSwarm's Event Mechanisms

OpenSwarm is designed to fit into severely-constrained robots, and low computational overhead is paramount. However, to enable the combining of computational resources across multiple devices, software (i.e., computational overhead) must be introduced to enable this distribution. As a result, OpenSwarm provides software events that can be distributed. Furthermore, OpenSwarm can process events asynchronously and synchronously. This allows high-priority events to be executed swiftly and efficiently, while lower-priority events are processed in idle time. OpenSwarm enables the optimisation of the robot's behaviour and prevents potential delays introduced by sequential execution, as can occur on Contiki.

Asynchronous events are buffered and sequentially processed in an isolated context to prevent race conditions. However, the buffering and the delayed processing creates a computational overhead. Synchronous events, on the other hand, are processed immediately within the context of the emitting function, which should be preferred if fast response times are required. However, this increases the execution time of the emitting functions, and, therefore, asynchronous events should be preferred.

Discussion

In robotics, it is often crucial to react within a defined response time. Events offer a low computational overhead, and relatively fast response times as the data is processed at its occurrence.

In OpenSwarm, events are used to signal the occurrence of a hardware or software condition (e.g., "a new camera fame is ready") and to transfer information (e.g., "this is the new frame"). Events are conveyed based on a publisher/subscriber architecture, which decouples sender (i.e., publisher) and receivers (i.e., subscriber) of events (i.e., messages). This facilitates modular design, which increases, in principle, maintainability, readability, portability and reusability of the code [Helmer et al. 2011; Mühl 2006]. Figure 3.2 shows how events can transfer information in OpenSwarm.

[3.2.V] Race conditions describe situations in which the outcome of the execution depends on the execution order of its components or other software.

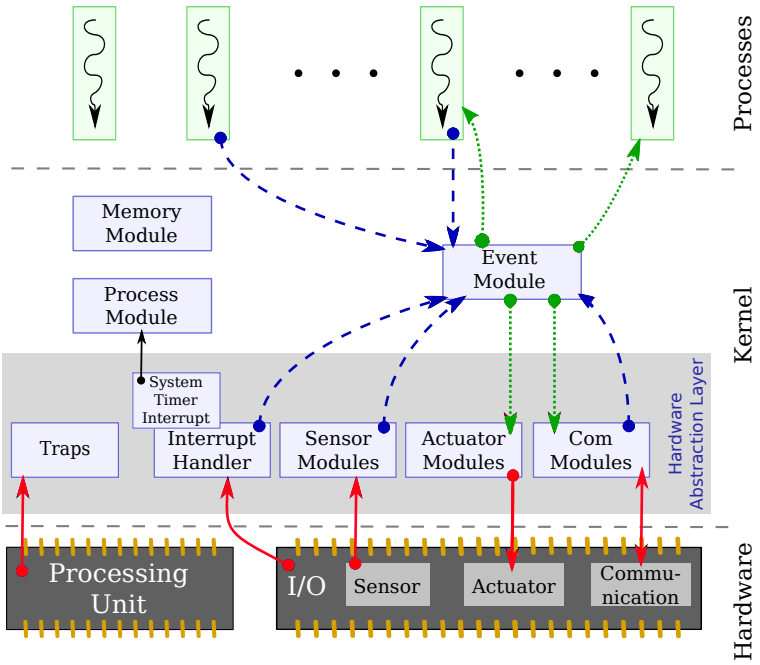


Figure 3.2: Information/event flow in OpenSwarm. When a module (blue rectangle) or a process (green rectangle) intends to emit an event, it executes a system call indicated by the blue dashed arrow. The event is then emitted by the system and executed as indicated by the green dotted line. Note that the red arrows are interactions between OpenSwarm and hardware.

Event-driven development is used in sensor network operating systems (Contiki and TinyOS) and robotic system software (ASEBA, and Urbi). OpenSwarm differs from these systems in the following ways:

- Contiki and TinyOS generate an event at the occurrence of an interrupt (both) or software function call (TinyOS only). An event is buffered (Contiki only) and executed by a single function that was assigned to it (both). The function cannot be paused and runs to completion. In contrast, OpenSwarm generates events at the occurrence of both interrupt and software function call. Each event can be executed synchronously (i.e., immediate execution as in TinyOS) and asynchronously (i.e., buffered and postponed execution as in Contiki). In OpenSwarm, the execution of events can be optimised on the basis of the importance of the event — e.g., time-critical events can be executed immediately and non-time-critical with short delays. Furthermore, events can be interrupted by higher-priority as some actions in robotics can take precedence over others (e.g., for safety). The two most significant differences to these operating systems are that OpenSwarm can subscribe multiple receivers to one event, and that each receiver could be executed on different[3.2.VI] devices/robots.
- ASEBA and Urbi use a script language that uses event-based syntax. Both systems process events iteratively. First, measurements and communication data are obtained. Then, the script is executed, and finally, new actuation and communication data is applied. In other words, data and events are processed once for every iteration. Long scripts can cause undefined behaviour as long scripts could be truncated or cause slower sampling rates (i.e., iteration rates). This directly impacts the performance of the system. In contrast, OpenSwarm events are generated and executed at any time. This allows faster response times and an interaction behaviour independent of the currently executed code.

[3.2.VI] The details and consequences of the distribution of events are discussed in Chapter 5.

3.2.2.2 Process Module

In this work, a *process* is an instance of a *program*, where a program is a collection of instructions and data that can be executed by a computational device. This instance contains a copy of the program, runtime memory, a call stack, and operating-system-specific data. A process can contain one or more *threads*, where a thread is a concurrent part with a separate call stack. All threads within a process share the same memory and, therefore, switching between threads generally causes less overhead than switching between processes. On the other hand, each process is contained within its own memory region that is not accessible to other processes.

Process Management

Process-based execution is common on any generic operating system (i.e., Linux, macOS, and Windows). Usually, programs are expected to run for long periods (e.g., minutes, hours, or continuously), and, therefore, initial computational overhead and changes between processes are negligible in comparison to the runtime. Furthermore, as these operating systems were designed for personal computers and commercial servers, they interact infrequently with I/O except communication devices. As a result, processes are paused when waiting for I/O device responses. When data from an I/O device is ready, the system performs the following steps. (I) The data is buffered, (II) the corresponding process is unblocked, and after the process is scheduled, (III) the data is processed. This introduces a significant time delay. As many weakly-constrained devices provide more computational resources than required, the latter delays are often small enough not to impact the behaviour. However, this is not the case for more constrained (in particular, severely-constrained) devices.

LiteOS, Mantis OS, Nano-RK, and NuttX are designed to be similar to Unix. In contrast to Unix and due to the lack of an MMU[3.2.VII], a process is not contained in an isolated memory region and, therefore, share the same memory space. As a result, a process has the properties of a thread as defined in [Stallings 2014].

Contiki and TinyOS are primarily event-driven systems but, in later versions, adopted thread-like processes to be executed in idle times. Contiki, however, executes cooperatively-scheduled non-concurrent functions, which can only be preempted by interrupts, in idle times. Other than promoted, Contiki does not provide processes, nor are they preemptive scheduled as defined in operating system research [Stallings 2014]. TinyOS, on the other hand, provides a single process that manages any additional process. At any event occurrence, the system changes to the control process, which then changes to the next process. As a result, fair scheduling is not guaranteed as processes could be starved of processing time. Furthermore, any process has a lower priority than that of events; therefore, this also can starve processes.

In OpenSwarm, the process module controls the creation, management, and deletion of thread-like processes. The process implementation follows similar concepts as used LiteOS, Mantis OS, Nano-RK, and NuttX. A process is defined by a process control block containing all process-related data, including a call stack. In contrast to Contiki, processes are concurrent and preempted by a periodic scheduler. In contrast to TinyOS, each process has assigned time slots that allow fair use of processing time. The differences to LiteOS, Mantis OS, Nano-RK, and NuttX are discussed next.

Process Execution

Depending on the design of the operating system, a process can be executed and can interact with the operating system in one of three ways, as shown in Figure 3.3.

- Split execution as used in Unix, LiteOS, and older generic operating systems — such as DOS — separates the execution of processes from operating system (see Figure 3.3a).

[3.2.VII] A memory management unit (MMU) is a part of an integrated circuit that allows concepts such as virtual memory. Virtual memory is an abstraction of physical memory. For more details see Section 3.2.3

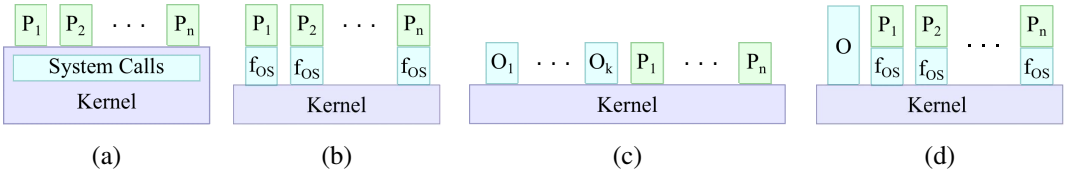


Figure 3.3: Process execution. (a) The split execution executes system calls in the context of the operating system. (b) The functional execution model executes system calls (f_{OS}) in the context of the user process. (c) The process-based execution model executes system call in the context of dedicated operating system processes (O_i). (d) The OpenSwarm's dual-execution model combines (b) and (c).

Whenever a process wants to execute a *system call*[3.2.VIII], first, the process is paused, and the control is handed over to the operating system which then executes the requested system call. Afterwards, the process continues. This protects the integrity of the kernel. Due to the invoked *context switches*[3.2.IX], this method causes significant processing overheads and delays, which is a significant drawback on severely-constrained devices.

- Functional execution describes the execution of operating system functions within the context of a process (see Figure 3.3b). The operating system takes control of the system in situations of faults, or to switch to other processes. This model reduces the number of context switches and, therefore, delays. However, the design and structure of the systems can be chaotic as the circumstances under which system calls are executed is uncontrolled and non-deterministic. Due to the minimal requirements of execution time and memory, functional execution is used in Nano-RK.
- Process-based execution describes a system where isolated functions are implemented in their processes on the same execution level as user processes (see Figure 3.3c). The kernel only provides minimal functions, for instance, detection and processing faults. Any other functions are performed by processes. As a result, each non-critical component is only scheduled if needed, and the operating system can be deployed on multi-core processors without design changes. However, the usage of operating system functions requires immediate or delayed context switches or process synchronisation methods. Similar to split execution, this causes significant delays. Despite these disadvantages, Mantis OS, NuttX, and Windows execute operating system functions with processes.

In comparison, OpenSwarm provides dual-execution model, as shown in Figure 3.3d. On the one hand, processes use functional execution to avoid computational overhead, and on the other hand, low-priority and non-time-critical functions are executed as an individual process. As performance is a priority, the majority of functions are directly executed within the context of the calling process similar Nano-RK. However, non-time-critical functions — such as *garbage collection*[3.2.X] — are executed as a process similar to Mantis OS. This execution method provides a trade-off to: (I) avoid delays for frequently used function (e.g., motor control), and (II) provide modular and interchangeable processes for non-critical functions.

Process States & Scheduling

Independently of how processes are executed, a process has a *life-cycle*, which defines the states of a process from creation to deletion. OpenSwarm uses a 5-state model, as shown in Figure 3.4. This model prevents processes from being rescheduled and, consequently, prevents unnecessary delays. This model is common for many operating systems, including LiteOS, Nano-RK, and NuttX. In comparison, most generic operating systems use a 7-state model,

[3.2.VIII] A system call is a function that requests a core function of the operating system (e.g., requesting file access). Unlike user libraries, system calls are integrated into the operating system.
[3.2.IX] A context switch is a procedure of unloading the current process and the loading of another.
[3.2.X] Garbage Collection is a procedure in which no-longer-needed memory is freed.

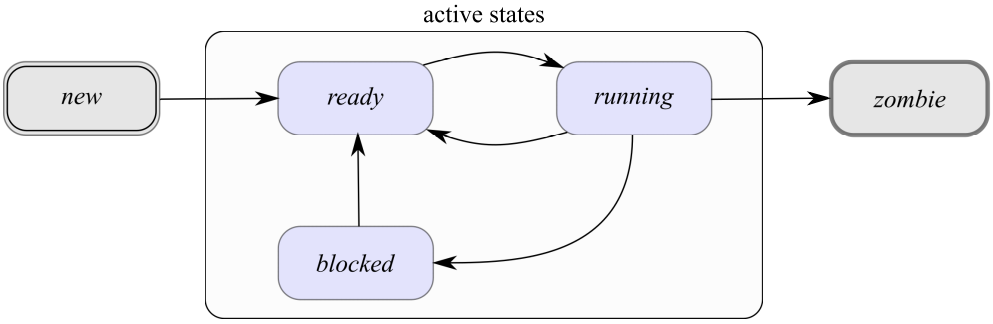


Figure 3.4: Process state model in OpenSwarm. When a program is created, it is in the state *new* (initial state). A process that is ready to be executed is *ready*. The currently executed process is *running*. Processes that wait for events to occur are *blocked*. Once that event has occurred, the process returns to be *ready*. When a process finished execution or crashed, it is terminated which is indicated by the state *zombie* (final state).

where the *ready* and *blocked* states are duplicated. Each duplicated state is a suspended state (i.e., the process is copied to secondary memory to free the primary memory making it possible to execute more processes). On severely-constrained robots, secondary memory is often not available in large volumes, which makes suspending processes difficult or not feasible.

The method that decides which process is executed next (i.e. changes from *ready* to *running*) is called the scheduler. Common operating systems — in this case, Mantis OS, LiteOS, Nuttx, Linux, Windows and macOS — use a priority-based round-robin scheduler[3.2.XI]. The only exception is Nano-RK, which uses rate-harmonised scheduling[3.2.XII] for power-consumption-aware *real-time*[3.2.XIII] scheduling.

In OpenSwarm, the scheduler is defined as a callback function, and each process control block holds generic scheduler-specific data allowing the implementation of any scheduler. Per default, an (equal-priority) round-robin algorithm is implemented to ensure starvation-free and fair scheduling. OpenSwarm also allows the removal of the scheduler stopping the execution of processes. This would cause OpenSwarm to behave similarly to TinyOS or Contiki.

3.2.2.3 Discussion

In OpenSwarm, behaviours can be implemented in two ways — via thread-based or events-based programming. Fundamentally, both paradigms can express the same behaviours; however, each provides benefits and disadvantages that particularly impact robots with severely-constrained resources.

As process-based execution is common on a wide range of operating systems — e.g., Linux, Windows and macOS —, it is well-understood among programmers. It is not surprising that Nano-RK, Nuttx, LiteOS, and Mantis OS aim to provide these established (i.e., Linux-like or POSIX) environments, and, therefore, replicate this execution paradigm. Generally, processes are designed for long periods of uninterrupted execution, and, when executed, have a small computational overhead. However, the creation and switching between processes takes relatively long, which can create significant overhead for short repeatedly-executed software.

[3.2.XI] Round-robin is an algorithm that assigns equal shares of a resource, here processing time, to every process [Stallings 2014]. This algorithm is simple and starvation-free. A priority-based round-robin algorithm assigns equal shares to all processes with the same priority. However, it schedules a higher-priority process more often, which can lead to starvation.

[3.2.XII] Rate-harmonised scheduling is a real-time algorithm that includes energy management. For more details see [Rowe et al. 2010].

[3.2.XIII] Real-time systems guarantee to respond to input within (hard real time) or mostly within (soft real time) a defined timeframe.

To avoid repeated overhead, event-based execution provides faster reaction times with lower memory usage. As short sequences of execution are common in sensor networks, the two of the most popular sensor network operating systems — TinyOS and Contiki — are primarily event-based. Multiple events run sequentially to completion. This makes these systems often not suitable to execute long-lasting algorithms as they would starve other parts of the system of processing time.

In robotics, robots are expected to react swiftly (e.g., for obstacle avoidance) and be able to process long algorithms (e.g., for path planning). As a result, one of OpenSwarm's novelties is the dual-execution model and its hybrid kernel that provides both event- and process-based execution. As described in Section 3.2.2.1 and 3.2.2.2, OpenSwarm differs from the presented systems in that it allows both types of execution. Furthermore, it is distinct from other systems in how events, as well as processes, are executed. Overall, OpenSwarm is suitable for short-code execution for fast response times (events) and long-lasting execution without monopolising the processor unit (processes). Furthermore, the hybrid kernel also enables distributed storage or processing, as explored in Chapter 5.

3.2.3 Memory Module

As the processing of code requires memory, organising this memory is part of the memory management. Similar to the sensor network operating systems, OpenSwarm was designed for MMU-less devices. As a result, the memory is flat, and any function has access to the entire memory.

3.2.3.1 Virtual Memory

Virtual memory abstracts physical memory, which enables contiguous access to fragmented memory. As the virtual addresses are translated by the MMU to physical addresses, it is possible to reposition memory segments during run time and to detect access violations. In particular, this can reduce *fragmentation*[3.2.XIV] and a more efficient use of the available memory.

Since many devices lack MMUs, software implementations of virtual memory have been attempted [Choudhuri and Givargis 2005; Bai et al. 2009]. In these works, a presented tool replaces every memory-access instruction with a function call that, first, translates the address and, then, loads/stores the value. Because memory access is one of the most frequent operations, this creates a significant overhead that is only acceptable if the execution is not time-critical. Due to the high-costs of a software virtual memory, it is not used in OpenSwarm. Even though virtual memory is a standard feature in any generic operating system since the 1990s, none of the presented operating systems for severely-constrained devices uses software virtual memory.

3.2.3.2 Memory Allocation

Memory can be allocated statically during compilation or dynamically during run time. Dynamic memory allocation provides flexibility to allocate memory when needed. This can be useful when the amount of needed memory is not known in advance. However, it can cause unpredictable delays (i.e., allocation time) and behaviour (i.e., unsuccessful allocation). As a consequence, hard real-time systems avoid this for better predictability as does Nano-RK. However, many sensor nodes have a low computational load; therefore, dynamic memory often can be deallocated before the next execution cycle. Consequently, dynamic memory allocation has a small impact on the node's behaviour and is used on Contiki, Nuttx, LiteOS, Mantis OS, and TinyOS.

[3.2.XIV] Memory fragmentation reduces the capacity or performance of the system due to unusable gaps between used memory. While a small degree of fragmentation is common, extreme cases cause reduced performance. However, the exact effects of fragmentation depend on the design of the system.

While some areas of robotics are safety-critical and require hard-real-time properties, swarm robots, in their current form, are mostly deployed in academic and non-safety-critical environments. As a result of this and due to the resource constraints, one can assume that the flexibility of dynamic allocation and its ability to use all available memory takes priority. That being the case, OpenSwarm provides dynamic memory allocation.

3.2.3.3 Discussion

OpenSwarm's memory module provides a dynamic memory allocation enabling the system to make use of all its primary memory. One of the limitations of OpenSwarm is that it provides a flat memory space allowing processes to access any data within the memory. While this does not create nor cause errors, it enables processes to introduce malfunction in other processes. For instance, a buffer overflow in one process could override data of another. As Contiki, Nuttx, LiteOS, Mantis OS, Nano-RK, and TinyOS also provide flat memory, OpenSwarm is no way inferior to them regarding memory management.

3.2.4 Hardware Abstraction Layer

Since the purpose of a robot is to interact with its environment, the ability to sense, actuate, and communicate is a crucial part of any robotic system. In OpenSwarm, the hardware abstraction layer (HAL) manages Input/Output (I/O) device and controls their access.

The large variety of I/O devices can be divided into three categories: input, output, and communication devices. Input devices are circuits measuring physical entities (e.g., voltages, temperature, or light intensity) digitally or analog. Output devices are circuits controlling actions (i.e., actuators such as motors) and indicators (e.g., displays or LEDs). Communication devices are circuits that transmit data to and receives it from other communication devices.

3.2.4.1 I/O Access

Due to the wide variety of I/O devices, the access to their functionality varies considerably. Commonly, on-chip I/O devices are accessed by dedicated registers and memory. While output devices can be controlled by setting values in registers at any point, input and communication devices need to notify the system of new data. In some cases, the software is required to check the respected registers (i.e., polling) and, in other cases, interrupts occur triggering the execution. Polling causes a periodical processing overhead and can cause delays while interrupts preempt current execution and can cause race conditions.

3.2.4.2 I/O Design

As shown in Figure 3.5, the Hardware Abstraction Layer (HAL) enables interaction with I/O devices, where each device is managed by a single I/O module. To simplify the hardware access, I/O modules provide a two-layered design. The lower layer, also called device-specific handler, provides transparency to hardware functions simplifying their use. The top layer of the module, called processor[3.2.XV], provides value abstraction by transforming hardware-specific to hardware-independent values and vice versa. For instance, a robot's speed could be $64\,\mathrm{mm\,s^{-1}}$, while the underlying hardware-dependent value would be 547. Based on the abstraction of access and value, this layer is the interface between the hardware-independent part (i.e. processes and events) and the robot's hardware.

[3.2.XV] Depending on the direction of the data flow, the processor is called pre-processor to convert from hardware-specific to hardware-independent data and post-processor to convert vice versa. Rx- and Tx-processor manage and buffer received or outgoing communication data, respectively.

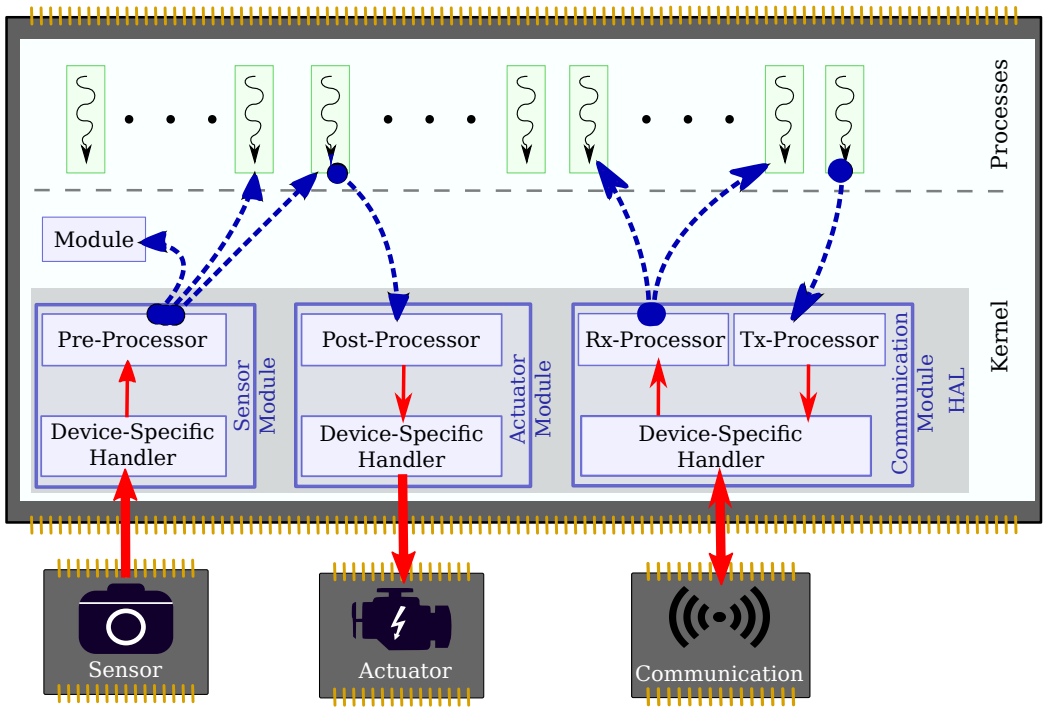


Figure 3.5: OpenSwarm's hardware abstraction layer (HAL). OpenSwarm processes data to/-from input, output, and communication devices with modules. Dashed blue arrows represent events. Hardware-specific data flow is indicated by thin red arrows. Data that is exchanged to/from hardware is indicated by thick red arrows.

3.2.4.3 Discussion

The delays when interacting with I/O devices can create a bottleneck to the system. As a result, inefficient use of these devices degrades the performance of the entire system. Robots are expected to interact with I/O frequently; therefore, the design of the I/O management is paramount in robotics.

OpenSwarm's design of I/O device interactions allows the use of hardware-independent data and hardware transparently. This is achieved by the module's two-layered design. A benefit of this design is that safety measures could be implemented as a top layer (i.e., processor) thereby preventing faults by limiting the actuation. Furthermore, virtual sensors/actuators can be implemented, functions of which are based on or derived from other available sensors/actuators. This allows the execution of a behaviour implementation that has been designed for different sensors and actuators. As shown in Chapter 5, this feature can be used to provide a virtual or augmented environment for a robot.

In contrast to OpenSwarm, Contiki and TinyOS mostly focus on measurements and communication. While they commonly provide transparent use of communication methods, there is no value abstraction. In addition, only one function can be linked to an input device, and multiple functions are only executed sequentially. This can result in significant delays. OpenSwarm, on the other hand, allows the preemption of code and I/O requests by I/O interrupts of higher priority. This provides faster response times for more important I/O devices.

Nano-RK, Nuttx, LiteOS, and Mantis OS are Linux-like (i.e., POSIX) operating systems, and so, they use character[3.2.XVI] device drivers to access I/O devices. While this creates a consistent *everything-is-a-file* resource representation, it also creates significant delays and computational overhead. First, an incoming signal is processed within the operating system context.

[3.2.XVI] Character device drivers allow the system to interact with a device by reading or writing characters into it. In other words, it behaves like a file.

Table 3.2: Embedded operating systems in comparison to OpenSwarm. The memory footprint[a] values for most systems are taken from the literature [Gay et al. 2005; Dunkels et al. 2004; Bhatti et al. 2005; Eswaran et al. 2005; Cao et al. 2008; Nutt 2016].

Operating System	Architecture	Execution		Process	Dynamic	Memory Usage	
		Events	Processes	Synch.	Memory	RAM	ROM
Contiki [Dunkels et al. 2004]	modular	yes	no[b]	-	yes	2 kB	60 kB
LiteOS [Cao et al. 2008]	modular	no	yes	mutex	yes	< 4 kB	< 128 kB
Mantis OS [Bhatti et al. 2005]	layered	no	yes	semaphores	yes	500 B[c]	14 kB
Nano-RK [Eswaran et al. 2005]	monolithic	no	yes	mutex & semaphores	no	2 kB	18 kB
NuttX [Nutt 2016]	modular	no	yes	POSIX[d]	yes	2 kB	32 kB[e]
TinyOS [Levis et al. 2005]	monolithic	yes	no[b]	-	no	512 B[c]	15 kB
OpenSwarm	**monolithic**	**yes**	**yes**	**event-based & semaphores**	**yes**	**2 kB**	**10 kB**

[a] The listed memory footprints are taken from literature and vary due to different hardware architectures (e.g. 8-bit MCU require smaller footprints than 64-bit architecture). Note that the memory footprint of OpenSwarm is based on the implementation presented in the next section.

[b] Only hardware interrupts can preempt the current execution. However, this is not a preemptive execution as defined in [Tanenbaum 2009; Stallings 2014].

[c] The reported memory usage contains only the core element of the system, and a deployed system is expected to have a significantly higher value. For instance, OpenSwarm could be reduced to 336 B of RAM; however, the fully deployed system consumes 1.9 kB.

[d] The POSIX standard provides interprocess communication methods, such as semaphores, shared memory, or message queues.

[e] NuttX requires 32 kB in its smallest configuration [Nutt 2016]. The memory footprint of a deployed system is not documented.

Then, the system unblocks the corresponding process. After the process has been scheduled, it processes the new data. Furthermore, these device drivers provide functionality abstraction, but no value abstraction.

3.2.5 Implementation

OpenSwarm's open-source implementation is freely-available at [Trenkwalder 2020*b*] under an adapted FreeBSD licence[3.2.XVII]. Documentation, tutorials and further details can be found at www.openswarm.org.

Since C compilers exist for most platforms, OpenSwarm is implemented in C. In version v0.17.09.25, the kernel contains 489 of 2783 = 17.6 % of hardware-specific lines of code without considering I/O modules. When deploying to other platforms, only 17.6 % require re-implementation. In other words, the remaining code can be ported unchanged. A list of system calls and implementation details, as well as examples of usage, can be found in Appendix B. Overall, the presented implementation, which supports the e-puck (see Appendix A), consumes 2 kB of RAM and 10 kB ROM.

3.2.6 Discussion

OpenSwarm is an embedded operating system designed for severely-constrained robots. As presented at the beginning of this chapter, OpenSwarm is designed with the following key properties:

- OpenSwarm uses a monolithic architecture. This results in a relatively **small memory footprint** which is lower or equal to other operating systems, as shown in Table 3.2.

[3.2.XVII] The full licence is available at https://openswarm.org/license/.

- Not to reduce the available computational resources further, OpenSwarm ensures **high execution efficiency** by executing software directly on the hardware without additional software layers (e.g., VMs or interpreters). It only takes control to manage I/O resources and to ensure fair use of any resources. The hybrid kernel enables a **concurrent** execution with events and processes. It can be used for fast-responsive and long-lasting execution, respectively. OpenSwarm's computational overhead is measured and compared in Section 3.3.
- OpenSwarm provides a two-layered **hardware abstraction** allowing a transparent use of hardware. Furthermore, OpenSwarm provides value abstraction, which provides further context for applications and reduces, in principle, platform-dependencies.
- OpenSwarm facilitates **modularisation** and events that are used as an **interface between modules and processes**.

3.2.6.1 Novelties of OpenSwarm

The novelties of OpenSwarm are:

- The dual-execution model that offers both process-based and event-based execution. While event- and process-based execution have been used on other robotic system software and operating systems individually, the combination of both features means that the code can be adapted to the needs of the user. In robotics, this is particularly beneficial in situations where robots must maintain operations (e.g., obstacle avoidance) while performing other tasks (e.g., path-planning). Furthermore, a related key aspect of OpenSwarm is its capability to distribute events through a network thus enabling distributed processing and storage, which is explored in Chapter 5.
- The structure of the hardware abstraction layer provides both functionality abstraction to allow the use of the devices transparently and value abstraction to reduce the platform dependency. Value abstraction provides context for measured entities, which separates their value from the underlying hardware. By combining both, OpenSwarm facilitates a transparent hardware-independent use of the robot's capabilities on a high level. Furthermore, this abstraction enables the implementation of virtual sensors/actuators. They can be used to simulate other sensors/actuators or to deploy the robot in a virtual or augmented environment, which is also explored in Chapter 5.

3.2.6.2 OpenSwarm Limitations

Due to the design choices of OpenSwarm, it faces certain limitations.

As it is designed to be a C API[3.2.XVIII], behaviour implementations can require more lines of code than other system software. In particular, domain-specific languages, such as ASEBA or Buzz, tend to use robotics-specific functions, reducing the number of lines.

Furthermore, due to the flat memory space, programs can access and manipulate data of other processes. This increases the likelihood of errors and could impact security. Having said that, measures to restrict such access (e.g., virtual memory) would significantly increase the computational overhead. Due to this overhead, providing flat memory is common practice for system software with severely-constrained devices.

3.3 Evaluation

This section investigates how OpenSwarm compares to other system software that has been deployed on severely-constrained robots. First, the memory and, then, the processing overhead of OpenSwarm and each of its modules is investigated. Thereafter, OpenSwarm is compared to

[3.2.XVIII] An API is an application programming interface, which enables the use of functions while developing software.

Table 3.3: Static and dynamic memory consumption of OpenSwarm and its components in bytes. The dynamic memory consumption depends on the number of running processes (p), registered events (e_r), buffered events (e_b), subscribers to the event (e_s), devices that require polling (d), subscribers to an ADC channel (a), and buffered messages (m). Furthermore, $E_{\uparrow}(s)$ symbolises the transmission of an event that contains data of the size, s. As a result, $E_{\uparrow}(s)$ increases the dynamic memory by s and increases e_b by one.

Component	ROM	RAM	
		static	dynamic
Process Module	1128	14	$278\,p$
Memory Module	363	22	0
Event Module	662	14	$e_r\,(6\,e_b + 10\,e_s)$
HAL	336	8	$4\,d$
Motors	520	18	0
Camera	4867	1368	$E_{\uparrow}(2)$
Remote Control	195	10	$E_{\uparrow}(2)$
Bluetooth	570	12	-
Selector	124	4	$E_{\uparrow}(2)$
ADC	219	42	$2\,a \leq 64$
Proximity	442	108	$p\,E_{\uparrow}(2)$
SwarmCom	387	67	$10\,m$

ASEBA regarding execution efficiency. Finally, the performance of different implementations is compared with that of OpenSwarm.

3.3.1 Memory Overhead

The presented implementation, which is deployed on the e-puck, consumes in total 2 kB of RAM and 10 kB ROM. However, when not every module is used, memory consumption can be further decreased. Table 3.3 shows the static and dynamic memory requirement for each module. Note that this data was obtained using the Microchip's MPLAB X compiler and emulator for the e-puck's microcontroller.

To measure the memory consumption, an empty project is emulated first and its memory consumptions — ROM and RAM — recorded. The memory consumption is then measured for every added module. Note that modules might not be independent of each other. As a result, the modules without dependencies are added first, followed by the ones with dependencies.

As evident from Table 3.3, OpenSwarm can be configured to consume as little as 336 B of RAM and 2489 B of ROM. With all modules, OpenSwarm consumes 1981 B of RAM and 9813 B of ROM which represents 24.6 % and 6.8 % of the respected memory. Note that the camera module of the e-puck robot requires most of the memory due to the allocated buffers. Removing only the camera module results in 613 B of RAM and 4946 B of ROM (i.e., 7.6 % and 3 %).

OpenSwarm's smallest configuration (i.e., the kernel without any processes and I/O modules) is smaller than TinyOS (512 B) and Mantis OS (500 B), as shown in Table 3.2. Even when adding all additional modules except for the camera module, OpenSwarm consumes approximately 20% more than TinyOS and Mantis OS in their smallest configuration. When adding the camera module, the memory consumption is equal to or lower than that of the other operating systems. The conclusion is that OpenSwarm is small considering memory consumption of other state-of-the-art operating systems.

Listing 3.1 The hardware-near implementation to periodically toggle an LED.

```
#include "p30F6014A.h"

//e-puck specific values
#define OUTPUT 0
#define LED _LATC1
#define DIRECTION_LED _TRISC1

#define THRESHOLD 0xFFFF

int main(void){
  unsigned int value = 0;
  DIRECTION_LED = OUTPUT;
  LED = 0; // turn off LED

  while(true){
    value = value + 1;
    if(value == THRESHOLD){
      LED = ~LED;
      value = 0;
    }
  }
  return 0;
}
```

3.3.2 Processing Overhead

Let the processing overhead be any additional instructions that increase the execution time, t, of an algorithm to $(1 + p)\,t$. As a result, p is a measure for the processing overhead. To measure it, one algorithm is implemented using different system software, which is deployed on the e-puck. Afterwards, their execution times are compared.

The used algorithm toggles an LED after incrementing a counter a specified number of times. Thereafter, the counter is reset, and the previous step repeated. As a result, the LED toggles periodically, and the periodicity is a measure of the execution time. The periodicity itself is measured with an oscilloscope at the LED's anode.

3.3.2.1 Implementation

First, the algorithm is implemented hardware-near (i.e., without any system software) as shown in Listing 3.1. This provides a reference point as any system software would execute additional operations increasing the execution time. Then, the same algorithm is implemented using OpenSwarm (see Listing 3.2), where several configurations are tested (see Table 3.4 for a detailed list). Finally, the algorithm is implemented using ASEBA (see Listing 3.3).

Based on the measured period, t_i, the computational overhead, p_i, can be calculated by

$$p_i = \frac{t_i}{t_0} - 1, \tag{3.1}$$

where t_0 is the measured period of the hardware-near implementation. Note that all implementations are deployed on the same robot to avoid a bias from hardware variations.

Listing 3.2 The OpenSwarm implementation to periodically toggle an LED. Lines 5–7 and 12 are only added when the processing overhead of multiple processes is investigated. Note that modules are en-/disabled with the corresponding macros within `os/system.h`.

```
1  #include "os/system.h"
2
3  #define THRESHOLD 0xFFFF
4
5  void busyThread(){  // only added when multiple threads
   ↪  are tested
6    while(true);
7  }
8
9  int main(void){
10
11   Sys_Kernel_Init();
12   Sys_Start_Process(busyThread,0); // only added when
     ↪  multiple threads are tested
13   Sys_Kernel_Start();
14
15   while(true){
16     value = value + 1;
17     if(value == THRESHOLD){
18       FRONT_LED = ~FRONT_LED;
19       value = 0;
20     }
21   }
22   return 0;
23 }
```

Listing 3.3 The ASEBA implementation to periodically toggle an LED.

```
1  var bool = 1
2  var value = 0
3
4  while bool == 1 do
5    value = value + 1
6
7    if value == 0xFFFF then
8      LED1 = ~LED1
9      value = 0
10   end
11 end
```

Table 3.4: Computational overhead of a hardware-near, OpenSwarm, and ASEBA implementation. Note that, if possible, only one module of OpenSwarm is tested at a time.

Configuration	Period / ms	p_i
Hardware-near	62.22	0
OpenSwarm (initialised)	62.21	0
Event Module	62.21	0
Memory Module	62.22	0
Process Module	62.28	$9.64\,10^{-4}$
+ one thread	128.4	$1.06\,10^{0}$
+ two threads	190.7	$2.07\,10^{0}$
I/O Module	62.40	$2.89\,10^{-3}$
+ Bluetooth Module	62.47	$4.02\,10^{-3}$
+ Remote Control Module	62.33	$1.77\,10^{-3}$
+ Motor Module (idle)		
idle	62.58	$5.79\,10^{-3}$
operating with $128\,\mathrm{mm\,s^{-1}}$	62.90	$1.09\,10^{-2}$
+ Selector Module	62.24	$3.21\,10^{-4}$
+ Communication Module[a] (Proximity Module)[a]		
for 310 bps	82.4	$3.24\,10^{-1}$
for 850 bps	117.2	$8.83\,10^{-1}$
for 1800 bps	178.4	$1.87\,10^{0}$
+ Camera Module	3852[b]	$2.87\,10^{0}$
+ all modules	4509[b]	$3.53\,10^{0}$
OpenSwarm (all w/o camera)	184.8	$1.97\,10^{0}$
ASEBA	15400	$2.47\,10^{2}$

[a] The proximity module is configured to sampling rates that allow the implementation of a communication system — SwarmCom — with 310, 850, and 1800 bps (see Chapter 4 for more details).

[b] As the camera records a few frames per second, the threshold needed to be changed from 65535 to 1048575. Otherwise, the LED would toggle irregularly.

3.3.2.2 Results & Discussion

The measured execution times and processing-overhead values, p_i, are shown in Table 3.4. It is evident that initialising OpenSwarm (i.e., providing its features but not using it) does not increase the processing overhead. As $n_p \in \{0, 1, 2\}$ processes are fairly executed alongside the system thread, each process is executed for $\frac{1}{n_p+1}$ of the time. As their execution time increases by (n_p+1), the processing overhead is $p_i \approx n_p$. This aligns with the observations in Table 3.4.

Overall, it can be seen that all I/O modules have a relatively small overhead (in the range of 0.01–$1\,\%$) with two exceptions — the proximity and the camera module. Both modules have an overhead of an additional 32.4–$186.7\,\%$ and $286.9\,\%$, respectively. When operating both proximity sensors and camera, the combined overhead is $3.5 < (1.87 + 2.87)$, suggesting that either data is lost or corrupted as one of the modules is not being fully executed. Note that this is a result of the design of the robot and can be expected on other system software that is deployed on the e-puck.

In comparison to OpenSwarm, ASEBA has a two magnitudes of power larger processing overhead. In other words, for a single instruction in the hardware-near setup, ASEBA executed an additional 247 in comparison to 3.53 (OpenSwarm) instructions. As a consequence, the limited resources of severely-constrained robots are further reduced, which in return limits the

complexity of deployed algorithms.

Note that the supervisory control framework as used in [Kaszubowski Lopes 2016] has been deployed on the e-puck. However, the used algorithm would require automata with more than 65535 states (for counting up to the threshold). Such automata would exceed the memory of the robot. Ergo, this system software could not be tested and compared.

3.3.3 Swarm Robotics Case Study

To investigate the impact of computational overhead on a swarm robotics task, an algorithm is implemented in four ways — hardware-near, with OpenSwarm, with the supervisory control framework (SCF) of [Kaszubowski Lopes 2016], and with ASEBA. Experiments are conducted and the performances of the system softwares compared.

In this section, object clustering [Gauci et al. 2014*a*] is used to show that the choice of system software can impact algorithms that perform with minimal computational resources. Object clustering is a task where several dispersed objects are pushed together to a cluster. Gauci et al. use a line-of-sight sensor to detect other robots, objects, or nothing (i.e. a wall as the environment is bounded) in front of the robot. The detection is performed through their colour — a robot (green), an object (red), or the environment (white). Each colour is then mapped to predefined wheel velocities.

3.3.3.1 Implementation

Based on [Gauci et al. 2014*a*], the line-of-sight sensor is a virtual sensor realised with the e-puck's onboard 640×480 pixel CMOS RGB camera. The camera is configured with an $8\times$ digital zoom providing a frame with 80×60 pixels. To avoid misalignments, a 20×20 sub-frame is used to detect the dominant colour. The hardware-near, OpenSwarm, and SCF implementation use the described method to avoid a bias resulting from camera calibration. Note that ASEBA provides only a row of 80 pixels, and, as a result, the inner 20 pixels are used to determine the dominant colour.

To execute the algorithm with OpenSwarm, the camera's pre-processor performs the image processing as described above. It also emits an `SYS_IO_1PXSENSOR` event containing the dominant colour. As shown in Listing 3.4, this event is then handled by `object_clustering` , which then performs the algorithm described by [Gauci et al. 2014*a*]. Note that the presented algorithm is implemented in OpenSwarm version 0.15.09.15 as used in [Trenkwalder et al. 2016].

Listing 3.5 shows the implementation in ASEBA. First, the virtual line-of-sight sensor value is calculated from the central 20 pixels. Then, based on the dominant colour, the corresponding velocities are set.

Finally, the SCF implementation is taken from [Kaszubowski Lopes 2016]. The listing for the hardware-near implementation is omitted due to its length.

3.3.3.2 Experiment Setup

For each implementation, ten trials are performed. Each trial deploys five robots to push 20 dispersed objects to a cluster, as shown in Figure 3.6. Each object has a diameter of 10 cm and a height of 10 cm. Both robots and objects are fitted with green and red paper jackets, respectively.

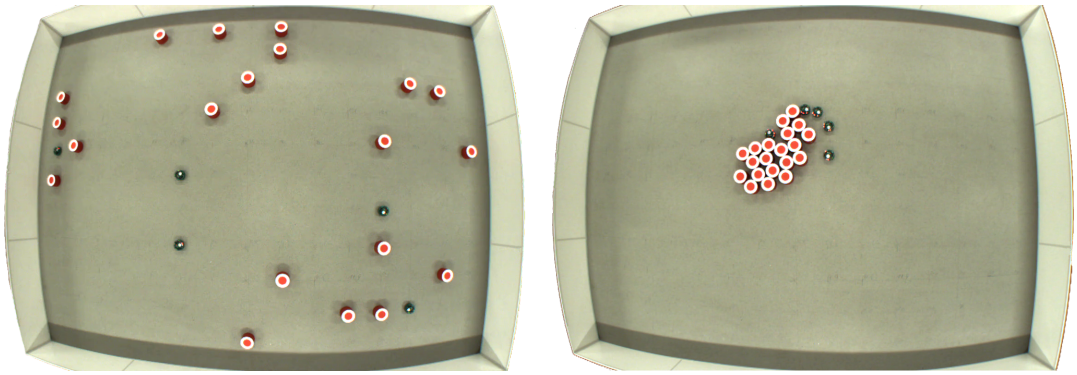
The used arena is 4×3 m, surrounded by 50 cm tall white walls. On the light-grey floor, 165 equally spaced pencil marks are arranged in a 15×11 grid. 25 marks are randomly selected to place first objects and then robots. The starting orientation of each robot is randomly chosen from $\left\{0, \frac{\pi}{4}, \ldots, \frac{7\pi}{4}\right\}$ to avoid bias. For each implementation, the same ten placements and

Listing 3.4 The object clustering algorithm implemented with OpenSwarm. OpenSwarm ensures that, on every occurrence of the camera event, `SYS_IO_1PXSENSOR`, the algorithm of [Gauci et al. 2014*a*] is executed.

```
#include "os/system.h"
bool object_clustering(uint16 PID, uint16 EventID,
 ↪  sys_event_data *data);

int main(void){
  Sys_Init_Kernel();//initialise OS

  Sys_Subscribe_to_Event(SYS_IO_1PXSENSOR, 0,
   ↪  object_clustering, NULL);

  Sys_Start_Kernel();//start OS
  Sys_Run_SystemThread(); //run system thread
}

bool object_clustering(uint16 PID, uint16 EventID,
 ↪  sys_event_data *data){
  sys_colour rx_color = *((sys_colour *) data->value);

  switch(rx_color){//what colour did the robot see?
  case GREEN://robot
    Sys_Send_IntEvent(SYS_LEFTMOTOR_SPEED, 114);
    Sys_Send_IntEvent(SYS_RIGHTMOTOR_SPEED,68);
    break;
  case RED://object
    Sys_Send_IntEvent(SYS_LEFTMOTOR_SPEED, 125);
    Sys_Send_IntEvent(SYS_RIGHTMOTOR_SPEED, 64);
    break;
  case WHITE://nothing or wall
    Sys_Send_IntEvent(SYS_LEFTMOTOR_SPEED, 70);
    Sys_Send_IntEvent(SYS_RIGHTMOTOR_SPEED, 127);
    break;
  default://anything else
    break;//do not change behaviour
  }
  return true;
}
```

Listing 3.5 The object clustering algorithm implemented with ASEBA. On every occurrence of the camera event, the inner 20 of 80 pixels are used to determine the dominant colour and then to execute algorithm of [Gauci et al. 2014*a*]. Note that the colour thresholds were obtained through calibration on the used robot.

```
var redness
var greenness
var whiteness
var i

onevent camera
redness = 0
greenness = 0
whiteness = 0
for i in 30:49 do
  if cam.red[i] >= 25 and cam.green[i] >= 20 then
    whiteness++
  elseif cam.red[i] >= 25 and cam.green[i] < 20 then
    redness++
  elseif cam.red[i] < 25 and cam.green[i] >= 20  then
    greenness++
  end
end
if whiteness > redness then
  if whiteness > greenness then
    speed.left = 70
    speed.right = 127
  else
    speed.left = 114
    speed.right = 68
  end
else
  if redness > greenness then
    speed.left = 125
    speed.right = 64
  else
    speed.left = 114
    speed.right = 68
  end
end
```



(a) (b)

Figure 3.6: Snapshots of the object clustering experiment: (a) initial distribution of robots and objects; (b) distribution after 10 min. (reprinted from [Trenkwalder et al. 2016])

orientations are used. Each trial is recorded with an overhead camera[3.3.I] and lasts approximately 15 minutes. Overall, all experiments followed the same protocol as described in [Gauci et al. 2014*a*].

3.3.3.3 Performance Metrics

Each trial is statically analysed by applying two performance metrics — dispersion and cluster metric. The dispersion metric, is based on the second moment of dispersed disks as presented in [Graham and Sloane 1990]. Gauci et al. [2014*a*] adapted this metric to

$$u = \frac{1}{d_{object}^2} \sum_i \| \boldsymbol{p}_i - \bar{\boldsymbol{p}} \|^2, \tag{3.2}$$

where d_{object} and $\boldsymbol{p}_i$ are the diameter and position of an object. $\bar{\boldsymbol{p}}$ represents the centroid of all 20 objects (i.e., $\bar{\boldsymbol{p}} = 0.05 \sum_i \boldsymbol{p}_i$). As the distances between objects can be arbitrarily long, the second moment, u, is unbound above. However, u has a lower bound of approximately 41.5 when considering $\| \boldsymbol{p}_i - \boldsymbol{p}_j \| \geq d_{object}$ for all $i \neq j$, based on [Graham and Sloane 1990].

While this metric shows how compact a cluster is, a small number of outliers can degrade the results disproportionally. To compensate for this drawback, the second metric, the cluster metric, counts the number of objects within the biggest cluster. As Gauci et al. does only provide a verbal description, let this metric be defined in this work as follows.

Let a graph, $G = (V, E)$, contain objects as vertices (in V), where an edge between two objects, i and j, is $(i, j, w) \in E$. The weight of an edge, w, is defined as the distance between two objects — $w = \| \boldsymbol{p}_i - \boldsymbol{p}_j \| = \| \boldsymbol{p}_j - \boldsymbol{p}_i \|$. Clusters can be defined as a connected graph $C_k = (V_k, E_k) \subseteq G$, where any object can only be in one cluster (i.e., $V_k \cap V_l = \emptyset$ for all $k \neq l$) and every object belongs to a cluster (i.e., $\bigcup_k V_k = V$). A single cluster is defined as

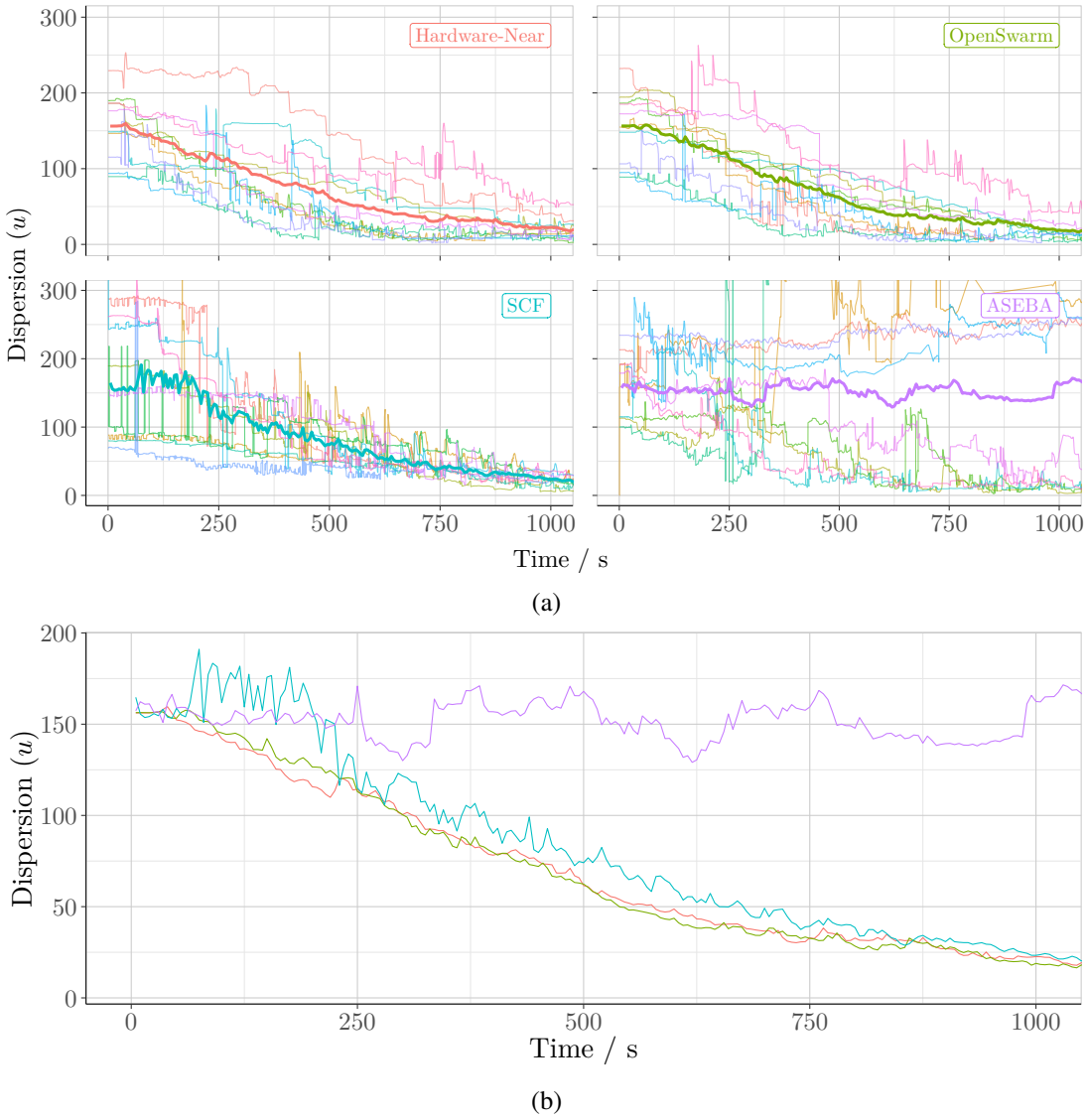
$$V_k = \{\, i, j \in V \mid (i \neq j)\, [\, w \leq 2\, d_{object}\,]\}\,, \tag{3.3}$$

$$E_k = \{\, (i, j, w) \mid (\forall i \in V_k)\, (\exists j \in V_k)\, [w \leq 2\, d_{object}]\}\,, \tag{3.4}$$

with an object diameter d_{object}. In other words, a cluster is a set of objects where any object is not further away than $2\, d_{object}$ to at least one other object of that cluster. With the indicator function, $I_k(i)$, that i belongs to a cluster C_k, the cluster metric is

$$c = \max_k \left\{ \frac{1}{20} \sum_i I_k(i) \right\}. \tag{3.5}$$

[3.3.I] All recorded experiments are available online [Trenkwalder 2020*d*]



(a)

(b)

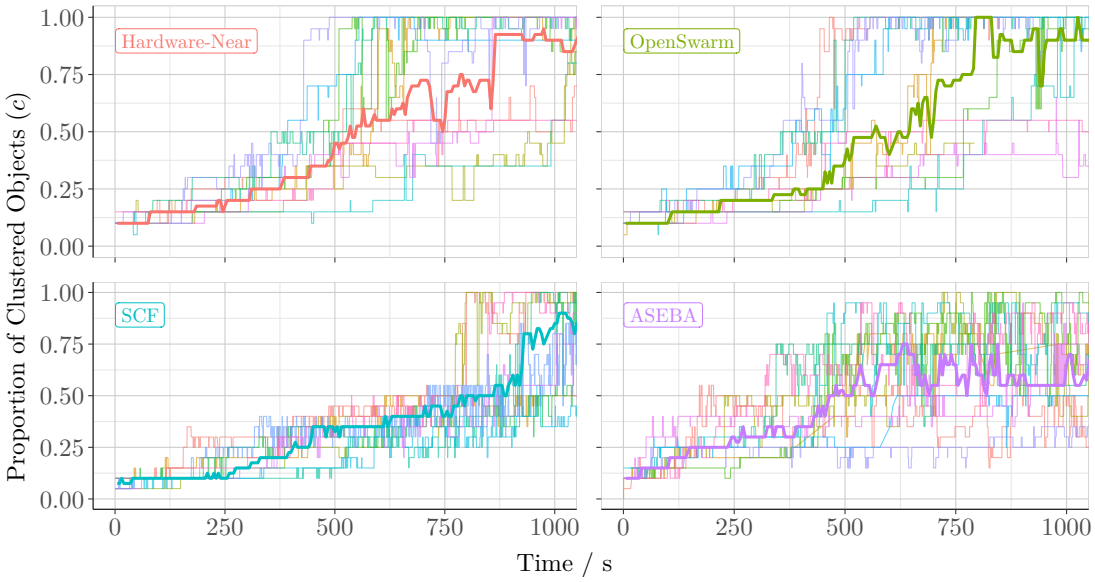
Figure 3.7: Dispersion metric applied to experimental data obtained in a 4×3 m arena. (a) shows the trials (thin lines) and the average (bold) for each implementation. (b) compares the averages over all trials. The colours of the averages correspond in both plots.

The cluster metric illustrates if all objects are pushed together. However, it does not take into account the spatial distribution of the cluster. For instance, a cluster reaching the maximum value can be compact or, in the worst case, a straight line. As a result, both metrics are used to describe the performance of the presented implementations, where better-performing systems converge faster to the final value.

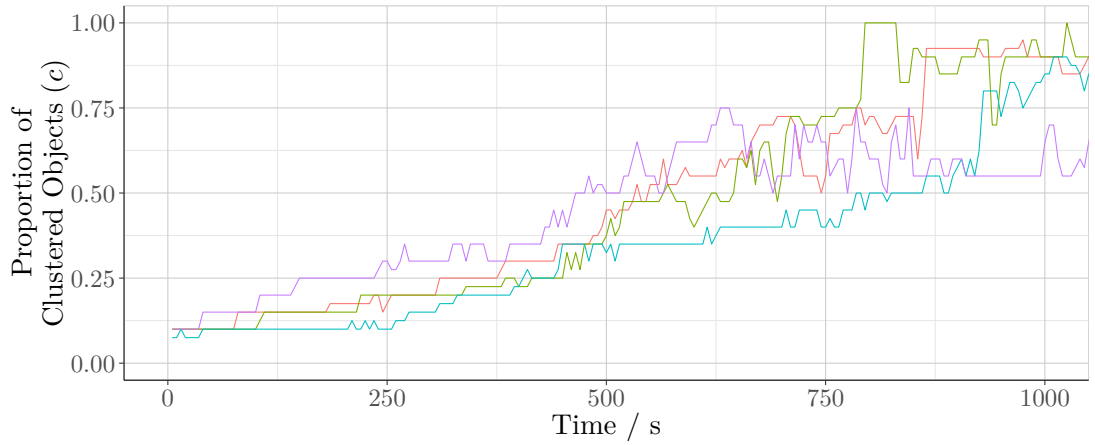
3.3.3.4 Results for an 4×3 m Arena

After conducting and recording each trial, a static analysis of the video is conducted. Both metrics are applied to each video frame. The results are shown in Figure 3.7 and 3.8.

In Figure 3.7a, the dispersion metric shows each of the ten trials represented as a coloured thin line and their average values as a bold line. When comparing the averages, as shown in Figure 3.7b, one sees that the performance of both the hardware-near and OpenSwarm implementation follow the same trend; hence, perform similarly. The SCF implementation follows the trend above; however, an offset can be seen. In contrast, the ASEBA implementation res-



(a)



(b)

Figure 3.8: Cluster metric applied to experimental data obtained in an 4×3 m arena. (a) shows the trials (thin lines) and the median (bold) for each implementation. (b) compares the median over all trials. The colours of the medians correspond in both plots.

ults do not follow the same trend. On average, objects remained dispersed. This results from only a subset of trials being successful while others create multiple clusters apart, which even increases the dispersion value.

These findings are also supported by Figure 3.8, where the hardware-near and OpenSwarm implementation increase the cluster size similarly swiftly. Furthermore, there is also a small offset to the SCF implementation, while ASEBA converges to cluster sizes that lie below the other systems.

The different trends for ASEBA can stem from multiple factors. The ASEBA implementation uses a different camera implementation than the other systems. Even though the detection of colour was calibrated to the used robot, this implementation missed objects that are relatively far away more frequently than other implementations. This can stem, for instance, from a lower frame rate (i.e., less angular sampling) or how pixels are extracted. In comparison, as the SCF, OpenSwarm, and hardware-near implementation use the same code to operate the camera, the offset to the latter must result from a computation overhead (i.e., a longer response time).

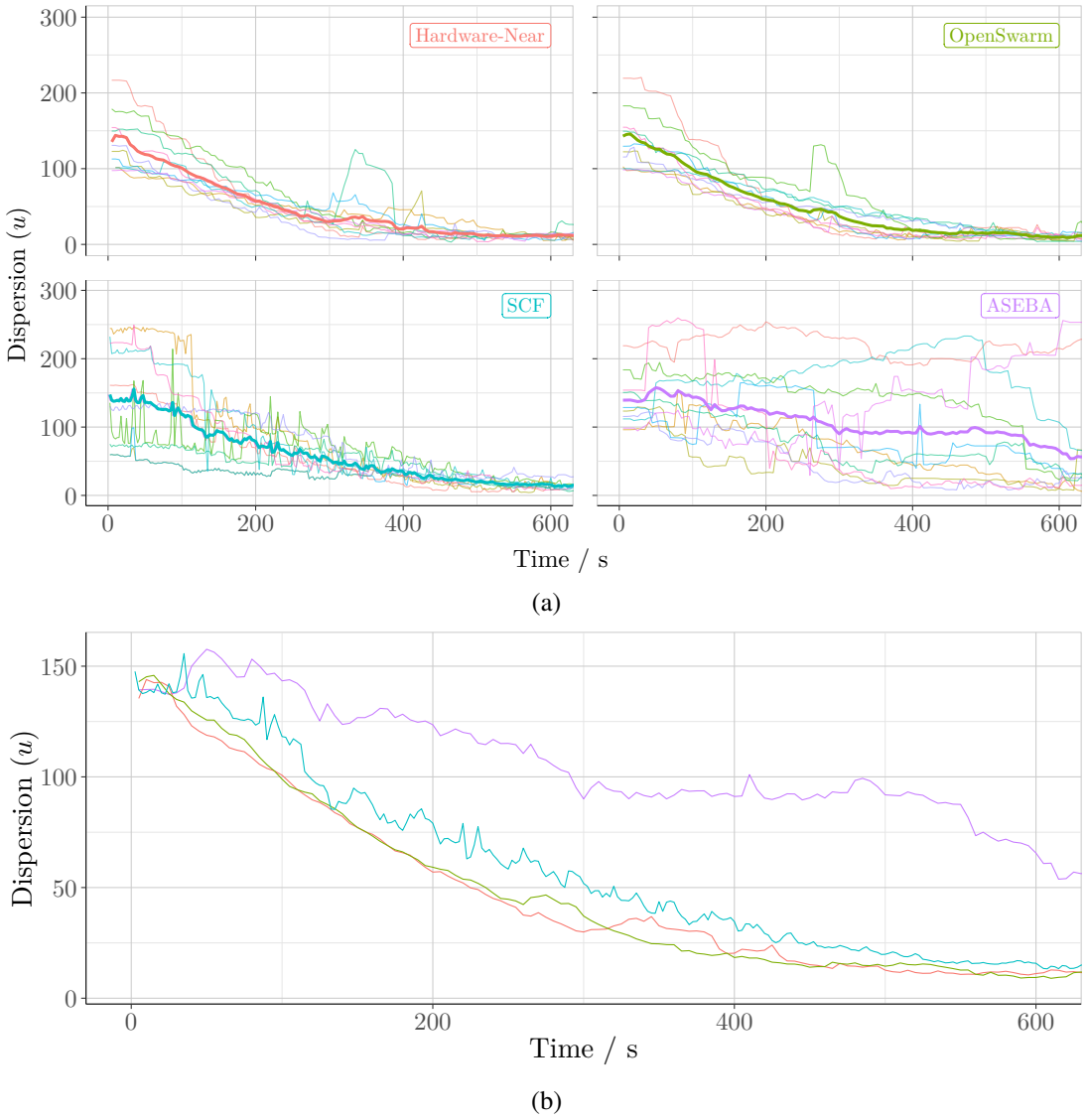
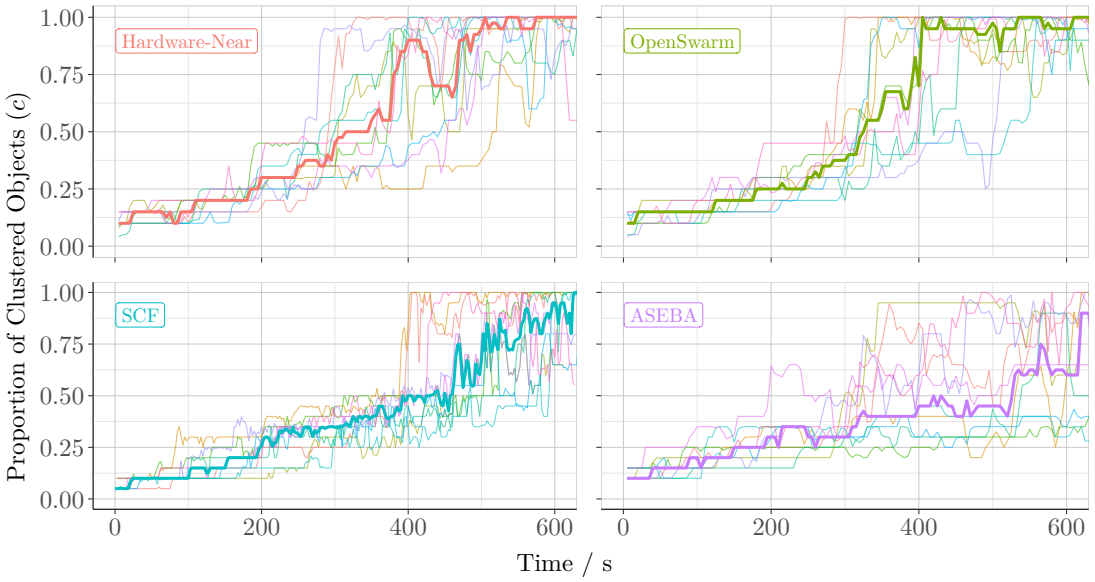


Figure 3.9: Dispersion metric applied to experimental data obtained in an 3×3 m arena. (a) shows the trials (thin lines) and the average (bold) for each implementation. (b) compares the averages over all trials. The colours of the averages correspond for both plots.

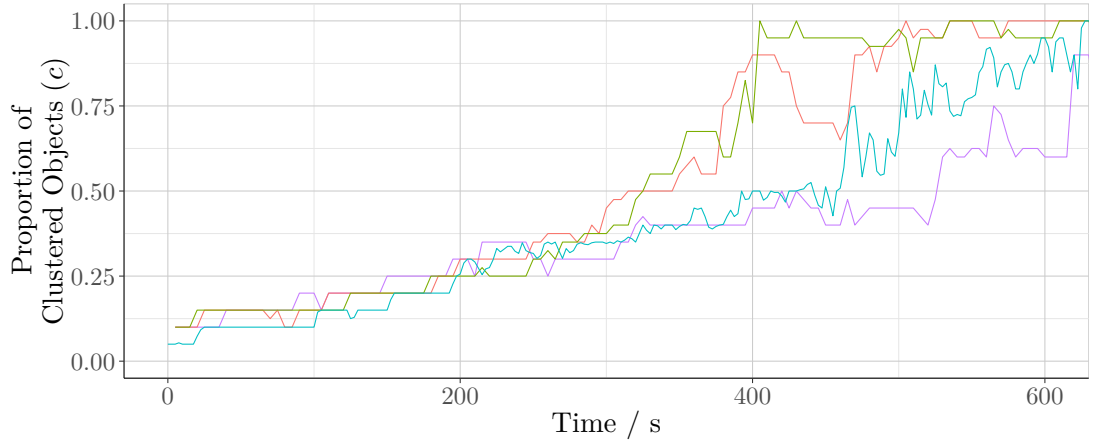
3.3.3.5 Results for an 3×3 m Arena

As the large distances caused ASEBA to underperform, the experiments were repeated within an arena with reduced size (from 4×3 to 3×3 m). After conducting an additional 10 trials for each implementation, both metrics are applied as in the previous section. The results are shown in Figure 3.9 and 3.10.

The hardware-near, OpenSwarm, and SCF implementation perform similarly with SCF, which has a notable offset, being the only exception. Furthermore, it can be seen that a reduced arena size improves the performance of the ASEBA implementation resulting in larger clusters; though, it still underperforms. Similarly, Figure 3.10 shows a wide range of outcomes for each trial for ASEBA. Overall, ASEBA performs worst, followed by SCF and then OpenSwarm.



(a)



(b)

Figure 3.10: Cluster metric applied to experimental data obtained in an 3×3 m arena. (a) shows the trials (thin lines) and the median (bold) for each implementation. (b) compares the medians over all trials. The colours of the medians correspond in both plots.

3.4 Discussion

OpenSwarm is the first operating system designed for severely-constrained robotic systems. It provides a novel dual-execution model and hybrid kernel that offers both process-based and event-based execution. While separately each execution model has been available on other system software, the combination of both features allows the development of both fast-responding and time-consuming code. Event-driven programming, being a more rule-based approach, often allows a more intuitive programming (e.g., if this event occurs, execute that function). In addition, it also provides a smaller overhead when called. Process-oriented programming, on the other hand, allows the execution of long-lasting algorithms, such as image processing. It is widely established and well understood. By enabling both programming paradigms in OpenSwarm's hybrid kernel, the user can optimise the resource consumption. In addition, OpenSwarm provides a two-layer hardware abstraction that allows the transparent use of hardware and provides additional context (e.g., the use of $128\,\mathrm{mm\,s^{-1}}$ instead of a unit-less 1000) when interacting with the robot's environment. Moreover, this allows, in principle, the creation

of more hardware-independent, general, and readable code, which potentially reduces errors [MacConnell 1993]. Additionally, the provided abstraction enables the use of virtual sensors/actuators and, therefore, virtual or augmented reality environments, which broadens the scope of how robots can be used. Furthermore, virtual sensor/actuators reduce dependency on specific hardware owing to the fact that not-available hardware can be simulated. Both properties are explored in Chapter 5.

In this chapter, OpenSwarm has been compared to other systems software. A detailed analysis of the memory consumption and processing overhead has shown that OpenSwarm has a small memory footprint and processing overhead. With the computational overhead being small, more resources are available to implement and execute robotic behaviours.

While a computational overhead does not necessarily have an impact on a robotic behaviour, additional experiments were conducted to measure whether a different system software does. It was shown that OpenSwarm closely matches the performance of the hardware-near implementation while SCF and ASEBA implementation have a small and notable performance reduction, respectively. The ASEBA implementation, in particular, had the worst performance and was, in many cases, not able to cluster all objects.

In general, it has been shown that the choice of system software can bias the outcome of even simple algorithms. When this bias is not known, any research results/findings could be rendered less reproducible or even void.

Finally, the computational overhead experiments revealed a general flaw of the e-puck platform. It was shown that the camera and proximity sensors interfere with each other's function. This can lead to missing or corrupt data. As a result, the e-puck is resource inadequate as defined in [Kopetz 2011], and both sensors should not be used together.

Overall, this chapter discusses the different variants of code execution when comparing OpenSwarm with other operating systems as well as robotic system software. While code of severely-constrained robots is often executed on a virtual machine, OpenSwarm presents execution directly on hardware, which has shown to be more execution efficient. It was demonstrated that the choice of system software does affect behaviours even computationally minimalistic ones.

4

SwarmCom: Communication on Severely-Constrained Robots

Contents

Distributed systems — in this case, swarms of robots sharing and combining resources — depend largely on managing local resources, communicating between robots, and managing of shared and combined resources. As the previous chapter is concerned with the management of local resources, this chapter addresses the communication between robots.

Communication is an essential aspect of robotics and has been proven to benefit cooperative multi-robot systems [Balch and Arkin 1994]. While robotics divides communication into *implicit*[4.0.1] and *explicit* communication, this work focuses on *explicit* communication, which allows the exchange of arbitrary data, and also enables a wide range of behaviours, such as [Hauert et al. 2010; Schmickl et al. 2011; Rubenstein et al. 2014; Garattoni and Birattari 2018].

In general, any communication system can be characterised by

- network *reliability*, which defines the probability that a message is transmitted and received correctly, and
- network *throughput*, which defines how much data can be sent within a period.

In robotics, in particular, both characteristics are paramount as distributed systems often require the exchange of correct data within a set time. In swarm robotics, it is often assumed that the

[4.0.1] Implicit communication describes the obtaining of information based on observations of other robots or the environment [Wang and Schwager 2016].

communication system works reliably. This results in limited research on communication, which may affect the validity and reproducibility of the deployed system.

Due to its large number of robots [Hamann 2018], its computational constraints [Trenkwalder 2019], and distributed behaviour [Barca and Sekercioglu 2013], a communication system for swarm robotics requires

- *simplicity* in design to be deployable on severely-constrained robots,
- *decentralisation* to allow large numbers of robots,
- *scalability* to allow close interaction between large numbers of robots, and
- *tolerance towards frequent topology changes* to allow mobility.

As the requirements are often conflicting, each system design is based on a trade-off benefiting the domain of use, resulting in a large variety of different systems.

4.1 Existing Communication Systems

Communication systems are utilised in many aspects of robotics and form a separate branch of robotics called networked robotics [Siciliano and Khatib 2016, Chapter 44]. It focuses on architectures, hardware, interfaces, and programs that enable behaviours requiring networking and is an intersection between telecommunication and robotics. It has three main subcategories — cloud robotics, telerobotics [Siciliano and Khatib 2016, Chapter 43] and networked multi-robot systems [Yan et al. 2013].

4.1.1 Cloud Robotics

As cloud robotics utilises the external infrastructures, robots require reliable and high bandwidth connection [Terrissa et al. 2015; Kehoe et al. 2015; Wan et al. 2016]. As described in Section 2.2.1, these systems often utilise well-established technologies such as Ethernet, IEEE 802.3, or Wifi, IEEE 802.11 [IEEE 2017]. However, these technologies are often unsuited for many swarm robots as they (I) have relatively high power consumption, (II) are not scalable to large numbers of robots [Ghazisaidi et al. 2009]. Furthermore, cloud robotics is designed for an individual robot and uses a centralised structure, which would be a bottleneck for large numbers of robots. In addition, Hamann [2018] categorises such systems as the more-generic multi-robot rather than swarm robotics systems due to individual operation of robots and their lack of scalability.

4.1.2 Telerobotics

Telerobotics focuses on robots that are operated remotely by a human (e.g., surgical robots [Munawar and Fischer 2016]). These systems also require a reliable and high-bandwidth connection with deterministic response times as they are often used in safety-critical environments. In contrast to the other areas networked robotics, these systems are often stationary and use wired connections allowing a high *quality-of-service*[4.1.1]. Due to (I) the stationary character, (II) its wired connection, and (III) the use of one operator per robot, telerobotics is less relevant to swarm robotics and, therefore, not discussed further.

4.1.3 Multi-Robot Systems

Multi-robot systems, in particular swarm robotics systems, are composed of a group of robots working cooperatively towards a common goal. They often involve synchronised actions or

[4.1.1] In digital communication, the quality of service (QoS) is a description of the overall performance, which includes packet loss, bit error, bit rate, and transmission delay.

collective decision making [Yan et al. 2013]. Overall, there are three categories of multi-robot systems in which communication plays a crucial role — modular/reconfigurable robotics, underwater robotics, and swarm robotics.

4.1.3.1 Modular and Reconfigurable Robotics

Modular and reconfigurable robots are devices that physically connect, resulting in larger robots [Chennareddy et al. 2017; Moubarak and Ben-Tzvi 2012]. While each module (i.e., an individual robot) is mobile, connected robots are often position-locked, which facilitates the use of wired connections. As a result, many of these systems use wired connections for inter-robot communication (e.g., [Kernbach et al. 2008]). Note there are platforms that use wireless communication (e.g., [Pacheco et al. 2014; 2015]); however, wired communication offers better a quality of service and calls for less complex hardware (i.e., easier miniaturisation). Common technologies are (wired) CAN, I^2C, UART, and Ethernet as well as (wireless) Wifi, ANT, Bluetooth, and Zigbee.

In contrast, swarm robotics uses mobile robots, which hinders the use of wired communication. Furthermore, many of the used wireless technologies are designed for smaller numbers of robots (e.g., Bluetooth can connect up to 7 slaves at a time) and are centralised, contradicting key aspects of swarm robotics.

4.1.3.2 Underwater Robotics

Another form of networked mobile multi-robot systems are underwater robotic platforms [Yuh 2000; Antonelli 2006]. Due to the limited range of optical and radio transmission, these robotic platforms are often connected through acoustic networks similar to underwater sensor networks [Chandrasekhar et al. 2006; Tuna and Gungor]. Acoustic waves can propagate far (between $100\,\mathrm{m}$ to several km) in dense media such as water. However, it requires low-frequency carrier waves, which results in small throughputs (up to 30 kbps) [Climent et al. 2014]. Furthermore, its slow propagation speed makes media access control[4.1.II], and the corrections of signal distortion challenging. In other words, when the number of robots is large (i.e., swarms), robots interfere in each other communication. These characteristics make acoustic communication less suitable for swarm robotics than other technologies.

4.1.3.3 Swarm Robotics

As swarms are formed by mobile robots, they use wireless communication, such as Bluetooth, Wifi, and Zigbee, [Hauert et al. 2010; Schmickl et al. 2011; Rubenstein et al. 2014; Garattoni and Birattari 2018]. While many of these technologies are centralised and limited in the number of connections, swarm robotics tends to uses these technologies with low numbers of robots (e.g., [Pickem et al. 2017]) or with small densities (e.g., [Hauert et al. 2010]). In swarm robotics, a communication system is often decentralised to enable distributed control. Decentralised communication is often achieved by networks, in which robots are connected, lacking a central managing hub. One such network type is MANET.

Mobile ad-hoc networks (MANETs) are decentralised and distributed networks. They wirelessly connect mobile devices — called nodes —, and work without additional infrastructure as each node can be the origin, end-point, or relay of a transmission. Until now, a wide range of MANETs have been investigated [Macker and Corson 1998; Chlamtac et al. 2003; Conti and Giordano 2014]. In swarm robotics, the merits of MANETs have been increasingly recognised, resulting in many radio-based [Li et al. 2009; Tutuko and Nurmaini 2014]

[4.1.II] Media access control is a method in which multiple devices regulate access to the communication channel to avoid interference and collisions.

and optical [Di Caro et al. 2009; Gutiérrez et al. 2009*a*; Rubenstein et al. 2012] MANETs. Fundamentally, both technologies are relevant to this work.

Each technology has a set of benefits that can aid swarm robotics:

- Generally, radio-based systems are widely used, which results in accessible and inexpensive components. Due to its standardisation, many systems are compatible directly off the shelves. This allows robots to be produced cheaply and in larger numbers. In comparison, optical communication systems are still at early development stages. They lack standardisation and often require repeated implementation efforts. However, optical communication is increasingly used as an alternative to radio-based systems. Research in free-space optical communication shows that a bit can be transmitted faster and with less energy [Anees and Bhatnagar 2015; Malik and Singh 2015; Khan 2017]. In particular in swarm robotics, low energy consumption is paramount as it allows longer operation of robots, and more throughput can enable more complex behaviours.
- Radio-based systems can penetrate many objects, which increases communication range and the number of robots that can be reached even in cluttered environments. However, the network devices are limited in regards the number of connections they can establish. For example, a single Bluetooth master can connect to up to 7 nodes within a 3 to 10 m radius. If the number of robots is large, they can interfere with each other and worsen communication quality. While many radio-based systems are designed for environments with relatively low densities of nodes (e.g., office), swarm robotics is likely to reach large densities[4.1.III] of robots; hence, they would exceed the capabilities of many state-of-the-art networks (e.g., Wifi, Bluetooth, Zigbee).
  Optical signals, on the other hand, are line-of-sight transmissions which can be obstructed by objects, reducing the range and the number of possible channels. As shown later in this chapter, this enables scalability and makes the communication system applicable to high-density scenarios.
- Radio-based signals are emitted and received by antennas, and its size depends on the carrier's wavelength (e.g., 6 cm for a 5 GHz signal used in Wifi (IEEE 802.11n)). This makes miniaturisation challenging. On the other hand, optical systems use LEDs to emit and photodetectors to receive signals. Both have good potential to be miniaturised and integrated, as shown by [Hirschman et al. 1996] or CMOS camera chips.

Due to their benefits, optical-based communication systems are increasingly used on the miniature robots [Seyfried et al. 2005; Caprari and Siegwart 2005; Gutiérrez et al. 2008; Arvin et al. 2009; Rubenstein et al. 2014; McLurkin et al. 2014].

4.1.4 Discussion

Robotics deploys a wide range of communication systems commonly based on acoustic, radio, or optical signal transmissions. Because of the technical challenges that arise with acoustic carrier signals, it is, in many cases, only deployed in environments that hinder the use of optical and radio transmissions. In contrast, communication systems based on radio signals are most common, not only in robotics. Radio waves can penetrate objects and reach a large number of nodes over relatively large distances, which is useful in many situations (e.g., search and rescue). However, the limited capability to miniaturise and to scale to a large number of robots makes this technology less suitable for areas such as swarm robotics. In contrast, miniaturisation and the use of large densities[4.1.IV] of robots are facilitated when using optical communication systems. The drawback of optical systems is that they are often unique and not compatible with other systems. As these unique systems are often not analysed regarding quality of service, the communication characteristics are often not well understood, and the

[4.1.III] For instance, up to 662 robots [Mondada et al. 2009] or 29980 robots [Rubenstein et al. 2014] within a 3 m radius.

[4.1.IV] Note that the scalability of optical systems is shown in a later section of this chapter.

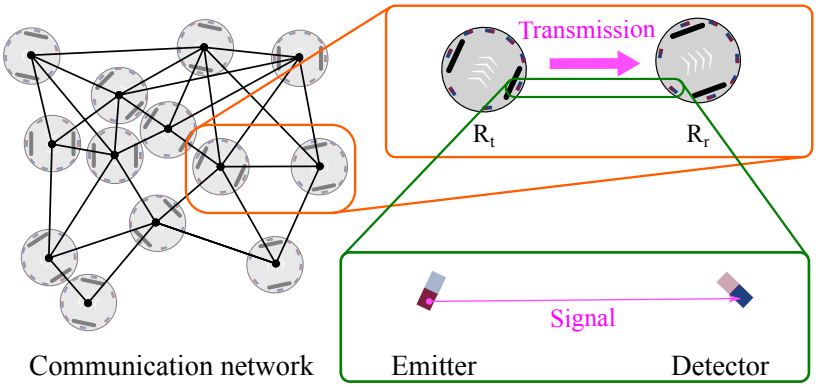


Figure 4.1: The SwarmCom communication network. The network (right) consists of links between robots (orange), where each robot transmits signals from emitters (red) to detector (blue).

system is assumed to operate correctly. This can lead to less reproducible research or undefined behaviour.

This chapter provides a framework to facilitate systematic studies of the communication channels of robotic swarms and the design of reproducible behaviour. First, a channel model is presented that describes the infra-red signal characteristics of a widely used swarm robotics platform, the e-puck. The proposed model is the first such model for any swarm robot. It shows in-detail how signals are received or transmitted, understanding of which is crucial to formally describe the communication channel. In the second part, SwarmCom, an optical (infra-red) MANET for severely-constrained robots, is proposed. It contains a dynamic detector that adapts to the environment and other robots. Finally, SwarmCom is compared to another state-of-the-art infra-red communication software for e-pucks — libIrcom [Gutiérrez et al. 2009*b*]. In addition to these contributions, this chapter answers two fundamental questions: "What is the impact of different communication ranges, mobility, and bit rates on the optical communication?" and "Can optical communication systems scale with the number of robots?". This allows a better understanding of such systems.

4.2 Swarm Robotics Network

To model and design a robotic network, the physical arrangements of the robots must be specified first. Let a group of identical robots operate on an unbounded plane. It is assumed that a robot is not further than $50\,\mathrm{cm}$ from at least one other robot if they cooperate. Overall, the robots form a partially connected mesh network, as shown in Figure 4.1.

4.3 Channel Model

The channel model describes how data is conveyed between two robots. It is composed of models that describe:

- how signals are transmitted between an emitter and a detector (emitter-to-detector model),
- how two robots transmit data with multiple emitters and detectors (robot-to-robot model), and
- how the signals are represented and understood by the robot (measurement model).

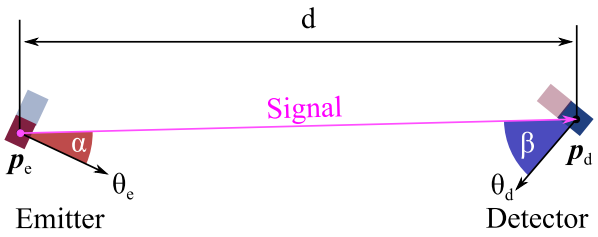


Figure 4.2: Signal propagation between an emitter (red) and a detector (blue).

4.3.1 Emitter-to-Detector Model

Let an emitter and a detector be located at $\boldsymbol{p}_e, \boldsymbol{p}_d \in \mathbb{R}^2$ with orientation $\theta_e, \theta_d \in (-\pi, \pi]$, respectively. As shown in Figure 4.2, a signal propagates in a direct line from emitter to detector with an emission and inclination angle, $\alpha \in (-\pi, \pi]$ and $\beta \in (-\pi, \pi]$, over the Euclidean distance $d \in \mathbb{R}$:

$$\alpha = \theta_e - \angle(\boldsymbol{p}_d - \boldsymbol{p}_e), \tag{4.1}$$
$$\beta = \angle(\boldsymbol{p}_e - \boldsymbol{p}_d) - \theta_d, \tag{4.2}$$
$$d = \| \boldsymbol{p}_d - \boldsymbol{p}_e \|_2, \tag{4.3}$$

where $\angle(\boldsymbol{v})$ is the angle between the vectors $\boldsymbol{v}$ and $[1\,0]^T$.

Optical communication systems commonly use on-off-modulation[4.3.I] to transmit one of two symbols, s_0 and s_1, for a symbol period T. Therefore, the normalised transmitted waveform is $s(t) : \mathbb{R} \to [0, 1]$, which is pseudo-static[4.3.II] for T. The normalised signal intensity, $y(t)$, arrives at the detector as

$$y(t) = h_c\, s(t) + n(t) \quad \text{with} \quad 0 \leq t < T, \tag{4.4}$$

where h_c is the channel attenuation (or shadowing) coefficient, and $n(t)$ is additive white Gaussian noise (AWGN). These are common assumptions for optical systems (e.g., [Carruthers and Kahn 1997]).

The channel attenuation is defined by

$$h_c = h_c(\alpha, d, \beta) = h_e(\alpha)\, h_m(d)\, h_d(\beta), \tag{4.5}$$

where h_e, h_m, and h_d are attenuation coefficients for the emitter, medium, and detector, respectively.

The emitter coefficient, h_e, models the attenuation caused by the signal's direction and the emitter's orientation (i.e., directionality). Consequently, h_e can be expressed as $h_e(\alpha) : (-\pi, \pi] \to [0, 1]$, where α is the emission angle. The maximum normalised attenuation is $h_e(0) = 1$ and, due to self-occlusion, $h_e(\alpha) = 0$ for all $\alpha \notin [-\frac{\pi}{2}, \frac{\pi}{2}]$. Similarly, $h_d(\beta) : (-\pi, \pi] \to [0, 1]$ denotes the detector attenuation, which depends on the inclination angle β with $h_d(0) = 1$ and $h_d(\beta) = 0$ for all $\beta \notin [-\frac{\pi}{2}, \frac{\pi}{2}]$.

The medium attenuation, h_m, describes effects on the signal when propagating through the medium. When the distance between emitter and detector tends to zero, the medium attenuation should not have an effect, resulting in $\lim_{d\to 0} h_m(d) = 1$. Similarly, if the distance is infinitely long, the signal intensity is infinitely small, resulting in $\lim_{d\to\infty} h_m(d) = 0$.

Note that the exact coefficients obtained through experiments are presented in Section 4.3.4.

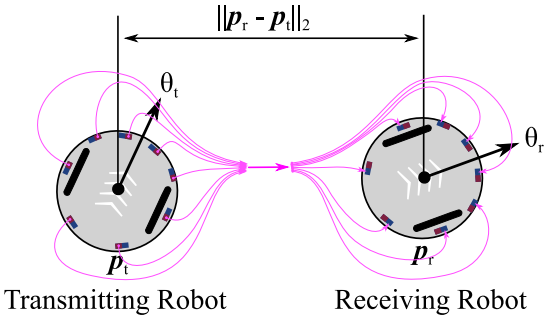


Figure 4.3: Setup of the robot-to-robot model, where eight emitters send a signal to eight detectors.

4.3.2 Robot-to-Robot Model

Let a transmitting and a receiving robot be at $\boldsymbol{p}_t, \boldsymbol{p}_r \in \mathbb{R}^2$ with orientation $\theta_t, \theta_r \in (-\pi, \pi]$, as shown in Figure 4.3. Based on the relative positions of each emitter $i \in \{1, 2, \dots 8\}$ and detector $j \in \{1, 2, \dots 8\}$ (see Appendix A for more details), the absolute positions are calculated by

$$\boldsymbol{p}_{e,i} = \begin{bmatrix} \cos(\theta_t) & -\sin(\theta_t) \\ \sin(\theta_t) & \cos(\theta_t) \end{bmatrix} \boldsymbol{p}_{e,i}^{loc} + \boldsymbol{p}_t, \tag{4.6}$$

$$\boldsymbol{p}_{d,j} = \begin{bmatrix} \cos(\theta_r) & -\sin(\theta_r) \\ \sin(\theta_r) & \cos(\theta_r) \end{bmatrix} \boldsymbol{p}_{d,j}^{loc} + \boldsymbol{p}_r. \tag{4.7}$$

With these coordinates, the superpositioned[4.3.III] signal intensity, $y_j(t)$ of a detector j is

$$\alpha_{i,j} = \theta_i - \angle(\boldsymbol{p}_{d,j} - \boldsymbol{p}_{e,i}), \tag{4.8}$$

$$d_{i,j} = d_{j,i} = \|\boldsymbol{p}_{d,j} - \boldsymbol{p}_{e,i}\|_2, \tag{4.9}$$

$$\beta_{j,i} = \angle(\boldsymbol{p}_{e,i} - \boldsymbol{p}_{d,j}) - \theta_j, \tag{4.10}$$

$$h_{c,j} = \min\left\{ \sum_i h_e(\alpha_{i,j})\, h_m(d_{i,j})\, h_d(\beta_{j,i}),\, 1 \right\}, \tag{4.11}$$

$$y_j(t) = h_{c,j}\, s(t) + n_j(t), \tag{4.12}$$

where $n_j(t)$ is AWGN. It is worth noting that a detector can saturate limiting $h_{c,j}$ to 1. This makes $y_j(t)$ the signal intensity at the detector j for a transmitted symbol, $s(t)$.

4.3.3 Measurement Model

When $y_j(t)$ reaches the detector, the measuring circuit transforms it into an electrical signal. This signal (i.e., voltage) is then sampled by an analog-to-digital-converter (ADC) of the MCU and makes the digital value available to any software.

As illustrated in Appendix A.1.2.2, $y_j(t)$ generates a proportional collector current[4.3.IV], $I_C(t)$, at the phototransistor of the detector j. This current causes a voltage drop from the supply voltage, V_{cc}, at the resistor R, which results in the collector-emitter voltage

$$V_{CE}(t) = V_{cc} - R\, I_C(t). \tag{4.13}$$

[4.3.I] On-Off Keying (OOK), also on-off modulation, is a modulation technique where one of two symbols is assigned to the presence and the other to the absence of the carrier.
[4.3.II] The signal can be seen pseudo-static because transient processes are negligible.
[4.3.III] The signal intensity is the sum of emitted signals of every emitter.
[4.3.IV] Note that all equation in this section apply to the detector j. For simplicity, the index j is dropped.

In other words, an increased signal intensity causes a decreased voltage at the ADC.

Due to the saturation of the phototransistor, $I_C(t)$ has an upper limit[4.3.V], $I_{C,\infty}$. When considering the ambient light that is superpositioned on the signal, the collector current is denoted to

$$I_C(t) = \min\left\{I_{C,S}(t) + I_{C,0}(t),\, I_{C,\infty}\right\}, \tag{4.14}$$

where $I_{C,S}$ and $I_{C,0}$ are current components proportional to the signal and ambient light intensity, respectively. Due to the ambient light current, $I_{C,0}$, the voltage $V_{CE,\text{max}}(t) = V_{cc} - R\, I_{C,0}(t)$ represents the upper bound. Similarly, $V_{CE,\text{min}} = V_{cc} - R\, I_{C,\infty}$ is the lower bound.

The ADC performs a linear time- and value-discretisation of $V_{CE}(t) \in [V_{CE,\text{min}}, V_{CE,\text{max}}(t)]$. Let $\mathcal{M}_V : [0,\, V_{cc}] \rightarrow \mathbb{M}$ be a measure that maps $V_{CE}(t)$ to the measurement set $\mathbb{M} = \{0, 1, \ldots, m_{sup}\} \subset \mathbb{N}$, where $m_{sup} = 2^{n_{ADC}} - 1$ is determined by the ADC width (n_{ADC} bits). Considering the boundaries $V_{CE,\text{max}}$ and $V_{CE,\text{min}}(t)$, the upper and lower limit of measurements are

$$m_{\text{max}}[k] = \mathcal{M}_V(V_{CE,\text{max}}(t)), \tag{4.15}$$

$$m_{\text{min}} = \mathcal{M}_V(V_{CE,\text{min}}). \tag{4.16}$$

Note that $[\cdot]$ indicates time-discrete values based on $k = \left\lfloor \frac{t}{t_s} \right\rfloor$ and the sampling period, t_s.

Due to $y_j(t) \propto I_C(t)$, $\mathcal{M}_V$ can be adopted to $\mathcal{M} : [0, 1] \rightarrow \mathbb{M}$ measuring the signal intensity as

$$m_{\text{max}}[k] = m_{sup} - m_0 = \mathcal{M}(0) \tag{4.17}$$

$$m_{\text{min}} = \mathcal{M}(1), \tag{4.18}$$

$$m_j[k] = \mathcal{M}(y_j(t)), \tag{4.19}$$

$$= \lfloor (m_{\text{max}}[k] - m_{\text{min}})\ (1 - y_j(t)) + m_{\text{min}} \rfloor + n_m[k], \tag{4.20}$$

where $n_m[k]$ is a discretised zero-mean AWGN as defined in [Roy 2003].

By combining the robot-to-robot model and the measurement model, it is possible to fully describe the signals between two robots.

4.3.4 Parameter Identification

Before the model can be used, the structure of h_e, h_m, and h_d as well as the value of m_{min} and m_{max} need to be defined. Three experiments were conducted identifying $n_m[k]$ as well as m_{max}, $h_d(\beta)$ as well as m_{min}, and $h_m(d)$ as well as $h_e(\alpha)$.

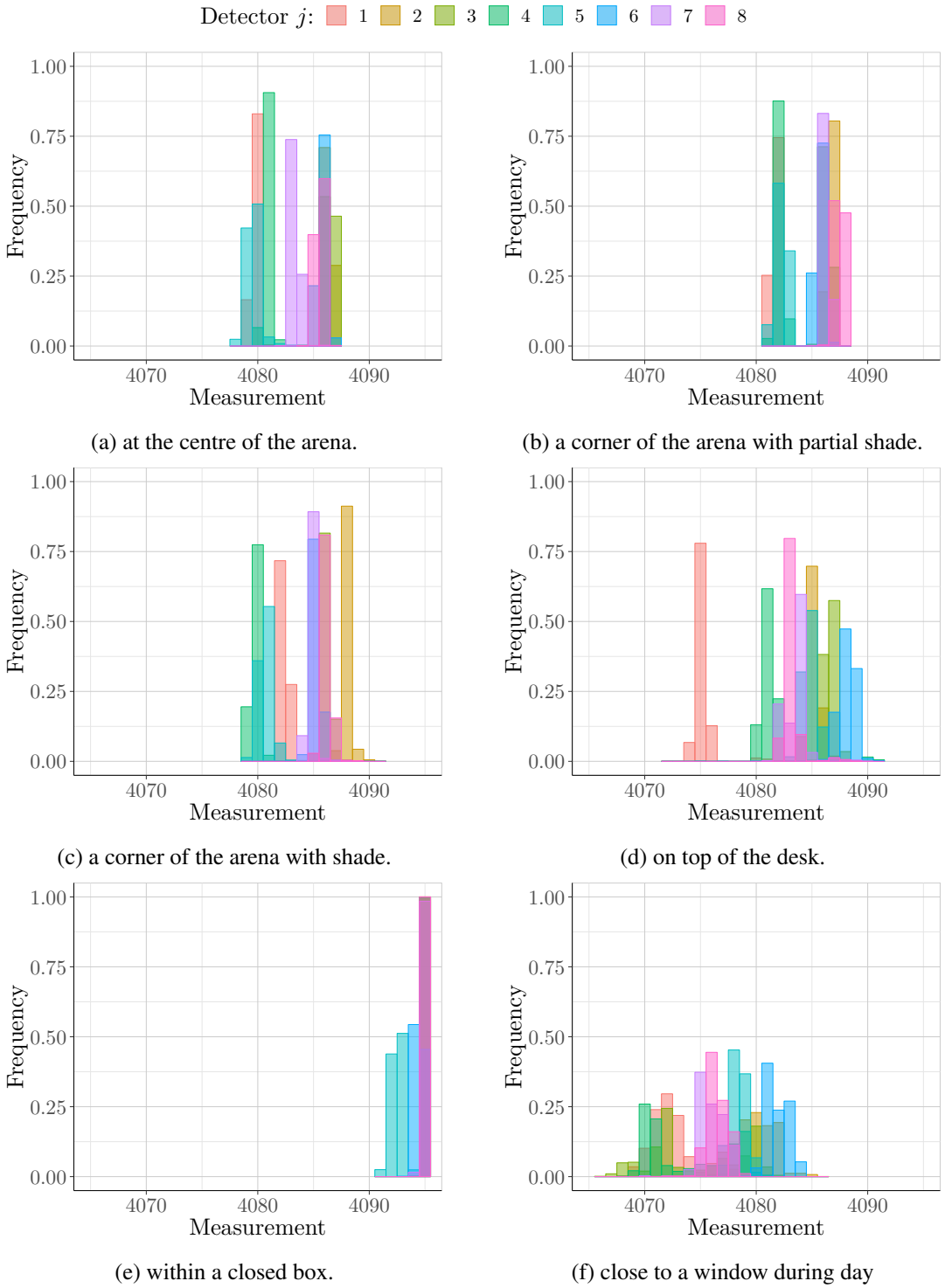
4.3.4.1 Experiments with a Single Robot in Ambient Light

First, the environmental impact on the robots is measured. As it cannot be determined generally, the following experiments determine the magnitude of $n_m[k]$ and m_{max} as well as their values for later experiments within the same environment.

In total, six experiments are conducted to characterise the magnitude of ambient light measurement, m_{max}. In the experiments, robots are placed at the centre of the robot arena[4.3.VI], in 2 different corners of the arena (one with shade and the other with partial shade), on the desk close to the arena, inside a closed box, and on a desk close to a window (during the day).

[4.3.V] The saturation current of the phototransistor is specific to that circuit; hence, it is considered time-invariant.

[4.3.VI] Note it is the same arena as described in Chapter 3. In addition, let the area be illuminated by six 1 m long fluorescent tubes. Two rows of three tubes are mounted in approximately 3 m high in parallel to the 4 m long walls. There are three windows around 2 m away from the arena's boundary.



(a) at the centre of the arena.

(b) a corner of the arena with partial shade.

(c) a corner of the arena with shade.

(d) on top of the desk.

(e) within a closed box.

(f) close to a window during day

Figure 4.4: Histograms of the measurements taken from each detector within different environments.

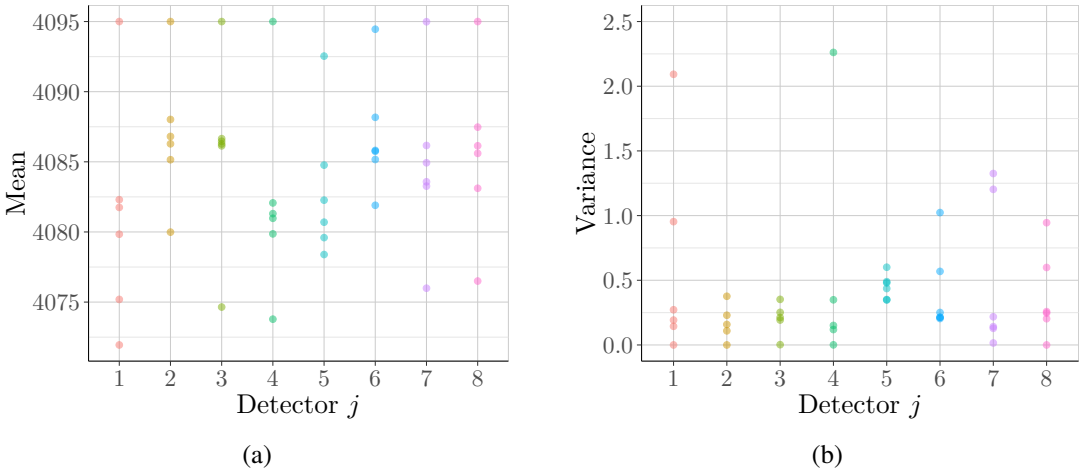


Figure 4.5: Sample (a) mean and (b) variance of more than 1500 environment measurements for each detector in six different configurations.

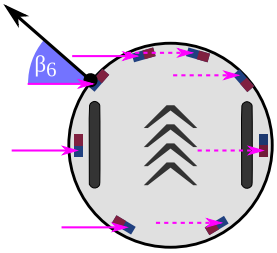


Figure 4.6: Setup of the experiments with a light source, where the light (pink).

For each experiment, more than 1500 measurements were taken by each detector — illustrated in Figure 4.4. Each measurement, $m[k]$, detects ambient light as $m_{\text{max}}[k] + n_m[k]$. As the sample mean is an unbiased estimator of the true mean, it estimates $m_{\text{max}}[k]$ for each detector in each setup. Similarly, the sample variance is a sufficient estimator for σ^2 of $n_m[k]$, for any $\sigma > 1$ [Roy 2003] and, hence, used.

Based on the measurements, the sample variance and mean were calculated for each detector and shown in Figures 4.5. In Figure 4.5a, it can be seen that m_{max} varies from 4095 (i.e., maximum measurable value, m_{sup}, indicating that no light was detected) from within a closed box to values around 4075 that were obtained in front of an open window. Overall, m_{max} can be approximated to 4080 (i.e., the ambient light offset, $m_0 = m_{sup} - m_{\text{max}}$ is 15) within the arena; hence, m_0 is set to 15 in further experiments. Based on Figure 4.5b, it can be seen that the variances do not exceed 2.5. Hence, the noise $n_m[k]$ is modelled as (worst-case) discrete Gaussian process with the variance of 2.5, which is also used in the subsequent experiments.

4.3.4.2 Experiments with a Single Robot and a Single Light Source

The next step is to characterise the saturation value, m_{min}, and the directionality of a detector, $h_d(\beta)$. This is achieved by performing experiments with one robot and a light source (10 W LED floodlight). The floodlight is placed at the height of the robot's detector and projects light parallel to the surface. The light's width is larger than the robot's diameter; hence, the light is assumed to be homogeneous and planar. In other words, the light reaches the robot's detectors from the same angle, as shown in Figure 4.6.

The saturation is measured by placing the robot 2 cm in front of the light source. Eight orientations are examined in which each detector is once oriented towards the light source. As the detector that is oriented towards the light saturates, $m_{\text{min}} + n_m[k]$ is measured. In each

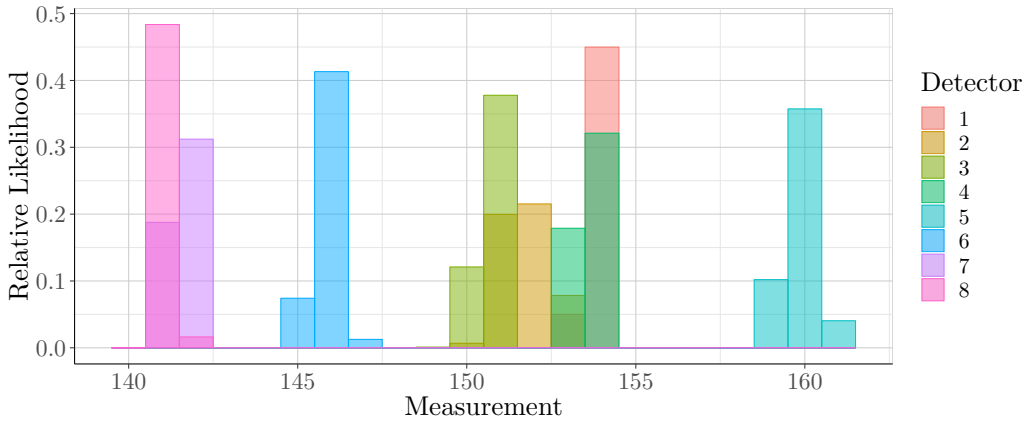


Figure 4.7: Histogram of the measured saturation values from the detectors closest to the light source.

Table 4.1: Sample mean (i.e., the saturation value, m_{min}) and variances obtained from measurements with a light source.

j	m_{min}	s^2
1	153.94	0.0608
2	152.52	1.0158
3	151.32	0.4140
4	153.88	0.2003
5	159.68	0.3858
6	146.37	0.3746
7	141.83	0.1604
8	141.11	0.1072

case, more than 2000 measurements per detector were collected and plotted in Figure 4.7. The sample mean (i.e., m_{min}) and variance (i.e., σ^2) were calculated and are displayed in Table 4.1. Over all sensors, m_{min} can be approximated to 149.80 (i.e., 150 on the robot).

The signal attenuation based on the inclination angle, β, is measured by placing the robot 10 cm from the light source. The robot is oriented with $\theta_k \in \{0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}\}$ towards the light source. Based on θ_k and the planar light, each detector has a different inclination angle similar to Figure 4.6. The inclination angle for a detector, j, is

$$\beta_{j,k} = \theta_j - \theta_k, \tag{4.21}$$

in each setup k.

At each orientation θ_k, more than 2000 measurements were taken for each detector and are shown in Figure 4.8. Similar to before, the true measurement is estimated by the sample mean, $\bar{m}_{j,k}$, and transformed into the estimated signal intensity $y_{j,k} = \mathcal{M}^{-1}(\bar{m}_{j,k})$. The detector attenuation, h_d, is the relation between $\beta_{j,k}$ and $y_{j,k}$. It is approximated by

$$h_d(\beta) = |\cos(\beta)^3|, \tag{4.22}$$

as shown in Figure 4.9.

4.3.4.3 Experiments with Two Robots

The emitter, h_e, and medium attenuation, h_m, are identified in a final experiment, where two robots are placed in the centre of the arena. Both robots are oriented in such a way that Emitter

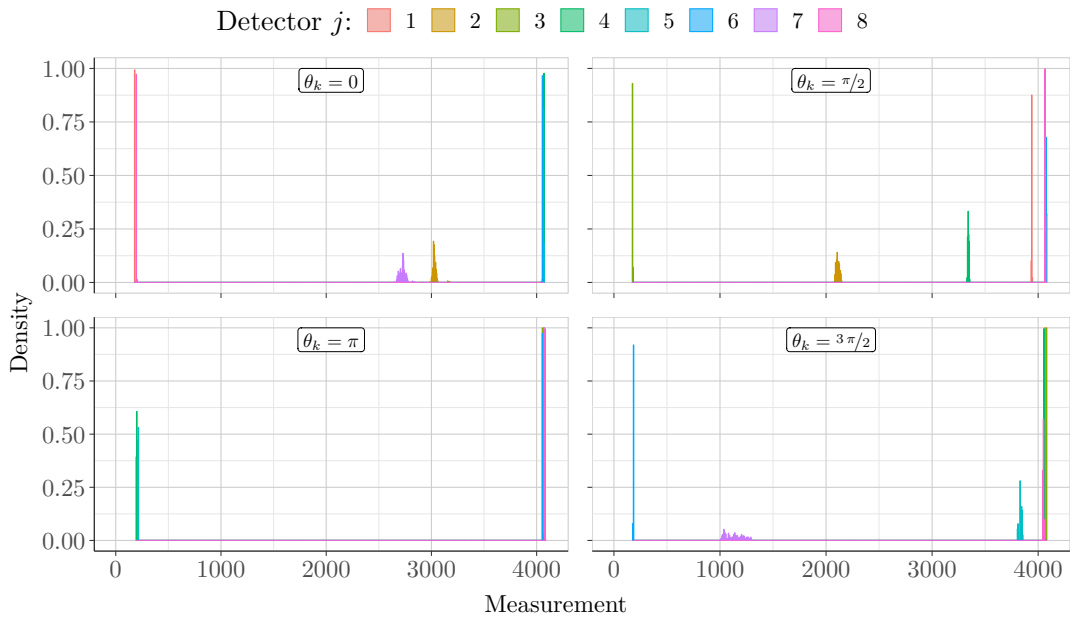


Figure 4.8: Histograms of the sensor values obtained when the robot was located 10 cm away from with orientation, θ_k, relative to the light source.

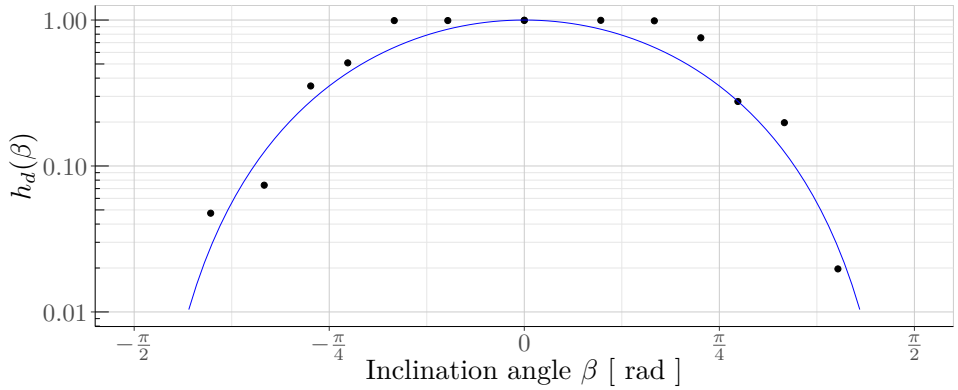


Figure 4.9: Detector attenuation, $h_d(\alpha)$. The mean values of the obtained signal intensity are plotted in relation to the inclination angle β (dots). These values are approximated by $h_d(\beta) = cos^3(\beta)$ (line). Note that inclinations angles $|\alpha| > \frac{\pi}{2}$ did not produce data points due to self-occlusion.

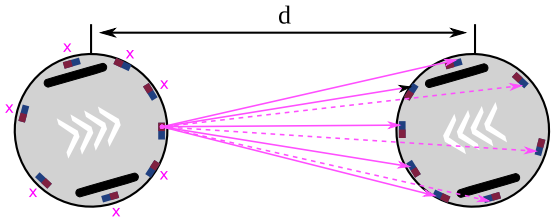


Figure 4.10: Setup of the experiment with two robots. Note that only one emitter is transmitting the signal, while all detectors of the other robot are measuring.

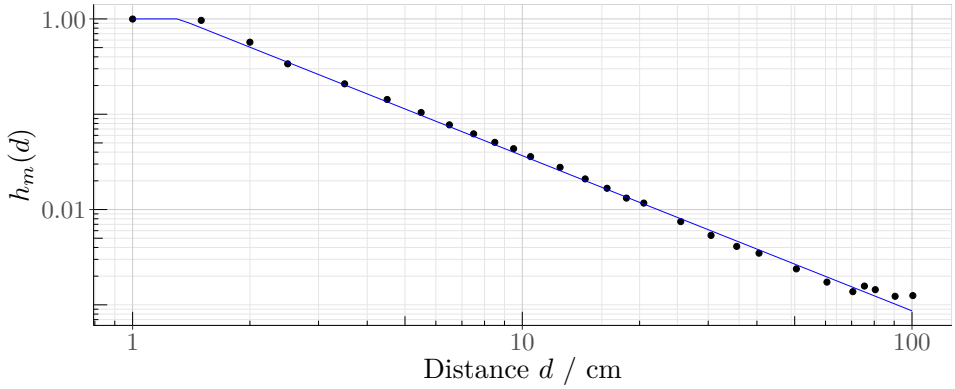


Figure 4.11: Signal intensities obtained for Detector 1 in relation to distance. The blue line indicates an approximation with $o_m \left(d_{ed}\right)^{k_m}$.

1 and Detector 1 of each robot faces the other (i.e., $\alpha_1 = \beta_1 = 0$), as shown in Figure 4.10. Initially, the robots are placed next to each other[4.3.VII]. While the emitting robot sends a constant signal with Emitter 1, the receiving robot detects it with all detectors. In total, the receiving robot takes 450 measurements and, after that, increases its distance to the emitting robot. This procedure is repeated 28 times and measurements are taken at the inter-robot distances of {7, 7.5, 8, 8.5, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 21, 23, 25, 27, 32, 37, 42, 47, 57, 67, 77, 87, 97, 107} cm.

The medium attenuation, $h_m(d)$, is estimated for measurements of Detector 1. Due to the alignment, $\alpha_1 = \beta_1 = 0$, the change in signal intensity can only result from the inter robot distance d. Similar to before, the true intensity is estimated by the transformed sample mean $y_{j,k} = \mathcal{M}^{-1}\left(\bar{m}_{j,k}\right)$. When plotting the signal strength against the distance, as shown in Figure 4.11, it can be seen that the medium attenuation can be approximated by

$$\widetilde{h}_m(d) = o_m \left(d\right)^{k_m} \tag{4.23}$$

$$h_m(d) = \begin{cases} 1 & \lim\limits_{d' \to d} \widetilde{h}_m(d') > 1 \\ \widetilde{h}_m(d) & \text{otherwise} \end{cases}, \tag{4.24}$$

with $k_m = -1.54557$ and $o_m = 1.12202$ (blue line).

After estimating the detector and medium attenuation, the emitter attenuation, $h_e(\alpha)$ can be estimated by using measurements of the other detectors (i.e., $j \neq 1$). As the emitter sends a normalised signal equal to 1, the emitter attenuation for each detector, j, in each setup, k, is

[4.3.VII] Note that the inter-robot distance is measured from and to a robot's centre. This results in a minimum of 7 cm (diameter of the robot).

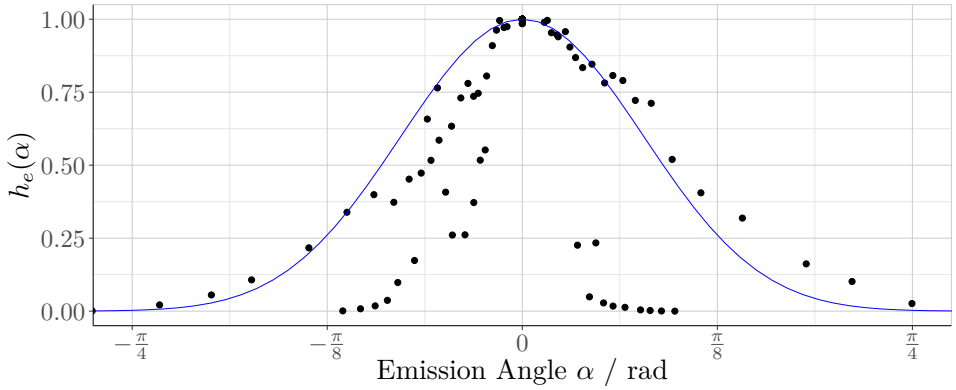


Figure 4.12: Signal intensity in relation to the emitting angle α. The trend is approximated by $h_e(\alpha) = \cos(\alpha)^7$ (blue line). Note that data points diverging from the trend are often values on the boundaries of $h_m(d)$ and $h_d(\beta)$ (e.g., in the range of $\beta \approx \pm\frac{\pi}{2}$).

Table 4.2: Model parameters

Parameter	Value	Description
n_{ADC}	12	Bit length for conversion
k_m	-1.54557	Medium attenuation decay
o_m	1.12202	Medium attenuation offset
m_{max}	4080	Ambient light offset
m_{min}	140	Saturation measurement

obtained by

$$y_{j,k} = \mathcal{M}^{-1}(\bar{m}_{j,k}) = h_e(\alpha_{j,k})\, h_m(d_{j,k})\, h_d(\beta_{j,k}) \cdot 1, \tag{4.25}$$

$$h_e(\alpha_{j,k}) = \frac{\mathcal{M}^{-1}(\bar{m}_{j,k})}{h_m(d_{j,k})\, h_d(\beta_{j,k})}. \tag{4.26}$$

For each distance, the emission angle and (4.26) were calculated and plotted in Figure 4.12. As shown in Figure 4.12 (blue line), the emitter attenuation function is approximated by

$$h_e(\alpha) = |\cos(\alpha)^7|. \tag{4.27}$$

4.3.5 Channel Model

By combining the previous sections, the full channel model is

$$y_j^\star(t) = \min\left\{\sum_i |\cos(\alpha_{i,j})^7 \ \min\{o_m\,(d_{i,j})^{k_m}, 1\}\ \cos(\beta_{i,j})^3|, 1\right\} s(t) \tag{4.28}$$

$$m_j[k] = \left\lfloor (m_{\text{max}}[k] - m_{\text{min}})\left(1 - y_j^\star(t)\right) + m_{\text{min}}\right\rfloor + n_m[k], \tag{4.29}$$

where $y_j^\star(t)$ is the noise-free intensity based on a sent symbol, $s(t) \in \{0, 1\}$. Its parameters are listed in Table 4.2. Furthermore, the emitter and detector directionality are shown in a radiation pattern plot illustrated in Figure 4.13.

4.3.6 Model Evaluation

Finally, the accuracy of the model is evaluated by performing an additional experiment. Two e-pucks were placed next to and oriented towards each other, where the inter-robot distance, d,

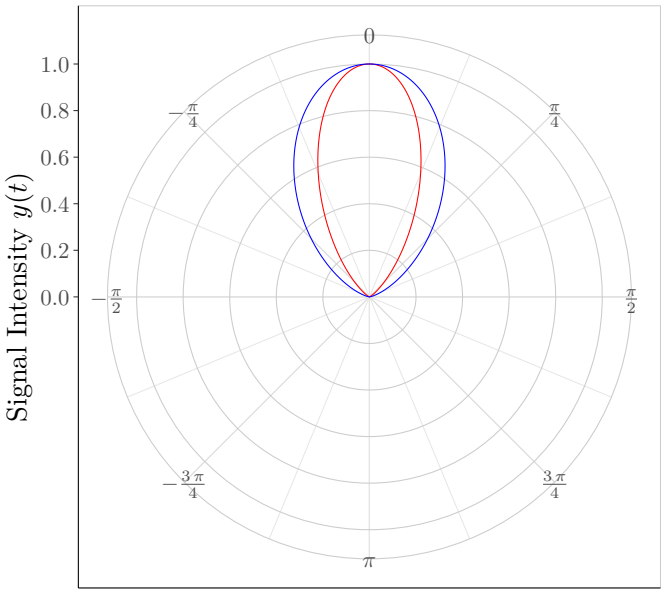


Figure 4.13: Radiation pattern of an emitter (red) and a detector (blue).

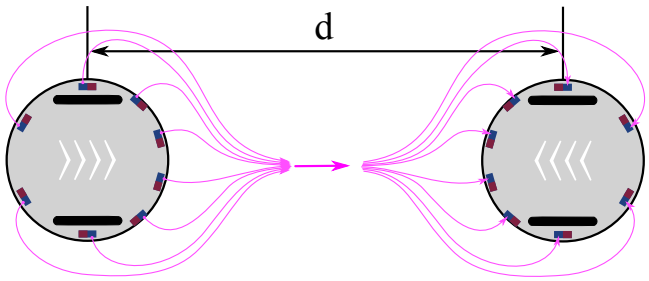


Figure 4.14: Setup of the experiments to evaluate the communication channel model. The transmitting robot (left) sends a signal with all emitters to all detectors of the receiving robot (right).

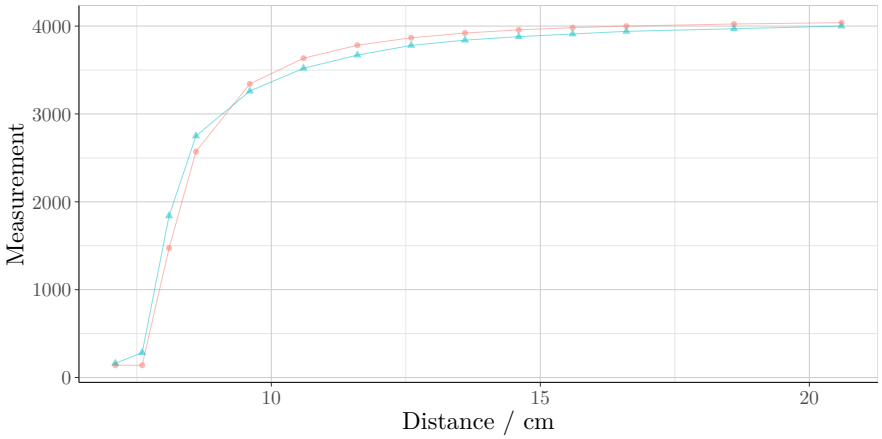
is chosen from $\{7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 22\}$ cm. At each location, one robot transmits a signal with all emitters, and the other receives it with all detectors (see Figure 4.14). The receiving robot sends all measurements to a computer via Bluetooth. After obtaining 250 measurements, the receiving robot is then moved to the next location.

In Figure 4.15, the model predictions are compared to the measurements from two e-puck robots. Similar to previous experiments, the sample mean estimates the true value for each distance. It can be seen that the predictions closely match the observed values.

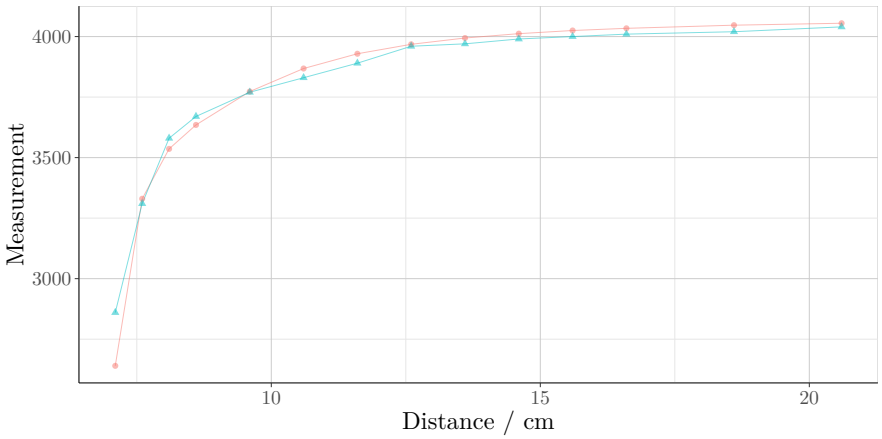
4.3.7 Discussion

The presented model shows how optical signals are emitted and received first between a pair of infra-red components and, after that, between e-puck robots. As the emitter-to-detector-model is specific to the used components (i.e., TCRT1000), it could be used for devices using that component. Furthermore, the robot-to-robot model is designed for the e-puck; however, it can also be applied to the e-puck 2[4.3.VIII] (release 2018), which uses the same components, circuits and sensor placements as the e-puck.

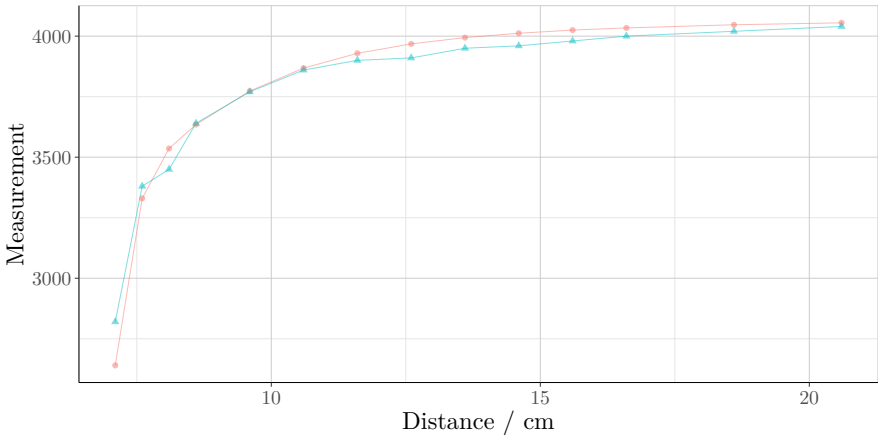
[4.3.VIII] Information was obtained through mail exchange with GCtronic (http://www.gctronic.com/); schematics of the e-puck 2 have not been released at the time of writing this work.



(a) Detector 1



(b) Detector 2



(c) Detector 7

Figure 4.15: Model predictions (red) in comparison to the measured results (blue).

Overall, it was shown that the predictions closely match the measurements. As a result, this model and the presented parameters are used in the subsequent sections.

4.4 SwarmCom: A MANET for Severely-Constrained Robots

After describing how signals behave, this section discusses the design of SwarmCom, a MANET for severely-constrained robots. With SwarmCom, a group of robots establishes a peer-to-peer mesh network dynamically. To create SwarmCom, multiple methods need defining:

- how data is transmitted (i.e., *modulation*),
- how data is received (i.e., *demodulation*),
- what data is transmitted to detect/remove transmission errors (i.e., *channel coding*), and
- how devices can access the network (i.e., *medium access control*).

4.4.1 Modulation

The method that defines how information is applied to the channel is called modulation. Many MANET technologies (e.g., Ethernet, Wifi, and Bluetooth) use sophisticated methods such as phase-/frequency-shift-keying or quadrature amplitude modulation [IEEE 2017]. However, these methods rely on matching and filtering waveforms, which is challenging in optical systems due to their high frequency.

In contrast, optical systems commonly measure signal intensities and not waveforms; hence, the modulation is limited amplitude-shift keying — in particular, on-off-keying (OOK). In OOK, information is represented as a sequence of two logical symbols, s_1 and s_0. Each symbol is modulated as emission with full intensity and the absence of any emission, respectively. As a result, this modulation method is susceptible to noise and ambient light changes, which increases the difficulty of correct demodulation/detection.

4.4.2 Demodulation

Demodulation is a method of detecting the sequence of symbols that have been transmitted while minimising the detection errors[4.4.1]. To detect a symbol within the sequence, the maximum likelihood decision rule, $\Upsilon(m) : \mathbb{M} \to \{s_0, s_1\}$, is used. It is defined as

$$\Upsilon(m) = \begin{cases} s_0, & \text{if } P(S{=}s_0|M{=}m) \geq P(S{=}s_1|M{=}m) \\ s_1, & \text{otherwise} \end{cases}, \tag{4.30}$$

where $P(S = s|M = m)$ denotes the likelihood of having received symbol $s \in \{s_0, s_1\}$ given measurement m. As this is an optimal decision rule, its accuracy solely depends on $P(S = s|M = m)$. As a result, this section investigates how this decision rule can be derived.

4.4.2.1 Assumptions

To calculate $P(S = s|M = m)$, assumptions about transmitted information, location, and orientations are required.

Symbol Probability

First, it is assumed that any sequence of symbols is equally likely. As a result, the a-priori probability of transmitting any symbol is

$$P\left(S = s_0\right) = P\left(S = s_1\right) = \frac{1}{2}. \tag{4.31}$$

[4.4.1] A detection error occurs if a symbol (e.g., s_0) is detected even though a different symbol has been sent (e.g., s_1).

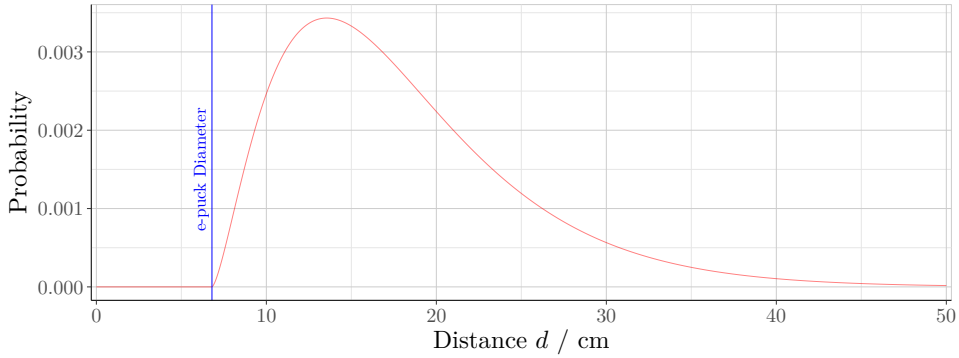


Figure 4.16: Probability density function of the inter-robot distance D. As objects cannot overlap, the minimal distance is the diameter of one robot.

In other words, the transmitted information contains no redundancy (i.e., maximum entropy) [Lapidoth 2009].

Location and Orientation Probability

Let the environment be unbound, and the transmitting robot be located at the origin of the coordinate system oriented towards the x-axis. As the relative location of the receiving robot is not known, it is represented as a random variable, $L = (D, \Theta)$, in polar coordinates. The distance random variable, D, is chosen from a gamma distribution [Forbes et al. 2010],

$$D \sim \mathcal{G}(k, \varphi), \tag{4.32}$$

with shape parameter, $k = 0.045$, and scale parameter, $\varphi = 2.5$, as shown in Figure 4.16.

The polar angle Θ is chosen from a uniform distribution with a probability density function $p(\theta) = \frac{1}{2\,\pi}$ for all $\theta \in (-\pi, \pi]$. Similarly, the receiver robot's orientation, O, is chosen from a uniform distribution with probability density function, $p(o) = \frac{1}{2\,\pi}$ for all $o \in (-\pi, \pi]$.

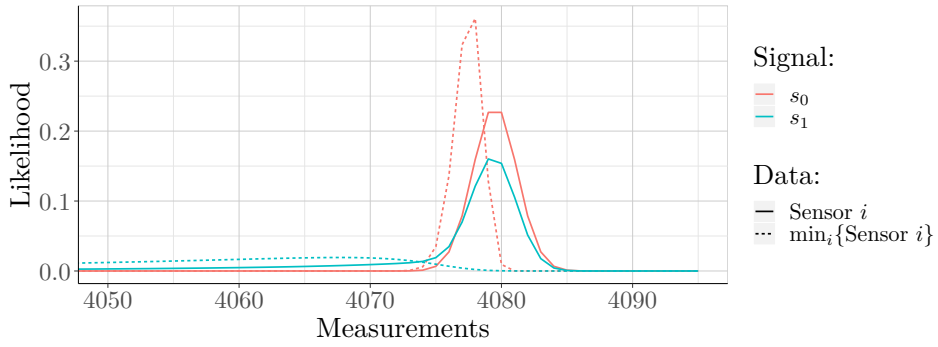
4.4.2.2 Signal Probability

During a transmission, the probability, $P\,(M = m|S = s)$, at the receiver defines the likelihood of obtaining the measurement m, when a symbol s has been sent. For s_0, $P\,(M = m|S = s_0)$ follows the distribution of $n_m[k]$. In contrast, $P\,(M = m|S = s_1)$ depends on multiple factors (e.g., inter-robot distance or orientation).

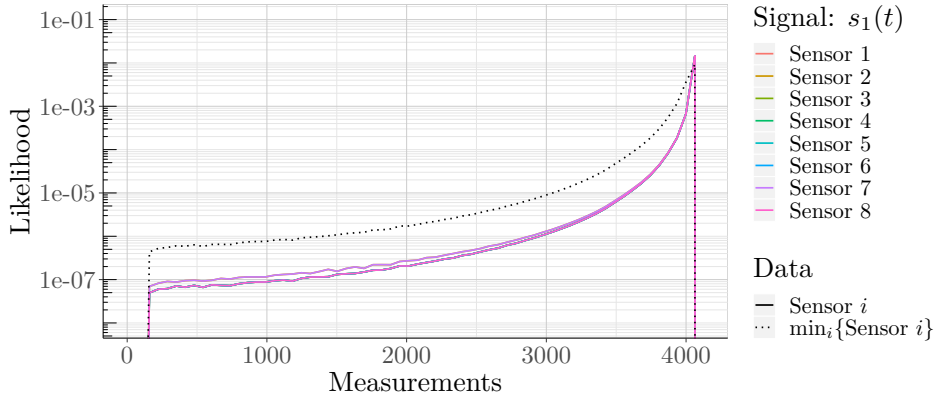
To obtain $P\,(M = m|S = s_1)$, a Monte Carlo-like[4.4.II] approach was applied, where all permutations of each 1000-quantile of D, 720-quantile of Θ, and 720-quantile of O are applied to the channel model of Section 4.3. This resulted in $5.184 \cdot 10^8$ observations for each sensor. Considering the statistical law of large numbers, the relative frequencies are used to approximate the theoretical probability of a noiseless channel. Then, it is convolved with the distribution of $n_m[k]$ to integrate noise. This results in the probability density function of $P\,(M = m|S = s_1)$, as shown in Figure 4.17.

When comparing the signal and the noise distribution (see Figure 4.17), it can be seen that the majority of measurements are similarly distributed (i.e., around m_{max} — 4080). As a result, errors during detection would be likely. One way to minimise these errors is to further limit the communication range, creating higher intensities. Alternatively, $\min\{\cdot\}$ can be applied to use all information available to the detectors. Taking the maximum signal for the decision making improves the signal distribution, as shown in Figure 4.17. While $\min\{\cdot\}$ reduces the number of potentially parallel communication channels, it does not reduce the communication range and has a small computational overhead. Hence, it is used in this work.

[4.4.II] A large number of data points are used to approximate the probability by applying the law of large numbers.



(a)



(b)

Figure 4.17: Relative frequency of measurements (solid) for each detector and the minimum (dotted) over all detectors.

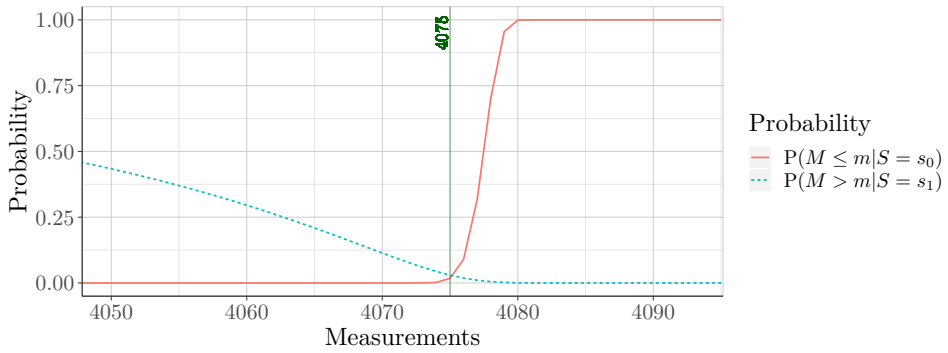


Figure 4.18: Detection threshold, m_t, and detection error probability.

4.4.2.3 Decision Rule

When applying the signal probability to (4.30), computationally expensive probability calculations would be conducted for each decision. To avoid this, a threshold can be used to decide which symbol is more likely. It reduces the computational requirements to compare values, as shown by

$$\Upsilon(m) = \begin{cases} s_0 & \text{if } m > m_t \\ s_1 & \text{otherwise} \end{cases}, \tag{4.33}$$

where m_t is the *decision threshold*. It is calculated by

$$m_t = \max_{m \in \mathbb{M}} \{m \,|\, P(S{=}s_1|M{\leq}m) \leq P(S{=}s_0|M{>}m)\}\,. \tag{4.34}$$

As $P(S{=}s_0|M{=}m)$ is not know, the Bayes' theorem is applied to (4.31) and (4.34) creating

$$m_t = \max_{m \in \mathbb{M}} \{m \,|\, P(M{\leq}m|S{=}s_1) \leq P(M{>}m|S{=}s_0)\}\,. \tag{4.35}$$

When applying (4.35) to the data of Figure 4.17, the decision threshold is calculated as $m_t = 4075$, as shown in Figure 4.18.

4.4.2.4 Evaluation

To validate the described detection (i.e., decision rule), two e-pucks, one transmitting and one receiving, are placed on a line. While the transmitting robot is oriented towards the receiving robot, the latter operates in four modes:

(I) static receiver robot with no LEDs illuminated,
(II) rotating receiver robot with no LEDs illuminated,
(III) static receiver robot with illuminated LEDs, and
(IV) rotating receiver robot with illuminated LEDs.

This makes it possible determine the influence of the motors and illuminated LEDs on the detection process. Note that the receiving robot is oriented towards the other when it is not rotating. The inter-robot distance is chosen from $d \in \{7, 12, 17, 22, 27, 32, 37, 42, 47, 52, 57\}$.

In each configuration, one symbol — s_0 or s_1 — is transmitted continuously until $2.5 \cdot 10^5$ decisions are performed. After that, the other symbol is transmitted. Because s_0 is the absence of an emitted signal, the detection error probabilities for s_0 are independent of the inter-robot distance and are shown in Table 4.3.

Figure 4.19 shows the rate of misdetection of s_1 in relation to the inter-robot distance, d. The detection error probability tends to be negligible for communication distances $\leq$ 32

Table 4.3: Average detection error probabilities for s_0 at each configuration.

Configuration	Error Probability
(I)	$5.8015\,10^{-7}$
(II)	$3.2963\,10^{-4}$
(III)	$3.8040\,10^{-4}$
(IV)	$6.8205\,10^{-4}$

cm. While a static robot not using its LEDs provide a low error probability up to 40 cm, the use of motors and LEDs increases the error probability significantly. This potentially stems from additional noise and reduction in the supply voltage, V_{cc}. The model estimates higher error probabilities than measured due to larger assumed noise with variance of 2.5 instead of (measured) 0.6657.

4.4.2.5 Discussion

In this section, a detection method was derived from the signal distribution between two robots. With the presented method, it is possible to decide between the two transmitted symbols — s_0 and s_1 — while requiring a small computational overhead. While data can be detected with a low error probability (i.e., most are even error-free), the error probability increases with distance — in particularly above 37 cm. Furthermore, the data has shown that using the robot's motor and LEDs has a negative impact on the detection. As a result, errors should be expected in experiments with mobile robots.

4.4.3 Channel Coding

As data is often transmitted in blocks, errors during the detection can change transmitted data, which could cause unpredictable or malicious behaviour. To prevent this, channel coding — a method to detect and correct errors — can be applied to improve the likelihood of receiving correct data. To ensure communication on severely-constrained robots, such as the e-puck, the encoding and decoding must be done in a small number of steps with a small memory footprint.

A fast encoding method based on vector-matrix arithmetic are binary-BCH codes [Yin et al. 2013]. A BCH codeword, $\boldsymbol{c} \in \mathbb{S}^n$, is a vector of n symbols of $\mathbb{S} = \{s_0, s_1\}$. The codeword is constructed by

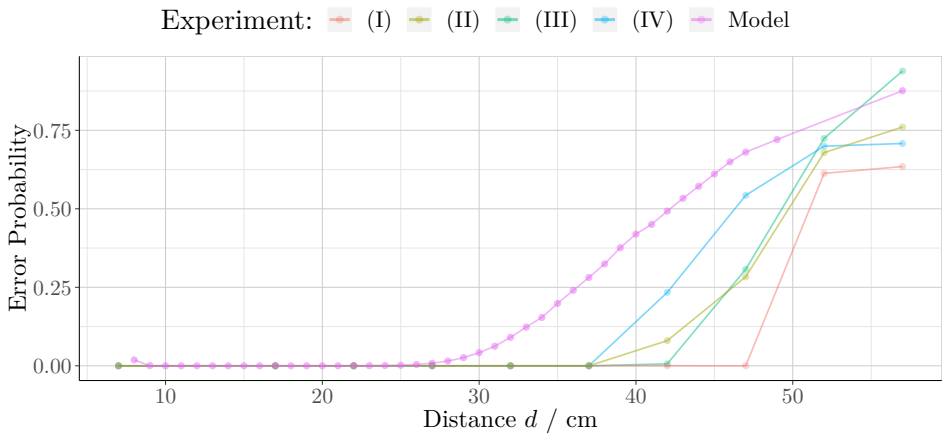
$$\boldsymbol{c} = \boldsymbol{d}\,\mathbf{G}, \tag{4.36}$$

where $\boldsymbol{d} \in \mathbb{S}^k$ and $\mathbf{G} \in \mathbb{S}^{k\times n}$ are the transmitted data and the generator matrix, respectively. Note that, in this work, $\boldsymbol{c}$ is systematic, where the codeword $\boldsymbol{c} = (\boldsymbol{d}, \boldsymbol{p})$ is composed of the data, $\boldsymbol{d}$, and a parity vector, $\boldsymbol{p}$. This allows a fast generation of codewords as only $\boldsymbol{p}$ needs to be calculated. When $\boldsymbol{c}$ is sent, errors can occur, and the received codeword is

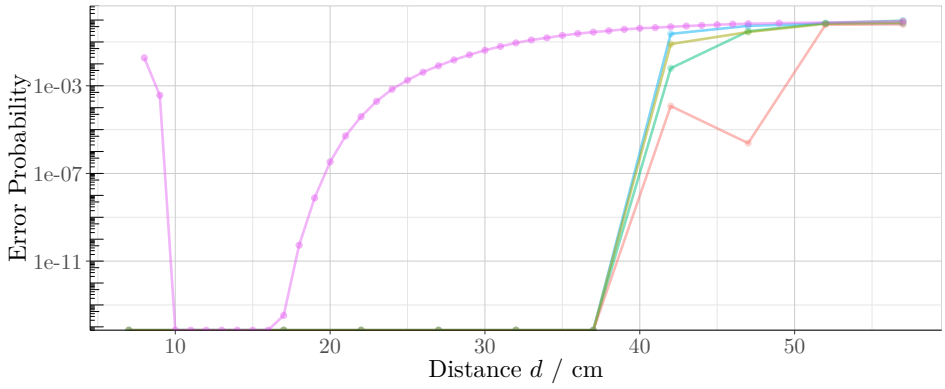
$$\boldsymbol{r} = \boldsymbol{c} \oplus \boldsymbol{e}, \tag{4.37}$$

where $\boldsymbol{e} \in \mathbb{S}^n$ is the error vector, and $\oplus(\cdot,\cdot)$ is a symbol-wise modulo-2 addition. After obtaining $\boldsymbol{r}$, the data is extracted with the decoder.

Due to the severe computational constraints, methods which have high computational costs (e.g., [Lapidoth 2009; Orhan et al. 2014; Arafa et al. 2017]) are often not practical. In this work, syndrome decoding is used as it is based on bitwise vector-matrix multiplication.



(a)



(b)

Figure 4.19: Misdetection probability for s_1 in relation to the inter-robot distance.

To decode $\boldsymbol{d}$ from $\boldsymbol{r}$, first, the parity matrix[4.4.III], $\mathbf{H}$, is used to detect errors as

$$\mathbf{0} = \mathbf{H}\,\boldsymbol{c}, \tag{4.38}$$

$$\boldsymbol{s} = \mathbf{H}\,\boldsymbol{r} = \mathbf{H}\,\boldsymbol{c} \oplus \mathbf{H}\,\boldsymbol{e} = \mathbf{H}\,\boldsymbol{e}, \tag{4.39}$$

where the syndrom $\boldsymbol{s} \neq \mathbf{0}$ if an error occurred (i.e., $\boldsymbol{e} \neq \mathbf{0}$). Based on $\mathbf{H}\,\boldsymbol{e}$, a translation table is created to link $\boldsymbol{s}$ to a specific $\boldsymbol{e}$. Finally, the error-free codeword is calculated by

$$\boldsymbol{c} = \boldsymbol{r} \oplus \boldsymbol{e}. \tag{4.40}$$

In this work, three channel codes are used — to correct up to 7 errors (i.e., repetition code, [1, 15]), to correct up to 1 error (i.e., [11, 15]), and not to correct any errors for best throughput (i.e., no encoding, [15, 15]). Each $[n, k]$ code indicates that n bits of data are transmitted as k bits blocks (i.e., $n \leq k$). The codeword length, k, is chosen based on the architecture of the robot (i.e., 16 bit for an e-puck robot). This allows fast processing as a single row of $\mathbf{G}$ and $\mathbf{H}$ can be computed in one operation. For any data that is longer than n bits, the data is split into n bit blocks and, after encoding, sequentially transmitted.

The [1, 15] code can be generated by repeating a single bit 15 times and decoded by majority decision. The [11, 15] code uses the generator polynomial, $g(x) = 1 + x + x^4$, to obtain

[4.4.III] The parity matrix, $\mathbf{H}$, can be derived from $\mathbf{G}$ as described in [Lapidoth 2009].

the generator and parity matrix

$$\boldsymbol{G} = \begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&1&1&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&1&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&1&1 \\ 0&0&0&1&0&0&0&0&0&0&0&1&1&0&1 \\ 0&0&0&0&1&0&0&0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&1&0&0&0&0&1&1&1&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0&1&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&0&1&0&1&0&1&1 \\ 0&0&0&0&0&0&0&0&0&0&1&1&0&0&1 \end{bmatrix}, \tag{4.41}$$

$$\boldsymbol{H} = \begin{bmatrix} 1&0&0&1&1&0&1&0&1&1&1&1&0&0&0 \\ 1&1&0&1&0&1&1&1&1&0&0&0&1&0&0 \\ 0&1&1&0&1&0&1&1&1&1&0&0&0&1&0 \\ 0&0&1&1&0&1&0&1&1&1&1&0&0&0&1 \end{bmatrix}, \tag{4.42}$$

as described in [Glover and Grant 2010]. The syndrome translation table corresponds to

$$\overbrace{\begin{bmatrix} 0&0&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \\ 0&0&1&1 \\ 0&1&0&0 \\ 0&1&0&1 \\ 0&1&1&0 \\ 0&1&1&1 \\ 1&0&0&0 \\ 1&0&0&1 \\ 1&0&1&0 \\ 1&0&1&1 \\ 1&1&0&0 \\ 1&1&0&1 \\ 1&1&1&0 \\ 1&1&1&1 \end{bmatrix}}^{\text{Syndromes, } \boldsymbol{s}} \rightarrow \overbrace{\begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \end{bmatrix}}^{\text{Error Vector, } \boldsymbol{e}}. \tag{4.43}$$

Overall, all three channel codes are available by default. As can be expected, not correcting any errors has the smallest computational overhead followed by correcting up to 7 errors and, then, correcting up to 1 error. In contrast, the throughput is most reduced by the repetition code followed by the [11, 15]-code to a $\frac{1}{15}$ and a $\frac{11}{15}$ of the $[15, 15]$ code, respectively. However, the impact of used channel code on the communication system is explored in Section 4.5.

The limitation of the chosen decoding method is the rapid growth of the syndrome table with increased codeword length. Consequently, codewords need to be short to allow efficient decoding. Considering that robots are mobile and can frequently change topology, short codewords benefit the system as they are more likely to transmit data fully.

4.4.4 Medium Access Control

Before data can be transmitted, the moment when the medium is accessed, and data is sent is defined by medium access control [Guerroumi et al. 2014; Hussain et al. 2017]. As multiple robots can transmit at the same time, medium access control prevents the interference of messages and improves throughput.

In MANETs, two forms of medium access control are common [Sesay et al. 2004]—controlled and random access.

- Controlled access divides the medium based on its physical properties (e.g., frequency or time) [Hadded et al. 2015; Glisic and Leppänen 2013]. As every transmission has a dedicated time or frequency slot, this method allows fair and efficient access, in particular, for high network loads like streaming. However, with the large numbers of robots in swarms, dividing the medium into equal slots is often not feasible. Furthermore, when sequentially used, a large numbers of slots can also cause long delays. Moreover, allowing robots to join/leave requires the adding/removing of slots, which is often performed by a central managing unit. However, central infrastructures oppose design principles of many swarms [Hamann 2018].
- Random medium access allows the competing for access. This competition is performed through local information; hence, it is decentralised and can be used independently of the

Table 4.4: Robotic infra-red communication systems.

System	Modulation	Error		Access
		Detection	Correction	Control
Colias	OOK (38 kHz)[a]	✖	✖	✖
libIrcom[b] (e-puck)	OOK[c]	CRC[d]	✖	CSMA
Kilobot	OOK	IBM-CRC-16	✖	CSMA
r-one	OOK (38 kHz)[a]	CRC-16-CCITT	✖	✖
SwarmCom (e-puck)	OOK	BCH[e]	✔[e]	CSMA

[a] Colias and r-one modulate the amplitude (OOK) of a 38 kHz carrier signal. This technology is common on infra-red remote controls for devices such as TVs.
[b] LibIrcom is a communication system designed for the e-puck robot. While being designed by the same group that designed the e-puck, it is an additional library and separately maintained.
[c] While libIrcom documentation describes the modulation through frequency modulation, it is an OOK method that uses two 32-bit codewords to encode a single bit.
[d] LibIrcom uses a self-defined 2-bit long CRC checksum.
[e] SwarmCom has three configurations in which it uses repetition code to correct up to 7 errors, a [11,15] BCH code to correct up to 1 error, and uncoded transmission to maximise throughput.

number of robots. However, high network loads can reduce performance as the number of collisions increases.

Due to infrequent communication within many swarm robotics systems, SwarmCom utilises random medium access called carrier sense multiple access (CSMA) [Shi et al. 2013; Wang et al. 2017]. In short, a robot only transmits a pending message, if no other robot is transmitting or waits until the transmission has ended. Consequently, CSMA is distributed as the robot uses only local information and is scalable as the detection solely depends on the network load and not on the number of robots listening. Overall, these properties match the requirements of swarm robotics and make CSMA suitable for SwarmCom.

4.4.5 Implementation

SwarmCom has been implemented in C on e-pucks running OpenSwarm. The source code is available under an open-source license.

For detection, SwarmCom uses time-multiplexed on-chip ADCs to sample the detector voltages. Each sample is used for the detection of a bit. Also, the detector determines the ambient light value, $m_{\max}$, during the start-up of the robot, and considers ambient light changes during run-time, as detailed in the following sections.

4.4.6 Discussion

In addition to SwarmCom, there are other infra-red communication systems used in swarm robotics — Colias [Arvin et al. 2014], e-puck [Gutiérrez et al. 2009*b*], I-Swarm [Seyfried et al. 2005], Khepera [Mondada et al. 1999], Kilobot [Rubenstein et al. 2012], r-one [McLurkin et al. 2014], and s-bot [Mondada et al. 2005]. Unfortunately, many of these communication systems are poorly documented and details are not published; this applies in particular to I-Swarm, Khepera, and s-bot and, therefore, they cannot be discussed here.

SwarmCom is compared to libIrcom (e-puck) as well as the communication systems of the Colias, r-one, and Kilobot in Table 4.4. Overall, each system uses OOK as a modulation method. However, Colias and r-one both use a single frequency carrier signal, which allows the use of signal filters (i.e., hardware) to improve detection. On the other hand, it requires hardware support and cannot be deployed on other platforms. LibIrcom advertises the use of frequency modulation, however, its software implementation shows the use of OOK and with

two 32-bit long codewords to encode a single bit reducing throughput to a $\frac{1}{32}$-th while not fully utilising channel coding (i.e., detection errors remain high).

While only the Colias robot does not utilise error detection or correction mechanisms, Kilobot and r-one provide 16-bit cyclic redundancy check (CRC) values that allow error detection. LibIrcom uses 2 bit CRC values for each byte of data. SwarmCom, on the other hand, uses the previously described BCH codes that allow the detection of errors through syndromes and the correction of 0, 1, or 7 errors depending on its configuration. This can reduce the likelihood of retransmissions and, consequently, reduce overhead.

While CSMA is used by libIrcom, Kilobot, and SwarmCom, the documentation on Colias and r-one do not report any access control methods. This increases the risk of message collisions and, subsequent, corrupt data. It is worth noting that r-one uses an ALOHA protocol for data transmission, which requires the acknowledgement of successfully received data. When a message is received, and an error detected, the message is not acknowledged and, subsequently, resent. However, noisy/busy channels or frequent topology changes can cause retransmissions, reducing the throughput.

Overall, SwarmCom is designed for systems with simple communication circuitry that lacks advanced signal processing hardware. It provides error detection and, in contrast to all other systems, error correction methods, where throughput can be reduced in favour of increasing reliability. Finally, it provides media access control that reduces the likelihood of retransmissions while maintaining its throughput. Insights on the performance of SwarmCom are presented as follows.

4.5 Evaluation

To allow a systematic verification of SwarmCom's performance and quality of service, a series of experiments are conducted. First, experiments are conducted with static robots to determine the quality of service based on the inter-robot distance d. Then, experiments with mobile robots are conducted to verify SwarmCom in more realistic environments. Finally, SwarmCom is compared to libIrcom to determine its performance against a widely used swarm robotics communication system.

4.5.1 Static-Robot Evaluation

To determine the communication quality of SwarmCom, a pair of static robots are used — a transmitting (R_t) and a receiving robot (R_r). Both robots use all of their emitters and detectors, as shown in the robot-to-robot model. Note that, unless otherwise stated, no channel coding is used to determine the true parameters (e.g., bit error rate).

4.5.1.1 Experimental Setup & Process

The experiments are conducted within a 4×3 m arena, as described in Section 3.3. The windows are shut to prevent light from entering. The experiments are automated with a computer. This computer is connected to each robot via Bluetooth[4.5.1] and controls the process (i.e., robot's behaviour) remotely.

Initially, both robots are placed in the middle of the arena, oriented towards each other (i.e., inter-robot distance is 7 cm). Then the experiment follows:

1. The computer uniformly randomly generates between 1 to 5 messages. Each message is 15 bit long, and each bit within a message is uniformly randomly chosen.

[4.5.1] To avoid a bias by the Bluetooth connection, each message to or from the computer is transmitted with a checksum and acknowledged to guarantee successful transmissions. In case of an error, the respected message is retransmitted.

2. The computer transmits the message to R_t via Bluetooth. When the message has been received, the checksum is calculated and compared. If the checksum does not match, the message is not acknowledged. If no acknowledgement is received within 500 ms, the computer retransmits the last message without affecting the result.
3. Once a correct message has been received by R_t, it starts the infra-red transmission to R_r. At the same time, R_t echoes the message back to the computer signalling the start of the transmission.
4. As soon as R_r receives 15 bit of data, it transmits it to the computer via Bluetooth.
5. The computer marks a message as lost if R_t acknowledged, but no message was received from R_r within 1 s.

After repeating the process until 100 successful messages have been obtained, R_r moves 1 cm away from R_t and the process is repeated. The experiment ends, when R_r has moved 100 cm or if ten consecutive messages have been marked as lost. In the latter case, the end of the communication range is assumed.

Based on the data, the computer calculates the following parameters:

- the *communication range*, which is the furthest inter-robot distance in which 100 successful messages were recorded,
- the *transmission time*, which is the difference between the times when data is received from R_t and R_r,
- the *bit rate* or throughput, which is the number of bits divided by the transmission time,
- the *bit error probability* or bit error rate, P_e, which is calculated by the Hamming distance[4.5.II] of the data from R_t and R_r divided by the length of the data,
- the *probability of error-free transmission*, P_f, which is

$$P_f(c) = \sum_{i=0}^{c} \binom{n}{i} P_e^i \, (1 - P_e)^{n-i}, \tag{4.44}$$

  where c is the maximum number of correctable errors of an n-bit message. In other words, P_f is the probability that at maximum c bit-errors occur. As up to c errors can be corrected, the transmission is considered error-free.
- the *probability of message loss*, P_l, which is

$$P_l = \frac{N_l}{N}, \tag{4.45}$$

  where N_l and N are the number of lost messages and the number of successful and lost messages, respectively.

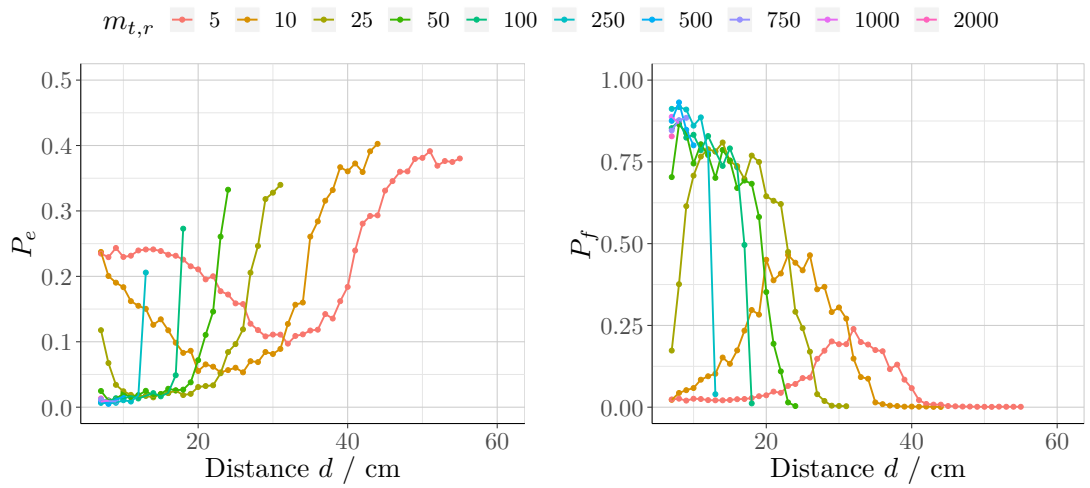
4.5.1.2 Fixed Threshold Experiments

This section investigates how different configuration of SwarmCom can influence the quality of communication. As the user can set bit rate and decision threshold, robots are configured to transmit with 310, 650, or 1160 bps (bit per second) and detect with relative thresholds of $m_{t,r} \in \{5,\ 10,\ 25,\ 50,\ 100,\ 250,\ 500,\ 750,\ 1000,\ 2000\}$. The relative decision threshold is calculated by

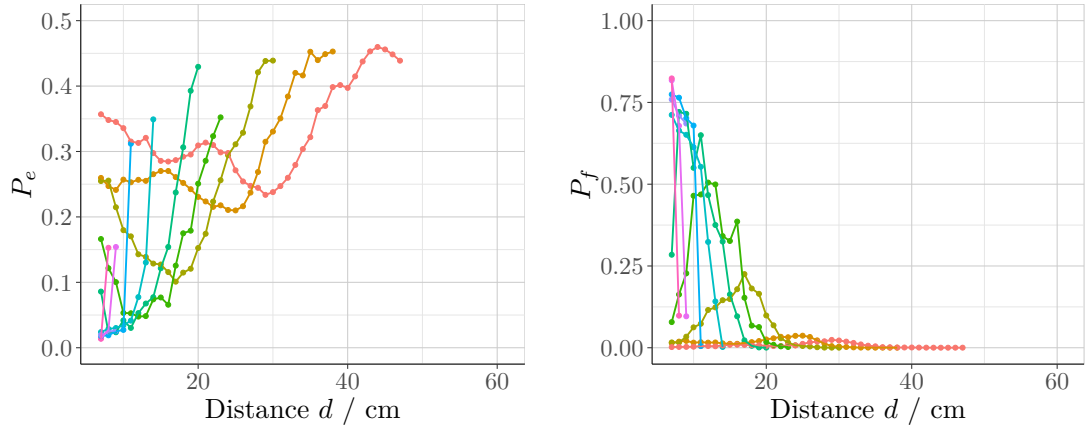
$$m_t = m_{\text{max}} - m_{t,r}. \tag{4.46}$$

Figure 4.20 shows the experimental results for each configuration. All trials show that a higher threshold reduces the communication range. While it is expected that closer robots produce lower bit-error probabilities, it can be seen that, in many cases, the decrease of inter-robot distance increases the bit error probability. As a result, an interim-region of low bit-error

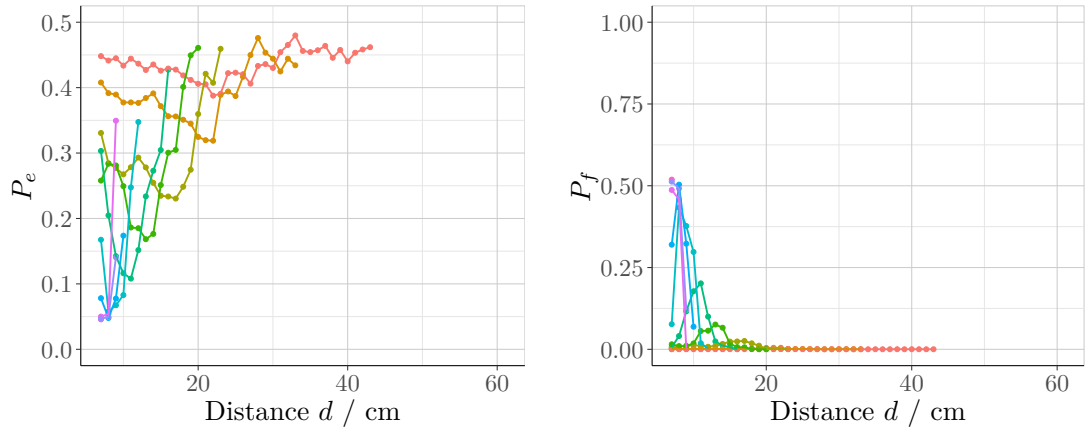
[4.5.II] The Hamming distance is the number of different bits between two vectors (e.g., the Hamming distance is 2 for 0101 and 0011).



(a) 310 bps



(b) 650 bps



(c) 1160 bps

Figure 4.20: Bit error probability, P_e, and the probability of error-free transmission, P_f, based on the inter-robot distance, d, of two static robots using different relative thresholds, $m_{t,r}$, and bit rates.

probability can be observed. Interestingly, when increasing the bit rate, the bit errors increase and the region of low bit-error probability is less distinct, which suggests that the increased bit errors at short ranges result from dynamic effects.

A possible reason for these observations is a low-pass characteristic (i.e., capacitive element) within the measuring circuit. As short distances result in high signal intensities (i.e., charging capacitive elements), the charges cannot be transported fast enough, resulting in inter-symbol interference. For higher bit rates, less time is available to move these charges; hence, the bit-error is larger. As lower thresholds are more sensitive to inter-symbol-interference, the bit-error would be larger for lower thresholds, as it was observed. As the circuit cannot be changed, a software solution needs to be found.

4.5.1.3 Dynamic Threshold Experiments

As a fixed threshold is either reducing the communication range or is sensitive to inter-symbol interference, a new method is proposed that decreases the impact of inter-symbol interference while maintaining larger communication ranges. The detection threshold is adapted dynamically based on the signal intensity.

If no signal has been detected, the detection threshold is $m_t = m_{\text{max}} - m_{t,r}$, as calculated in Section 4.4.2.3 to maintain high sensitivity, and, ergo, a large communication range. When a signal with amplitude A is detected, the dynamic decision threshold, $m_{t,dyn}$, is calculated by

$$m_{t,dyn} = \left\lfloor \frac{|m_t + A|}{2} \right\rfloor = \left\lfloor \frac{|m_{\text{max}} - m_{t,r} + A|}{2} \right\rfloor . \tag{4.47}$$

Note that (4.47) is based on m_t instead of m_{max} to ensure $m_{t,dyn} < m_t$. This dynamic threshold is used for the detection of each subsequent bit. When the transmission ends, the detection threshold is reset again to m_t.

Dynamic Threshold Evaluation

With the dynamic threshold, new sets of trials were conducted following the protocol, setup, and configurations as described in the previous section. The results are illustrated in Figure 4.21. Figure 4.21a shows that the bit-error probability is reduced by magnitudes of power in comparison to the previous experiment. Figure 4.21b illustrates that the communication range decreases with increased bit rates similar to Figure 4.20. The increased bit rates also increase inter-symbol interferences, which can lead to a region of increased bit-error probability. To avoid this region, $m_{t,r}$ can be selected to limit the communication range to low bit error probabilities.

Bit Error Limits

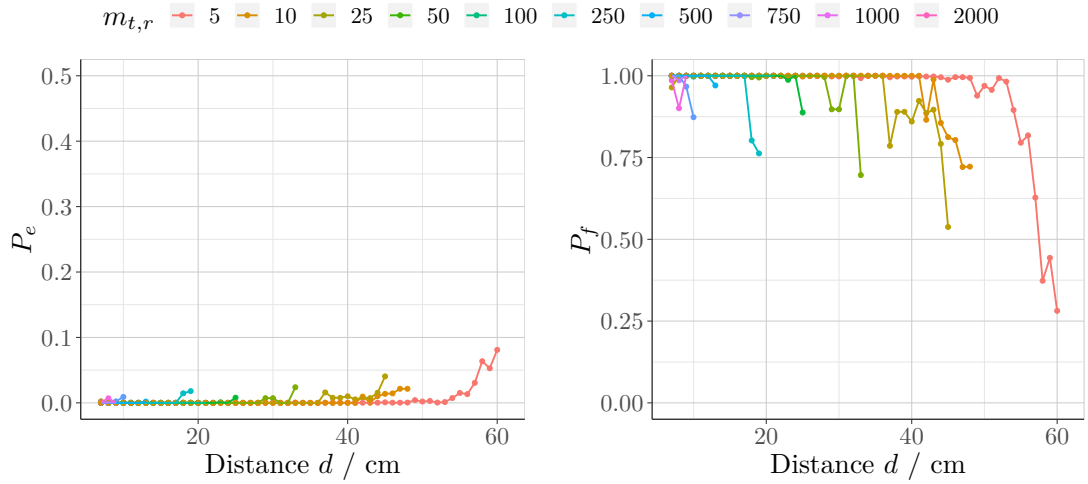
Additional experiments were performed to determine configurations which ensure $P_e < 0.01$. In these experiments, the set thresholds are matched with the bit rates in such a way that,

$$\max \{d \mid 1 - (1 - P_{e,b}(d)) \cdot (1 - P_{e,t}(d)) < 0.01\} , \tag{4.48}$$

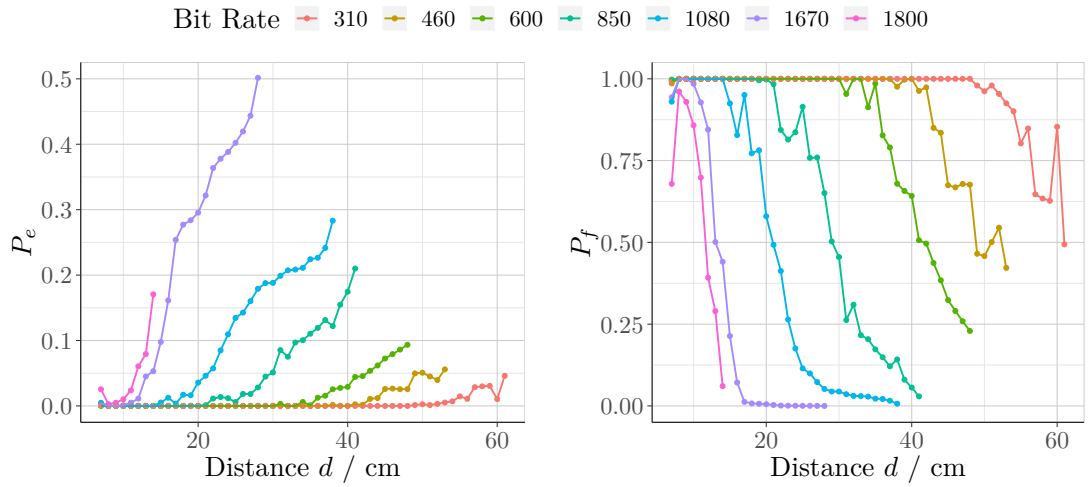
applies, where $P_{e,b}(d)$ and $P_{e,t}(d)$ are the error probabilities of the selected bit rate and the selected threshold at an inter-robot distance, d, respectively. When the obtained data is applied to (4.48), Figure 4.22 shows an area of configurations in which $P_e < 0.01$. In other words, it shows that $m_{t,r}$ and bit rate cannot be chosen independently without impacting the bit-error probability. Note that the limit of $P_e < 0.01$ is arbitrarily chosen and can be adapted to the need of the user.

Channel Coding

SwarmCom uses no channel coding per default to maximise throughput. However, it can be configured to correct up to 1 or up to 7 errors within a 15-bit message. To investigate the

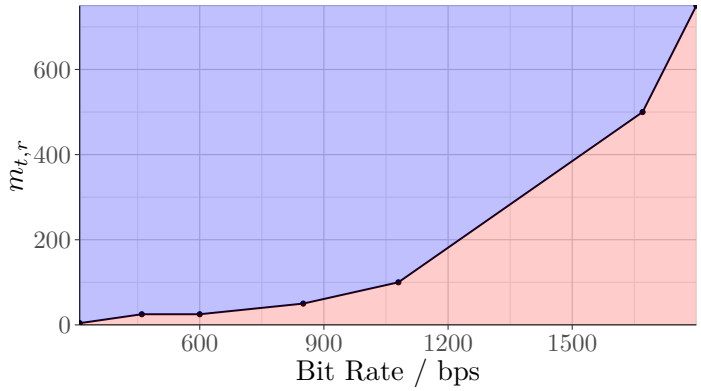


(a) Variable Threshold with 310 bps



(b) Variable Bit Rate with $m_{t,r} = 5$

Figure 4.21: Bit error probability, P_e, and the probability of error-free transmission, P_f, based on the inter-robot distance, d, of two static robots using dynamic threshold detection with a set of initial relative thresholds $m_{t,r}$ and bit rates.

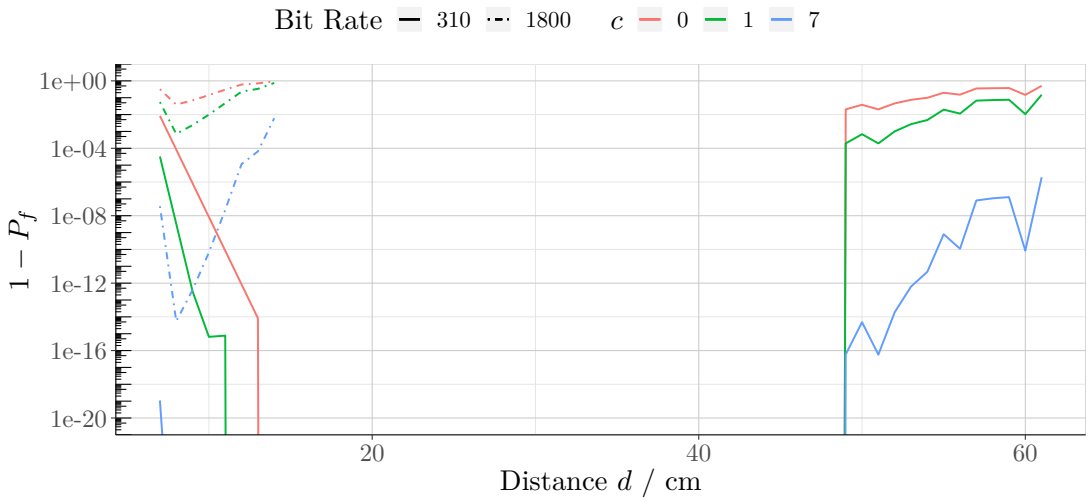


(a)

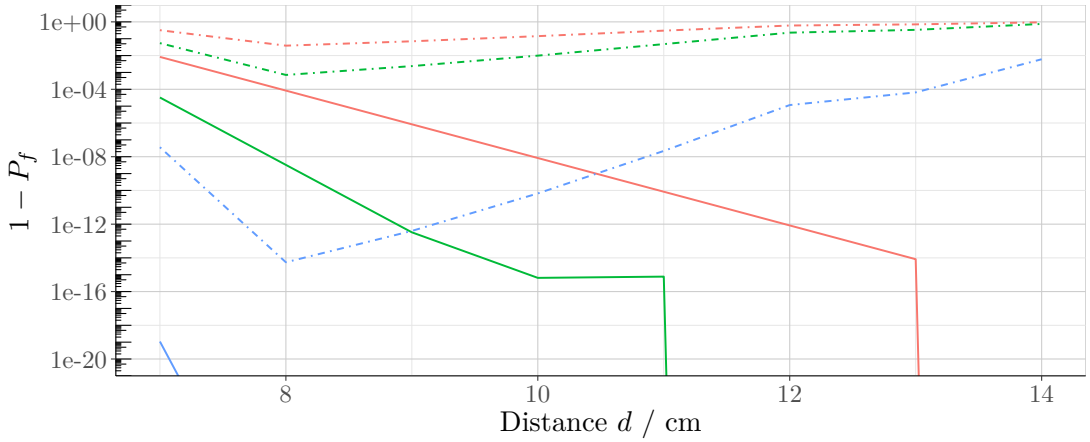
Bit Rate (bps)	$m_{t,r}$	d_{max} (cm)
1800	750	9
1670	500	10
1080	100	19
850	50	25
600	10	45
310	5	61

(b)

Figure 4.22: SwarmCom configuration space to ensure $P_e < 0.01$. (a) The blue area shows configurations in which $P_e < 0.01$. (b) The communication range resulting from each configuration.



(a) Full Range



(b) Zoomed Range

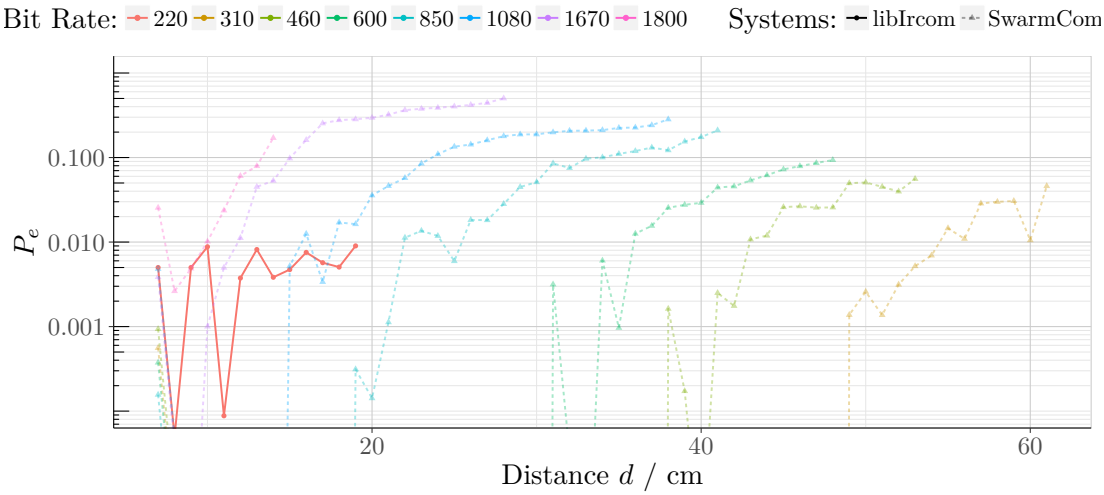
Figure 4.23: Applied channel coding to correct $c \in \{0, 1, 7\}$ errors within 15 bit. Shown is the probability of transmission with errors, $1 - P_f$, at 310 bps and 1800 bps.

impact of channel coding on the systems, the three channel codes are compared. Based on the data presented in Figure 4.21, channel coding is applied to transmissions with 310 bps and 1800 bps. The results are compared in Figure 4.23.

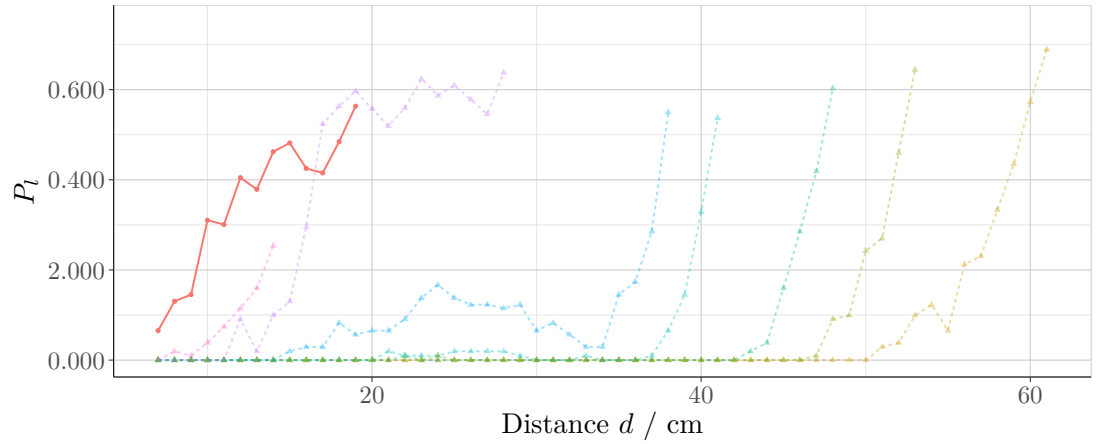
As illustrated, the probability of transmissions with errors, $1 - P_f$, decreases by several orders of magnitude when correcting 1 or 7 errors. This enables transmissions to be virtually error-free (i.e., $1 - P_f \leq 10^{-6}$) over a range of up to 60 cm. However, when considering transmissions with 310 bps, even without channel coding, they are, to a large proportion, error-free. On the downside, channel coding reduces the throughput to approximately 120 and 21 bps for bit rates of 1800 bps and 310 bps.

When comparing transmissions with 1800 and 310 bps, repetition-coded 1800 bps transmissions are less error-prone for $d < 10$ cm (i.e., ≤ 4 cm gap between robots), than an uncoded 310 bps transmission. When the distance between robots is likely to be larger than a few centimetres, choosing uncoded 310 bps provides a four times larger communication range with 2.5 times higher throughput while transmitting with fewer errors.

Generally, choosing the right channel code depends on the application and its robustness against communication errors. Overall, in situations where throughput is paramount, no channel coding is recommended; that being the case, this is SwarmCom's default.



(a)



(b)

Figure 4.24: Comparison of libIrcom (red) with SwarmCom (other colours): (a) bit error probability, P_e, and (b) probability of message loss, P_l.

Comparison with libIrcom

To fairly compare SwarmCom to an existing communication system, both systems are implemented with OpenSwarm and deployed on the same e-puck robot. This state-of-the-art communication software is libIrcom, which is often used with e-puck robots (e.g., [Murray et al. 2013; Prieto et al. 2010]). The implementation of libIrcom uses the identical functions to control the robot and exchange messages via Bluetooth as the SwarmCom implementation. The experiment was conducted following the same procedure and within the same environment as in Section 4.5.1.1.

Figure 4.24 shows the comparison between SwarmCom and libIrcom concerning the bit-error probability, P_e, and the probability of message loss, P_l, over the inter-robot distance. Based on the measurements, the bit rate of libIrcom is 220 bps (i.e., 12.2–71.0% of SwarmCom). At bit rates up to 850 bps, SwarmCom consistently outperforms libIrcom regarding range (i.e., increase of +116–221%), throughput (i.e., increase of +386–40.9%), and the probability of message loss by orders of magnitudes.

As libIrcom differs from SwarmCom in many aspects, it is difficult to attribute these improvements to a single factor. On the one hand, SwarmCom utilises the underlying channel model to maximise communication range and an adaptive threshold to reduce errors. Furthermore, each transmitted symbol carries data, maximizing throughput. On the other hand, by

Algorithm 1 Random Walk

Require: t_m, the forward moving interval
1: **procedure** MOVE
2: **loop**
3: Choose a random rotation angle, $\theta \in (-\pi, \pi]$
4: Rotate by θ
5: Choose a random velocity, $v \in [-128, 128]\ \frac{\text{mm}}{\text{s}}$
6: **repeat**
7: Move forward with v
8: **until** time, t_m, has elapsed
9: **end loop**

attempting the digital recreation of frequency modulation, libIrcom ends up with 32-bit codewords for a single symbol, considerably reducing throughput. While these codewords reduce the bit-error probability, the potential of channel coding is not fully used for two reasons: (I) the two used codewords have a high overlap (i.e., Hamming distance) reducing the effectiveness of the code and (II) only two combinations of a 32-bit long sequence are used further limiting the amount of information that can be sent.

4.5.2 Mobile-Robot evaluation

For a more realistic evaluation, additional experiments with a larger number of robots are conducted, where robots communicate while performing a random movement. First, the mobility is investigated, as it directly impacts the inter-robot distance and the quality of transmission. Then, the performances of two different modes of communication are compared. After that, the impact of changing the moving velocities, the communication range, the bit rate, or the density of robots are investigated. Finally, SwarmCom is compared to libIrcom within this more realistic environment.

4.5.2.1 Experimental Setup & Process

The experiments described in this section are conducted in a 72×72 cm arena. The arena has a grey floor and is surrounded by 30 cm tall white walls. It is illuminated, as described in Section 4.5.1.2.

Within the arena, 2 to 7 robots are placed in random locations and with random orientations. Each robot is connected to a computer via Bluetooth and performs a random walk continuously following Algorithm 1 with a motion time of $t_m = 2$ s. During this time, the computer automatically performs the following steps.

1. The computer connects to a random robot within the set.
2. Then, the computer randomly generates one-to-five 15-bit-long random messages similar to Section 4.5.1.1 and transmits these to the selected robot. If the robot does not acknowledge the message within 500 ms, the message is retransmitted without affecting the results.
3. While the data is echoed back to the computer, the messages are transmitted to other robots.
4. Any robot that receives data also transmits it to the computer. Note that robots utilise the dynamic threshold detection, as described in the previous section.
5. The computer waits for 2 s to receive messages and then records:
   - number of robots,
   - bit rate,
   - bit-error probability for each robot,

- probability of error-free transmission, and
- how many robots received a message.

The computer repeats this process 1000 times for every configuration.

Note that the experiments are conducted with two ways of communication:

***local broadcast*,** where a message is sent by only one robot and received by its neighbours. In this form, robots can only communicate in direct line of sight and within their communication range.

***flooding*,** where any received message is also transmitted at its first occurrence. This allows robots to communicate with robots outside of their communication range as other robots relay that message.

Each experiment is recorded by a camera mounted on top of the arena. A program[4.5.III] is used to calculate the robots' positions, potential occlusions, and inter-robot distances. This is subsequently used to estimate the probability of message loss, $\hat{P}_l$, in a static analysis.

The probability $\hat{P}_l$ is estimated by first detecting the transmitting robot (illuminated LEDs). Then, every other robot in direct line of sight is considered and its distance to the transmitting robot calculated. Based on the inter-robot distance, the probability of message loss is estimated by interpolating the data shown in Figure 4.21. If the transmission was a local broadcast, this value is recorded. If the transmission was flooded, $\hat{P}_l$ is calculated for every robot. First, all non-cyclic paths from the transmitting robot to any other robot are taken. For each candidate path, p, $\hat{P}_{l,p}$ is calculated by

$$\hat{P}_{l,p} = \prod_i (1 - \hat{P}_{l,p,i}), \tag{4.49}$$

where $\hat{P}_{l,p,i}$ is the probability of message loss for each hop i within p. The probability of message loss from the transmitting to a receiving robot is, then,

$$\hat{P}_l = 1 - \max_p \left\{ 1 - \hat{P}_{l,p} \right\}. \tag{4.50}$$

In other words, the probability of message loss is defined by the largest probability of successful transmission, $1 - \hat{P}_{l,p}$.

To compare the outcome of two experiments (e.g., the distributions of P_e), a two-sample Kolmogorov-Smirnov test (KS test) is used as defined in [Young 1977]. It is a non-parametric statistical test that compares two empirical cumulative distribution functions (CDF)[4.5.IV], $F_{1,n}$ and $F_{2,m}$. They are statistically significantly different when the maximum difference between both CDFs is

$$D_{m,n}^{(i,j)} = \sup_x \left| F_{i,n}(x) - F_{j,m}(x) \right|, \tag{4.51}$$

$$\underbrace{D_{m,n}^{(i,j)} \sqrt{\frac{m\,n}{m+n}}}_{\Delta^{(i,j)}} \geq \sqrt{-\frac{1}{2} \ln\left(\frac{\alpha}{2}\right)}. \tag{4.52}$$

where $\Delta^{(i,j)}$ is referred to as KS value and where n and m are the sample sizes of experiment i and j, respectively. For an significance level of $\alpha = 0.01$, (4.52) simplifies to

$$\Delta^{(i,j)} \geq 1.628. \tag{4.53}$$

[4.5.III] The used program is written in Qt 5.10 and OpenCV (version 3.4).

[4.5.IV] An empirical cumulative distribution function, $F_{i,k}$, of an experiment i is based on a sample of k measurements.

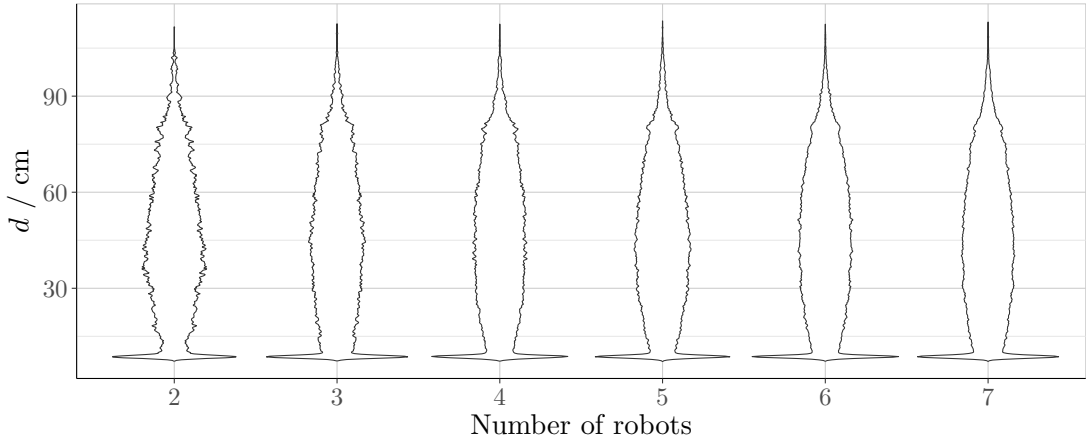


Figure 4.25: The distribution of inter-robot distance, d, between two moving robots within line of sight for experiments with 2, 3, . . . , 7 robots.

Note that, in this work, signed KS tests [Press et al. 2007] are used to decide if an experiment i produces statistically smaller values than another, j, when

$$L_{m,n}^{(i,j)} = \max_x \left(F_{i,n}(x) - F_{j,m}(x) \right), \tag{4.54}$$

$$L_{m,n}^{(i,j)} \sqrt{\frac{m\,n}{m+n}} > 1.628. \tag{4.55}$$

If not stated otherwise, statistical equality is tested with $D_{m,n}^{(i,j)}$.

In this work, a mobile-robot experiment consists of 6 trials. While the experiment setup and robot configurations remain the same, the number of deployed robots is different in each trial (i.e., two to seven robots). To compare two experiments, the distributions of the trials need comparing. In order to do that, this work introduces KS vectors and matrices. A KS vector contains six KS values, $\Delta^{(i,j)}$ comparing the trials with the same number of robots. A KS matrix consists of 36 KS values, where each combination of trials is compared.

In this work, KS vectors are used to compare two experiments with different configurations. KS matrices, on the other hand, are used to investigate changes (i.e., based on the number of robots) across the trials of an experiment.

4.5.2.2 Inter-Robot Distance Experiments

As the robots are moving across the arena, the distances between robots and, thus, their communication properties are changing continuously. As Section 4.5.1.2 showed, the distance between robots, d, directly relates to the quality of service. Therefore, first, the distribution of d is determined to indicating its expected shape during an experiment. As transmissions always occur in a line of sight, the distribution of d is determined by analysing the video recordings, and each pair of robots is considered.

Figure 4.25 shows the distribution of inter-robot distances for different numbers of robots as violin plots. The peak at 7 cm indicates that two robots have collided and have a high probability of restricted movement for a period of time. Another peak can be seen at around 70–80 cm, where two robots' movement is restricted by opposing walls. However, two robots restricting each other movement is more prominent.

Even though the distance distributions appear similar across the trials, a KS matrix is calculated to test their equality (i.e., the null hypothesis is that the distributions of d are equal).

The resulting KS matrix,

$$\begin{pmatrix} 0 & 16.941 & 16.973 & 7.310 & 13.336 & 18.702 \\ 16.941 & 0 & 4.281 & 16.420 & 12.791 & 5.135 \\ 16.973 & 4.281 & 0 & 16.249 & 10.632 & 2.883 \\ 7.310 & 16.420 & 16.249 & 0 & 11.587 & 21.165 \\ 13.336 & 12.791 & 10.632 & 11.587 & 0 & 11.831 \\ 18.702 & 5.135 & 2.883 & 21.165 & 11.831 & 0 \end{pmatrix}, \tag{4.56}$$

shows that all elements exceed the threshold 1.628 (highlighted in red). As a result, the null hypothesis is rejected for any trial pairing. In other words, the number of robots significantly changes the distribution of d. However, when considering the signed KS matrix,

$$\begin{pmatrix} 0 & 0.272 & 0.024 & 0.045 & 0.247 & 0.084 \\ 16.941 & 0 & 4.281 & 16.420 & 12.791 & 5.135 \\ 16.973 & 3.244 & 0 & 16.249 & 8.337 & 2.883 \\ 7.310 & 14.984 & 0.412 & 0 & 3.806 & 1.261 \\ 13.336 & 11.947 & 10.632 & 11.587 & 0 & 3.861 \\ 18.702 & 2.514 & 0.983 & 21.165 & 11.831 & 0 \end{pmatrix}, \tag{4.57}$$

it can be seen that only trials with 2 robots tend to maintain larger inter-robot distances than trials with more robots. Trials with more robots are significantly different, but robots are not statistically closer or further away than in the other trials.

4.5.2.3 Modes of communication

As swarms of robots can operate in large numbers and nearby, it must be expected that some robots are occluded by other robots, resulting in missed transmissions. Consequently, flooding can be used to convey data across the swarm. As there are two types of communication, the impact of choosing local broadcast or flooding is investigated.

For each type, an experiment is conducted with 2–7 robots. Each robot transmits with 310 bps and uses $m_{r,t} = 10$ (i.e., up to 45 cm of communication range).

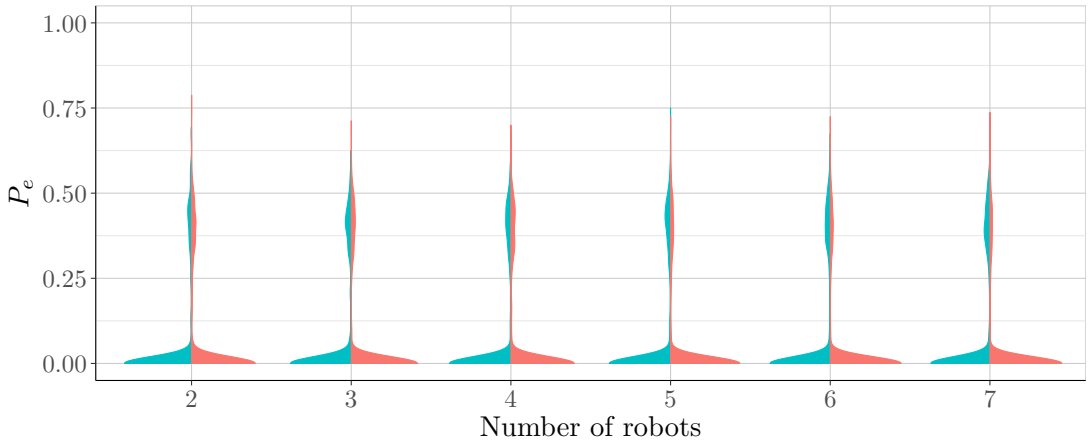
When investigating the probability of message loss, P_l, as shown in Figure 4.26b, it is clear that increasing the number of robots has different effects on flooded and local broadcast messages. As expected, increasing the number of robots leads to a decrease in messages losses when messages are flooded. As more robots are present, it is more likely that at least one path to a robot exists (even occluded ones). On the other hand, when transmitting only locally, increasing the number of robots also increases the likelihood of robots being occluded, resulting in a higher P_l.

Figure 4.26b shows that $\hat{P}_l$ follows the trend of the observed P_l with an offset. This is likely to stem from the computer vision analysis as it only identifies the robots' connectivity for a single video frame. This static analysis makes binary decisions on robots being occluded. Consequently, robots that obtain partial transmissions (i.e., entering or leaving robots during a transmission) are not captured.

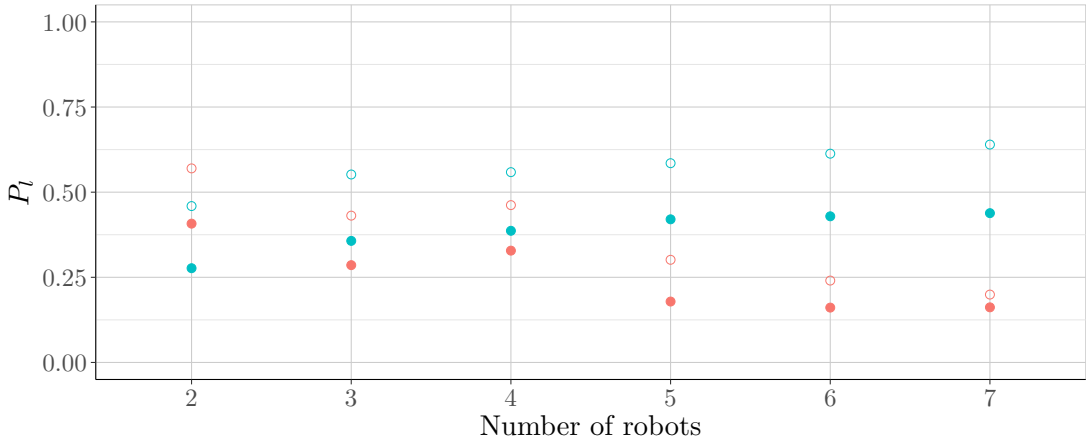
Figure 4.26a illustrates the distribution of the measured bit error probability, P_e, for both types of communication. To compare both distributions, the calculated KS vector,

$$[0.466, 0.336, 1.361, 0.603, 1.452, 1.131], \tag{4.58}$$

indicates no significant impact on P_e between the communication type. However, when invest-



(a)



(b)

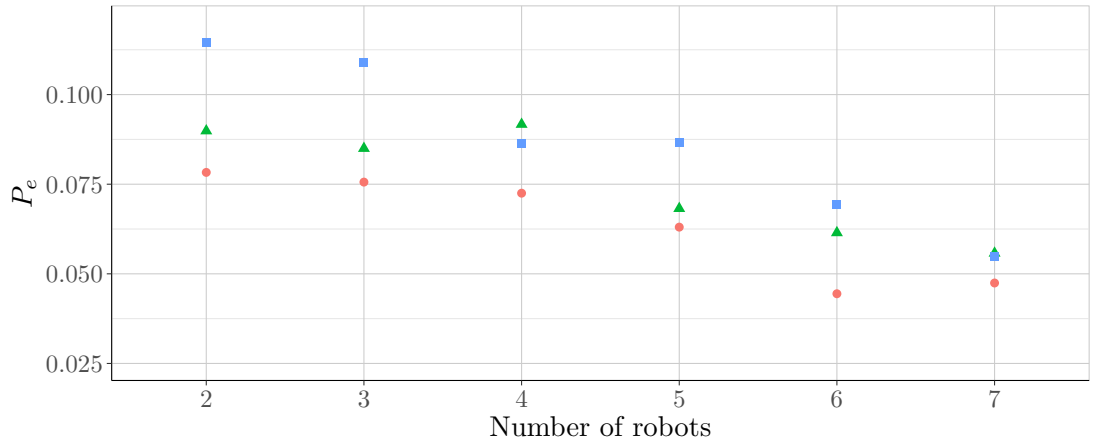
Figure 4.26: Comparison of two communication types: local broadcast (blue) and flooding (red). The distribution of the bit-error probability, P_e, and the average probability of message loss, P_l, are shown in (a) and (b) respectively. In (b), solid circles indicate average values and empty circles indicate the estimates of $\hat{P}_l$.

igating the effects of having different numbers of robots on P_e, the KS matrices,

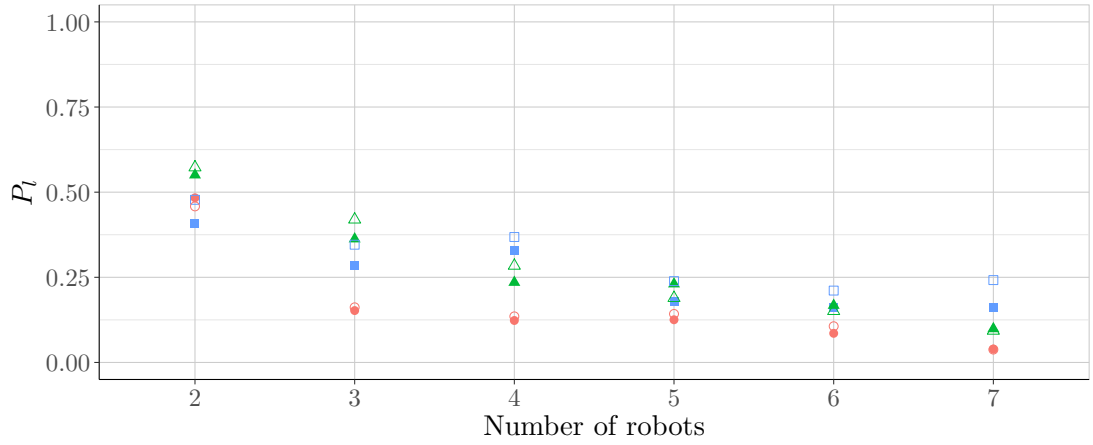
$$\left.\begin{pmatrix} 0 & 1.588 & 1.666 & 1.714 & 2.034 & 2.339 \\ 0.160 & 0 & 0.427 & 0.454 & 0.445 & 0.866 \\ 0.252 & 0.271 & 0 & 0.325 & 0.532 & 0.978 \\ 0.262 & 0.219 & 0.379 & 0 & 0.521 & 1.102 \\ 0.204 & 0.089 & 0.169 & 0.337 & 0 & 0.757 \\ 0.214 & 0.126 & 0.176 & 0.432 & 0.127 & 0 \end{pmatrix}\right\} \text{Loc. broadcast} \tag{4.59}$$

$$\left.\begin{pmatrix} 0 & 0.262 & 0.295 & 0.025 & 0.000 & 0.000 \\ 0.669 & 0 & 1.013 & 0.019 & 0.008 & 0.013 \\ 0.098 & 0.192 & 0 & 0.082 & 0.034 & 0.041 \\ 1.566 & 1.340 & 2.559 & 0 & 0.003 & 0.024 \\ 2.016 & 1.825 & 3.355 & 0.820 & 0 & 0.034 \\ 2.330 & 2.337 & 3.933 & 1.447 & 0.836 & 0 \end{pmatrix}\right\} \text{Flooding} \tag{4.60}$$

show two trends. Firstly, the number of robots has a small impact on the bit-error probability. One exception is the trial with two robots when transmitting locally, which shows significantly higher numbers of errors than trials with more robots. This is likely to result from significantly larger distances between robots, as shown in (4.57). Secondly, when flooded, increased



(a)



(b)

Figure 4.27: Experiment results for robots moving with $v_1 = 43\,\mathrm{mm\,s^{-1}}$ (red disk), $v_2 = 85\,\mathrm{mm\,s^{-1}}$ (green triangle) and $v_3 = 128\,\mathrm{mm\,s^{-1}}$ (blue square). (a) and (b) show the average bit error probability, P_e, and the probability of message loss, P_l. Solid and hollow shapes indicate the average of measured P_l and calculated $\hat{P}_l$.

numbers of robots result in higher P_e as any repeated message increases the error probability to $1 - (1 - P_e)^h$, where h is the number of hobs (i.e., repeats). Generally, it can be seen that increasing the number of robots increases the level of significance, and it can be expected that this trend continues.

When looking at the shape of the distribution of P_e as shown in Figure 4.26a, two peaks can be seen across all experiments. The dominant peak is at $P_e \approx 0$, where 68–75% of a trial's messages are error-free. The less pronounced peak is at $P_e \approx 0.4$. It is likely to stem from burst errors[4.5.V], resulting from robots obtaining only partial transmissions. Therefore, it would be a result of the robots' mobility.

4.5.2.4 Variation of the Velocities

This section examines the impact of different moving velocities (i.e., mobility) on communication. In the conducted experiments, Algorithm 1 is changed to limit the linear velocities to 43, 85, and $128\,\mathrm{mm\,s^{-1}}$. Note that robots are configured to flood messages.

Figure 4.27 illustrates the average values of (a) P_e and (b) P_l. Both probabilities appear to

[4.5.V] A burst error creates a sequence of successive faulty bits.

reduce with decreased velocity. To test this, three null hypotheses — that P_e of a trial is equal or larger for the respective trial with smaller velocity — are used, and a corresponding signed KS vectors calculated,

$$[1.631, 2.530, 1.902, 2.606, 3.192, 1.325], \tag{4.61}$$
$$[1.746, 1.721, 2.534, 1.704, 2.306, 1.501], \text{ and} \tag{4.62}$$
$$[1.715, 1.780, 0.677, 1.970, 1.108, 0.219], \tag{4.63}$$

for trials with v_1 against v_2, v_1 against v_3, and v_2 against v_3.

Based on the KS vectors, the null hypotheses can be rejected (with $\alpha = 0.01$) for trials with few robots. In other words, when increasing the velocity, the error probability increases for small numbers of robots. This is likely a result of robots more frequent entering or leaving the communication range, even during transmission. However, for increased numbers of robots, effects on P_e are less prominent as the likelihood of more possible communication paths increases.

Figure 4.27b shows that robots with the velocity v_1 tend to have fewer message losses. Interestingly, when comparing P_l with $\hat{P}_l$, it can be seen that $\hat{P}_l$ predict P_l better for lower velocities. This supports the claim that the prediction error results from robots passing through lines of transmissions or entering/leaving the communication range during transmission.

4.5.2.5 Variation of the Communication Range

As a sufficient communication range is paramount to connectivity in a network, this section investigates how different communication ranges change the communication quality. Three experiments were conducted in which the communication range is limited to 21 cm ($m_{r,t} = 75$), 33 cm ($m_{r,t} = 25$), and 45 cm ($m_{r,t} = 10$), respectively. Note that the robots continue to transmit with 310 bps.

Figure 4.28a illustrates the averages of bit-error probability, P_e, for three communications ranges. Similar to the previous section, the impact on P_e is tested by creating three null hypotheses – that P_e for the respective smaller communication range is less or equal to P_e of the respective larger range. When calculating the signed KS vector,

$$[0.607, 1.027, 0.521, 1.075, 1.772, 1.799], \tag{4.64}$$
$$[1.570, 2.266, 2.646, 3.151, 3.610, 4.300], \text{ and} \tag{4.65}$$
$$[1.108, 1.440, 1.765, 2.494, 4.271, 4.246], \tag{4.66}$$

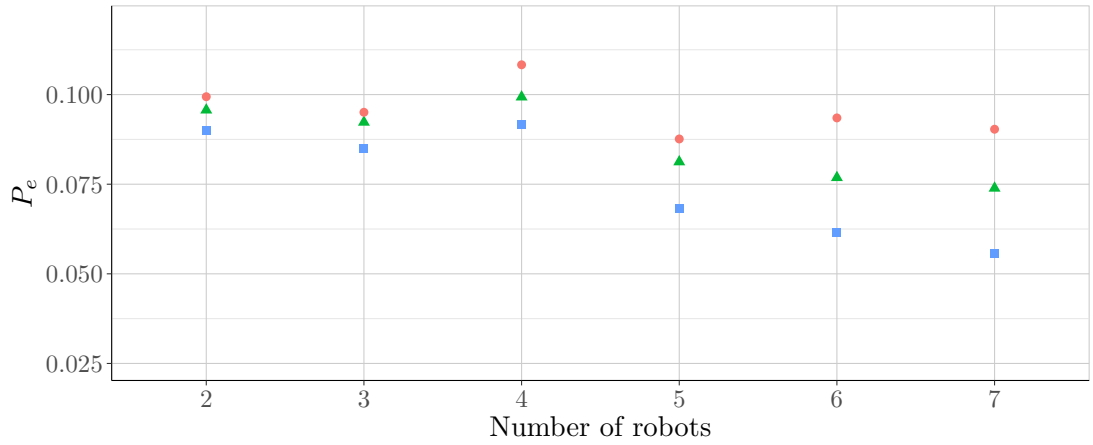
for testing the communication ranges 21 against 33, 21 against 45, and 33 against 45 cm, respectively. The vectors show that the null hypotheses are rejected in many cases, in particular for increased number of robots. In other words, smaller communication ranges tend to produce significantly more errors. This potentially results from robots entering/leaving or being obstructed during transmissions (i.e., causing burst errors).

In Figure 4.28b, the probability of message loss, P_l increases with shorter communication ranges. This is likely caused by reduced coverage and connectivity with shorter ranges.

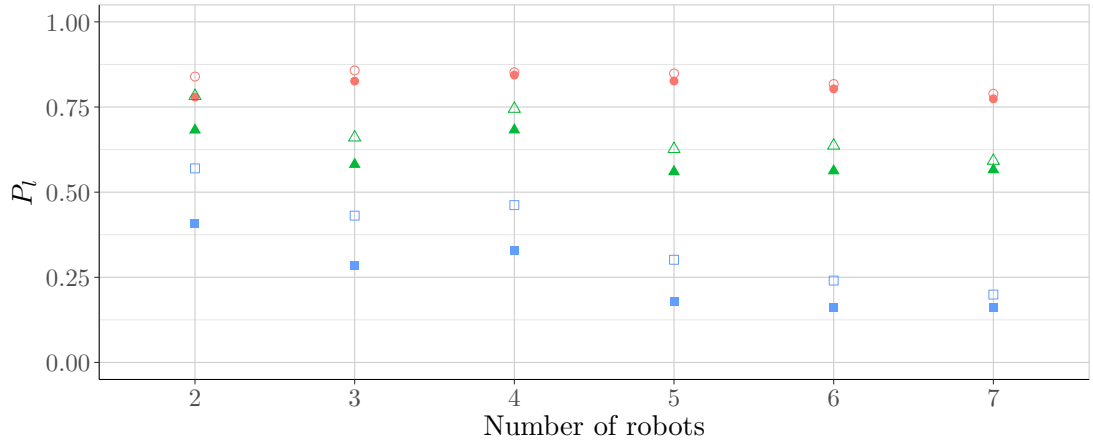
4.5.2.6 Variation of Bit Rate

SwarmCom can be configured to use different bit rates. To investigate the effects of changing the bit rate, three experiments with bit rates of 310, 1080, and 1670 bps were conducted. As shown in Section 4.5.1.2, a higher throughput is often traded for a shorter communication range. For a fair comparison, all robots are configured to communicate within 21 cm.

Figure 4.29a shows the average values of P_e for each bit rate, and it appears that lower bit rates are less error-prone. To test this observation, three null hypotheses — that P_e for the



(a)



(b)

Figure 4.28: Experiment results for robots communicating within a restricted communication range: (a) and (b) showing the average bit error probability, P_e, and the probability of message loss, P_l, for communication ranges of 21 (red, circle), 33 (green, triangle), and 45 cm (blue, square). Note that solid and hollow shapes indicate the average of the measured P_l and the calculated $\hat{P}_l$.

respective larger bit rate is less or equal than the respective smaller bit rate. For three pairs of bit rates, the signed KS vectors are

$$[0.609, 0.842, 1.411, 1.057, 1.352, 1.779], \tag{4.67}$$

$$[0.432, 0.999, 1.013, 1.502, 1.876, 2.371], \text{ and} \tag{4.68}$$

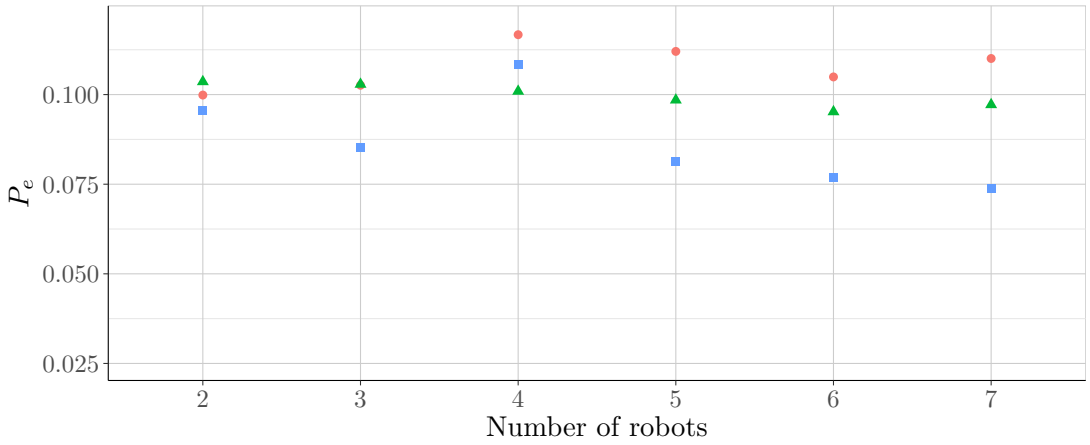
$$[0.432, 0.613, 0.746, 0.898, 0.705, 1.115], \tag{4.69}$$

when testing 310 against 1080, 310 against 1670, and 1080 against 1670 bps, respectively. Even though the significance values of the vectors increase with the number of robots, it can be seen that the null hypotheses cannot be rejected unless the number of robots is relatively large. Interestingly, the results suggest that the error probabilities are less influenced by the bit rate than by the communication range.

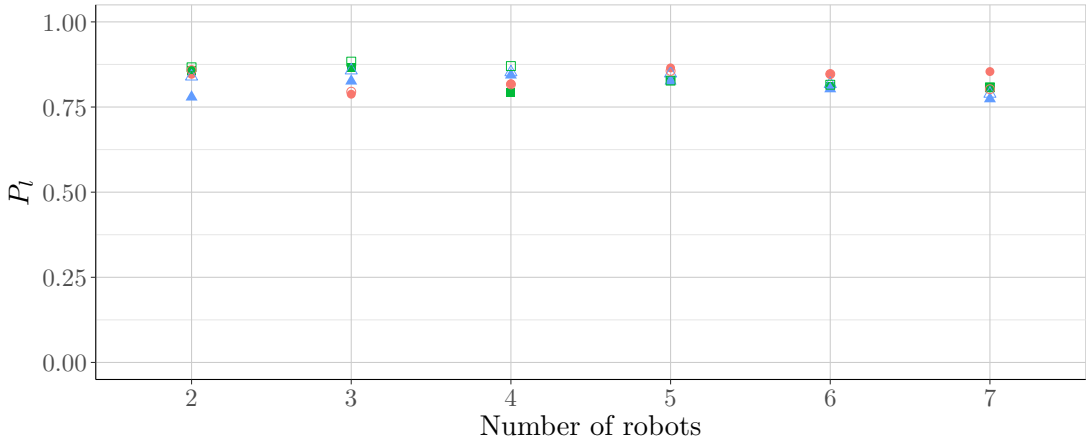
Finally, Figure 4.29b indicates that a change in bit rate has only a moderate impact on P_l.

4.5.2.7 Scalability

After investigating the impact of different configurations of SwarmCom, an additional property is investigated: scalability. Two studies were conducted — a computer simulation and



(a)



(b)

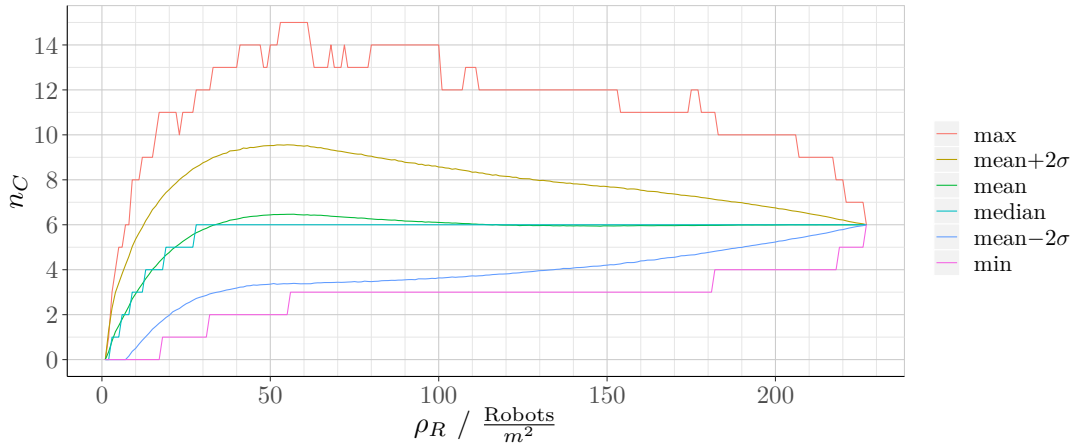
Figure 4.29: Experiment results for robots transmitting with different bit rates — 1670 bps (red, circle), 1080 bps (green, triangle), and 310 bps (blue, square). (a) and (b) show the average bit error probability, P_e and the average probability of message loss, P_l. Solid shapes indicate the averages of measured values and hollow shapes indicate the predictions.

experiments with real robots to evaluate the simulation. The simulation itself examines how the density of robots impacts the network connectivity.

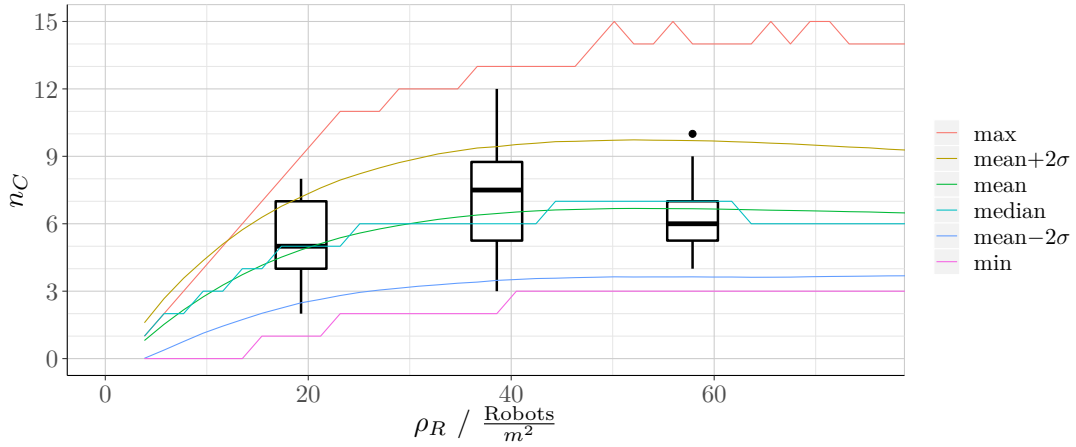
In the simulation, a transmitting robot creates a circular communication area of 61 cm radius. This arena is then populated by additional n randomly placed robots, where $n \in \{1, 2, \ldots, N\}$. The placing algorithm could place up to $N = 272$ robots, which is close to the theoretical limit of 285 robots[4.5.VI]. This results in a robot density, $\rho_R \in [1, 234]$ $\frac{\text{robots}}{\text{m}^2}$. For every density, the number of establishable channels, n_C, to the central robot is determined for 10000 different robot placements. A channel is considered to be established if a robot can distinguish between s_0 and s_1 based on the robot-to-robot model of Section 4.3.2.

Figure 4.30 shows the number of established channels, n_C, based on the robot density, ρ_R. It can be seen that n_C is bound by a relatively small constant (≈ 15). In other words, while hundreds of robots may be within range, the transmitting robot is only required to communicate with < 15 robots. Other robots are occluded. As the number of connections (i.e., load) is bound by a constant, the network is scalable. Furthermore, due to this relatively low upper bound, severely-constrained robots can participate in that network. Interestingly, if the network

[4.5.VI] 285 is the number of robots with 7 cm diameter that can fit within a circle of 61 cm radius according to http://hydra.nat.uni-magdeburg.de/packing/cci/cci285.html.

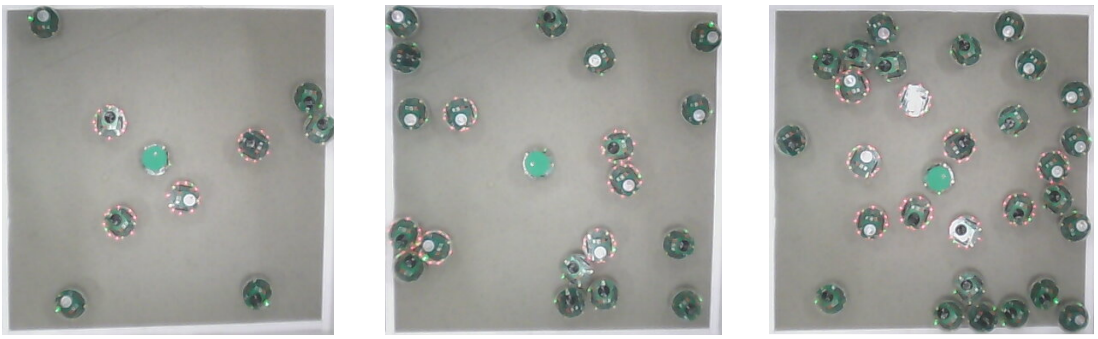


(a)



(b)

Figure 4.30: The (a) simulation and (b) real-robot experiment results showing the number of establishable channels, n_C depending on the number of robots. The data is based on a communication area of (a) a circle with 62 cm radius and (b) a 72 × 72 cm square.



(a) (b) (c)

Figure 4.31: Snapshots with (a) 10, (b) 20, and (c) 30 e-pucks within a 72 × 72 cm environment. The central static robot (green cap) continuously transmits the signal, and the other randomly-moving robots illuminate their LEDs when receiving the signal. Note that the number of channels, n_C, are (a) 4, (b) 6, and (c) 9.

works with high densities of robots, which is common in many academic experiments (e.g., [Rubenstein et al. 2014]), the number of connections converges to $n_C = 6$ (i.e., corresponding to a hexagonal arrangement).

The experiments with real robots (i.e., e-pucks) investigate how the robot density relates to the connectivity in practice. In total, 10, 20, and 30 e-pucks are used within a 72×72 cm arena, resulting in the robot densities, $\rho_R \in \{19.3, 38.6, 57.9\} \frac{\text{robots}}{\text{m}^2}$, as shown in Figure 4.31. In each of three experiments, one static robot is placed in the centre, while the other robots perform a random walk following Algorithm 1. While the central robot transmits a signal continuously, all other robots detect. When a robot receives the signal, it illuminates its LEDs. By emitting a continuous signal, a bias by bit-rates, velocities, or transmission errors can be avoided. The number of established channels, n_C, is determined by taking a snapshot every 30 seconds and counting the illuminated. In total, 90 trials are performed in three experiments (i.e., 30 per experiment).

Figure 4.30b shows the experiment results as box plots in comparison to simulation. To allow a fair comparison, the simulation was repeated with the only difference being that robots were randomly placed in a 72 × 72 cm arena. Overall, the observed data is in good agreement with the simulation results. These results suggest that the resource allocation and utilisation of SwarmCom are scalable and, in principle, could be applied to any number of robots.

4.5.2.8 Comparison with libIrcom

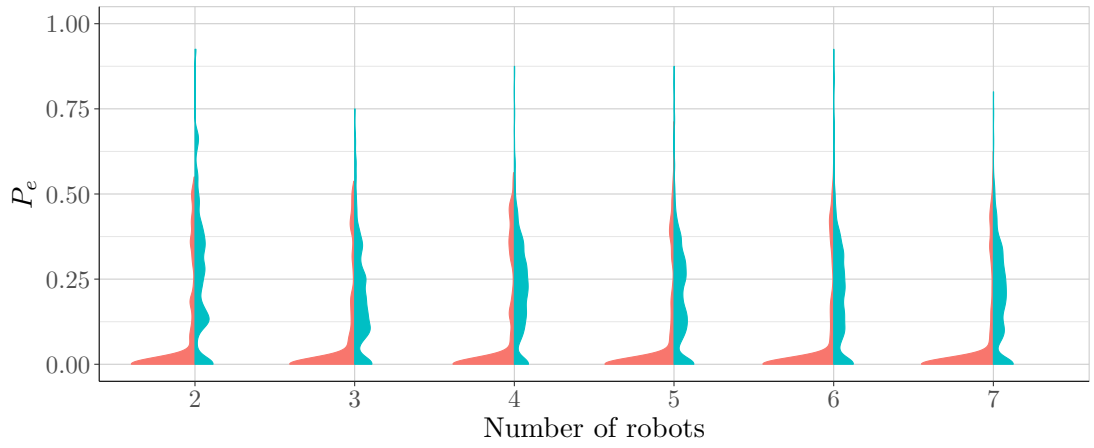
Finally, the overall performance of SwarmCom and libIrcom is compared in more realistic conditions than Section 4.5.1. To allow a fair comparison, SwarmCom is configured to use the same communication range (i.e., 21 cm) and a similar bit rate[4.5.VII] as libIrcom.

Figure 4.32a and 4.32b compare the distributions and average values of bit-error probability, P_e. Overall, libIrcom transmits only 14–20% of the messages error-free, which is a $\frac{1}{5}$ to $\frac{1}{3}$ of the error-free transmission with SwarmCom (68–75%). When comparing the averages of P_e (see Figure 4.32b), it appears that libIrcom produces significantly more errors than SwarmCom. To test this relation, the signed KS vector,

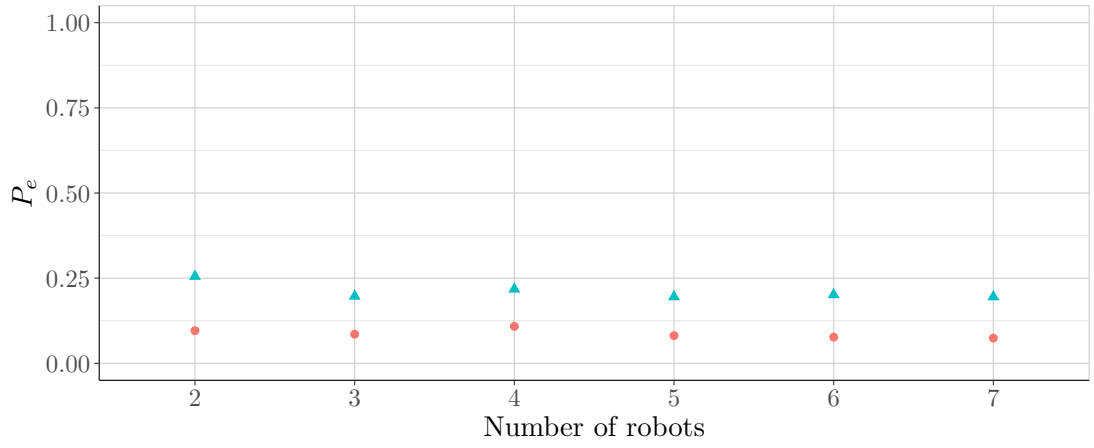
$$[4.207, 6.397, 7.630, 9.030, 10.316, 12.382]\,, \tag{4.70}$$

is calculated. It shows that libIrcom indeed produces significantly more errors than SwarmCom. Furthermore, comparing (4.64)–(4.69) to (4.70) shows that changing from SwarmCom

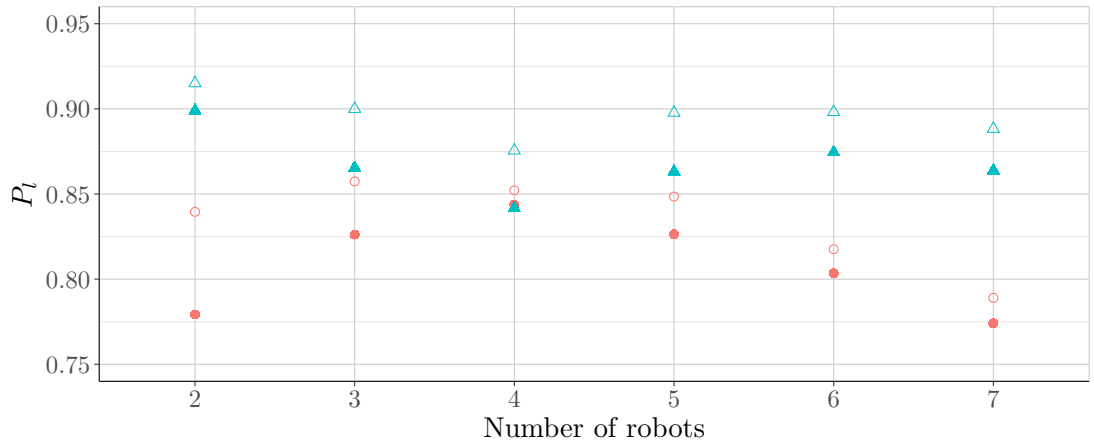
[4.5.VII] Note that 310 bps is the slowest possible bit rate in SwarmCom and, hence, is used to compare to libIrcom with its 220 bps.



(a)



(b)



(c)

Figure 4.32: Experiments with groups of mobile robots comparing SwarmCom (red, circles) and libIrcom (blue, triangles). (a) Distribution of P_e; (b) mean of P_l (solid) and $\hat{P}_l$ (hollow).

to libIrcom has a significantly larger impact on the bit-error probability than changing the bit rate or communication range.

Figure 4.32c shows the probability of message loss, P_l, for both systems. Similar to Section 4.5.2.5, both systems have comparable values of P_l further suggesting P_l is mainly a result of the communication range.

4.5.3 Discussion

In this section, a new dynamic-threshold detector was proposed to reduce the occurring inter-symbol interference. It has a low computational overhead of calculating one mean value per transmission. The proposed detector reduces the detection error by magnitudes of power, resulting in regions that can be considered error-free.

This section has demonstrated that SwarmCom is scalable towards the number of connections and the access of the medium. This allows the system to be deployed in situations with high densities of robots.

In general, SwarmCom was evaluated in two configurations — with static and mobile robots. Fundamentally, static robots provide significantly better quality of service with magnitudes of power lower error probability. In particular, results suggest that mobile robots increasingly obtain partial transmissions, which considerably increases the bit errors. As a result, it is recommended to exchange data when robots do not move.

The quality of communication between mobile robots is affected by the choice of movement, range, and bit rate. Experiments have shown that the quality of communication is most affected by the robots' velocities, then their communication range, and least influenced by the choice of bit rate. Note that the choice of one parameter can limit others (e.g., a higher bit rate reduces the maximum configurable communication range).

Overall, the biggest change in communication quality is caused when changing from SwarmCom to libIrcom with both static and mobile robots. It was shown that SwarmCom outperforms libIrcom regarding range, bit rate, error probability, and the probability of message loss.

4.6 Discussion

This chapter proposed the first model describing the infra-red signal characteristics of a widely-used swarm robot — the e-puck. Experiments showed that the model predictions closely match measurements of the real robots. This model can be applied not only to the e-puck but also to the e-puck 2 as it uses the same proximity sensors and measuring circuit as the original version[4.6.I]. While this model is used to describe the exchange of information between robots, it potentially can be used to improve the proximity measurements due to the use of the same signal.

The second part of this chapter proposed SwarmCom, an optical MANET for severely-constrained robots. Its key property is the dynamic detector that adapts to the ambient light during runtime and, more importantly, adapts its decision threshold to the incoming signal. This enables SwarmCom to reduce the bit-error probability by orders of magnitudes (i.e., more reliable communication). In addition, SwarmCom is configurable regarding bit rates, communication ranges, and channel coding (i.e., further improving communication quality). Experiments have shown that SwarmCom can communicate reliably within up to 61 cm at 310 bps and up to 14 cm at 1800 bps. Furthermore, transmissions up to 50 cm can be configured to be virtually error-free (i.e., a probability of error-free transmission is $1 - 10^{-16}$).

During the evaluation experiments, SwarmCom was tested in a more realistic environment where up to seven mobile robots communicated. It was shown that flooding can dramatic-

[4.6.I] Information obtained through mail exchange with GCtronic (http://www.gctronic.com/); schematics of the e-puck 2 have not been released at the time of writing this thesis.

ally improve the connectivity of the network. It allows robots to reach other robots that are occluded or too-distant to be reached otherwise without notability affecting the bit error probability. Furthermore, while a change of bit rate has no significant impact on the transmission errors, experiments showed that reducing the communication range increases transmission errors as well as reduces the connectivity of the network. Similarly, the robots speed impacts the transmission errors considerably as fast robots are likely to change connectivity during transmissions.

Based on the latter observation, the key insight of this chapter is that mobility significantly impacts the communication quality. This occurs for any system with relatively small communication range (i.e., cm to dm-scale), low bit rates (i.e., bps to kbps), and relatively high mobility. As these properties apply to a large set of communication systems in swarm robotics (e.g., Colias, e-puck, i-Swarm, Kilobot, and r-one), these findings highly suggest that any of these systems can suffer from this type of error. This should be considered during the design of swarming algorithms. For instance, where throughput is not the primary criterion, channel coding is recommended. If it is the primary criterion, it is recommended to cease motion during transmissions. In case robots cannot cease motion, the algorithm must be robust against communication errors.

Another key insight of this chapter is its investigation on scalability. Evaluations with up to 30 mobile e-pucks and computer simulations showed that the numbers of connections per robot have an upper bound of 15. This upper limit applies for any number of robots within communication range. Note that this is a property that can be applied to any network, where signals can be occluded, even though the exact value of the upper bound might vary based on a robot's size and shape. As a network's medium access and utilisation operate independently of the total number of robots, it is scalable.

In contrast, radio-based signals would reach any device within its communication range, resulting in potentially hundreds or thousands of robots. However, in practice, this exceeds the capability of many radio-based devices. Overall, swarm robotics benefits more from optical communication as it allows the use of severely-constrained robots, enables communication within a reasonable range, and copes well with any density of robots.

A possible extension for SwarmCom could include situated communication, where in addition to conveying messages, the receiving robot would detect the direction or relative position of the transmitting robot. This has been utilised in swarm robotics as it can prove useful, for example, to remain connected to a local neighbourhood. However, as there has been extensive work on this subject (e.g., [Støy 2001; Gutiérrez et al. 2009*a*]), investigating such features would not provide additional insights.

Finally, SwarmCom is compared to libIrcom, the only other infra-red communication library for the e-puck. It was shown that SwarmCom transmits without error 3–5$\times$ more often than libIrcom. In summary, SwarmCom provides 0.8–3 times wider range, 1.4–8 times higher bit rates, and 50%–63% lower bit error rates than libIrcom, without using channel coding. For instance, the user can prioritise 3 times larger communication range and 63% lower bit error rates while providing 1.4 times of the bit-rate.

A current limitation of SwarmCom is the use of flooding (i.e., a rudimentary routing method) to convey messages across the swarm. Due to many limitations (i.e., low throughput, large numbers of devices, and high mobility), the most modern routing algorithm cannot be deployed as obtaining the required topological information is, in practice, not feasible. As flooding does not require any topology information, it remains a valid option. However, if robots engage in frequent communication, the flooding protocol can consume most of the network's throughput as it is not scalable.

When considering existing optical systems of other disciplines [Malik and Singh 2015; Mallick 2016; Khan 2017; Zhang et al. 2018], it can be seen that they (hard- and software) are more reliable and offer a higher throughput. If similar developments can be achieved in

swarm robotics, topological information could be collected faster, enabling more sophisticated routing protocols and networking in general. It should be noted that dedicated communication hardware, such as [Gutiérrez et al. 2008; 2009*a*; Millard et al. 2017*b*], can outperform Swarm-Com, at least in bit rate and communication range. However, these systems significantly alter the robot's hardware adding additional costs, weight, and energy demand.

Lastly, this chapter has shown that a rigorous communication model, in combination with a well-designed communication system, can have significant effects on swarm communication. When this communication is not adequately investigated as is common practice, it could bias the outcome of robotic behaviour. Further research in swarm communication could help swarm robotics to transition into real-world environments.

5

Distributed Processing on Severely-Constrained Robots

Contents

After investigating how robots can compute locally in Chapter 3 and how they can communicate in Chapter 4, this chapter explores how these two components can be united to enable distributed computation. It would enable access to and processing of memory across multiple robots. While it is common in swarm robotics that a large number of relatively simple robots need to work together physically to perform a given task, distributed computation on robotic swarms would enable robots to work together computationally and enable the processing of algorithms that a single robot is too computationally constrained to perform.

Systems in which a group of independent devices processes and acts on local information to solve a task are *distributed systems* [Tanenbaum and Van Steen 2007]. These systems are commonly based on a large number of connected servers (i.e., clusters) providing services, such as distributed computation and distributed storage. In this field, the most prominent systems are Hadoop [Lam 2010], Kafka [Garg 2013], and Spark [Zaharia et al. 2016]. While research within this field is well understood, the latter systems mostly use high-performance infrastructures and networks, where computational or networking overhead can be tolerated. In swarm robotics, however, similar infrastructures and networks are often not available. As a result, many insights are not transferable to severely-constrained robots, such as e-pucks.

Distributed systems that are more similar to swarm robotics can be found in cyber-physical systems, where a large number of sensors and actuator are distributed over space (e.g., smart grid or smart cities) [Shi et al. 2011]. One of the largest infrastructures in this area is TerraSwarm [Lee et al. 2012]. It is designed to process data from thousands of mostly static sensors distributed over large spaces (e.g., a city). While the number of sensors and actuators is high (similar to swarm robotics), the relative low deployment densities allow the use of modern state-of-the-art technologies — in particular, 5G [Lien et al. 2019; Al-Turjman 2019] — and infrastructure similar to the previously discussed distributed systems. As mentioned earlier,

many swarms cannot adopt this research/technology due to the high deployment densities as well as the networking and computational constraints.

In robotics, systems similar to cyber-physical systems have been attempted with success — for instance, RCC, RaaS, cloud robotics and RobotEarth. In each of these systems, computational infrastructure either provides resources to the robot (i.e., RobotEarth and cloud robotics) or utilises the robot remotely (i.e., RCC and RaaS). However, these typically non-constrained systems commonly utilise weakly-constrained robots communicating via high-speed networks. Similarly to the last two types of systems, many swarms cannot adopt these solutions.

One work, in particular, is relevant for this chapter as it presents a similar approach coming from a cyber-physical perspective [Graff 2017]. Graff presents a system where a group of weakly-constrained robots is connected wirelessly to weakly-constrained devices. While the concept of only using robots has been presented in [Graff et al. 2014], it has never been realised. Overall, system-wide behaviour is implemented in a centralised manner, where robots and system have complete system knowledge. However, a common assumption in swarm robotics is that system-wide behaviour is achieved by controllers with limited system knowledge. While Graff [2017] acknowledges that the presented system requires full system knowledge, he discusses the decentralisation as a potential future work.

This chapter fills this gap, whereby the computational resources of robots are combined. However, unlike before, this system is fully decentralised. In addition, the presented work is based on severely-constrained robots with constrained network capabilities, which ordinarily limits the system functions. The presented work is a novel attempt to lift the often overlooked severe computational constraints of many swarm robots.

5.1 Distributed Extension of OpenSwarm

To create a system for distributed computation on top of loosely coupled systems, such as swarms of robots, each robot needs to manage its local resources and needs to provide access to memory via a network. In this work, OpenSwarm manages local resources (see Chapter 3) and SwarmCom manages the network access and usage (see Chapter 4). In order to create a system for distributed computation, OpenSwarm is extended to perform *remote procedure calls* (RPCs) [Bershad et al. 1990], to synchronise the behaviour of a group of robots and to achieve *consensus* [Correia et al. 2011].

5.1.1 Remote Procedure Call

An RPC is a mechanism whereby a function (i.e., procedure) is called and executed on a different device. In comparison to local procedure calls, which invoke the allocation of call stack elements and execution of instructions on the same device, RPCs invoke the generation of a message that is passed to another device and interpreted (i.e., translated into a local procedure call).

Through the hybrid kernel of OpenSwarm, events can be used to implements behaviours. At the occurring of an event, the data of it is passed to a locally executed function. As OpenSwarm has been designed with distribution in mind, the function of an event resembles the function of an RPC with the difference being that RPCs transmit the data via a network.

To implement RPCs in OpenSwarm, additional system calls have been added, which transmit the event identifier (i.e., enventID) and the corresponding data as one message via the network. At arrival, the event is then treated as a local event. In particular, system calls were added that give the user the control to send an event locally (existing) or globally (new). When used, the global sending of an event also emits the event locally. In addition, other added system calls can register an event as global, which means that a local emitting of an event is always performed locally and globally. For more details, see the usage examples in Appendix B. In

this way, any function or data access can be implemented through event handlers as described in Chapter 3.

A special case of RPCs is remote memory access. It is a way to manipulate addressable memory on a remote device. While this can be implemented as an event, a specific system call has been added. This system call writes given values to a specified location in memory locally and globally. As the memory location is specified within the RPC, this function should only be applied on statically allocated memory as it can produce undefined behaviour on dynamically allocated memory.

5.1.2 Consensus

A key aspect of a distributed system is the decision making or the control of the overall behaviour. In many systems, this is performed by a centralised unit or infrastructure. However, systems in swarm robotics often avoid centralisation as they provide a single point of failure and poor scalability to large numbers. Consequently, the robots need to achieve a consensus if decisions need to be made or behaviour controlled.

For consensus, each robot transmits its data. At arrival, each data is stored separately and, when comparing these values, the consensus is found by selecting the most frequent values. Note that this approach to achieve consensus can be considered minimalistic. However, it is sufficient for the presented proof-of-concept (see Section 5.2) as the communication is configured in such a way that the probability of error-free transmission is approximately $1-10^{-16}$; hence, failures are unlikely during the experiments. If this system is deployed elsewhere, the author recommends to use an established algorithm, such as PBFT [Castro and Liskov 2002], or to consult the large body of existing work in this field [Ren et al. 2005; Correia et al. 2011].

5.1.3 Synchronisation

In practice, it might be necessary for robots to synchronise their behaviour. There are multiple ways to achieve this. Firstly, by performing the consensus algorithm, the robots synchronise their behaviour as each robot waits for the arrival of all values. The second option is to use remote memory access, where for instance, each robot sets globally a dedicated memory location. If all locations have been set, the robots are synchronised and can perform the next task.

5.1.4 Discussion

By sharing events across multiple robots, it is possible to store and process data remotely. However, this has multiple implications for the operated swarm.

- The system is restricted to the same build of OpenSwarm to prevent inconsistent data access or execution across robots. In practice, this limitation has a small impact as using the same build is often required to guarantee consistent behaviours.
- Due to the use of a mobile and wireless network, the execution and delays of RPCs are unpredictable; hence, the system is always a soft real-time system [Kopetz 2011]. As current swarm robotics systems are applied mostly within academic environments, this limitation does not have a significant impact. In future applications, however, hard real-time properties might be desirable. This can only be achieved with further advancements of digital communication for swarm robots.
  In addition, for the conducted experiments, the uncertainty of communication is minimised by robots only communicating when stationary. Moreover, as discussed in Section 5.2, SwarmCom's configuration and the distance between robots is selected in such a way that it only operates in a region of high probability of error-free transmission of $1-10^{-16}$, which is based on findings from Chapter 4.

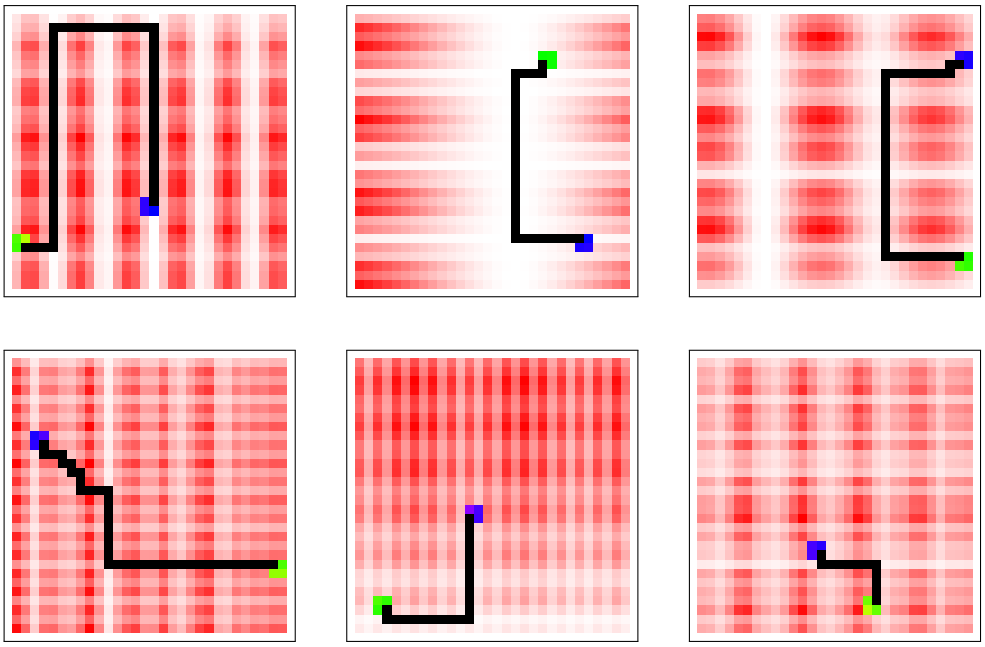


Figure 5.1: Six experiment environments (i.e., a grid of 30×30 cells) filled with inhomogeneously distributed radiation (red). Four robots start at the start cells (green) and need to move to the goal cells (blue), while being minimally exposed to the radiation.

Besides these limitations, the distribution of events offers additional advances for swarm robots. Other than distributed storage and processing, it allows robots to be used in augmented environments as sensors/actuators can be virtualised. While this itself might be useful to improve development times by facilitating testing directly on robots, it could also help swarm robotics to further bridge the reality gap by providing more data (e.g., from external sensors) or functions (e.g., from external infrastructure) to robots.

5.2 Evaluation

In this section, multiple properties of the extended operating system are investigated. This is being done by proposing a task that is inspired by search and rescue. In this task, robots need to (I) map the environment, (II) compute a path to the goal, and then (III) follow a path as illustrated in Figure 5.1. The specific challenge of this evaluation lies in the fact that a single robot is computationally too constrained to perform mapping and path planning individually. As a result, the solution discussed below serves as a proof-of-concept that overcoming severe constraints is feasible.

5.2.1 Environment

The area to be searched is a grid of $W \times H$ cells with height, $H \in \mathbb{N}$, and width, $W \in \mathbb{N}$. There can be a maximum of one robot at a time in each cell. From any location, a robot can move horizontally or vertically to other adjacent cells within the grid.

At the beginning of the experiment, n_R robots are placed in neighbouring cells, hereafter referred to as start cells, which are aligned in a rectangle or a square with a width of $\lceil \sqrt{n_R} \rceil$ and a height of $\left\lceil \frac{n_R}{\lceil \sqrt{n_R} \rceil} \right\rceil$, as shown in Figure 5.1. The start cells as a group are located at a random position[5.2.I] within the grid. The target that needs to be found is a group of n_R neighbouring

[5.2.I] To select a random position for the start cells, a single location is randomly chosen from

cells, hereafter referred to as goal cells. They are aligned similarly to the start cells and are positioned at a random location without overlapping any start cells.

Each cell has a radiation[5.2.II]-value, $\mathfrak{R} : \mathbb{N}^2 \to \mathbb{N}$. The radiation, $\mathfrak{R}(x, y)$, is defined at a location through its discrete frequency components

$$\mathfrak{R}(x, y) = \sum_i \sum_j A_i \, A_j \, \mathrm{e}^{\mathrm{i}(k_i \, x + \phi_i) + \mathrm{i}(k_j \, y + \phi_j)}, \tag{5.1}$$

where $A_i \, \mathrm{e}^{\mathrm{i}(k_i \, x + \phi_i)}$ and $A_j \, \mathrm{e}^{\mathrm{i}(k_j \, y + \phi_j)}$ are orthogonal wavenumber components across the width and height, respectively. Note that A_i as well as A_j and k_i as well as k_j are amplitudes and spatial wavenumbers. Due to the Petersen-Middleton theorem [Petersen and Middleton 1962], $k_i \in \mathbb{R}$ and $k_j \in \mathbb{R}$ is bound by $[0, \pi)$. Note that the ranges of the coefficients of 5.1 are described in Section 5.2.4

When a robot moves from one cell, i, to an adjacent cell, j, the exposure to radiation is calculated by

$$r_{j,i} = \frac{\mathfrak{R}(x_j, y_j) + \mathfrak{R}(x_i, y_i)}{2}. \tag{5.2}$$

When moving along a path, the path exposure is calculated by accumulating the individual cell exposures.

Finally, let z bytes be the memory that is required to store the radiation and exposure value for a cell. For this evaluation, let the available memory of a single robot, M, be smaller than the memory that is required for mapping and path planning (i.e., all measurements and accumulative exposure for each cell) results,

$$M < H \, W \, z. \tag{5.3}$$

5.2.2 Solution

Due to the computational constraints of robots, the grid is split into n_R segments (i.e., one per robot). Each segment is a vertical slice of the grid, which is stored and processed by the respected robot. To distributively process the segment, each segment shares the first and last column with the neighbouring segments resulting in $\lceil W \, n_R^{-1} \rceil + 1$ columns per segment. In other words, having more robots increases the memory overhead as more redundant cells are stored (i.e., $n_R - 1$ redundant columns). As a result, the number of columns is at minimum two for $n_R = W$ robots.

The proposed behaviour is divided into 3 stages: mapping (i.e., filling the segment with measurements), path planning (i.e., processing the segment), and path following (i.e., extracting the path from segments and following it). Figure 5.2 illustrates the proposed behaviour as a flow chart. At the beginning when all robots are at their start cells, the robots need to agree which segment is mapped by whom as illustrated in Listing 5.1. Thereafter, the mapping of the assigned segment is performed, which ends with returning to the corresponding start cell. When all robots have arrived at their start cells, as shown in Listing 5.2, the distributed path planning is performed. When finished, the path (i.e., the sequence of adjacent cells) to the goal is shared with the robots and subsequently followed by every robot. As soon as all robots have arrived at their goal cell, the task is completed. For more details on the mapping, path planning, and the path following, see the sections below.

$\left\{ (x, y) \,|\, x \in [0, W - \lceil \sqrt{n_R} \rceil) \wedge y \in \left[0, H - \left\lceil \frac{n_R}{\lceil \sqrt{n_R} \rceil} \right\rceil \right) \right\}$ as root of the rectangle containing all start cells. Note the same procedure applies to the goal cells

[5.2.II] Note that radiation can be seen as a generic cost for a robot.

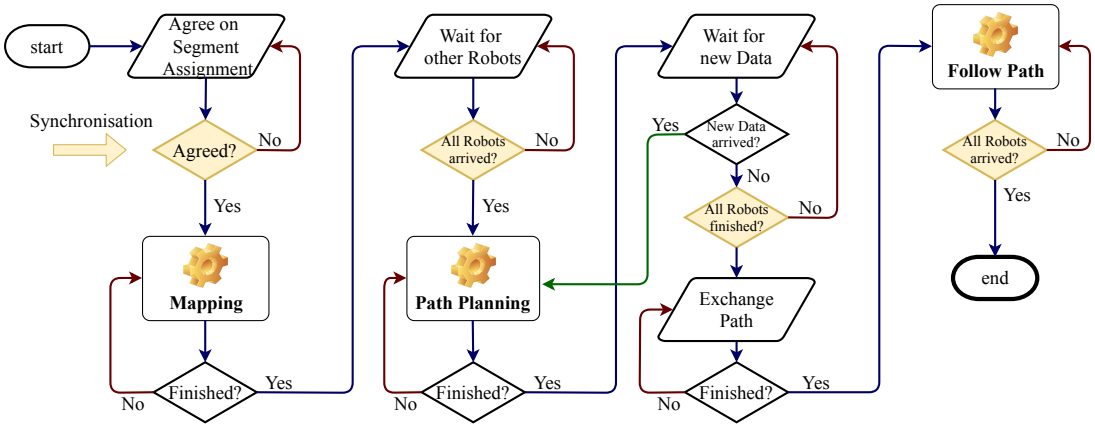


Figure 5.2: The behaviour illustrated as a flow chart. The gears indicate that the respective robot is performing one of three stages of the proposed behaviour. Yellow boxes indicate that the all robots' behaviours are synchronous at that point.

```
static volatile char segments[] = {ROBOT_1, ROBOT_2,
 ↪  ROBOT_3, ROBOT_4};

// ...
  if(Sys_AgreeOn(segments, 4)){// has been agreed on?
    for(int i = 0; i < 4; i++){//which segment?
      if(segment[i] == THIS_ROBOT){
        startMapping(i);// initialise mapping
        break;
  }}}
// ...
```

Listing 5.1: A code segment showing how a consensus is achieved based on a data vector `segments`. It is achieved by following the description in Section 5.1.2. After agreeing on the vector, the vector is searched to identify the current robot's segment number. The segment number is afterwards used to initialise and start the mapping process (i.e., a state change to `m_state = doing_mapping`) with `startMapping`. This code is a code snippet from the implementation of Figure 5.2. Note that `ROBOT_1`, `ROBOT_2`, `ROBOT_3`, `ROBOT_4`, and `THIS_ROBOT` are preprocessor variables containing the ID for each robot and the ID of the current robot.

```
volatile bool atHome[] = {false, false, false, false};

void areAllAtHome(){
    if(atGoal()){//reached home locally
        Sys_GlobalSet(&atHome[THIS_ROBOT], true);
    }

    for(uint i = 0; i < 4; ++i){
        if(!atHome[i]){
          return;
```

```
11        }}
12
13      startProcessing();// initialises & starts processing
14    }
```

Listing 5.2: A code snippet showing how robots can synchronise with remote memory access, `Sys_GlobalSet`. In this code, all robots need to move to their start cell (already set as `goal`). When all robots have arrived, the segment processing is initialised and started with `startProcessing`. This code is a code snippet from the implementation of Figure 5.2. Note that `THIS_ROBOT` is a preprocessor variable containing the ID of the current robot, which is between 0 and 3.

5.2.2.1 Mapping

After the segments have been assigned, the mapping is initialised in which the system calculates the segment boundaries, starts moving towards the first cell within the segment, and then starts the mapping by changing the mapping state. The first cell within a segment is the cell with the smallest column and row index. At the arrival at that cell, the robot measures the radiation[5.2.III], as shown in Listing 5.3. Thereafter, the robot moves vertically (next row) to the next cell. This process is repeated until the robot has measured all cells within a column. After moving horizontally to the next column, the previous steps are repeated until all cells in all columns are visited and measured. After measuring the last cell, the robot moves to its start cell.

```
1   extern int height;//of segment
2   extern volatile Position goal;//to move to
3   extern volatile Position mapping_end;
4   extern volatile mapping_state m_state;
5
6   void mapping(){
7       if(m_state != doing_mapping) return;
8
9       if(atGoal()){
10          measure();
11          goal = nextCell();//robot moves to goal
12      }
13  }
14
15  static int delta_y = 1;
16  Position nextCell(){
17      uint x = goal.x;
18      uint y = goal.y;
19
20      y += delta_y;//move vertically
21
22      if(y >= (uint)height){//move horizontally
23          x += 1;
24          y = goal.y;
```

[5.2.III] The radiation is assumed to be homogeneous within the cell.

```
25            delta_y = -delta_y;
26        }
27        Position p = {x,y};
28
29        if(x > mapping_end.x){//done?
30            p = home;
31            m_state = idle;
32        }
33        return p;
34    }
```

Listing 5.3: The implemented mapping algorithm. This code is executed whenever new localisation data is available as it has been implemented through events. Note that setting `goal` makes the robot move to the new goal location unless it has already reached it. `m_state` is the state of the mapping behaviour; it can be `idle` or `doing_mapping`. `mapping_end` is a struct containing the limits of the segment.

5.2.2.2 Path Planning

After the mapping has taken place, the distributed path planning starts. To reduce computational overhead, the path planning algorithm calculates a path having minimum cumulative exposure from goal to start (i.e., the exposure of the goal is set to zero).

The used algorithm is a variation of the Floyd–Warshall algorithm [Sedgewick and Wayne 2015], in which the minimal cumulative exposure to the goal cell is calculated for every cell within the grid. As each robot contains a different part (i.e., segment) of the grid, each segment is independently calculated as shown in Listing 5.4. Once the local processing has been completed, the shared columns are then exchanged with neighbouring robots.

While there are several different and well-established approaches to path finding, namely Dijkstra and A*, the choice of using the Floyd–Warshall algorithm was based on the following reasons:

- In this task, a robot cannot store enough data to map and compute the path by itself. As a result, memory consumption is the most critical resource in this case. While Dijkstra and A* have an improved processing time enabled by storing sorted queues, the Floyd–Warshall algorithm requires more processing time; however, does not require to additional lists of cells (i.e., queues).
- When considering the processing and communication time, it can be seen that the transmission of 30 cells consumes as much time as approximately $8.5\,10^6$ instruction cycles. In other words, the transmission of one shared column takes as much time as applying approximately 18000, 37000, and 75000 instructions on each cell when using 2, 4, and 6 robots, respectively. As can be seen, communication time is a more critical resource. As a distributed version of A* would require consensus on "which cell to compute next" after computing a single cell, this algorithm would create a communication overhead. While Dijkstra would also need to achieve the same consensus to keep their queues sorted across the swarm, the Floyd–Warshall algorithm can compute the entire segment before it needs to exchange shared columns.

The Floyd–Warshall algorithm has been chosen to reduce memory consumption and communication time.

In detail, the used variation of the Floyd–Warshall algorithm works as shown in Listing 5.4. When a segment is processed, the algorithm sequentially calculates the cumulative exposure

for each cell, based on (5.2), for each neighbour. The new value of the cell is the minimum of the calculated values and the current[5.2.IV] value, as shown with `calculate`. To reduce the memory consumption (i.e., avoiding to store which elements still need calculating), the entire segment is processed whenever a cell has changed during the previous processing (i.e., iteration) of the segment. Note that the algorithm first goes through cells by increasing the row and column indexes, which benefits the propagation of exposure values towards higher row and column indexes. To reduce the number of required iterations, the next processing iteration goes through cells by decreasing their indexes, which benefits the propagation of exposures towards lower row and column indexes. If a processing iteration of a segment resulted in no changes, no further processing iteration is performed. If the value within a shared column has changed since the processing has started, then this shared column is transmitted to neighbouring robots. Once a column has been received, it is compared to the local cells, and if the exposure of a cell is smaller than that of the corresponding local one, the value is saved while triggering the processing of the local segment.

As soon as every robot has processed their segment at least once, every robot has finished any processing, and no columns are currently transmitted, the path planning is completed. Note that the calculated exposure values at the start cells represent the minimal cumulative exposure to the goal cells.

```
extern int width;// segment width
extern int height;// = 30
extern int size;//= width*height;
extern unsigned long *radiationMap;//see Mapping
extern unsigned long *segment;//calculated exposure

void planning(){
  bool leftColumnChanged = false;
  bool rightColumnChanged = false;

  int e = 0;//segment element index
  int de = 1;//element selection direction

  bool has_changed = false;
  do{
    has_changed = false;

    for(; e >= 0 && e < size; e += de){
      unsigned long original = segment[e];
      unsigned long new_value = calculate(e);

      if(new_value < original){
        has_changed = true;
        segment[e] = new_value;

        if((e % width) == 0 ){
          leftColumnChanged = true;
        }else if( (e % width) == width-1 ){
          rightColumnChanged = true;
        }
```

[5.2.IV] Note that the current value is taken into consideration to avoid overwriting the initial processing condition (i.e., goal cells have an exposure of zero).

```
        }
      }

      de *= -1;//turn around processing direction
    }while(has_changed);

    if(leftColumnChanged)  transmitLColumn();
    if(rightColumnChanged)  transmitRColumn();
  }

  unsigned long calculate(const int e){
    unsigned long out = segment[e];
    unsigned long local_rad = radiationMap[e];

    const int neighbours[] = {-width, 1, width, -1};

    int neighbour = e-width;//top neighbour
    if(neighbour >= 0)
      out = min(exposure(e, neighbour), out);

    if((e % width)-1 >= 0)//left neighbour
      out = min(exposure(e, e-1), out);

    neighbour = e+width;//bottom neighbour
    if(neighbour < size)
      out = min(exposure(e, neighbour), out);

    if((e % width)+1 < width)//right neighbour
      out = min(exposure(e, e+1), out);

      return out;
  }

  unsigned long exposure(const int p, const int n){
    if(segment[n] != 0xFFFFFFFF ){//has it been calculated
     ↪  already?
      return (radiationMap[p] + radiationMap[n])/2 +
       ↪  segment[n];
    }
    return segment[p];
  }
```

Listing 5.4: The implementation of the used variation of the Floyd–Warshall algorithm. Note that the algorithm, implemented in `planning`, is executed within a thread as it is expected to have a long run time. `calculate` shows how the current cell's exposure value is calculated and `exposure` demonstrates how the cumulative exposure value is calculated between two points. Note that the exposure value for cells that have not been calculated is set to `0xFFFFFFFF`. `transmitLColumn` and `transmitRColumn` are functions that transmit the shared columns on the left (i.e., lower column index) and on the right (i.e., higher column index).

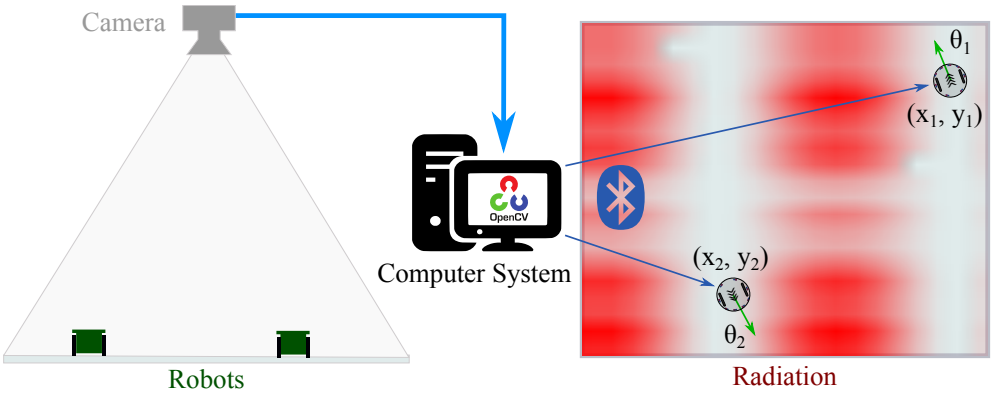


Figure 5.3: Experiment Setup.

5.2.2.3 Path Following

To extract the path from the segments, the robot containing the start cells in its segment transmits the start cell with minimal exposure. The next cell within the path is the cell adjacent to the current cell with the smallest accumulated exposure value. Every cell is transmitted until the goal cells have been reached or if adjacent cells have higher values (i.e., a shared column has been reached). In the latter case, the robot which shares that column would then continue to transmit the next cells. In the former case, the robots start following the path.

During the path following, the robots move from cell to cell and record their exposure. When all robots reach their goal cell, the recorded accumulative exposure is logged.

5.2.3 Implementation

The described behaviour is implemented on OpenSwarm (v0.19.09.07) with SwarmCom (as presented in Chapter 4). SwarmCom is configured to transmit 850 bps within 25 cm. This enables the fastest data throughput while maintaining a sufficient communication range (see Section 4.5.1.3). Due to the cell size of 10 cm width and height, the inter-robot distance for two robots is between 7 and 20 cm. As the behaviour ensures that the robots remain static during transmissions (i.e., remain on their start cells), SwarmCom operates within a region of high probability of error-free transmission of $1-10^{-16}$, which is based on findings in Chapter 4.

5.2.4 Experiment Setup

The experiment is conducted in a 3×3 m arena with light grey floor bound by 50 cm tall white walls. The arena is illuminated, as described in Chapter 4. The radiation patterns within the arena are calculated by (5.1) with $i \in \{1, 2, \ldots, n_i\}$ and $j \in \{1, 2, \ldots, n_j\}$, where the parameters are uniformly randomly chosen from $n_i, n_j \in \{1, 2, \ldots, 5\}$, $k_i \in \left[\frac{\pi}{W}, \pi\right)$, $k_j \in \left[\frac{\pi}{H}, \pi\right)$, and $\phi_i, \phi_j \in [0, 2\pi)$.

As a robot needs to measure location and radiation within the arena, the environment is augmented with the help of a computer system, as shown in Figure 5.3. To detect the robot's location, a 3264×2448 camera[5.2.V] captures frames from a bird's eye view. With OpenCV 4.1, the two-coloured markers[5.2.VI] of each robot are detected, and their location and orientation calculated. Based on the previously recorded location, the closest markers are assigned to the respected robot, which allows the tracking of robots. Finally, the location and orientation are sent to the robots via Bluetooth. On the robots, the messages are converted to events, which is used to simulate a virtual localisation sensor (such as GPS). The virtual radiation sensor is

[5.2.V] The camera is a Svpro SV-USB8MP02G-SFV.
[5.2.VI] Each marker is mounted on top of the robot facing upwards to be detectable by the overhead camera.

calculated from the generated radiation patterns and the calculated location. With every newly calculated location, the new radiation value is also transmitted to the respected robot. The use of the computer system is a necessary tool to provide virtual sensor data. However, it does not influence the decentralised character of swarm robotics as it only acts as a sensor and has no other influence over the behaviour or its execution.

5.2.5 Experiment Procedure

The experiment consists of 18 trials in which 2, 4, and 6 robots perform the previously described solution. First, the robots are placed at random locations in the arena, and, after the computer system has been configured[5.2.VII], the experiment is started. At the start, the computer system generates a random radiation map (e.g., see Figure 5.3) and, then, instructs the robots to move to their start cells. When all robots have reached their start cells, the robots start their behaviour as described in Section 5.2.2. Finally, the software detects if all robots have reached a goal cell (i.e., termination condition). The computer system then records the experiment results and starts a new experiment automatically.

Overall, there is no limit to how long a trial can last. However, in case a robot restarts, switches off, or loses its connection to the computer system, the trial is aborted and restarted. Note that this has not been experienced during all 18 trials.

In addition to the performed experiments, the experiment is repeated with a single robot remotely controlled by the computer system. The system initiates the mapping, performs the path planning, and guides the robot along the path. This is performed to provide a reference point to which the distributed approach can be compared.

5.2.6 Results

The outcomes of the recorded experiments are illustrated and discussed below.

5.2.6.1 Mapping

As the robot speed and the measuring is not affected by other robots, the overall mapping time, t_m, was recorded. Figure 5.4 shows t_m and that an increased number of robots reduces the mapping time, as expected. However, t_m has a lower bound based on the time a robot requires to move to and from the segment. Moreover, additional robots also increase the number of redundantly mapped cells (i.e., shared cells).

5.2.6.2 Path Planning

After the mapping, the arrival of all robots at their starting cells triggers the execution of the path planning. The path planning time, t_p, is the time from triggering the execution to the time when all robots agree that they finished processing their segments and no further shared cells are exchanged. Based on the number of transmitted shared columns per trial, n_c, the minimal communication time is calculated by

$$t_c = \frac{n_c \, H \, z_e}{b}, \tag{5.4}$$

where $H = 30$ cells, $z_e = 32$ bits per cell, and $b = 850$ bps are the grid height, memory consumption for the exposure value of a cell, and the transmission bitrate. Note that this is the minimal communication time as it does not contain collisions, retransmissions, or any other delays that could have occurred.

[5.2.VII] To configure the computer software, first, the camera distortion matrix is loaded and, then, the arena corners are set. Finally, the connection between robots and computer system is established.

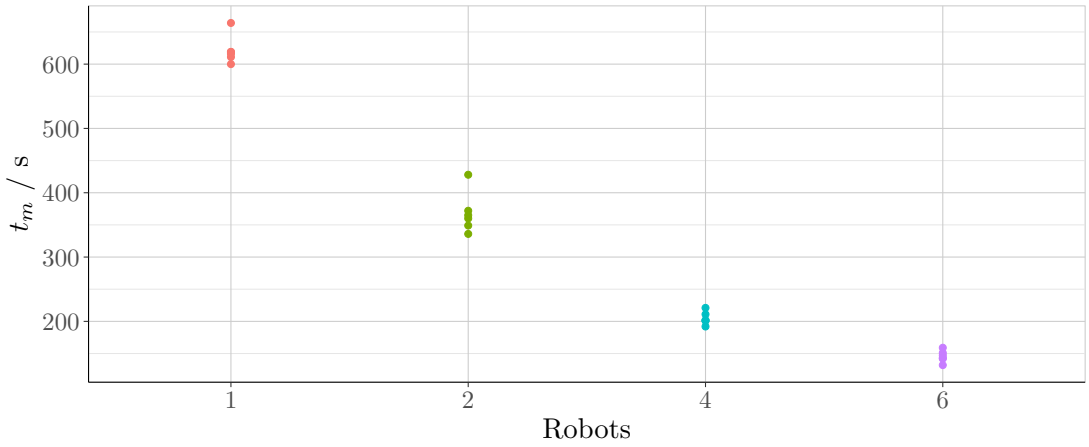


Figure 5.4: The recorded mapping times, t_m, for 1, 2, 4, and 6 robots.

Figure 5.5 shows the path planning time, t_p, and its increase with additional robots. While it could have been expected that the time decreases through parallelisation of the segment processing, Figure 5.5a shows that the majority of the time is spent on communication (i.e., t_c), which serialises the processing time. As the difference between t_p and t_c includes potential retransmissions and communication delays as well as the decision making to agree on the end of the planning, $t_p - t_c$ cannot be denoted to the processing time. When looking at the number of cells that have been processed by any robot, n_P, Figure 5.5b shows that n_P does not follow the trend of t_p; hence, it can be expected that the overall processing time does not follow the trend of t_p, too. As a result, the dominantly contributing component is likely to be communication. Note that segments can be processed while transmissions are conducted.

Finally, in each trial, the computer system performed the Dijkstra algorithm in parallel to the robots and compared the resulting exposure values of robots and computer system. In 100 % of the cases, the calculated exposure values were equal, which verifies that the deployed algorithm is correct and that communication errors do not occur or do not impact the results.

5.2.6.3 Path Following

After agreeing that every robot has finished their path planning, the robots share the path and, thereafter, follow it to the goal cells. At arrival, the robots transmit the measured exposure to the computer system, which then compares it to the calculated values. On example is shown in Figure 5.7, which shows snapshots of 4 robots following the path illustrated in Figure 5.7f.

Figure 5.6 shows the deviation from the calculated exposure. While the majority of measured exposure values are equal to the calculated values, for a larger number of robots, the measured values differ up to 7.8 %. This can be explained by the fact that more robots are more likely to block or influence each other movements by collisions. One such collision is shown in Figure 5.7c and 5.7e. This most frequently occurred at the beginning when all robots moved to the first cell of the path.

5.2.7 Discussion

This section focuses on experiments that demonstrated that data could be stored and processed distributively by multiple severely-constrained robots. Each trial was conducted with 2, 4, and 6 robots. As the robot's virtual sensors require Bluetooth, trials with larger numbers of robots could not be investigated; it would cause unpredictable latencies and unreliable connections due to the limitations of Bluetooth.

Each trial was split into three steps: mapping, path planning, and path following. The ex-

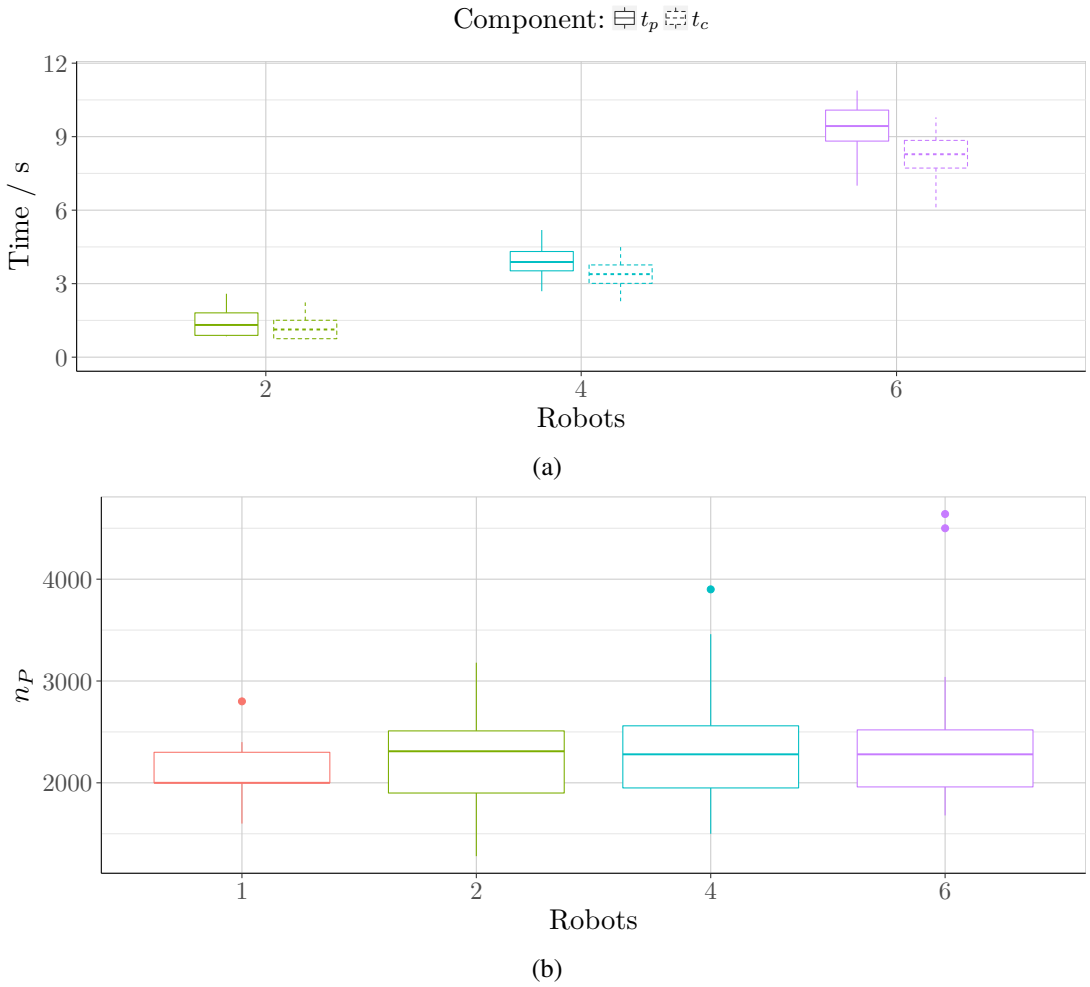


Figure 5.5: Box plots for (a) the path planning time, t_p, (a) the calculated communication time, t_c, and (b) the number of processed cells, n_P, for different numbers of robots.

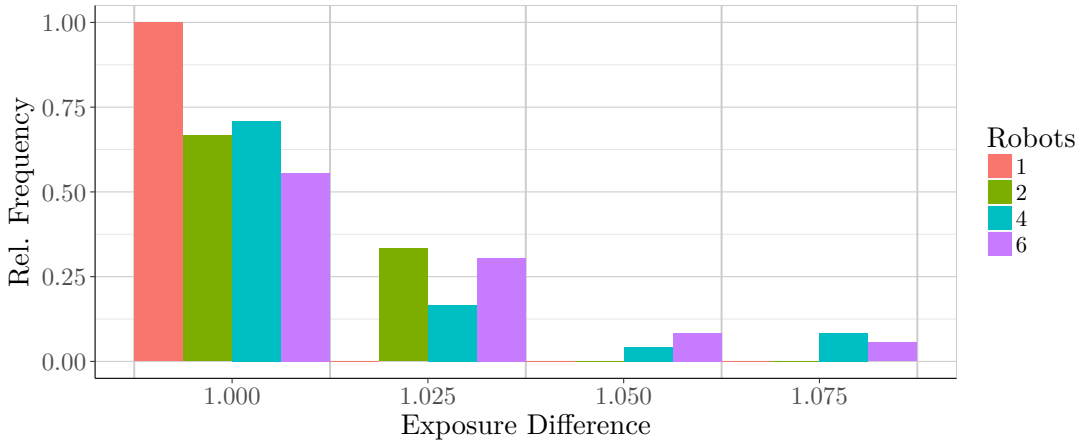
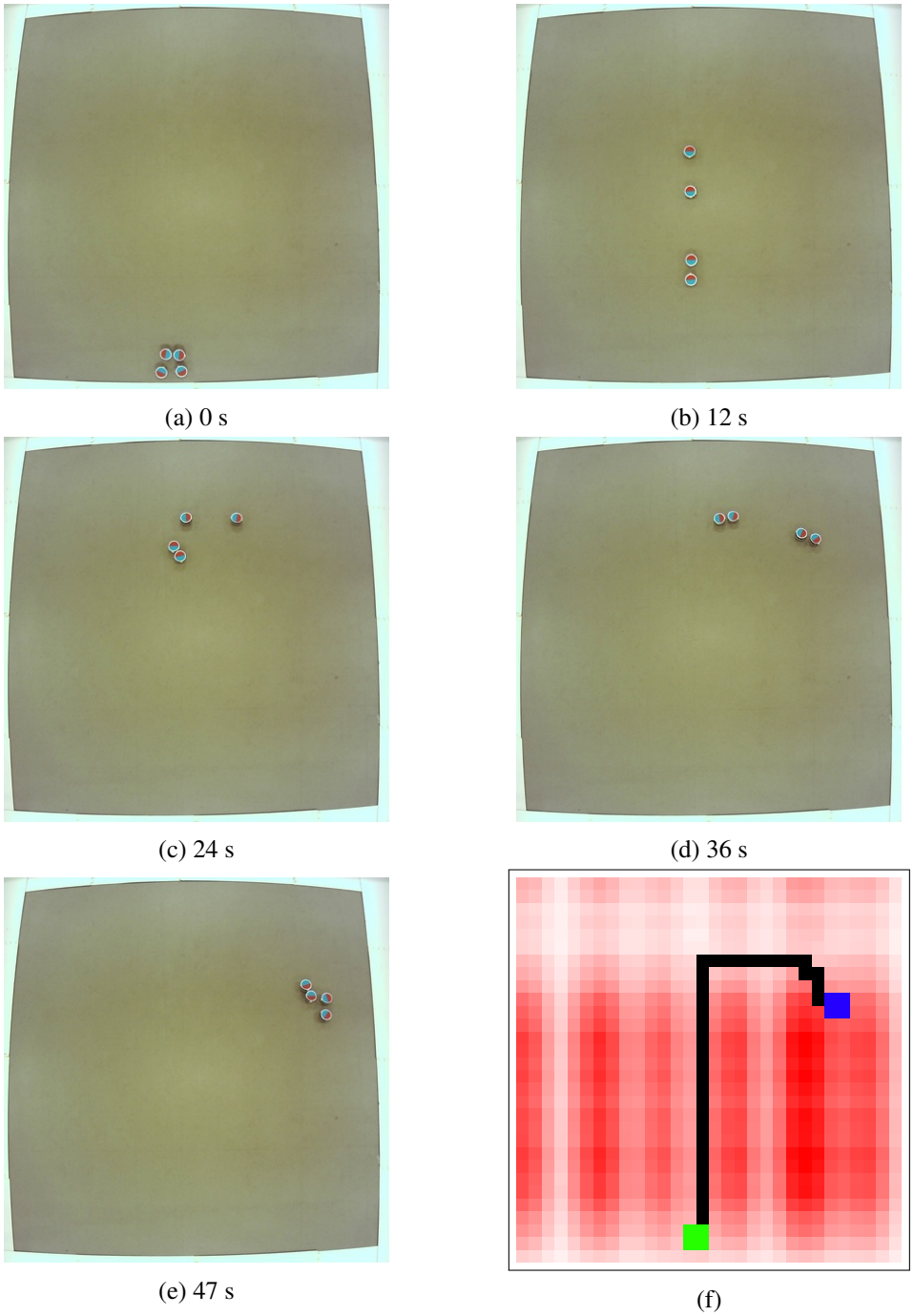


Figure 5.6: Histogram of the differences between calculated and measured exposure values.



(a) 0 s (b) 12 s

(c) 24 s (d) 36 s

(e) 47 s (f)

Figure 5.7: Snapshots of four robots following the calculated path. (a)-(e) Snapshots at different times. (f) shows the calculated path that should be followed. Not that, at 24 s, one robot collided with another causing it to be pushed away from the path.

periments have shown that the mapping is the most time-consuming activity. However, when using more robots, the time decreases to approximately 150 s (for 6 robots), which still is approximately 10 times larger than the time required to perform path planning. It was demonstrated that the run time of the path planning algorithm is to a large extent caused by the communication of the data. In other words, advancing communication systems on severely-constrained robots is likely to reduce the run times considerably.

While the calculated exposures had been verified to be correct, it was observed that robots often obstruct or divert each other's movement, which is the likely cause of the deviation of up to 7.8 % from the calculated exposures. Moreover, the experiment showed that a group of severely-constrained robots succeeded in solving a task that, in terms of memory requirements, was too difficult for any single robot.

5.3 Discussion

This chapter shows an approach whereby computation constraints of individual robots can be overcome by the distribution of data and processing. While the procedure to distribute solutions and algorithms is task-specific and cannot be generalised, the conducted proof-of-concept experiment has shown a viable method of overcoming computational constraints whereby a large set of data is split among a number of robots, locally computed, and shared data are exchanged. While this requires an algorithm that can be split and concurrently processed, the author believes that this result is meaningful within a broader context, and paves the way for more distributed computation and computational cooperation on severely-constrained robots. Furthermore, on a more general note, the following key properties have been identified.

- It has been demonstrated that remote procedure calls, in particular remote memory access, and consensus can be used to synchronise a robot's behaviour and make decisions. This prevents the use of centralised control infrastructure and facilitates the distributed nature of swarm robotics.
- The use of virtual positioning and radiation sensor demonstrated the benefits of simulated sensors and augmented environments. To the robot and the developer, the use of virtual sensors is transparent as they behave as if they were physical sensors. Overall, this expands the possibilities of how robots are used and how complex the environment can be.
- Finally, it was demonstrated that heterogeneous systems could be integrated. On the one hand, when the robot was performing the experiment, it was aided by the computer system with virtual sensors data. On the other hand, when the computer system was performing the experiment, it was utilising (i.e., remote controlling) one robot to extend its reach. Consequently, it enables the building of both system types, such as RaaS [Chen et al. 2010] or cloud robotics [Wan et al. 2016].

While the described properties open the possibility of novel uses of swarms and their computational resources, the distribution of OpenSwarm comes with side effects. For instance, the design of software and algorithm becomes more difficult as they are distributed as discussed in [Selic 2000]. Moreover, it was shown that the choice of algorithm depends not only on computational resources but also on the amount of data that is exchanged — resulting in a trade-off between computational resources and communication. Furthermore, additional theories and established problems within distributed systems research require consideration. The CAP theorem is one such problem, which states that the system (in this case, a swarm) can only have two attributes out of three (i.e., *consistency*[5.3.I], *availability*[5.3.II], and *partitioning tolerant*[5.3.III]). Therefore, further research with regard to network and computational constraints

[5.3.I] When data is stored on multiple devices, the data is consistent if all copies of that data are equal.
[5.3.II] Data is available if it can be accessed.
[5.3.III] Partition tolerance is when a system maintains function even when a device within the system malfunctions.

could benefit swarm robotics.

In addition, the experiments showed that the majority of the processing time is spent on communicating data between robots. This communication bottleneck can be lifted in two ways. First, by increasing the computational power, which would result in weakly-constrained robots, the communication demand can be reduced, going towards the extreme of a robot performing the task individually. However, this would not benefit the large set of severely-constrained robots, which is the focus of this work. Alternatively, further research on short-range optical communication, as used on swarm robots, could increase throughput allowing more data to be exchanged more swiftly.

Overall, the presented implementation constitutes the first step to combine resources on severely-constrained robots with constrained networking. Further research on networking, specifically with regard to the requirements in swarm robotics, could significantly improve the performance of such a system and enable the use of more advanced and already established concepts from other areas, such as distributed systems. The usage of data and process redundancy, for instance, could make the computation on swarms more robust.

Conclusions

Swarm robotics is a new and promising discipline with many potential applications. However, it has not been able to transition from the academic environments to the real world. In an attempt to narrow this gap, this work investigates a potential cause of this imbalance between the conceptual and the practical — computational constraints.

This thesis explores how computationally-constrained robots can collectively solve problems that are computationally too demanding for individual robots. A presented case study has shown that individual computational constraints can be overcome by distributed computation, which combines local computation and the communication of data between robots. While this study is not a universal solution to all the complex challenges within swarm robotics, the presented approach, whereby large data is distributively collected, stored, and processed, is a step closer towards the solution. The author believes that the results are meaningful within a broader context as they open an intersection between research on distributed systems and swarm robotics, and pave the way for further work that would bridge the gap between academia and the real world.

To enable research on distributed computation, first, this thesis investigated the computational constraints of individual swarm robots. To quantify computational resources, a computational index is proposed. A quantitative study based on 5264 devices revealed different computational classes of these devices. This makes the comparison of robots easier and can be used in future work to show trends with respect to a robot's resources. In particular, this thesis analysed a large number of swarm robots showing that the majority of them are severely-constrained, which means they provide the fewest resources of all robots. When considering that inefficient use of resources affects the most robots with few resources, the execution and the programming of software become a crucial aspect of how complex the behaviours are.

Furthermore, to the author's knowledge, this thesis presents the design, implementation, and study of the first operating system designed for and deployed on severely-constrained robots, OpenSwarm. It provides a novel dual-execution model, which results in the reduction of the computational overhead depending on run-time requirements. In addition, it was shown that OpenSwarm has a comparably small memory footprint and, when compared to two other systems (i.e., ASEBA and SCF), it outperforms them. On the other hand, implementing behaviours can be more complicated and require more code in OpenSwarm than in other system software, such as ASEBA or Buzz. This stems from additional layers of abstraction that these languages provide. In general, this shows that, at least, for the compared systems, there is a trade-off between the level of abstraction and the execution-efficiency. In other words, systems with fewer resources might benefit more from execution efficiency; while less-constrained systems can afford the additional layers of abstraction making the development of behaviours easier. In a broader context, this has shown that the choice of system software can affect executed behaviours, even when a computationally minimalistic approach is used.

OpenSwarm can be used to implement event-driven behaviour. While other robotic system software (e.g., ASEBA or URBI) also provide local events, OpenSwarm's events are based on a message-passing mechanism. As a result, events can be transmitted to other robots enabling

the distributed access to data and processes. Furthermore, this work demonstrated that this capability of distribution is a key to overcoming of the computational constraints of an individual robot.

As demonstrated in this work by combining distributed events with the provided hardware abstraction, external data can be used to situate the robot within an augmented/virtualised environment. While that requires a sufficient network throughput, it can be beneficial for the development process as sensors and actuators can be emulated transparently. However, a much greater benefit lies in the capability to integrate external resources with the robot. The author believes this to be of significance within a broader context; robots could receive preprocessed or simplified data, which could facilitate the performing of decision making and behaviours with fewer internal resources. Potential future work could look into the aspects of such a merger, for instance, whereby swarm robotics is integrated with and acts on information provided by IoT or cyber-physical systems. This would link the research of this thesis with other research, such as TerraSwarm.

Another aspect that requires consideration when performing distributed computation is the network between robots as it defines computation time and error as well as the uncertainty of execution. Therefore, this thesis presents the modelling, design, implementation, and study of a free-space infra-red MANET, SwarmCom. Its key property is the dynamic detector that adapts to the ambient light and the incoming signal strength, improving the quality of service by orders of magnitude (i.e., it offers a more reliable communication). It was demonstrated that SwarmCom is competitive with regard to throughput and communication range and outperforms libIrcom in most of its configurations. While there are systems that outperform SwarmCom, they often require hardware alterations adding to the robot's complexity, costs and power consumption.

While those technical properties enabled the transmission of events and subsequently the proof-of-concept experiment, this work also investigated fundamental properties of free-space optical communication systems, such as SwarmCom and libIrcom. One such property is the mode of communication. It was shown that flooding significantly improves the connectivity of the network. While flooding is fundamentally not scalable and state-of-the-art routing protocols either have a significant communication overhead or cannot cope with frequent topology changes, there is currently no alternative to flooding. This opens the possibility of further research on routing protocols that should provide minimal overhead while being adaptive to frequent topology changes. Both qualities would be required as swarm robotics networks provide relatively low throughput while robots move swiftly in comparison to the communication range.

This work has shown that mobility has a negative effect on the quality of service. As this results from geometric properties and occlusion, these findings are relevant to many swarm robots, such as Colias, e-puck, i-Swarm, Kilobot, and r-one. Because the probability of error increases by orders of magnitude, it is recommended for any of these systems to cease motion during transmissions or to create software that is robust against communication errors.

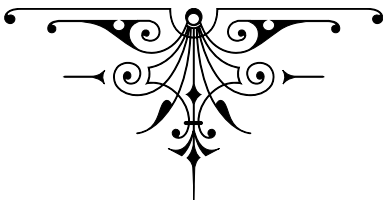
In contrast, it was demonstrated that the network's scalability benefits from occlusion and the short communication range. Experiments showed that the number of robots within communication range is bound enabling robots to communication even in high-density situations. In comparison, radio-based signals would reach any device nearby, resulting in potentially hundreds or thousands of robots exceeding the capability of any radio-based devices.

On a broader scale, the author is convinced that research on communication systems suitable for swarm robotics is the most overlooked aspect of the discipline. While this thesis introduced insights into fundamental properties of swarm robotics communication, there is significant room for further research. While there are many aspects in which further research would contribute to the field, from the author's perspective, there is a particular need for (I) technological advancements, as shown in [Khan 2017; Zhang et al. 2018], (II) the aforemen-

tioned routing protocols, and (III) more suitable approaches based on information theory. In comparison to existing communication systems which use large transmissions (e.g. 1500 bytes for Ethernet) that allow the use of sophisticated theory and practices, swarm robotics communication commonly uses short transmission (e.g. one or two bytes) where these practices are no longer efficient. In summary, further research on any of those aspects would not only reduce the in-this-work-identified communication bottleneck when distributing resources; it could improve cooperation of swarm robots in general.

Overall, this thesis demonstrated that the majority of robots have few computational resources and that the choice of system software has an impact on their performance. The communication on swarm robots is limited, yet the limitations can be overcome by distributively utilising the computational resources of swarm robots. While the experiments had been performed in an often simplified environment, the author believes that shining the light on computational resources, communication, and distributed computation paves the way for further research on these often-overlooked subjects. Looking forward, the author hopes that his contribution will, in the long run, enable swarm robotics to make the transition into the real world and result in more practical applications.

Appendix



"*Almost everything you do will seem insignificant, but it is important that you do it.*"

— Mahatma Gandhi

A

e-Puck Robot

Contents

One of the most used platforms in swarm robotics is the open-hardware miniature robot called e-puck[A.0.I] [Mondada et al. 2009]. It was designed at the École Polytechnique Fédérale de Lausanne (Switzerland) as a successor of the Khepera robot [Mondada et al. 1999]. E-pucks have been used in various areas of swarm robotics — for instance, to illustrate evolutionary algorithms [Pugh and Martinoli 2007; Li et al. 2016], fuzzy control [Mohammad et al. 2013], and collective transport [Chen et al. 2013]. This makes an e-puck an excellent candidate for this work.

A.1 Properties

Each e-puck is a cylindrical shaped robot with a diameter of $7\,\mathrm{cm}$ and a height of $5\,\mathrm{cm}$ as shown in Figure A.1. It can move on flat surfaces with its differential wheels. It provides a range of sensors and actuators listed in Table A.1.

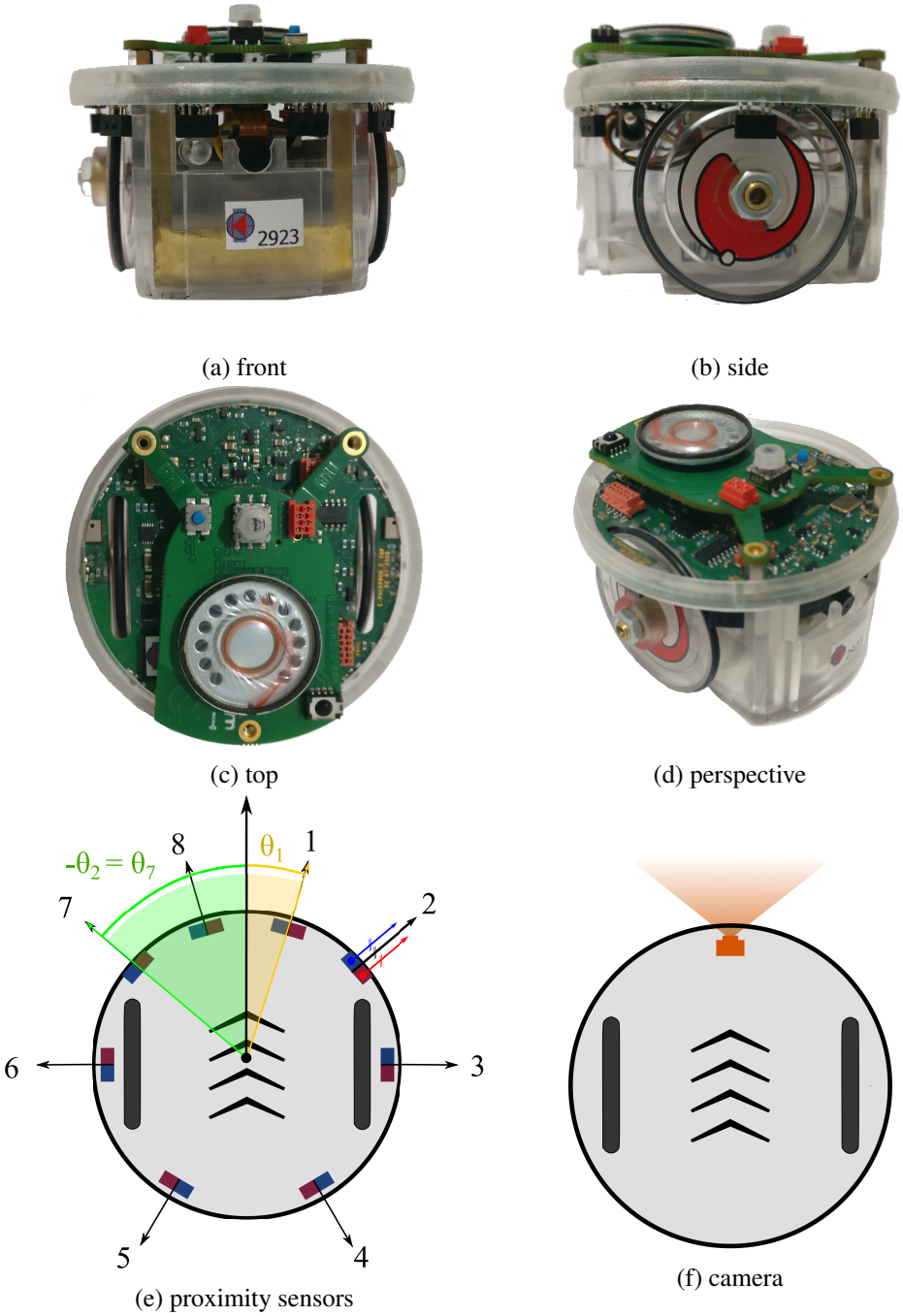
A.1.1 Computational Properties

An e-puck is equipped with a Microchip dsPIC30F6014a[A.1.I] as MCU. The MCU operates a 16 bit modified Harvard architecture that processes 14.75 MIPS. It provides $8\,\mathrm{kB}$ of RAM and $144\,\mathrm{kB}$ of ROM. With this processing unit, the robot has a computational index of $C_I = 17.7$ and is classified as a severely-constrained robot (C_1) as described in Section 2.1.

A.1.2 Physical Properties

In this work, three actuators/sensors are frequently used: the stepper motors, the proximity sensors, and the camera.

[A.0.I] For more details visit: www.e-puck.org (e-puck).
[A.1.I] For further information see [Microchip Technology Inc. 2018].



(a) front (b) side

(c) top (d) perspective

(e) proximity sensors (f) camera

Figure A.1: e-puck robot. (a)–(d) show an e-puck from different perspectives. (e) shows a schematic e-puck highlighting all 8 proximity sensors and their sensing directions as well as the orientation of the transmitter (red) and receiver (blue). (f) illustrates the relative position and orientation of the on-bard camera (orange).

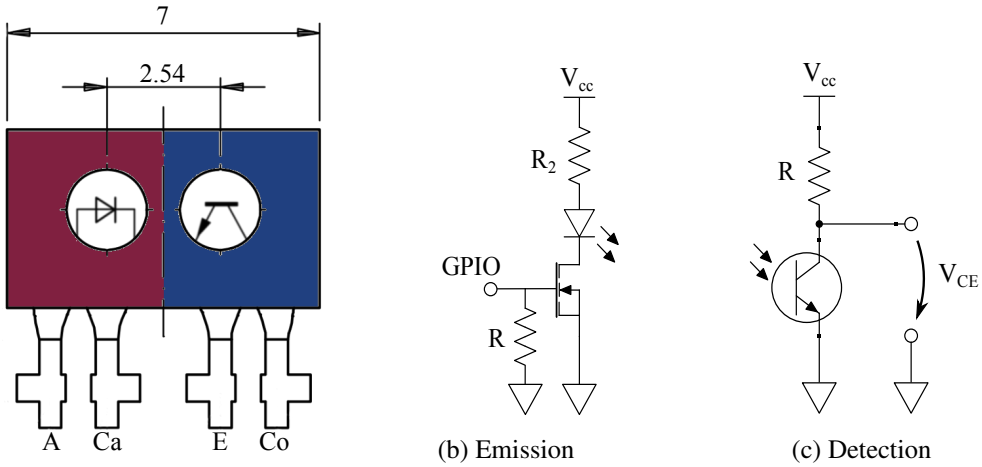
Table A.1: e-puck's sensors and actuators.

Type	Number	Feature
Indicator	1	red forward-facing LED
Indicator	1	green downward-facing LED
Indicator	8	red LED ring
Indicator	1	speaker
Propulsion	2	stepper motors
Sensor	3	microphones
Sensor	8	infra-red proximity sensors
Sensor	1	3-axis accelerometer
Sensor	1	infra-red remote control receiver
Sensor	1	640×480 RGB CMOS Camera
Sensor	1	4-bit rotary encoder switch
Communication	1	Bluetooth

A.1.2.1 Stepper Motors

Two independently operated stepper motors are connected to four GPIO[A.1.II] ports of the MCU. To move from one configuration to another (i.e., performing a step), the MCU changes the signals on these ports, which causes a periodically computational overhead. The robot can perform up to 1000 steps (i.e., 1 revolution) per second and, with a wheel diameter of $41\,\mathrm{mm}$, the maximum velocity is approximately $128\,\mathrm{mm\,s^{-1}}$.

A.1.2.2 Proximity Sensors



(a) TCRT1000 (b) Emission Circuit (c) Detection Circuit

Figure A.2: e-puck proximity sensor. (a) shows the physical shape of the sensor (in millimetres). (b) shows the e-puck's emission circuit for the LED (red). (c) shows the e-puck's detection circuit for the phototransistor (blue). Note that circles indicate a connection to the MCU, R indicates a $10^5\,\Omega$ resistor, and R_2 indicates a $18\,\Omega$ resistor.

The proximity sensors are positioned and oriented as shown in Figure A.1e and Table A.2. Each sensor is a Reflective Optical Sensor (TCRT1000) from Vishay Electronics that is composed of an infra-red LED (red) and a phototransistor (blue) as shown in Figure A.2. The LEDs

[A.1.II] A general-purpose I/O port is a digital I/O port on an MCU that can be configured as output or input port and operated as such.

Table A.2: e-puck's proximity sensor locations in polar coordinates (r, θ) relative to the robot's centre. Note that the sensor orientation is equal to θ.

i	r / mm	θ / rad
0	32.5	0.308
1	34.5	0.860
2	32.5	1.570
3	32.5	2.618
4	32.5	−2.618
5	32.5	−1.570
6	34.5	−0.860
7	32.5	−0.308

are connected to GPIO ports and are illuminated by software. The transistor is connected to a time-multiplexed 12-bit ADC, which can sample with up to $100\,\mathrm{kHz}$ for a single channel or $6.3\,\mathrm{kHz}$ when all channels are sampled. The analog-to-digital conversion is conducted by an independent part of the MCU, which does not require processing time.

Based on the emission circuit of Figure A.2b, the emitted signal intensity is proportional to the voltage applied to the GPIO port. In other words, if the GDIP port outputs voltage, the LED emits the maximum signal intensity (i.e., which is limited by the circuit). Similarly, based on the detection circuit of Figure A.2c, the received light intensity, $y(t)$, is proportional to the phototransistor's collector-emitter current, I_{CE}. As a result, the measured collector-emitter voltage,

$$V_{CE} = V_{cc} - R\,I_{CE}, \tag{A.1}$$

is indirectly proportional to the incoming signal intensity (V_{cc} and R are the supply voltage and a resistor, respectively).

A.1.2.3 Camera

The e-puck is equipped with a 640×480 RGB pixel camera[A.1.III]. In addition to eight GPIO ports to transmit colour values, three additional connections trigger a new pixel, line, or frame interrupt. Handling these interrupts and storing the pixels is conducted by software and, therefore, the capturing itself causes significant computational overhead.

Due to the hardware design, the usage of the camera is impaired with significant drawbacks. Firstly, pixels arrive every eight instructions, which forces the MCU to collect an entire row of pixels without interruption. This monopolises the processing time and, if there is an interruption, data corruption is likely. Furthermore, an entire frame of 640×480 grey-scale pixels would consume 307.2 kB of memory. Considering the available $8\,\mathrm{kB}$ RAM or $144\,\mathrm{kB}$ ROM, the robot is incapable of storing or processing a single frame. As a consequence, the e-puck is *resource inadequate* — as defined in [Kopetz 2011].

A.1.3 Communication Properties

The e-puck provides two wireless communication methods — Communication via Bluetooth and via a proprietary infrared system called libIrcom.

[A.1.III] Depending on the model of the e-puck, the camera is a PixelPlus PO3030, PO6030, or PO8030.

A.1.3.1 Bluetooth

The Bluetooth component, LMX9820A [Semiconductor 2019] can send data byte-wise to and from the e-puck's microcontroller via UART1[A.1.IV]. These bytes are then sent/received transparently by the Bluetooth component to another connected Bluetooth device.

In general, Bluetooth is a *master-slave* protocol[A.1.V] based on the IEEE 802.15.1 standard [IEEE 2017]. Theoretically, it can send 0.732–2.2 Mbps within a communication range of 10–$3\,\mathrm{m}$. However, on the e-puck, a throughput of 0.018 Mbps was reported [Ecole polytechnique federale de Lausanne 2014].

On the e-puck, Bluetooth is used to program the robot or to connect from a computer. However, it has several disadvantages with respect to swarm robotics. (I) The master-slave character requires the networks to be hierarchical, which acts against the peer-to-peer character of swarm robotics. (II) Bluetooth allows having up-to 255 devices and only 7 active slaves within one piconet. This hinders robots to form large networks. (III) Each piconet has a unique frequency channel (one of 39) which limits its scalability. This limits the number of robots that can be connected. However, the frequency overlap between Wifi and Bluetooth reduces the number of available channels considerably. Furthermore, *frequency-hopping*[A.1.VI] can reduce the performance considerably.

A.1.3.2 libIrcom

libIrcom is designed as a peer-to-peer network that uses infrared signals emitted and received by the proximity sensors [Gutiérrez et al. 2009*b*]. Two e-pucks can communicate within line-of-sight of up-to $19\,\mathrm{cm}$[A.1.VII] radius with a measured 200 bps[A.1.VIII].

For signal detection, 32 measurements are taken to detect 4 or 8 pulses representing the symbols 0 and 1, respectively. Even though this procedure is described as frequency modulation, the system performs channel coding that uses two codewords — 1100110011001100 and 1010101010101010 — to represent 0 and 1, respectively. This encoding provides a *Hamming distance*[A.1.IX] of 8 increases its robustness against noise. However, a Hamming distance of 8 is relatively low[A.1.X] considering that only two of $2.15 \cdot 10^9$ possible codewords are used. Furthermore, each byte is transmitted with 2 CRC[A.1.XI] bits to allow error detection. No further error correction has been implemented. To avoid package *collisions*[A.1.XII], each robot uses CSMA.

[A.1.IV] A Universal Asynchronous Receiver Transmitter (UART) is an circuit realising a serial communication to other electronic components on a PCB.

[A.1.V] A master-slave protocol describes communication systems, where a singe node establishes and controls the connections to slave nodes. The master also controls the information flow to and from nodes.

[A.1.VI] Frequency-hopping describes situations where channels are changed from one frequency to another due to faults or to access different piconets. When performed often, this method reduces the performance of the channel significantly.

[A.1.VII] Note that Gutiérrez et al. [2009*b*] reports $25\,\mathrm{cm}$ of communication range. However, this value could not be reproduced.

[A.1.VIII] Note that Gutiérrez et al. [2009*b*] reports 30 bytes per seconds (i.e., 240 bps). However, this value could not be reproduced.

[A.1.IX] The Hamming distance is a value calculated from two codewords and indicates in how many symbols it differs from another code.

[A.1.X] For two codewords only, a Hamming distance should be close to 32, which would allow the optimal robustness against errors.

[A.1.XI] A cyclic redundancy check (CRC) is a method to detect errors where a vector of parity bits is extracted by a polynomial of the transmitted data. If the received parity bits and the calculated bits do not match, an error must have occurred.

[A.1.XII] A package collision describes a case where at least two transmitter send at the same time. As a result, the received data is unspecified.

A.2 Extensions

The e-puck's hardware provides several restrictions on computation, communication, and vision. Therefore, many extensions have been built to loosen these restrictions. Note that two sets of extensions are relevant for this work — computational and communication extensions.

The Linux, Gumstix, Pi-puck, and Xpuck extension boards are additional hardware increasing the robot's computational resources [Liu and Winfield 2011; Millard et al. 2017*a*; Jones et al. 2018]. Each board increases the computational index of the e-puck from $C_I = 17.7$ to 24.5, 26.5, 26.7, and 29.5, respectively. However, these extensions[A.2.I] consume between 280–665 mA, which is an approximate increase of 50-100 % in comparison to the entire robot.

As the communication on the e-puck (with libIrcom) is capable of 220 bps, the Zigbee and Range & Bearing extension boards are additional hardware used to enhance the e-puck's communication capabilities. Each board increases throughput and range to 250 kbps and $5\,\mathrm{m}$ as well as 5 kbps and $0.80\,\mathrm{m}$, respectively. Both boards are equipped with additional processors and increase the computational index to $C_I = 18.98$ as well as 20.06, and they consume $23.9\,\mathrm{mA}$ ($+4$ %) as well as $48\,\mathrm{mA}$ ($+8$ %), respectively.

A full list of available extensions can be found at [Mondada and Bonani 2018].

A.3 e-Puck 2

The e-puck 2 is a commercial upgrade released in the second half of 2018. The mechanical properties (i.e., size, wheels, and motors) are the same as the original. However, the processor was upgraded to a 32 bit STM32F407 (ARM Cortex M4) with 210 DMIPS instead of 15 MIPS. It also provides $192\,\mathrm{kB}$ of RAM and $1024\,\mathrm{kB}$ of ROM. This increases the computational index from $C_I = 17.7$ to 21.93 (i.e., close to the edge of severely-constrained robots). In addition, it provides a series of additional features — for instance, micro SD card slot, magnetometer, and USB connection.

[A.2.I] Note that the Xpuck extension provides additional battery ($+200$ %) but also consumes approx. $1191\,\mathrm{mA}$ ($+200$ %).

B

OpenSwarm Implementation Details

Contents

This chapter extends Chapter 3.2.5 and provides additional details on implementation and usage. The presented implementation is deployed on the e-puck which operates a Microchip dsPIC30F6014A with 8 kB RAM, and 144 kB ROM. The MCU is driven by a quartz crystal oscillating at 7.3728 MHz and it processes 14.7456 MIPS[B.0.I]. To improve the convenience for experienced e-puck programmers, names and labels of hardware have been taken from the e-puck library from `e_epuck_ports.h`, `e_init_port.h`, and `e_init_port.c` [Ecole polytechnique federale de Lausanne 2014]. Finally, the source code is compiled by Microchip's MPLAB XC16 Compiler v1.31 (released 20.02.2017 for Linux 64-bit).

OpenSwarm provides an application programming interface (API) for C and does not alter the programming language in any way. However, it provides multiple layers of abstraction and additional features that are intended to facilitate the development of swarm robots. OpenSwarm features are provided by system calls, which are described below.

B.1 System Calls

System calls are functions that access and provide features of the kernel of OpenSwarm. They can be used to execute, manage, communicate with, and synchronise processes and events. A selection of the system calls are presented in Table B.1.

[B.0.I] 14.7456 MIPS result in 67.8 ns per instruction.

Table B.1: A selection of available system calls in OpenSwarm. A complete list (including arguments and return values) can be found in [Trenkwalder 2020*c*]. Note that system calls in green are functions which require communication; in this work, they use SwarmCom.

Group	System Call	Description
Initialise & Start	`Sys_Init_Kernel`	initialises OpenSwarm
	`Sys_Start_Kernel`	starts all functions of OpenSwarm
	`Sys_Run_SystemThread`	executes the System Thread indefinitely
Task Management	`Sys_Create_Process`	creates a process based on an arbitrary function
	`Sys_Kill_Process`	terminates a process forcefully
	`Sys_Yield`	cooperatively yields the current process to execute another
Event Management	`Sys_Register_Event`	tells OpenSwarm that an event with a unique ID can occur
	`Sys_Unregister_Event`	removes a registered event from the system
	`Sys_IsEventRegistered`	checks if an event has been registered
	`Sys_Subscribe_to_Event`	subscribes an event handler to a specific event
	`Sys_Unsubscribe_from_Event`	removes the event handler
Interprocess Communication	`Sys_Send_Event`	emits an event (default is synchronous emission)
	`Sys_Send_SyncEvent`	emits an event synchronously
	`Sys_Send_AsyncEvent`	emits an event asynchronously
	`Sys_Wait_for_Event`	blocks a process until the specified event occurs
Execution		

Table B.1: System calls of OpenSwarm. (continued)

Group	System Call	Description
Interprocess Synchronisation	`Sys_Init_Semaphore`	creates and initialises a semaphore with a starting value
	`Sys_Acquire_Semaphore`	decreases the semaphore counter and might block process
	`Sys_Release_Semaphore`	increases the semaphore counter and might release other processes
	`Sys_Start_CriticalSection`	prevents the scheduler from being executed
	`Sys_End_CriticalSection`	returns to a normal scheduling behaviour
	`Sys_Start_AtomicSection`	prevents any interrupt from occurring
	`Sys_End_AtomicSection`	returns to a normal scheduling behaviour
Distributed Extension	`Sys_Register_GlobalEvent`	registers a global event
	`Sys_Send_GlobalEvent`	emits a global event
	`Sys_GlobalSet`	stores a value to a memory across a network
	`Sys_AgreeOn`	tries to achieve consensus on data across a network

B.1.1 Initialise and Start

Before OpenSwarm can be used, it requires initialisation by calling `Sys_Init_Kernel`. This performs the following tasks:

- establishing names and definitions of hardware pins and ports of the MCU,
- creation of the default process, called System Thread
- configuration of the System Timer, which is used to preempt processes,
- configuration of the I/O Timer, that is needed for polling on certain I/O devices,
- initialisation of all available I/O devices and modules, and
- registration of all OpenSwarm events.

After initialising OpenSwarm, additional processes and events can be created and registered. It is worth noting that processes and events are not executed or emitted before the system has started.

When the system is ready, `Sys_Start_Kernel` starts all the initialised and configured elements of OpenSwarm. All interrupts are enabled, events can be emitted and processes are executed.

`Sys_Init_Kernel` and `Sys_Start_Kernel` were separated to provide the user with more flexibility when it comes to preparing the system. For instance, processes can be created before OpenSwarm was started. In this situation, the concurrent execution of all processes starts at the same time. This can be important to measure performances (e.g., throughput). Similarly, events can be registered before the execution of any process. This prevents the loss of events before the event handler is subscribed.

After starting all functions, it must be guaranteed that the System Thread continuously performs its action. If the System Thread would terminate, the MCU performs a reboot, which would disrupt the robots behaviour. Therefore, `Sys_Run_SystemThread` executes indefinitely.

B.1.2 Process Management

To add a new process, `Sys_Create_Process` allocates the process image (i.e., PCB, stack, and event register). The process image is created in such a way that a function that should be executed as a process is called when the process is executed for the first time. During the creation, a unique identifier, $processID$, is assigned to the process, which serves as a reference across OpenSwarm. Once the process is due, the function that was passed as a parameter is executed. When this function ends or the process should be terminated, `Sys_Kill_Process` referencing $processID$ is called.

It is worth noting, that without any user processes, the System Thread is processed without preemtption. In case multiple processes are available (i.e., *ready*), each process is executed for a defined period (i.e., $50\,\mathrm{ms}$ as default). However, if a process needs to be prematurely scheduled, it yields with `Sys_Yield`. For instance, the System Thread yields after every iteration of its functions. This improves the throughput by avoiding necessary iterations of sporadically needed functions.

B.1.3 Event Management

Before an event can be used, first, it must be registered by using `Sys_Register_Event`. This functions allocates an event registration struct (i.e., a linked list element) that is appended to a linked list of registered events. Similarly, `Sys_Unregister_Event` is used to delete the associated struct and to remove an event from the system. Also, the user can check if an event has been registered with `Sys_IsEventRegistered`.

If users want to obtain events, they can either subscribe a handler function (i.e., callback function) to that event with `Sys_Subscribe_to_Event` or can block a process with `Sys_Wait_For_Event` until that event arrives. Both functions provide the possibility to

define a second callback function, called condition, that can preselect events. As a result, unwanted processing of events can be avoided increasing the system's efficiency. For instance, when considering an obstacle avoidance algorithm, a condition can be formulated that executes an obstacle avoidance algorithm only when obstacles are closer than a certain threshold.

To avoid execution of an event handler, it can be removed with `Sys_Unsubscribe_from_Event`. Note that one event handler can only be subscribed once to a specific event.

B.1.4 Interprocess Communication

To communicate with other processes or to perform cooperative execution, data can be send with the non-blocking functions, `Sys_Send_AsyncEvent` for asynchronous and `Sys_Send_SyncEvent` for synchronous sending. When using asynchronous events, data is buffered by OpenSwarm and executed asynchronously between the scheduling of two processes. This prevents the consumption of execution time of the running processes. However, this causes an unpredictable time-delay (i.e., jitter). For example, the worst case delay happens when the event is emitted immediately after a process has been scheduled (i.e., the delay is the full scheduling period). These delays are particularly problematic for time-critical execution.

For time-critical execution, the unbuffered synchronous event execution is preferable, because the event handler is directly executed after emission of the event. Therefore, the response time is minimal. However, the event is processed within the context of the current process or Interrupt Service Routine (ISR), which consumes processing time or extends the execution time of the ISR. Per default, the generic `Sys_Send_Event` uses the synchronous `Sys_Send_SyncEvent` to transmit data.

When sent, events can then be obtained by an subscribed event handler or blocking function that has been waiting for the event (see above). It is worth noting that `Sys_Wait_for_Event` not only unblocks when the event occurs, it also returns a pointer to the obtained data. Thereafter, the process continues its execution when rescheduled.

B.1.5 Interprocess Synchronisation

Running processes might contain sections that manipulate shared data or changes values of hardware registers. When these sections are interrupted, they can lead to malfunction and/or data corruption. To avoid these interruptions, processes or handler functions can declare such sections as atomic (i.e., uninterruptible) with `Sys_Start_AtomicSection` and `Sys_End_AtomicSection`. Within these two commands, any interruptions are prevented and if an interrupt occurs, its ISR is postponed to the end of the atomic section.

While protecting sections from malfunctions or data corruption, time-critical ISR might not be able to execute their code in time, which can lead to malfunction or data corruption as well. Hence, it is good practice to protect only short sequences of code, if possible.

To minimise the impact of atomic sections, `Sys_Start_CriticalSection` and `Sys_End_CriticalSection` provide a weaker form — called critical sections. This section only prevents the rescheduling of processes (i.e., System Timer interrupt). Both atomic and critical sections influence the execution indirectly by starving other pieces of code of execution time. As a result, these sections can monopolise the processing unit, if used excessively.

In OpenSwarm, direct interprocess synchronisation can be done by semaphores. Once created with `Sys_Init_Semaphore`, a semaphore contains a value[B.1.1] and a queue of processes waiting to be executed. This structure is stored in a doubly linked list and can be removed from the system by `Sys_Delete_Semaphore`.

When a process wants to access a resource protected by a semaphore, it tries to acquire a semaphore by calling `Sys_Aquire_Semaphore`. If it is successful, the process continues its execution. Otherwise, the process is blocked until enough processes released the semaphore

[B.1.1] The initial value of the semaphore defines how many processes can be used in parallel.

with `Sys_Release_Semaphore`. For further information how semaphores can be used, see [Downey 2005].

B.2 System Events

OpenSwarm provides a series of predefined platform-independent and platform-dependent events (see Table B.2). Each event has a unique event identifier, $eventID$, and can be used to distribute (source) or obtain (sink) data of a specific type and within a certain range. Platform-independent events are available on any deployed OpenSwarm, while platform-dependent events are only available if OpenSwarm incorporates specific modules.

Two platform-independent events are present in OpenSwarm— `SYS_EVENT_REBOOT` and `SYS_EVENT_10ms_CLOCK`. `SYS_EVENT_REBOOT` is a data-free event that initiates a reboot of the entire system. It is worth noting that rebooting takes several milliseconds and should only be used in exceptional circumstances. `SYS_EVENT_10ms_CLOCK` is an event that occurs periodically every $10\,\mathrm{ms}$. It also contains the value of the current system time in milliseconds.

In OpenSwarm, platform-dependent events are used by I/O modules to provide a function (e.g., distribute sensor values or perform an actuation). As a result, these event are closely related to the module's function and are discussed together with their I/O modules in the next section.

B.3 I/O Modules

In OpenSwarm, each I/O device is operated by the MCU, which requires OpenSwarm to manage it through I/O modules[B.3.1]. The accelerometer, microphones, and proximity sensors provide analogue values and, hence, are converted by the on-chip ADC.

B.3.1 ADC Module

The e-puck's MCU provides 16 convertible analog pins, where each analog signal on one of the pins is converted by the successive approximation register ADC. Each input channel (i.e., signal on a specific pin) is sequentially converted. After converting all input channels, an ADC interrupt signals that the ADC buffer has been filled. The hardware-specific function obtains these values and executes pre-processors sequentially. The pre-processor that is executed for a channel has been registered to it by other I/O modules.

B.3.2 Bluetooth Module

The Texas Instruments LMX9820/LMX9838 (i.e., Bluetooth device) is connected to the UART 1 on the e-puck's MCU. As a result, the Bluetooth module manages the UART 1 device with its device-specific handler.

This module manages the interrupt-driven hardware that reads and writes via the UART interface. When a byte is obtained, the reading interrupt executes the RX processor, which analyses the obtained byte or sequence of bytes. Subsequently, the results are emitted as an event `SYS_EVENT_IO_FROM_BLUETOOTH`. As soon as data should be transmitted, a process emits `SYS_EVENT_IO_TO_BLUETOOTH`. This data is then copied onto a writing buffer (i.e., linked list containing any-sized data). The buffer is emptied when the UART 1 device is ready to receive more data. This transfers the data to the component that does the radio transmissions.

[B.3.1] It is worth noting that OpenSwarm v0.17.09.25 does not currently manage the accelerometer, speaker and microphones.

Table B.2: OpenSwarm's available events. A complete list can be found in [Trenkwalder 2020*c*].

Module	Event Label	*eventID*	Direction	Data Type	Value Range
—[a]	SYS_EVENT_REBOOT	0x01	sink	—[b]	—
—[a]	SYS_EVENT_10ms_CLOCK	0x02	source	**unsigned long**	0 – 4 294 967 295 ms
Motors	SYS_EVENT_IO_MOTOR_LEFT	0x03	sink	**unsigned char**	-128 – 128 mm s^{-1}
Motors	SYS_EVENT_IO_MOTOR_RIGHT	0x04	sink	**unsigned char**	-128 – 128 mm s^{-1}
Camera	SYS_EVENT_IO_CAMERA	0x05	source	syscolour[c]	—
Remote Control	SYS_EVENT_IO_REMOTECONTROL	0x06	source	**unsigned char**	command
Bluetooth	SYS_EVENT_IO_TO_BLUETOOTH	0x07	sink	**char** *	data
Bluetooth	SYS_EVENT_IO_FROM_BLUETOOTH	0x14	source	**unsigned char**	data
Selector	SYS_EVENT_IO_SELECTOR_CHANGE	0x08	source	**unsigned char**	0 – 15
Proximity	SYS_EVENT_IO_PROX_0	0x0A	source	**unsigned short**	0 – 100 mm
Proximity	SYS_EVENT_IO_PROX_1	0x0B	source	**unsigned short**	0 – 100 mm
Proximity	···	···	source	**unsigned short**	0 – 100 mm
Proximity	SYS_EVENT_IO_PROX_7	0x11	source	**unsigned short**	0 – 100 mm
Proximity	SYS_EVENT_IO_PROX_ALL	0x09	source	—[d]	—
Infra-red Communication	SYS_EVENT_COM_RX_MSG	0x12	source	**unsigned long**	data
Infra-red Communication	SYS_EVENT_COM_TX_MSG	0x13	sink	**unsigned long**	data

[a] This is an platform-independent event.
[b] SYS_EVENT_REBOOT does not transmit data. Its occurrence signals OpenSwarm to initiate a reboot.
[c] SYS_EVENT_IO_CAMERA provides in its current setup a single color information. Its value is one of red, green, blue, white, or black.
[d] SYS_EVENT_IO_PROX_ALL does not send any data. It signals that all proximity events have been emitted.

B.3.3 Camera Module

The Pixelplus PO8030[B.3.II] (i.e., camera device with up-to 640×480 pixels) is connected to three external timer interrupt ports (Timer 0, 4, and 5) to indicate new pixels, rows, or frames. Eight General-Purpose I/O (GPIO)[B.3.III] pins transfer pixel information from camera to the device. It is worth noting that the e-puck's MCU does not provide enough memory to store a single frame. As a result, the camera must be configured in such a way that the obtained data can be stored and processed by the robot.

When using the camera, an external interrupt signals the arrival of a new pixel. When this interrupt occurs, the pixel can be taken from the digital input port and stored in memory for. However, the hardware design prevents the use of it as starting the ISR takes longer than the pixel is available. As a result, the row interrupt is used to collect all pixels[B.3.IV] consecutively. Note that this is a time-critical execution and any delay or interruption can cause data corruption. To decrease the MCU load, the frame timer interrupt, which indicates rows that belong to a frame, is used to en/disable the row interrupt.

The data is collected by the hardware-specific handler. When a frame has been filled, a pre-processor is called to extract the information from it. This information is then emitted as `SYS_EVENT_IO_CAMERA` event. In the current version, the pre-processor extracts a single colour value (i.e., blue, red, green, black, or white) from the centre of view (used in Section 3.3.3).

Note that the hardware design of the e-puck prevents an efficient use of the camera due to the lack of memory, the time-critical and long-lasting collection of pixels, and the high CPU load when using it. Hence, features of OpenSwarm and other processes cannot be executed while the camera fetches a frame. This reduces the responsiveness of the system and might limit the number of usable modules.

B.3.4 Selector Module

The selector (i.e., rotary incremental encoder) is connected to four GPIO pins. These pins can be read at any time and does not provide interrupts. Therefore, the selector is periodically checked by its device-specific hander. When the measured value has changed, a `SYS_EVENT_IO_SELECTOR_CHANGE` event with its value is emitted.

B.3.5 Remote Control Module

The Vishay Semiconductors TSOP36230 (i.e., Infra-red Remote Control Receiver Module) is connected to a single digital external interrupt pin. When a remote control command is being received, it causes an interrupt, which enables the periodic measurement of the input pin. To obtain the sequence of bits, the device-specific handler is called by the I/O Timer. As soon as all bits are obtained, the pre-processor emits the value as an `SYS_EVENT_IO_REMOTECONTROL` event.

B.3.6 Motors Module

As the only actuating module on the e-puck, the motor module manages the two stepper motors. The left and right motor are connected to 8 GPIO pins (i.e., four pins per motor). Due to the hardware design, software switches from one configuration to the next (i.e., performing a step). This means the module's device specific handler is called periodically by the I/O Timer. To achieve a desired velocity, the module must ensure that the correct durations between steps as calculated and applied.

[B.3.II] Older versions of the e-puck robot contain a Pixelplus PO3030 or PO6030.
[B.3.III] A GPIO is a digital I/O port on an MCU which can be set/unset and read by the running program. It can be configured as output and as an input port.
[B.3.IV] A row can be up to 640 pixels.

When a process aims to change the velocity of a wheel, it emits SYS_EVENT_IO_MOTOR_RIGHT or SYS_EVENT_IO_MOTOR_LEFT with the forward velocity as data. The transmitted data ($\mathrm{mm\,s^{-1}}$) is then converted by the post-processor into time duration per step[B.3.V]. The periodically called hardware-specific handler measures how much time has passed and, if appropriate, applies the next step.

It is worth noting that this module provides a power saving function. It applies a voltage only for the duration required to move the motor into the correct position. Thereafter, all output values are retuned to zero to avoid additional power consumption. This was required due to the hardware design of the e-puck, which makes it possible to draw a higher current than the battery would provide. This would result in restarts or incapacitated robots. To avoid this, the power saving functions were applied.

B.3.7 Proximity Module

Each of the eight Vishay Semiconductors TCRT1000 (i.e., proximity sensor) is connected to an analog input pin. The ADC module periodically samples all sensor inputs and executes the specified proximity pre-processor on the eight ADC channels.

The used processor linearises the sensor reading and transforms it into millimetres. The result is then stored in an single buffer to enable fast access. In addition the processor emits the value as SYS_EVENT_IO_PROX_x event, where x indicates which sensor provides the value. When all sensor readings have been sent, SYS_EVENT_IO_PROX_ALL is emitted.

B.3.8 Infra-red Communication Module: SwamCom

Instead of using the eight Vishay Semiconductors TCRT1000 for proximity measurements, they can be used to emit and receive infra-red signals. The raw signal values are obtained by the ADC similarly to before. However, the rx-processor decides which information (i.e., bit) has been obtained. Thereafter, the bits are collected to messages, decoded, and send as an SYS_EVENT_COM_RX_MSG event.

When a process transmits data, it emits a SYS_EVENT_COM_TX_MSG event, which is processed by the tx-processor. When configured, the tx-processor encodes the data into messages and applies all symbols sequentially to the digital output pins to emit the infra-red signals.

The modelling and design of this communication channel is presented in Chapter 4.

B.4 Case Studies

This section discusses how to use modules to design a behaviour. To do so, a set of didactic examples is presented to highlight certain aspects of OpenSwarm.

B.4.1 How to use OpenSwarm

To illustrate how to use OpenSwarm, this section presents an obstacle avoidance algorithm. Let a robot move indefinitely forward on an empty plane. When it detects an obstacle, the robot drives away from it as illustrated in Figure B.1.

This behaviour can be achieved in four steps: acquiring sensor readings, calculating the vector pointing towards the object, calculating the desired motion, and applying the calculated speed values to the motors. As described before, sensor readings are obtained by events from the proximity module. Therefore, event handlers are used to store data into an array.

Let an obtained value be expressed as a vector, $\boldsymbol{s}_i$, with the length that is indirectly proportional to the measured distance and oriented in alignment with the sensor's orientation. As

[B.3.V] The maximum velocity is $\pm 128\,\mathrm{mm\,s^{-1}}$. This is achieved by $1\,\mathrm{ms/step}$.

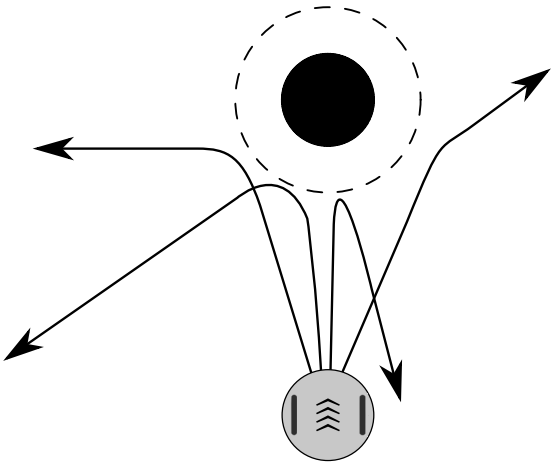


Figure B.1: Desired Obstacle avoidance behaviour. The robot (grey) moves forward unless an object (black) blocks its way. Depending on the approaching trajectory, the robot alters its direction (arrows). Note that the dashed line indicates the minimum radius between robot and object.

multiple objects could occur, all vectors are superpositioned according to

$$\boldsymbol{o} = \begin{bmatrix} o_x \\ o_y \end{bmatrix} = \sum_{\forall i} \boldsymbol{s}_i, \tag{B.1}$$

creating a vector that points to the direction were most objects are expected. With $\boldsymbol{o}$, the motion algorithm is composed of two parts: the rotation velocity, $\boldsymbol{r}$, and forward velocity, $\boldsymbol{f}$. First, the robot should always rotate away from a detected object with the rotation speeds

$$\boldsymbol{r} = \begin{bmatrix} r_{left} \\ r_{right} \end{bmatrix} = \begin{bmatrix} -\sin\left(\arg\left(\boldsymbol{o}\right)\right) v_{max} \\ \sin\left(\arg\left(\boldsymbol{o}\right)\right) v_{max} \end{bmatrix}. \tag{B.2}$$

Second, the robot should move forward with full speed unless it detects an object and, subsequently, reduce the speed until it stops in front of the object to avoid collisions. When, the object is on the back side of the robot, it moves away from it. This can be achieved with the following forward velocities

$$\boldsymbol{f} = \begin{bmatrix} f_{left} \\ f_{right} \end{bmatrix}, \tag{B.3}$$

$$f_{left} = f_{right} = \begin{cases} \frac{\|\boldsymbol{o}\|_2}{100} v_{max}, & \text{if } o_x \geq 0 \\ v_{max}, & \text{otherwise} \end{cases}. \tag{B.4}$$

The resulting wheel velocities are defined as

$$\boldsymbol{v} = \boldsymbol{r} + \boldsymbol{f}. \tag{B.5}$$

To achieve this behaviour on the real robot, it can be implemented as follows. Listing B.1 shows the implementation of the main function. At the top, OpenSwarm's main header file, `system.h`, is included, which declares all functions and features of OpenSwarm. Then the array to store the proximity values, `proxValues`, is declared. Furthermore, a boolean variable `isEmpty` is created to indicate if the process processed all values in `proxValues`. In `main`, OpenSwarm is initialised by `SYS_Init_Kernel()`. Then, the event handler, `proxHandler`, is subscribed to all proximity sensor events with a condition function, `condition`, and user data (i.e., a pointer to the element in the array where the sensor value should be stored).

Following this, a process, `process`, is created that implements the robot's behaviour. Finally, OpenSwarm is started with `Sys_Start_Kernel()` and `Sys_Run_SystemThread()` guarantees the further execution of OpenSwarm.

```
#include "os/system.h"
#define PROXIMITIES 8 // How many sensors?

int proxValues[PROXIMITIES];//buffers the proximity values
bool isEmpty = true;//has proxValues been emptied

void main(void)
{
  //Initialise OpenSwarm
  Sys_Init_Kernel();

  //get data from sensors & store them into proxValues
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX0, proxHandler,
   ↪  condition, &proxValues[0]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX1, proxHandler,
   ↪  condition, &proxValues[1]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX2, proxHandler,
   ↪  condition, &proxValues[2]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX3, proxHandler,
   ↪  condition, &proxValues[3]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX4, proxHandler,
   ↪  condition, &proxValues[4]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX5, proxHandler,
   ↪  condition, &proxValues[5]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX6, proxHandler,
   ↪  condition, &proxValues[6]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX7, proxHandler,
   ↪  condition, &proxValues[7]);
  Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX_ALL,
   ↪  proxFinish, condition, 0);

  //Create process that implements the robotic behaviour
  Sys_Create_Process(behaviour);

  //Start OpenSwarm and all of its modules
  Sys_Start_Kernel();

  //Keep OpenSwarm running
  Sys_Run_SystemThread();
}
```

Listing B.1: OpenSwarm implementation of the obstacle avoidance algorithm. This code shows how OpenSwarm is initialised and started as well as how event handlers are subscribed and how processes are started.

Listing B.2 shows how sensor values are collected and buffered with event handlers. When

the proximity modules emit an event (e.g., `SYS_EVENT_IO_PROX0`), OpenSwarm decides if the event handler should be called or not by executing `condition`. If `proxHandler` is executed, the data is extracted from the event and, if an object was detected within 100 mm, it is stored at the location in the buffer. Note that a weighted value indirectly proportional to the sensor value is buffered. After all events have been emitted, the `SYS_EVENT_IO_PROX_ALL` event is emitted to signal that all proximity events have been sent out. This triggers the execution of `proxFinish` that sets the flag to false to avoid overwriting unprocessed data.

```
1  bool condition(uint eventID, sys_event_data *data, void
   ↪  *user_data){
2    return isEmpty; //only buffer new elements if the old
     ↪  ones have been processed
3  }
4
5  bool proxReader(uint eventID, sys_event_data *data, void
   ↪  *user_data){
6    unsigned int distance = *((unsigned int *) data->value);
     ↪  //get data from event and store it in the correct
     ↪  element of the array
7    if(distance > 100){//mm // is value out of range
8      Sys_Start_AtomicSection();
9        *user_data = 0;//store s_i in proxValues
10     Sys_End_AtomicSection();
11   }else{//create a weight towards closest object
12     Sys_Start_AtomicSection();
13       *user_data = 100-distance;//store weighting in
           ↪  proxValues
14     Sys_End_AtomicSection();
15   }
16
17   return true;
18 }
19
20 bool proxFinish(uint eventID, sys_event_data *data, void
   ↪  *user_data){
21   Sys_Start_AtomicSection();
22     isEmpty = false;
23   Sys_End_AtomicSection();
24   return true;
25 }
```

Listing B.2: Event handling for the obstacle avoidance implementation. This shows how sensor values are obtained and stored.

When `SYS_EVENT_IO_PROX_ALL` has been sent out, it continues the execution of `process` as illustrated in Listing B.3. `behaviour` first calculates $\boldsymbol{o}$, which is stored in `pObject`, with a precision of 1%. Following this, the motor velocity is calculated by `calculateMotorSpeed`, which implements (B.2) to (B.5). The velocities are then emitted to the motors modules with the events `SYS_EVENT_IO_MOTOR_LEFT` and `SYS_EVENT_IO_MOTOR_RIGHT`.

```
typedef struct vector_s{
    int x;
    int y;
    long length;
} vector;

typedef struct motor_s{
    int left;
    int right;
} motor_speeds;

void behaviour(){
  //a priori calculated components for each sensor
  const int transform_x[] = {96, 70,   0, -88, -88,    0,  70,  96}; // divide by 100
  const int transform_y[] = {30, 72, 100,  48, -48, -100, -72, -30}; // divide by 100

  motor_speeds robot_speed = {0};
  vector pObject = {0};

  long prox_x = 0;
  long prox_y = 0;
  int i;

  while(true){
    Sys_Wait_For_Event(SYS_EVENT_IO_PROX_ALL);

    prox_x = 0;
    prox_y = 0;

    Sys_Start_AtomicSection();
      //calculate the overall vector, o
    for(i = 0; i < PROXIMITIES; i++){
      prox_x += ((long) proxValues[i] * (long) transform_x[i])/100; //calculate the X component
      prox_y += ((long) proxValues[i] * (long) transform_y[i])/100; //calculate the Y component
    }
    Sys_End_AtomicSection();

    pObject->x = (int) prox_x;
    pObject->y = (int) prox_y;
    pObject->length = sqrt(prox_x*prox_x + prox_y*prox_y);

    calculateMotorSpeed(&pObject, &robot_speed); //calculate the motor speed

    //apply motor speeds
    Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_LEFT, robot_speed.left);
```

```
46        Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_RIGHT,
          ↪  robot_speed.right);
47
48        Sys_Start_AtomicSection();
49          isEmpty = true;
50        Sys_End_AtomicSection();
51      }
52    }
53
54    void calculateMotorSpeed(vector *o, motor_speeds *speeds){
55
56      if(o == 0 || o->length == 0){//if there is no vector
57        speeds->left  = 0;
58        speeds->right = 0;
59        return;
60      }
61
62      if(object->length < 10){//if you do not see anything
63        speeds->left =  MAX_SPEED;
64        speeds->right = MAX_SPEED;
65        return;
66      }
67
68      //calculate sin(arg(o)) · v_max
69      signed int sinMax = (((o->y * 100)/o->length) *
          ↪  MAX_SPEED)/100;
70
71      //rotation if object is on the side
72      speeds->left  =  -sinMax;
73      speeds->right =   sinMax;
74
75      //add forward speed
76      if(vec->x >= 0){//is the target in front
77        if(vec->x < 100){//if the object gets close
78          //get slower when getting closer
79          speeds->left  += (o->x)/100*MAX_SPEED;
80          speeds->right += (o->x)/100*MAX_SPEED;
81        }else{
82          speeds->left  += MAX_SPEED;
83          speeds->right += MAX_SPEED;
84        }
85      }else{//if the object is behind you
86        speeds->left  += MAX_SPEED;
87        speeds->right += MAX_SPEED;
88      }
89    }
```

Listing B.3: Process-based behaviour implementation of obstacle avoidance.

With this, the robot can achieve the desired behaviour. It is worth noting that this imple-

Listing B.4 A didactic code illustration of data sharing across event handlers. Both event handlers — `handler1` and `handler2` — use the same global variable. Depending on how they are executed (synchronously or asynchronously), race conditions can occur.

```
1  #include "os/system.h"
2
3  int variable = 0;
4
5  bool handler1(unsigned
   ↪  int, sys_event_data
   ↪  *, void *){
6    // Note "variable++;"
     ↪  would do the same
     ↪  as
7    int temp = variable;
8    temp++;
9    variable = temp;
10 }
```

```
11 bool handler2(unsigned
   ↪  int, sys_event_data
   ↪  *, void *){
12   // Note "variable--;"
     ↪  would do the same
     ↪  as
13   int temp = variable;
14   temp--;
15   variable = temp;
16 }
```

mentation is a didactic example to illustrate important features of OpenSwarm. Consequently, this behaviour could have been implemented in various different ways (e.g., only with processes or only with event handlers). Each variant provides different benefits or disadvantages. In general, it is recommended to implement the calculation of $\boldsymbol{o}$ as pre-processor of the proximity module as the calculation could be performed faster by avoiding multiple event handlers and additional buffering.

B.4.2 Data Sharing: Synchronous vs. Asynchronous Events

When writing software, it is often inevitable to share data within a system and OpenSwarm is no exception to it. In OpenSwarm, processes and event handlers are executed in a flat memory, which means data can be shared anywhere in the working memory (i.e., global variables). In this section, data sharing for cooperative execution and process-based execution is discussed.

While cooperative execution already provides a sequential execution, data sharing can provide challenges because race conditions can occur when interrupt nesting is enabled (i.e., interrupts with higher priorities can interrupt other ISR). In this example, two event handlers — `handler1` and `handler2` — are available and manipulating a global variable — `variable` (see Listing B.4). `handler1` loads a global variable, increases its value, and saves the value back. `handler2` loads a global variable, decreases its value, and saves the value back. If both handlers are called equally often, the value of `variable` should remain around zero.

Asynchronously executed handler functions are sequentially executed, so if `handler1` and `handler2` are triggered by asynchronous events, race conditions cannot occur. However, the disadvantage of asynchronous execution is the unpredictable time delay.

When using synchronous events, the event handlers are executed within the context of their event's emission (i.e., a process or ISR). If the event handler is executed within a low interrupt-priority context, it can be interrupted by any higher-priority ISR. As a result, race conditions can occur. For instance, `handler1` is executed and loaded the shared variable. Then, stopped at line 8 by an interrupt that executes `handler2`. Consequently, `handler2` uses the current value, decreases it, and stores it back in the shared variable. When returning to the paused `handler1`, it increases the loaded value, and stores it back. As a result, it seems

like handler2 has not been executed. If this occurs regularly due to unfortunate timing, it would seem like handler2 has never been executed.

With a synchronous event handler, race conditions can only be avoided when sections that manipulate shared data cannot be interrupted. This can be achieved by creating an atomic section with Sys_Start_AtomicSection and Sys_Stop_AtomicSection around the used shared data. This code is then able to prevent race conditions. It is worth noting that excessive use of atomic sections can result in priority inversion; a situation in which a lower priority function is executed while a higher priority function is waiting for execution. This can lead to unpredictable behaviour as well as errors or faults.

B.4.3 Preemptive vs. Cooperative Behaviour Design

To illustrate the design and implementation process, this section describes the control of the wheels with the measured proximity value of a sensor on the front. In this example, a robot is placed on a plane with an obstacle in front of it. Depending on the distance to the obstacle, the robot moves away from the object (faster when closer).

In Listing B.5, a cooperative program is shown. With every occurrence of SYS_EVENT_IO_PROX0, handler is executed. First it extracts the measurement from the event. Then, the new wheel speed is calculated. Finally the speeds are sent to the motors module with the events SYS_EVENT_IO_MOTOR_LEFT and SYS_EVENT_IO_MOTOR_RIGHT.

```
1  #include "os/system.h"
2
3  bool handler(unsigned int eventID, sys_event_data *data,
   ↪  void *user_data){
4    unsigned int value = *((unsigned int *) data->value);
     ↪  //get data from event
5    if(value == -1){ //Is there no obstacle in range
6      value = 0;
7    }else{//there is an obstacle detected
8      value = MAX_WHEEL_SPEED_MM_S/(value+1);
9    }
10   Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_LEFT, value);
11   Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_RIGHT, value);
12   return true;
13 }
14
15 int main(void){
16   Sys_Init_Kernel(); //initialise OpenSwarm
17   Sys_Subscribe_to_Event(SYS_EVENT_IO_PROX0, handler, 0,
     ↪  0);
18   Sys_Create_Kernel(); //starts OpenSwarm
19   Sys_Run_SystemThread();
20 }
```

Listing B.5: A didactic code illustration to show event-based programming in OpenSwarm. Depending on the measured distance (SYS_EVENT_IO_PROX0), the robot moves faster backwards when the distance is smaller.

In Listing B.6, a preemptive process is implemented. With the start of OpenSwarm, process is executed and, shortly after its first execution, the process is blocked while waiting for

a SYS_EVENT_IO_PROX0 event. As soon as SYS_EVENT_IO_PROX0 occurs, the process continues by extracting the value from the event. After calculating the new wheel speeds, they are sent to the motors module with SYS_EVENT_IO_MOTOR_LEFT and SYS_EVENT_IO_MOTOR_RIGHT as before. Then, the process waits for the next event to occur.

```
#include "os/system.h"

void process(){
  while(true){
    sys_event_data *data =
     ↪  Sys_Wait_For_Event(SYS_EVENT_IO_PROX0);
    unsigned int value = *((unsigned int *) data->value);
     ↪  //get data from event
    if(value == -1){ //Is there no obstacle in range
      value = 0;
    }else{//there is a obstacle detected
    value = MAX_WHEEL_SPEED_MM_S/(value+1);
    }
    Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_LEFT, value);
    Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_RIGHT, value);
    Sys_Free(data); // free event data
  }
}

int main(void){
  Sys_Init_Kernel(); //initialise OpenSwarm
  Sys_Create_Process(process);
  Sys_Start_Kernel(); //starts OpenSwarm
  Sys_Run_SystemThread();
}
```

Listing B.6: A didactic code illustration to show process-based programming in OpenSwarm.

When comparing Listing B.5 and B.6, it can be seen that programs can be designed either event-driven or process-driven. Both programming paradigms can be used to achieve the similar behaviour. In both cases, the System Thread executes the majority of the time. When SYS_EVENT_IO_PROX0 occurs, the event handler is synchronously executed (Listing B.5) or process is placed to be executed. It is worth noting that the time from emitting the event to processing it is less predictable in the process-based solution because the delay is dependent on the relative occurrence of that event within the system. For instance, when the event occurs at the beginning of the System Thread's execution cycle, then the delay is bigger than it would be towards its end.

Besides the different response time, the program reacts differently if the periodicity of SYS_EVENT_IO_PROX0 becomes faster. In the case the times between events, t_e, is bigger than both the execution time, t_x[B.4.1], and the time between scheduling, t_s, (i.e., $t_x \ll t_s \ll t_e$) the program behaves as described above, where both solutions behave similarly despite the response time jitter.

When $t_x \ll t_e \ll t_s$, the process-based solution can no longer guarantee that every event is processed because the scheduling delay can cause the missing of events. To improve this,

[B.4.1] In this example, it is assumed that the execution time of event and process is t_x.

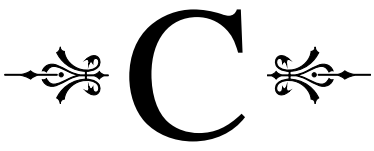
OpenSwarm has been adapted to provide *process* with a list of events that occurred while waiting. Consequently, `process` can process all events listed before blocking (shown in Listing B.7). This can increase the performance and responsiveness of the system. It is worth noting that events occurring during this execution are lost. In both cases, the event-based implementation provides a faster and more reliable solution[B.4.II].

```
#include "os/system.h"

void process(){
  while(true){
    sys_event_data *data =
     ↪  Sys_Wait_For_Event(SYS_EVENT_IO_PROX0);

    while(data != 0){//go through each event that occurred
      unsigned int value = *((unsigned int *)
       ↪  data->value); //get data from event
      if(value == -1){ //Is there no obstacle in range
        value = 0;
      }else{//there is a obstacle detected
        value = MAX_WHEEL_SPEED_MM_S/(value+1);
      }
      Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_LEFT, value);
      Sys_Send_IntEvent(SYS_EVENT_IO_MOTOR_RIGHT, value);

      sys_event_data *temp = data;
      data = data->next;//next occurred event
      Sys_Free(temp); // free event data
    }
  }
}

int main(void){
  Sys_Init_Kernel(); //initialise OpenSwarm
  Sys_Create_Process(process);
  Sys_Start_Kernel(); //starts OpenSwarm
  Sys_Run_SystemThread();
}
```

Listing B.7: A didactic code illustration to show process-based programming in OpenSwarm.

However, if $t_e \ll t_x \ll t_s$, then both event-based and process-based solutions require more time to process than the event requires to occur. The event-based solution would execute one event handler after the other, which monopolises the CPU. Furthermore, the increasing amount of buffered events would inevitably cause faults (e.g., memory/stack overflow). On the other hand, the process-based solution executes one event, misses events that occur during its execution, and then waits for the next events to occur. While waiting, the System Thread is executed for one iteration and then `process` continues. As a result, the system is stable but does not process all occurring events.

[B.4.II] This is the reason why embedded operating systems, such as Contiki, provide this form of execution model.



Computer Systems

Contents

This section provides an overview of current computer systems and analyses how many resources are available and how do they compare to the other systems. In particular, three groups of devices are investigated: microcontroller-based, sensor-network node, and embedded computer systems. The data for each subsection can be found at [Trenkwalder 2020*a*].

C.1 Microcontroller

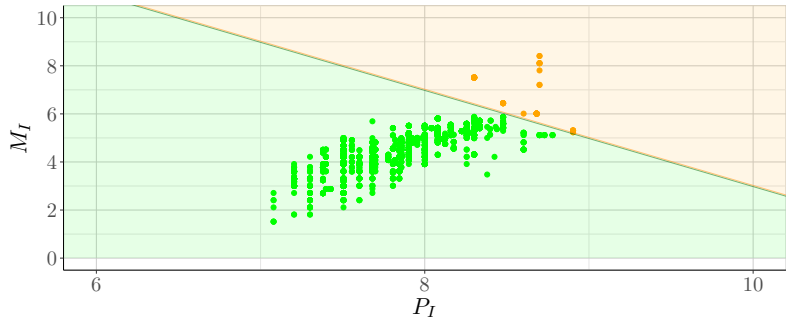
Microcontroller, also referred to as microcontroller units (MCUs), are integrated circuits that provides processor, RAM, ROM, and I/O devices on a single chip. They are designed to operate devices and are popular on mobile or low-power devices. Many robots, such as Kilobot or e-puck, are solely powered by a single MCU and, hence, this group of devices is relevant in robotics.

Figure C.1 shows a histogram 2768 state-of-the-art MCUs produced by Infineon, Microchip, STMicroelectronics, and Texas Instruments. As shown, the majority (98.1 %) of microcontrollers are severely-constrained.

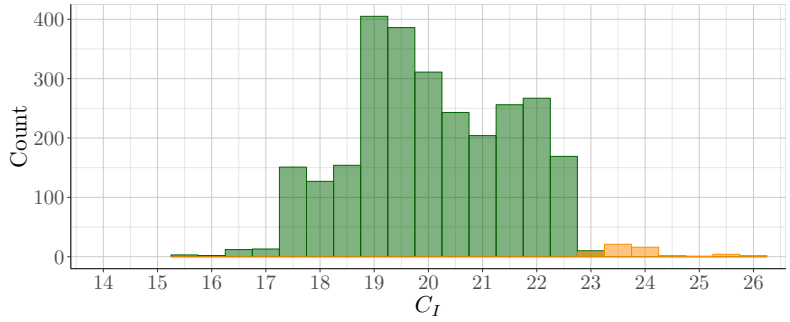
C.2 Sensor Network Nodes

In the area of sensor networks, a large number of elements, referred to as a sensor node or mote, are used to measure physical entities, such as temperature or pressure, covering large areas. Each sensor node transmits its measurements wirelessly or via a tether to a basis station.

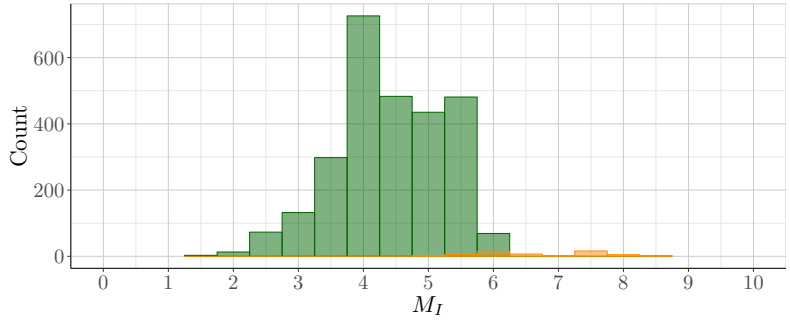
Commonly, a sensor node is designed to periodically measure and transmit data. Many devices are designed to be used in large numbers (i.e., they need to be cost-effective) and for long times (i.e., they need to consume low power). As a result, the nodes provide small computational resources, similar to many severely-constrained robots.



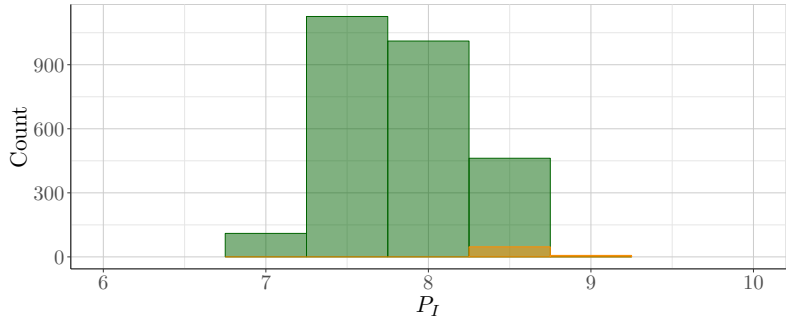
(a)



(b)



(c)



(d)

Figure C.1: Computational resources of 2768 MCUs from [Trenkwalder 2020*a*]. (a) shows a scatter plot of computational indices. (b), (c), and (d) show the histograms of the computational index (C_I), the memory index (M_I), and the processing index (P_I), respectively.

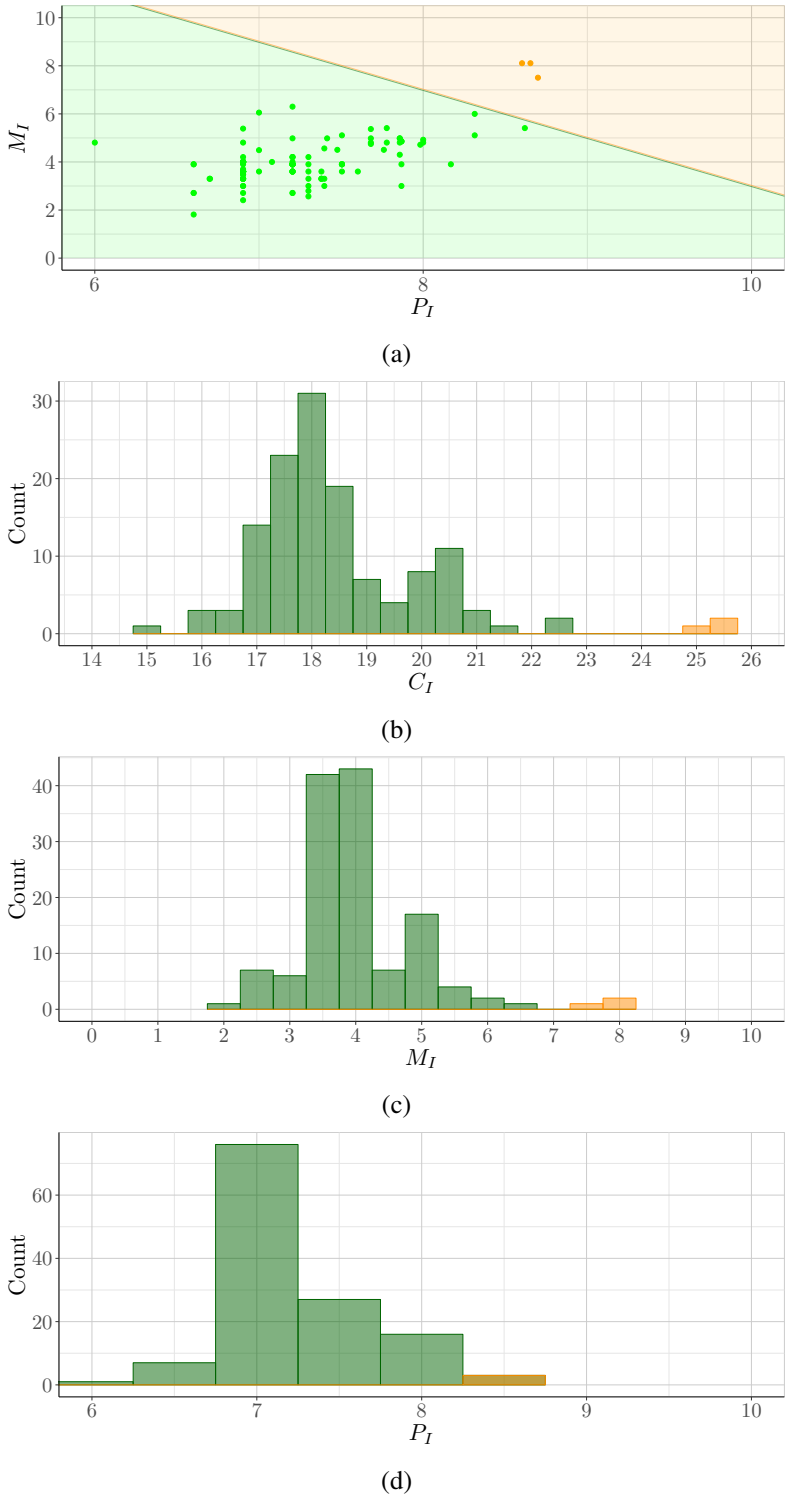


Figure C.2: Computational resources of 133 sensor nodes from [Trenkwalder 2020*a*]. (a) shows a scatter plot of computational indices. (b), (c), and (d) show the histograms of the computational index (C_I), the memory index (M_I), and the processing index (P_I), respectively.

Figure C.2 shows the computational resources of 133 sensor nodes. It is evident that $97.7\,\%$ of the presented sensor nodes are severely-constrained devices.

C.3 Embedded Computer Systems

While the definition varies across resources [Dean 2017], embedded computer systems are a large subset of computer systems, which are designed for a specific purposes in contrast to general purpose computers — such as personal computers. These systems are commonly embedded in an greater system; hence, its name. These systems are deployed in many areas of modern life — for instance, networking (e.g., routers), home assistants (e.g., Amazon Alexa Echo or Google Home), single board computers (e.g., Raspberry Pi), and robotics (see Section 2).

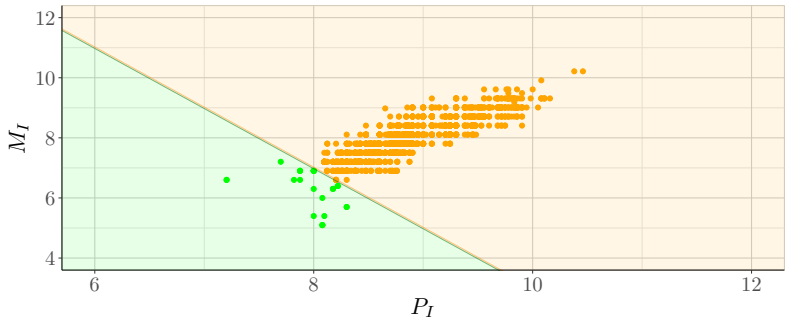
Figure C.3 shows 2326 embedded computer systems taken from [Wikimedia Foundation 2019*b*]. It is evident that $99.4\,\%$ of the presented devices are weakly-constrained devices.

C.4 Discussion

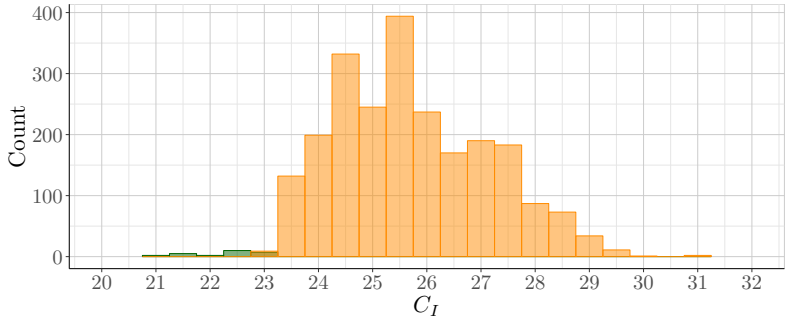
When comparing the types of systems as shown in Figure C.4, it can be seen that sensor network nodes tend to have the fewest resources some of the current nodes use legacy MCUs and others operate modern MCUs with lower frequency (see Figure C.4d) to reduce the power consumption. However, as sensor nodes are often MCU-based devices, it is not surprising that both MCUs and sensor nodes populate the same region for any of their computational indices (see Figure C.4b–C.4d).

While the populations of all types have an overlap when considering P_I, it can be seen that the C_I and, specifically, M_I shows two distinct populating regions. Figure C.4a reveals that each set has a small number of devices across the threshold. However, the majority of MCUs and sensor network are and severely-constrained and embedded computer systems are weakly-constrained devices. In particular, the scatter plot shows a gap between both sets of devices. As described in Chapter 2, this stems from transitioning from from integrated to discrete components. Furthermore, it is interesting to see that the threshold (i.e., the relation between memory and processing power) in Figure C.4a connects the upper and lower end of each population suggesting that the processing-time–memory trade-off is a valid assumption to represent computational power.

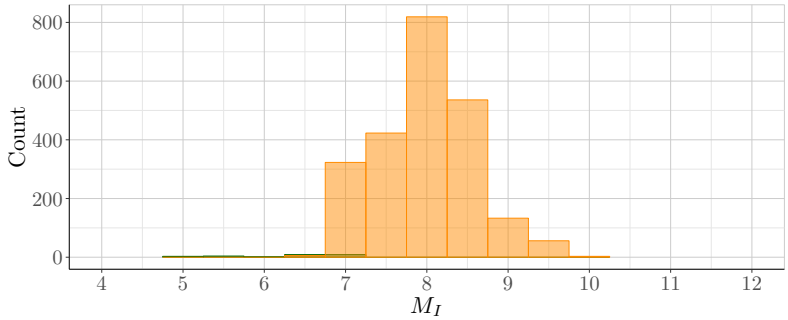
Note that the classification introduced by [Bormann et al. 2014] does not have an impact or does not show any benefit for severely-constrained devices as they are uniformly distributed around the thresholds.



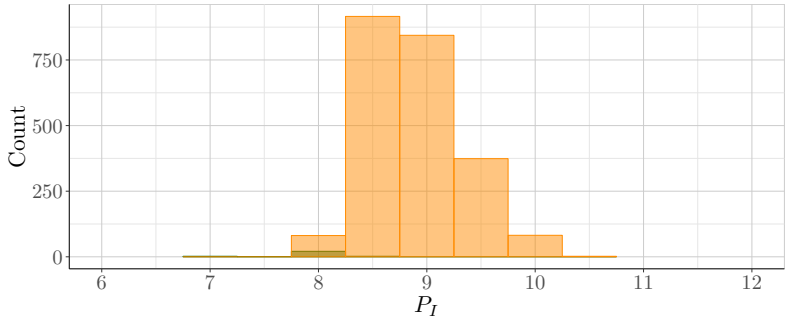
(a)



(b)

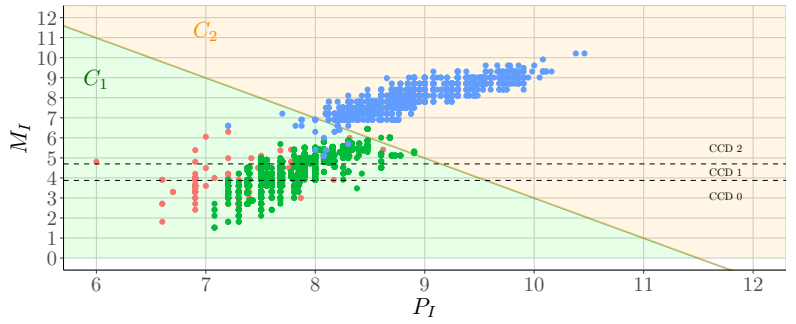


(c)

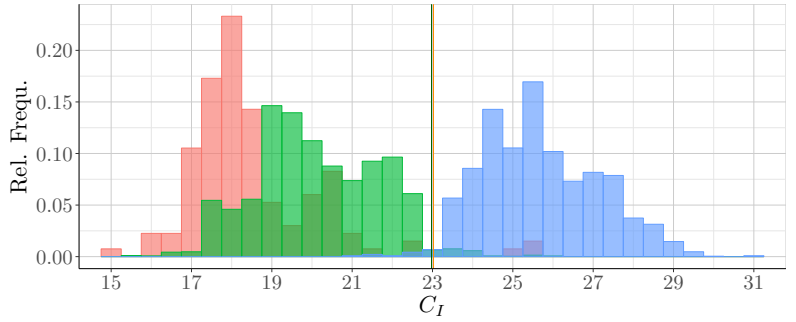


(d)

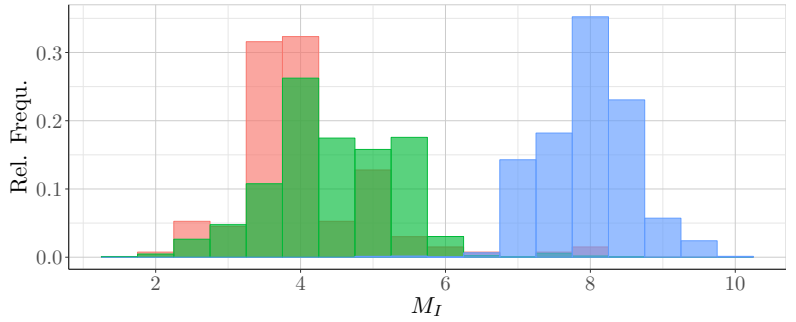
Figure C.3: Computational resources of 2326 embedded computer systems from [Trenkwalder 2020*a*]. (a) shows a scatter plot of computational indices. (b), (c), and (d) show the histograms of the computational index (C_I), the memory index (M_I), and the processing index (P_I), respectively.



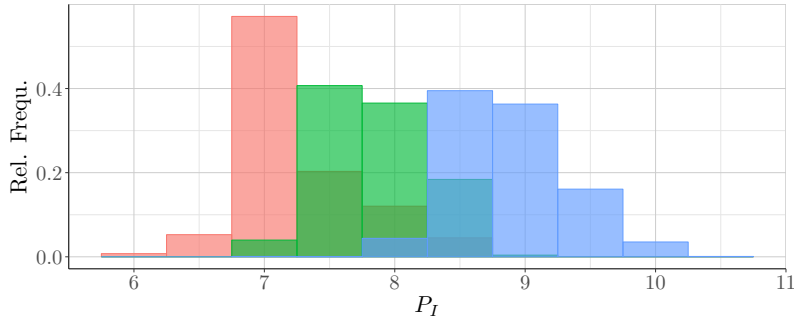
(a)



(b)



(c)



(d)

Figure C.4: Computational resources of 5227 computer systems from [Trenkwalder 2020*a*]. (a) shows a scatter plot of computational indices. (b), (c), and (d) show the histograms of the computational index (C_I), the memory index (M_I), and the processing index (P_I), respectively. Note (a) also shows the classification of Bormann et al. [2014].

References

E. U. Acar, H. Choset, Y. Zhang, and M. Schervish (2003). 'Path planning for robotic demining: Robust sensor-based coverage of unstructured environments and probabilistic methods.' *Int. J. Robot. Res.*, 22(7-8):441–466.

I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci (2002). 'Wireless sensor networks: a survey.' *Computer Networks*, 38(4):393–422.

F. Al-Turjman (2019). '5G-enabled devices and smart-spaces in social-IoT: an overview.' *Future Generation Comp. Sys.*, 92:732–744.

S. Anees and M. R. Bhatnagar (2015). 'Performance evaluation of decode-and-forward dual-hop asymmetric radio frequency-free space optical communication system.' *IET Optoelectronics*, 9(5):232–240.

G. Antonelli (2006). *Underwater Robots: Motion and Force Control of Vehicle-Manipulator Systems (Springer Tracts in Advanced Robotics)*. Berlin, Germany: Springer.

A. Arab and Q. Feng (2014). 'Reliability research on micro- and nano-electromechanical systems: a review.' *IEEE Int. J. Adv. Manuf. Technol.*, 74(9):1679–1690.

A. Arafa, A. Baknina, and S. Ulukus (2017). 'Energy Harvesting Two-Way Channels With Decoding and Processing Costs.' *IEEE Trans. Green Commun. Netw.*, 1(1):3–16. doi: 10.1109/TGCN.2016.2603588.

F. Arvin, J. Espinosa, B. Bird, A. West, S. Watson, and B. Lennox (2018). 'Mona: an affordable open-source mobile robot for education and research.' *J. Intell. & Robot. Sys.*, (pp. 1–15).

F. Arvin, J. Murray, C. Zhang, and S. Yue (2014). 'Colias: An autonomous micro robot for swarm robotic applications.' *Int. J. Adv. Robot. Syst.*, 11(7):113.

F. Arvin, K. Samsudin, and A. R. Ramli (2009). 'A Short-Range Infrared Communication for Swarm Mobile Robots.' In '2009 Int. Conf. Sig. Process. Syst.', (pp. 454–458). Piscataway, NJ: IEEE. doi: 10.1109/ICSPS.2009.88.

F. Arvin, K. Samsudin, A. R. Ramli, et al. (2009). 'Development of a miniature robot for swarm robotic application.' *Int. J. Comp. Elect. Eng.*, 1(4):436–442.

E. Bahceci, O. Soysal, and E. Sahin (2003). 'A review: Pattern formation and adaptation in multi-robot systems.' *Robot. Inst.* Tech. Rep. CMU-RI-TR-03-43.

L. S. Bai, L. Yang, and R. P. Dick (2009). 'MEMMU: Memory Expansion for MMU-less Embedded Systems.' *ACM Trans. Embed. Comput. Syst.*, 8(3):1–33.

J. C. Baillie (2004). 'Urbi: A universal language for robotic control.' *Int. J. Humanoid Robot.*, (pp. 7–29).

T. Balch and R. C. Arkin (1994). 'Communication in reactive multiagent robotic systems.' *Auton. robots*, 1(1):27–52.

W. Bangert, A. Kielhorn, F. Rahe, A. Albert, P. Biber, S. Grzonka, S. Haug, A. Michaels, D. Mentrup, M. Hänsel, et al. (2013). 'Field-robot-based agriculture:"RemoteFarming. 1" and "BoniRob-Apps".' *VDI-Berichte*, (2193):439–446.

J. C. Barca and Y. A. Sekercioglu (2013). 'Swarm robotics reviewed.' *Robotica*, 31(3):345–359.

L. Bayındır (2016). 'A review of swarm robotics tasks.' *Neurocomputing*, 172:292–321.

G. Beltrame and A. Dentinger (2019). 'BittyBuzz repository.' https://github.com/MISTLab/BittyBuzz. Accessed on: 31-March-2019.

G. Beni (2005). 'From Swarm Intelligence to Swarm Robotics.' In 'Swarm Robot.', volume 3342 of *Lecture Notes in Computer Science*, (pp. 1–9). Berlin, Germany: Springer.

B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy (1990). 'Lightweight Remote Procedure Call.' *ACM Trans. Comput. Syst.*, 8(1):37–55.

J. Betthauser, D. Benavides, J. Schornick, N. O'Hara, J. Patel, J. Cole, and E. Lobaton (2014). 'WolfBot: A distributed mobile sensing platform for research and education.' In 'Proc. Conf. American Soc. Eng. Edu. (ASEE 2014 Zone 1),' (pp. 1–8). Piscataway, NJ: IEEE.

S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han (2005). 'MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms.' *Mob. Netw. Appl.*, 10(4):563–579.

R. Bischoff, U. Huggenberger, and E. Prassler (2011). 'Kuka youbot-a mobile manipulator for research and education.' In 'Proc. 2011 IEEE Int. Conf. Robotics and Autom. (ICRA 2011),' (pp. 1–4). Piscataway, NJ, USA: IEEE.

M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada (2010). 'The marXbot, a miniature mobile robot opening new perspectives for the collective-robotic research.' In 'Proc. 2010 IEEE/RSJ Int. Conf. on Intell. Robots and Syst. (IROS 2010),' (pp. 4187–4193). Piscataway, NJ, USA: IEEE.

C. Bormann, M. Ersue, and A. Keranen (2014). 'Terminology for Constrained-Node Networks.' https://tools.ietf.org/html/rfc7228, RFC 7228, Internet Engineering Task Force (IETF). Accessed on: 09-Oct-2016.

P. Bouchier (2013). 'Embedded ROS [ROS Topics].' *IEEE Robot. Autom. Mag.*, 20(2):17–19.

M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo (2013). 'Swarm robotics: a review from the swarm engineering perspective.' *Swarm Intell.*, 7(1):1–41.

H. Bruyninckx, P. Soetens, and B. Koninckx (2003). 'The real-time motion control core of the Orocos project.' In 'Proc. 2003 IEEE Int. Conf. Robot. and Autom. (ICRA 2003),' volume 2, (pp. 2766–2771). Piscataway, NJ, USA: IEEE.

Cambridge University Press (2016). 'Cambridge Dictionary - Definition Robot.' http://dictionary.cambridge.org/de/worterbuch/englisch/robot. Accessed on: 02-March-2016.

Q. Cao, T. Abdelzaher, J. Stankovic, and T. He (2008). 'The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks.' In 'Proc. 2008 IEEE Int. Conf. Inform. Process. Sens. Netw. (IPSN 2008),' (pp. 233–244). Piscataway, NJ, USA: IEEE.

G. Caprari and R. Siegwart (2005). 'Mobile micro-robots ready to use: Alice.' In 'Proc. 2005 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2005),' (pp. 3295–3300). doi: 10.1109/IROS.2005.1545568.

J. B. Carruthers and J. M. Kahn (1997). 'Modeling of nondirected wireless infrared channels.' *IEEE Trans. Commun.*, 45(10):1260–1268.

M. Castro and B. Liskov (2002). 'Practical Byzantine fault tolerance and proactive recovery.' *ACM Trans. Comp. Syst. (TOCS)*, 20(4):398–461.

V. Chandrasekhar, W. K. Seah, Y. S. Choo, and H. V. Ee (2006). 'Localization in Underwater Sensor Networks: Survey and Challenges.' In 'Proc. 1st ACM Int. Workshop on Underw. Netw. (WUWNet 2006),' (pp. 33–40). New York, NY, USA: ACM.

H. Chao, Y. Cao, and Y. Chen (2010). 'Autopilots for small unmanned aerial vehicles: A survey.' *Int. J. Contr., Autom. and Syst.*, 8(1):36–44.

J. Chen, M. Gauci, and R. Groß (2013). 'A Strategy for Transporting Tall Objects with a Swarm of Miniature Mobile Robots.' In 'Proc. 2013 IEEE Int. Conf. Robotics and Autom. (ICRA 2013),' (pp. 863–869). Piscataway, NJ: IEEE.

J. Chen, M. Gauci, W. Li, A. Kolling, and R. Groß (2015). 'Occlusion-Based Cooperative Transport with a Swarm of Miniature Mobile Robots.' *IEEE Trans. Robot.*, 31(2):307–321.

Y. Chen, Z. Du, and M. García-Acosta (2010). 'Robot as a Service in Cloud Computing.' In 'Proc. 2010 IEEE Int. Symp. Serv. Orient. Syst. Eng. (SOSE 2010),' (pp. 151–158). Piscataway, NJ, USA: IEEE.

Z. Chen (2000). *Java card technology for smart cards: architecture and programmer's guide*. Upper Saddle River, NJ, USA: Addison-Wesley.

S. Chennareddy, A. Agrawal, and A. Karuppiah (2017). 'Modular Self-Reconfigurable Robotic Systems: A Survey on Hardware Architectures.' *J. Robot.*, 2017.

I. Chlamtac, M. Conti, and J. J.-N. Liu (2003). 'Mobile ad hoc networking: imperatives and challenges.' *Ad Hoc Networks*, 1(1):13–64.

C.-Y. Chong and S. P. Kumar (2003). 'Sensor networks: evolution, opportunities, and challenges.' *Proc. IEEE*, 91(8):1247–1256.

S. Choudhuri and T. Givargis (2005). 'Software virtual memory management for MMU-less embedded systems.' *Center for Embed. Comput. Syst.*

M. Chui, J. Manyika, and M. Miremadi (2015). 'Four fundamentals of workplace automation.' *McKinsey Quarterly*, (pp. 1–9).

S. Climent, A. Sanchez, J. V. Capella, N. Meratnia, and J. J. Serrano (2014). 'Underwater acoustic wireless sensor networks: advances and future trends in physical, MAC and routing layers.' *Sensors*, 14(1):795–833.

T. H. Collett, B. A. MacDonald, and B. P. Gerkey (2005). 'Player 2.0: Toward a practical robot programming framework.' In 'Proc. Australasian Conf. Robot. and Autom. (ACRA 2005),' (p. 145). Sydney, Australia: ARAA.

M. Conti and S. Giordano (2014). 'Mobile ad hoc networking: milestones, challenges, and new research directions.' *IEEE Communications Magazine*, 52(1):85–96.

M. Correia, G. S. Veronese, N. F. Neves, and P. Verissimo (2011). 'Byzantine consensus in asynchronous message-passing systems: a survey.' *Int. J. Crit. Computer-Based Syst.*, 2(2):141–161.

M. S. Couceiro (2016). 'An overview of swarm robotics for search and rescue applications.' In 'Handbook of Research on Design, Control, and Modeling of Swarm Robotics,' (pp. 345–382). IGI Global.

A. G. Dean (2017). *Embedded Systems Fundamentals with ARM Cortex-M based Microcontrollers: A Practical Approach*. Cambridge, UK: ARM Education Media.

C. Degryse (2016). 'Digitalisation of the economy and its impact on labour markets.' *ETUI Res. Paper - Working Paper 2016.02*.

D. Deville, A. Galland, G. Grimaud, and S. Jean (2003). 'Smart card operating systems: Past, present and future.' In '5th USENIX/NordU Conf.', Berkeley, CA, USA: USENIX Association.

G. A. Di Caro, F. Ducatelle, and L. M. Gambardella (2009). 'Wireless communications for distributed navigation in robot swarms.' In 'Workshops on Appl. Evol. Comput.', (pp. 21–30). Berlin, Germany: Springer.

E. Diller, M. Sitti, et al. (2013). 'Micro-scale mobile robotics.' *Found. and Trends® in Robot.*, 2(3):143–259.

A. Dobra (2014). 'General classification of robots. Size criteria.' In 'Proc. 2014 IEEE Int. Conf. Robot. in Alpe-Adria-Danube Region (RAAD),' (pp. 1–6). Piscataway, NJ: IEEE.

M. Dorigo, D. Floreano, L. M. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, M. Bonani, M. Brambilla, A. Brutschy, et al. (2013). 'Swarmanoid: a novel concept for the study of heterogeneous robotic swarms.' *IEEE Robot. Autom. Mag.*, 20(4):60–71.

M. Dorigo, E. Tuci, V. Trianni, R. Groß, S. Nouyan, C. Ampatzis, T. H. Labella, M. Bonani, F. Mondada, et al. (2006). 'SWARM-BOT: Design and implementation of colonies of self-assembling robots.' Technical report, IEEE Comput. Intell. Soc.

A. B. Downey (2005). 'The Little Book of Semaphores.' *Version*, 2(5):11–15.

M. J. Doyle, X. Xu, Y. Gu, F. Perez-Diaz, C. Parrott, and R. Groß (2016). 'Modular hydraulic propulsion: A robot that moves by routing fluid through itself.' In 'Proc. 2016 IEEE Int. Conf. Robotics and Autom. (ICRA 2016),' (pp. 5189–5196). Piscataway, NJ, USA: IEEE.

Z. Du, W. Yang, Y. Chen, X. Sun, X. Wang, and C. Xu (2011). 'Design of a Robot Cloud Center.' In 'Proc. 2011 IEEE Int. Symp. Auton. Decentr. Syst. (ISADS 2011),' (pp. 269–275). Piscataway, NJ, USA: IEEE.

F. Ducatelle, G. A. Di Caro, C. Pinciroli, and L. M. Gambardella (2011*a*). 'Self-organized cooperation between robotic swarms.' *Swarm Intell.*, 5(2):73.

F. Ducatelle, G. A. Di Caro, C. Pinciroli, F. Mondada, and L. Gambardella (2011*b*). 'Communication assisted navigation in robotic swarms: self-organization and cooperation.' In 'Proc. 2011 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2011),' (pp. 4981–4988). Piscataway, NJ, USA: IEEE.

A. Dunkels, B. Gronvall, and T. Voigt (2004). 'Contiki - a lightweight and flexible operating system for tiny networked sensors.' In 'Proc. 2004 IEEE Int. Conf. Local Comput. Netw.', (pp. 455–462). Piscataway, NJ, USA: IEEE.

Ecole polytechnique federale de Lausanne (2014). 'e-puck Library.' URL http://www.e-puck.org/. Accessed on: 28-Feb-2015.

Y. Edan, S. Han, and N. Kondo (2009). *Automation in Agriculture*, (pp. 1095–1128). Berlin, Germany: Springer.

A. Elkady and T. Sobh (2012). 'Robotics middleware: A comprehensive literature survey and attribute-based bibliography.' *J. Robot.*

A. Ellery (2000). *An Introduction to Space Robotics*. Berlin, Germany: Springer.

W. Elmenreich, B. Heiden, G. Reiner, and S. Zhevzhyk (2015). 'A low-cost robot for multi-robot experiments.' In 'Proc. 2015 IEEE Int. Works. Intel. Solutions Embed. Syst. (WISES 2015),' (pp. 127–132). Piscataway, NJ: IEEE.

T. Erl (2012). *Service-oriented architecture*, volume 12. Upper Saddle River, NJ, USA: Prentice Hall Press.

M. A. Ertl and D. Gregg (2003). 'The structure and performance of efficient interpreters.' *J. Instruction-Level Parallelism*, 5:1–25.

J. A. Escalera, M. Doyle, F. Mondada, and R. Groß (2016). 'Evo-bots: A Simple, Stochastic Approach to Self-Assembling Artificial Organisms.' In 'Proc. 2016 Int. Symp. Distrib. Auton. Robot. Syst. (DARS 2016),' EPFL-CONF-221161, (pp. 373–383). Berlin, Germany: Springer.

A. Eswaran, A. Rowe, and R. Rajkumar (2005). 'Nano-RK: an energy-aware resource-centric RTOS for sensor networks.' In 'Proc. 26th IEEE Int. Real-Time Syst. Symp. (RTSS 2005),' (pp. 265–275). Piscataway, NJ, USA: IEEE.

M. O. Farooq and T. Kunz (2011). 'Operating Systems for Wireless Sensor Networks: A Survey.' *Sensors*, 11(6):5900–5930.

N. Farrow, J. Klingner, D. Reishus, and N. Correll (2014). 'Miniature six-channel range and bearing system: algorithm, analysis and experimental validation.' In 'Proc. 2014 IEEE Int. Conf. Robot. and Autom. (ICRA 2014),' (pp. 6180–6185). Piscataway, NJ: IEEE.

A. Flores-Abad, O. Ma, K. Pham, and S. Ulrich (2014). 'A review of space robotics technologies for on-orbit servicing.' *Progress in Aerospace Sciences*, 68:1–26.

C. Forbes, M. Evans, N. Hastings, and B. Peacock (2010). *Gamma Distribution*, (pp. 109–113). Hoboken, NJ: John Wiley & Sons.

M. Fujita and R. Enteretainment (2000). 'Entertainment Robot: AIBO.' *J. Inst. Image Inform. and Telev. Eng.*, 54(5):657–661.

S. Fusco, M. S. Sakar, S. Kennedy, C. Peters, S. Pane, D. Mooney, and B. J. Nelson (2014). 'Self-folding mobile microrobots for biomedical applications.' In 'Proc. 2014 IEEE Int. Conf. Robot. and Autom. (ICRA 2014),' (pp. 3777–3782). Piscataway, NJ, USA: IEEE.

L. Garattoni and M. Birattari (2018). 'Autonomous task sequencing in a robot swarm.' *Science Robotics*, 3(20).

V. García-López, F. Chen, L. G. Nilewski, G. Duret, A. Aliyan, A. B. Kolomeisky, J. T. Robinson, G. Wang, R. Pal, and J. M. Tour (2017). 'Molecular machines open cell membranes.' *Nature*, 548(7669):567.

N. Garg (2013). *Apache Kafka*. Birmingham, UK: Packt Publishing Ltd.

M. Gauci, J. Chen, W. Li, T. J. Dodd, and R. Groß (2014*a*). 'Clustering Objects with Robots That Do Not Compute.' In 'Proc. 12th Conf. Auton. Agents and Multi-Agent Syst. (AAMAS 2014),' (pp. 421–428). Richland, SC: IFAAMS.

M. Gauci, J. Chen, W. Li, T. J. Dodd, and R. Groß (2014*b*). 'Self-Organised Aggregation without Computation.' *Int. J. Robot. Res.*, 33(8):1145–1161. doi: 10.1177/0278364914525244.

D. Gay, P. Levis, and D. Culler (2005). 'Software Design Patterns for TinyOS.' *ACM SIGPLAN*, 40(7):40–49.

N. Ghazisaidi, H. Kassaei, and M. S. Bohlooli (2009). 'Integration of WiFi and WiMAX-Mesh Networks.' In 'Proc. 2009 IEEE/RSJ Int. Conf. Adv. Mesh Netw.', (pp. 1–6). Piscataway, NJ: IEEE. doi: 10.1109/MESH.2009.8.

W. Giernacki, M. Skwierczyński, W. Witwicki, P. Wroński, and P. Kozierski (2017). 'Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering.' In 'Proc. 2017 IEEE Int. Conf. Methods and Models in Autom. and Robot. (MMAR),' (pp. 37–42). Piscataway, NJ: IEEE.

S. G. Glisic and P. A. Leppänen (2013). *Wireless communications: TDMA versus CDMA*. Berlin, Germany: Springer Science & Business Media.

I. Glover and P. M. Grant (2010). *Digital communications*. London, UK: Pearson Education.

K. Goldberg and B. Kehoe (2013). 'Cloud robotics and automation: A survey of related work.' *Tech. Rep. UCB/EECS-2013-5*.

M. A. Goodrich and A. C. Schultz (2007). 'Human-robot Interaction: A Survey.' *Found. Trends Hum.-Comput. Interact.*, 1(3):203–275.

D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier (2008). 'The nao humanoid: a combination of performance and affordability.' *CoRR abs/0807.3223*.

D. Graff (2017). *Programming and Managing Swarms of Mobile Robots: A Systemic Approach*. Ph.D. thesis, Technical University of Berlin, Germany.

D. Graff, J. Richling, and M. Werner (2014). *jSwarm: distributed coordination in robot swarms*. New York, NY, USA: ACM.

R. L. Graham and N. J. Sloane (1990). 'Penny-packing and two-dimensional codes.' *Discr. & Comput. Geom.*, 5(1):1–11.

W. Grega and A. Pilat (2008). 'Real-time control teaching using LEGO® MINDSTORMS® NXT robot.' In 'Int. Multi-Conf. Comput Sci. and Inform. Techn. (IMCSIT 2008),' (pp. 625–628). Piscataway, NJ, USA: IEEE.

M. S. Grewal and A. P. Andrews (2014). *Kalman Filtering: Theory and Practice with MATLAB*. Hoboken, NJ: John Wiley & Sons.

T. Grift, Q. Zhang, N. Kondo, and K. Ting (2008). 'A review of automation and robotics for the bioindustry.' *J. Biomech. Eng.*, 1(1):37–54.

M. P. Groover (2007). *Automation, Production Systems, and Computer-Integrated Manufacturing*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3 edition.

M. Guerroumi, A.-S. K. Pathan, N. Badache, and S. Moussaoui (2014). 'On the medium access control protocols suitable for wireless sensor networks-a survey.' *Int. J. Commun. Netw. and Inform. Security*, 6(2):89.

E. Guizzo and E. Ackerman (2017). 'The TurtleBot3 Teacher.' *IEEE Spectrum*, 54(8):19–20.

S. Gupte, P. I. T. Mohandas, and J. M. Conrad (2012). 'A survey of quadrotor Unmanned Aerial Vehicles.' In 'Proc 2012 IEEE Southeastcon,' (pp. 1–6). Piscataway, NJ, USA: IEEE.

Á. Gutiérrez, A. Campo, M. Dorigo, D. Amor, L. Magdalena, and F. Monasterio-Huelin (2008). 'An Open Localization and Local Communication Embodied Sensor.' *Sensors*, 8(11):7545–7563. doi: http://dx.doi.org/10.3390/s8117545.

Á. Gutiérrez, A. Campo, M. Dorigo, J. Donate, F. Monasterio-Huelin, and L. Magdalena (2009*a*). 'Open e-puck range & bearing miniaturized board for local communication in swarm robotics.' In 'Proc. 2009 IEEE Int. Conf. Robotics and Autom. (ICRA 2009),' (pp. 3111–3116). Piscataway, NJ, USA: IEEE.

Á. Gutiérrez, E. Tuci, and A. Campo (2009*b*). 'Evolution of Neuro-Controllers for Robots' Alignment using Local Communication.' *Int. J. Adv. Robot. Syst.*, 6(1):6.

M. Hadded, P. Muhlethaler, A. Laouiti, R. Zagrouba, and L. A. Saidane (2015). 'TDMA-Based MAC Protocols for Vehicular Ad Hoc Networks: A Survey, Qualitative Analysis, and Open Research Issues.' *IEEE Commun. Surveys Tuts.*, 17(4):2461–2492.

H. Hamann (2010). *Space-Time Continuous Models of Swarm Robotic Systems*. Berlin, Germany: Springer.

H. Hamann (2018). *Swarm Robotics: A Formal Approach*. Berlin, Germany: Springer.

S. Hauert, S. Leven, J.-C. Zufferey, and D. Floreano (2010). 'Communication-based swarming for flying robots.' In 'Proc. 2010 IEEE Int. Conf. Robotics and Autom. (ICRA 2010),' Piscataway, NJ: IEEE.

M. Hellman (1980). 'A cryptanalytic time-memory trade-off.' *IEEE Trans. Inf. Theory*, 26(4):401–406.

S. Helmer, A. Poulovassilis, and F. Xhafa (2011). *Reasoning in Event-Based Distributed Systems*. Studies in Computational Intelligence. Berlin, Germany: Springer.

K. Herrick (2000). 'Development of the unmanned aerial vehicle market: forecasts and trends.' *Air & Space Europe*, 2(2):25–27.

K. Hirschman, L. Tsybeskov, S. Duttagupta, and P. Fauchet (1996). 'Silicon-based visible light-emitting devices integrated into microelectronic circuits.' *Nature*, 384(6607):338.

N. R. Hoff III (2011). *Multi-robot foraging for swarms of simple robots*. Citeseer.

F. Hussain, A. Anpalagan, and R. Vannithamby (2017). 'Medium access control techniques in M2M communication: survey and critical review.' *Trans. Emerging Telecommun. Technol.*, 28(1).

IEEE (2017). 'IEEE 802 Local Networks.' http://www.ieee802.org/. Accessed on: 25-Aug-2017.

IEEE (2019). 'Robots grouped by IEEE.' https://robots.ieee.org/learn/types-of-robots/. Accessed on: 15-Mar-2019.

L. Iocchi, D. Nardi, and M. Salerno (2001). *Reactivity and Deliberation: A Survey on Multi-Robot Systems*, (pp. 9–32). Berlin, Germany: Springer.

S. Jones, M. Studley, S. Hauert, and A. F. T. Winfield (2018). 'A Two Teraflop Swarm.' *Frontiers in Robotics and AI*, 5:11.

Z. Ju, C. Yang, and H. Ma (2014). 'Kinematics modeling and experimental verification of baxter robot.' In 'Proc. 33rd Chin. Control Conf.', (pp. 8518–8523).

G. Kantor, S. Singh, R. Peterson, D. Rus, A. Das, V. Kumar, G. Pereira, and J. Spletzer (2003). 'Distributed search and rescue with robot and sensor teams.' In 'Field and Serv. Robot.', (pp. 529–538). Berlin, Germany: Springer.

G. Kapellmann-Zafra (2017). *Human-Swarm Robot Interaction with Different Awareness Constraints*. Ph.D. thesis, The University of Sheffield.

Y. Kaszubowski Lopes (2016). *Supervisory Control Theory for Controlling Swarm Robotics Systems*. Ph.D. thesis, University of Sheffield.

B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg (2015). 'A Survey of Research on Cloud Robotics and Automation.' *IEEE Trans. Autom. Sci. Eng.*, 12(2):398–409.

C. F. Kemerer (1995). 'Software complexity and software maintenance: A survey of empirical research.' *Annals of Softw. Eng.*, 1(1):1–22.

S. Kernbach, E. Meister, F. Schlachter, K. Jebens, M. Szymanski, J. Liedke, D. Laneri, L. Winkler, T. Schmickl, R. Thenius, et al. (2008). 'Symbiotic robot organisms: REPLICATOR and SYMBRION projects.' In 'Proc. 8th Workshop on Perf. Metrics for Intell. Syst.', (pp. 62–69). New York, NY, USA: ACM.

A. Kettler, M. Szymanski, J. Liedke, and H. Wörn (2010). 'Introducing wanda-a new robot for research, education, and arts.' In 'Proc. 2010 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2010),' (pp. 4181–4186). Piscataway, NJ: IEEE.

L. U. Khan (2017). 'Visible light communication: Applications, architecture, standardization and research challenges.' *Dig. Comm. and Netw.*, 3(2):78–88.

J. Y. Kim, T. Colaco, Z. Kashino, G. Nejat, and B. Benhabib (2016). 'mROBerTO: A modular millirobot for swarm-behavior studies.' In 'Proc. 2016 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2016),' (pp. 2109–2114). Piscataway, NJ: IEEE.

A. Kolling, P. Walker, N. Chakraborty, K. Sycara, and M. Lewis (2016). 'Human Interaction With Robot Swarms: A Survey.' *IEEE Trans. Hum.-Mach. Syst.*, 46(1):9–26.

H. Kopetz (2011). *Real-time systems: design principles for distributed embedded applications*. Berlin, Germany: Springer.

T. Kosar, P. E. Martı, P. A. Barrientos, M. Mernik, et al. (2008). 'A preliminary study on various implementation approaches of domain-specific language.' *Inform. Softw. Techn.*, 50(5):390–405.

D. Kruger, I. Van Lil, N. Sunderhauf, R. Baumgartl, and P. Protzel (2006). 'Using and extending the Miro middleware for autonomous mobile robots.' In 'Proc. 2006 Int. Conf. Towards Auton. Robot. Syst. (TAROS 2006),' Berlin, Germany: Springer.

A. Kushleyev, D. Mellinger, C. Powers, and V. Kumar (2013). 'Towards a swarm of agile micro quadrotors.' *Auton. Robots*, 35(4):287–300.

C. Lam (2010). *Hadoop in action*. Shelter Island, NY: Manning Publications Co.

A. Lapidoth (2009). *A foundation in digital communication*. Cambridge, UK: Cambridge University Press.

E. A. Lee, J. D. Kubiatowicz, J. M. Rabaey, A. L. Sangiovanni-Vincentelli, S. A. Seshia, J. Wawrzynek, D. Blaauw, P. Dutta, K. Fu, C. Guestrin, et al. (2012). 'The terraswarm research center (TSRC)(a white paper).' *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-207*.

S. Lee, S. Kim, S. Kim, J.-Y. Kim, C. Moon, B. J. Nelson, and H. Choi (2018). 'A Capsule-Type Microrobot with Pick-and-Drop Motion for Targeted Drug and Cell Delivery.' *Adv. Healthc. Mater.*, 7(9):1700985. doi: 10.1002/adhm.201870036.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler (2005). 'TinyOS: An Operating System for Sensor Networks.' In 'Proc. Ambient Intell.', (pp. 115–148). Berlin, Germany: Springer.

Q. Li, X. Yang, Y. Zhu, and J. Zhang (2017). 'Self-organized Task Allocation in a Swarm of E-puck Robots.' In 'Chin. Intell. Autom. Conf.', (pp. 153–160). Berlin, Germany: Springer.

W. Li, M. Gauci, and R. Groß (2016). 'Turing learning: a metric-free approach to inferring behavior and its application to swarms.' *Swarm Intell.*, 10(3):211–243.

Y. Li, S. Du, and Y. Kim (2009). 'Robot swarm manet cooperation based on mobile agent.' In '2009 IEEE Int. Conf. Robot. and Biomimetics (ROBIO 2009),' (pp. 1416–1420). Piscataway, NJ, USA: IEEE.

S.-Y. Lien, C.-C. Tseng, I. Moerman, and L. Badia (2019). 'Recent Advances in 5G Technologies: New Radio Access and Networking.' *Wireless Commun. Mob. Comp.*, 2019.

W. Liu and A. F. Winfield (2011). 'Open-hardware e-puck Linux extension board for experimental swarm robotics research.' *Microprocessors and Microsystems*, 35(1):60–67.

Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß (2016). 'Supervisory control theory applied to swarm robotics.' *Swarm Intell.*, 10(1):65–97.

Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß (2017). 'Probabilistic Supervisory Control Theory (pSCT) Applied to Swarm Robotics.' In 'Proc. 16th Conf. Auton. Agents and Multi-Agent Syst. (AAMAS 2017),' (pp. 1395–1403). Richland, SC: IFAAMS.

S. MacConnell (1993). *Code complete: A practical handbook of software construction*. Redmont, WA, USA: Microsoft Press.

J. P. Macker and M. S. Corson (1998). 'Mobile Ad Hoc Networking and the IETF.' *SIGMOBILE Mob. Comput. Commun. Rev.*, 2(1):9–14. doi: 10.1145/584007.584015. URL http://doi.acm.org/10.1145/584007.584015.

S. Magnenat, P. Rétornaz, M. Bonani, V. Longchamp, and F. Mondada (2011). 'ASEBA: A modular architecture for event-based control of complex robots.' *IEEE/ASME Trans. Mechatronics*, 16(2):321–329.

A. Malik and P. Singh (2015). 'Free space optics: current applications and future challenges.' *Int. J. Optics*, 2015.

M. R. Mallick (2016). 'A Comparative Study Of Wireless Protocols With Li-Fi Technology: A Survey.' *Int. J. Adv. Comput. Eng. and Netw.*, 4(6):123–127.

A. Manecy, N. Marchand, F. Ruffier, and S. Viollet (2015). 'X4-MaG: a low-cost open-source micro-quadrotor and its linux-based controller.' *Int. J. Micro Air Veh.*, 7(2):89–109.

A. D. Marchese, C. D. Onal, and D. Rus (2014). 'Autonomous soft robotic fish capable of escape maneuvers using fluidic elastomer actuators.' *Soft Robotics*, 1(1):75–87.

S. Martel, M. Sherwood, C. Helm, W. G. De Quevedo, T. Fofonoff, R. Dyer, J. Bevilacqua, J. Kaufman, O. Roushdy, and I. Hunter (2001). 'Three-legged wireless miniature robots for mass-scale operations at the sub-atomic scale.' In 'Proc. 2001 IEEE Int. Conf. Robot. and Autom. (ICRA 2001),' (pp. 3423–3428). Piscataway, NJ: IEEE.

P. Marwedel (2006). *Embedded system design*, volume 1. Berlin, Germany: Springer.

W. Mauerer (2017). *Professional Linux kernel architecture*. Hoboken, NJ: John Wiley & Sons.

R. Mayet, J. Roberz, T. Schmickl, and K. Crailsheim (2010). 'Antbots: A Feasible Visual Emulation of Pheromone Trails for Swarm Robots.' In M. Dorigo, M. Birattari, G. A. Di Caro, R. Doursat, A. P. Engelbrecht, D. Floreano, L. M. Gambardella, R. Groß, E. Şahin, H. Sayama, and T. Stützle (Eds.), 'Swarm Intell.', (pp. 84–94). Berlin, Germany: Springer.

J. McLurkin, A. McMullen, N. Robbins, G. Habibi, A. Becker, A. Chou, H. Li, M. John, N. Okeke, J. Rykowski, S. Kim, W. Xie, T. Vaughn, Y. Zhou, J. Shen, N. Chen, Q. Kaseman, L. Langford, J. Hunt, A. Boone, and K. Koch (2014). 'A robot system design for low-cost multi-robot manipulation.' In 'Proc. 2014 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2014),' (pp. 912–918). doi: 10.1109/IROS.2014.6942668.

J. McLurkin, J. Rykowski, M. John, Q. Kaseman, and A. J. Lynch (2013). 'Using multi-robot systems for engineering education: teaching and outreach with large numbers of an advanced, low-cost robot.' *IEEE Trans. Educ.*, 56(1):24–33.

G. Metta, G. Sandini, D. Vernon, L. Natale, and F. Nori (2008). 'The iCub humanoid robot: an open platform for research in embodied cognition.' In 'Proc. 8th Workshop on Perform. Metrics for Intell. Syst.', (pp. 50–56). New York, NY, USA: ACM.

Microchip Technology Inc. (2018). 'Microchip: Manual and Description of dsPIC30F6014a.' https://www.microchip.com/wwwproducts/en/dsPIC30F6014A. Accessed on: 08-Aug-2018.

Microsoft (2017). 'Microsoft Windows Embedded Product Overview.' https://www.microsoft.com/windowsembedded/en-us/products-solutions-overview.aspx. Accessed on: 07-Jul-2017.

A. G. Millard, R. Joyce, J. A. Hilder, C. Fleşeriu, L. Newbrook, W. Li, L. J. McDaid, and D. M. Halliday (2017*a*). 'The Pi-puck extension board: a Raspberry Pi interface for the e-puck robot platform.' In 'Proc. 2017 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2017),' (pp. 741–748). Piscataway, NJ, USA: IEEE.

A. G. Millard, R. A. Joyce, J. A. Hilder, C. Fleseriu, L. Newbrook, W. Li, L. McDaid, and D. M. Halliday (2017*b*). 'The Pi-puck extension board: a Raspberry Pi interface for the e-puck robot platform.' In 'Proc. 2017 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2017),' (pp. 741–748). Piscataway, NJ: IEEE.

N. Mohamed, J. Al-Jaroodi, and I. Jawhar (2008). 'Middleware for Robotics: A Survey.' In 'Proc. 2008 IEEE Conf. Robot. Autom. Mechatr.', (pp. 736–742). Piscataway, NJ: IEEE.

N. Mohamed, J. Al-Jaroodi, and I. Jawhar (2009). 'A review of middleware for networked robots.' *Int. J. Computer Sci. and Netw. Security*, 9(5):139–148.

S. H. A. Mohammad, M. A. Jeffril, and N. Sariff (2013). 'Mobile robot obstacle avoidance by using Fuzzy Logic technique.' In 'Proc. 2013 IEEE Int. Conf. Syst. Eng. Technol.', (pp. 331–335). Piscataway, NJ, USA: IEEE.

G. Mohanarajah, D. Hunziker, R. D'Andrea, and M. Waibel (2015). 'Rapyuta: A Cloud Robotics Platform.' *IEEE Trans. Autom. Sci. Eng.*, 12(2):481–493.

F. Mondada and M. Bonani (2018). 'e-puck Extension Boards.' http://www.e-puck.org/index.php?option=com_content&view=article&id=12&Itemid=16. Accessed on: 15-Sep-2018.

F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli (2009). 'The e-puck, a robot designed for education in engineering.' In 'Proc. 9th Conf. Auton. Robot Syst. and Competitions,' volume 1, (pp. 59–65). Castelo Branco, Portugal: IPCB.

F. Mondada, E. Franzi, and A. Guignard (1999). 'The development of khepera.' In 'Proc. 1st Int. Khepera Workshop Experiments with the Mini-Robot Khepera,' (pp. 7–14).

F. Mondada, L. M. Gambardella, D. Floreano, S. Nolfi, J. L. Deneuborg, and M. Dorigo (2005). 'The cooperation of swarm-bots: physical interactions in collective robotics.' *IEEE Robot. Autom. Mag.*, 12(2):21–28.

F. Mondada, G. C. Pettinaro, A. Guignard, I. W. Kwee, D. Floreano, J.-L. Deneubourg, S. Nolfi, L. M. Gambardella, and M. Dorigo (2004). 'SWARM-BOT: A new distributed robotic concept.' *Auton. Robots*, 17(2-3):193–221.

P. Moubarak and P. Ben-Tzvi (2012). 'Modular and reconfigurable mobile robotics.' *Robot. Auton. Sys.*, 60(12):1648–1663.

G. Mühl (2006). *Distributed Event-Based Systems*. Berlin, Germany: Springer.

A. Munawar and G. Fischer (2016). 'A Surgical Robot Teleoperation Framework for Providing Haptic Feedback Incorporating Virtual Environment-Based Guidance.' *Front. Robot. and AI*, 3:47.

L. Murray, J. Timmis, and A. Tyrrell (2013). 'Modular self-assembling and self-reconfiguring e-pucks.' *Swarm Intell.*, 7(2):83–113.

M. Namoshe, N. Tlale, C. Kumile, and G. Bright (2008). 'Open middleware for robotics.' In 'Proc. 2008 IEEE Int. Conf. Mechatronics and Mach. Vision in Pract. (M2VIP 2008),' (pp. 189–194). Piscataway, NJ, USA: IEEE.

R. Nardi and O. Holland (2007). 'UltraSwarm: A Further Step Towards a Flock of Miniature Helicopters.' In E. Şahin, W. Spears, and A. Winfield (Eds.), 'Swarm Robot.', volume 4433 of *Lecture Notes in Comput. Sci.*, (pp. 116–128). Berlin, Germany: Springer.

I. Navarro and F. Matía (2012). 'An introduction to swarm robotics.' *Isrn robotics*, 2013.

G. Nutt (2016). 'NuttX - Online documentation.' URL http://www.nuttx.org/Documentation/NuttX.html. Accessed on: 30-Aug-2016.

R. Okuda, Y. Kajiwara, and K. Terashima (2014). 'A survey of technical trend of ADAS and autonomous driving.' In 'Proc. 2014 IEEE Int. Symp. VLSI Techn., Syst. and Appl. (VLSI-TSA 2014),' (pp. 1–4). Piscataway, NJ, USA: IEEE.

O. Orhan, D. Gündüz, and E. Erkip (2014). 'Energy Harvesting Broadband Communication Systems With Processing Energy Cost.' *IEEE Trans. Wireless Commun.*, 13(11):6095–6107. doi: 10.1109/TWC.2014.2328600.

C. Osterloh, T. Pionteck, and E. Maehle (2012). 'MONSUN II: A small and inexpensive AUV for underwater swarms.' In 'Proc. 7th German Conf. Robot. (ROBOTIK 2012),' (pp. 1–6). Frankfurt, Germany: VDE.

Oxford University Press (2016). 'Oxford Dictionary - Definition Robot.' https://en.oxforddictionaries.com/definition/robot. Accessed on: 02-March-2016.

A. Özgür, S. Lemaignan, W. Johal, M. Beltran, M. Briod, L. Pereyre, F. Mondada, and P. Dillenbourg (2017). 'Cellulo: Versatile handheld robots for education.' In 'Proc. 2017 ACM/IEEE Int. Conf. Human-Robot Interaction (HRI 2017),' (pp. 119–127). Piscataway, NJ: IEEE.

M. Pacheco, R. Fogh, H. H. Lund, and D. J. Christensen (2014). 'Fable: A modular robot for students, makers and researchers.' In 'Proc. IROS Workshop on Modular and Swarm Syst.', Piscataway, NJ, USA: IEEE.

M. Pacheco, R. Fogh, H. H. Lund, and D. J. Christensen (2015). 'Fable II: Design of a modular robot for creative learning.' In 'Proc. 2015 IEEE Int. Conf. Robotics and Autom. (ICRA 2015),' (pp. 6134–6139). Piscataway, NJ, USA: IEEE.

C. Parrott, T. J. Dodd, and R. Groß (2016). 'HyMod: A 3-DOF Hybrid Mobile and Self-Reconfigurable Modular Robot and its Extensions.' In 'Proc. 2016 Int. Symp. Distrib. Auton. Robot. Syst. (DARS 2016),' (pp. 401–414). Berlin, Germany: Springer.

F. Perez-Diaz, S. M. Trenkwalder, R. Zillmer, and R. Groß (2016). 'Emergence and inhibition of synchronization in robot swarms.' In 'Proc. 2016 Int. Symp. Distrib. Auton. Robot. Syst. (DARS 2016),' Berlin, Germany: Springer.

J. Pestana, J. L. Sanchez-Lopez, P. Campoy, and S. Saripalli (2013). 'Vision based gps-denied object tracking and following for unmanned aerial vehicles.' In 'Proc. 2013 IEEE Int. Symp. Safety, Security, and Rescue Robotics (SSRR 2013),,' (pp. 1–6). Piscataway, NJ, USA: IEEE.

D. P. Petersen and D. Middleton (1962). 'Sampling and reconstruction of wave-number-limited functions in N-dimensional Euclidean spaces.' *Information and Control*, 5(4):279–323.

D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt (2017). 'The robotarium: A remotely accessible swarm robotics research testbed.' In 'Proc. 2017 IEEE Int. Conf. Robotics and Autom. (ICRA 2017),' (pp. 1699–1706). Piscataway, NJ: IEEE.

D. Pickem, M. Lee, and M. Egerstedt (2015). 'The GRITSBot in its natural habitat-a multi-robot testbed.' In 'Proc. 2015 IEEE Int. Conf. Robot. and Autom. (ICRA 2015),' (pp. 4062–4067). Piscataway, NJ: IEEE.

A. Pierson, Z. Wang, and M. Schwager (2017). 'Intercepting Rogue Robots: An Algorithm for Capturing Multiple Evaders With Multiple Pursuers.' *IEEE Robot. Autom. Lett.*, 2(2):530–537.

C. Pinciroli, A. Lee-Brown, and G. Beltrame (2015). 'Buzz: An extensible programming language for self-organizing heterogeneous robot swarms.' *arXiv:1507.05946*.

L. Prechelt (2000). 'An empirical comparison of seven programming languages.' *IEEE Computer*, (10):23–29.

W. H. Press, S. A. Teukolsky, B. P. Flannery, and W. T. Vetterling (2007). 'Kolmogorov-Smirnov Test.' In 'Numerical Recipes in FORTRAN: The Art of Scientific Computing,' (pp. 617–620). Cambridge, UK: Cambridge University Press.

A. Prieto, J. Becerra, F. Bellas, and R. Duro (2010). 'Open-ended evolution as a means to self-organize heterogeneous multi-robot systems in real time.' *Robot. and Auton. Syst.*, 58(12):1282–1291.

J. Pugh and A. Martinoli (2007). 'Parallel learning in heterogeneous multi-robot swarms.' In 'Proc. IEEE Congr. Evolut. Comput. (CEC 2007),' (pp. 3839–3846). Piscataway, NJ, USA: IEEE.

M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng (2009). 'ROS: an open-source Robot Operating System.' In 'ICRA workshop on open source software,' volume 3. Kobe, Piscataway, NJ: IEEE.

N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, A. S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater, et al. (2015). 'Valkyrie: Nasa's first bipedal humanoid robot.' *J. Field Robot.*, 32(3):397–419.

P. J. Ramadge and W. M. Wonham (1987). 'Supervisory control of a class of discrete event processes.' *J. Contr. Optimiz.*, 25(1):206–230.

W. Rankl and W. Effing (2004). *Smart Card Handbook*. Wiley InterScience electronic collection. Hoboken, NJ: John Wiley & Sons.

A. Reina, A. J. Cope, E. Nikolaidis, J. A. R. Marshall, and C. Sabo (2017). 'ARK: Augmented Reality for Kilobots.' *IEEE Robot. Autom. Lett.*, 2(3):1755–1761. doi: 10.1109/LRA.2017.2700059.

W. Ren, R. W. Beard, and E. M. Atkins (2005). 'A survey of consensus problems in multi-agent coordination.' In 'Proc. 2005 IEEE American Contr. Conf. (ACC 2005).', (pp. 1859–1864). Piscataway, NJ: IEEE.

V. Richards (2016). '2016 Nobel Prize in Chemistry: Molecular machines.' *Nature chemistry*, 8(12):1090.

F. Riedo, M. Chevalier, S. Magnenat, and F. Mondada (2013). 'Thymio II, a robot that grows wiser with children.' In 'Proc. IEEE Workshop on Adv. Robot. Soc. Impac. (ARSO 2013),' (pp. 187–193). Eidgenössische Technische Hochschule Zürich, Autonomous System Lab, Piscataway, NJ, USA: IEEE.

B. Ristic (2015). *Particle Filters for Random Set Models*. Berlin, Germany: Springer.

B. Robič (2015). *The foundations of computability theory*. Berlin, Germany: Springer.

J. Rojas (2018). 'Plastic Waste is Exponentially Filling our Oceans, but where are the Robots?' In 'Proc. 2018 IEEE Region 10 Humanitarian Techn. Conf. (R10-HTC 2018),' (pp. 1–6). Piscataway, NJ: IEEE.

J. W. Romanishin, K. Gilpin, S. Claici, and D. Rus (2015). '3D M-Blocks: Self-reconfiguring robots capable of locomotion via pivoting in three dimensions.' In 'Proc. 2015 IEEE Int. Conf. Robot. and Autom. (ICRA 2015),' (pp. 1925–1932). Piscataway, NJ: IEEE.

T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy (1996). 'The structure and performance of interpreters.' *ACM SIGPLAN*, 31(9):150–159.

A. Rowe, K. Lakshmanan, H. Zhu, and R. Rajkumar (2010). 'Rate-Harmonized Scheduling and Its Applicability to Energy Management.' *IEEE Trans. Ind. Informat.*, 6(3):265–275.

D. Roy (2003). 'The discrete normal distribution.' *Commun. Stat.-theo. and Methods*, 32(10):1871–1883.

M. Rubenstein, C. Ahler, and R. Nagpal (2012). 'Kilobot: A Low Cost Scalable Robot System for Collective Behaviors.' In 'Proc. 2012 IEEE Int. Conf. Robotics and Autom. (ICRA 2012),' (pp. 3293–3298). Piscataway, NJ: IEEE.

M. Rubenstein, A. Cornejo, and R. Nagpal (2014). 'Programmable self-assembly in a thousand-robot swarm.' *Science*, 345(6198):795–799.

A. Ruckelshausen, P. Biber, M. Dorna, H. Gremmes, R. Klose, A. Linz, F. Rahe, R. Resch, M. Thiel, D. Trautz, et al. (2009). 'BoniRob–an autonomous field robot platform for individual plant phenotyping.' *Preci. Agric.*, 9(841):1.

E. Şahin (2005). *Swarm Robotics: From Sources of Inspiration to Domains of Application*, (pp. 10–20). Berlin, Germany: Springer.

E. Şahin and A. Winfield (2008). 'Special issue on swarm robotics.' *Swarm Intell.*, 2(2):69–72.

E. Sakthivelmurugan, G. Senthilkumar, K. Prithiviraj, and K. T. Devraj (2018). 'Foraging behavior analysis of swarm robotics system.' In 'Proc. Int. Conf. Res. Mech. Eng. Sci. (RiMES 2017),' volume 144, (p. 01013). Les Ulis, France: EDP Sciences.

T. Schmickl, R. Thenius, C. Moslinger, J. Timmis, A. Tyrrell, M. Read, J. Hilder, J. Halloy, A. Campo, C. Stefanini, et al. (2011). 'CoCoRo–The Self-Aware Underwater Swarm.' In 'Proc. 2011 IEEE Conf. Self-Adapt. Self-Organ. Syst. Workshops (SASOW 2011),' (pp. 120–126). Piscataway, NJ, USA: IEEE.

M. Schwager, B. J. Julian, M. Angermann, and D. Rus (2011). 'Eyes in the sky: Decentralized control for the deployment of robotic camera networks.' *Proc. IEEE*, 99(9):1541–1561.

R. Sedgewick and K. Wayne (2015). 'Algorithms: 24-part Lecture Series.'

B. Selic (2000). 'Distributed Software Design: Challenges and Solutions.' *Embedded Syst. Progr.*, 13(12):127–144.

N. Semiconductor (2019). 'LMX9820A - Bluetooth Serial Port Module.' https://www.promelec.ru/UPLOAD/fck/image/for_news/texas_news/LMX9820ASM_ds_0.71.pdf. Accessed on: 11-February-2019.

S. Sesay, Z. Yang, and J. He (2004). 'A survey on mobile ad hoc wireless network.' *Inform. Techn. J.*, 3(2):168–175.

J. Seyfried, M. Szymanski, N. Bender, R. Estaña, M. Thiel, and H. Wörn (2005). 'The I-SWARM Project: Intelligent Small World Autonomous Robots for Micro-manipulation.' In 'Proc. 2004 Int. Conf. Swarm Robot. (SAB 2004),' (pp. 70–83). Berlin, Germany: Springer.

M. Shan, J. Guo, and E. Gill (2016). 'Review and comparison of active space debris capturing and removal methods.' *Prog. Aerospace Sci.*, 80:18–32.

T. B. Sheridan (2016). 'Human–Robot Intera.' *Human Factors*, 58(4):525–532.

J. Shi, J. Wan, H. Yan, and H. Suo (2011). 'A survey of cyber-physical systems.' In 'Proc. 2011 IEEE Int. Conf. Wireless Commun. Signal Proces. (WCSP 2011),' (pp. 1–6). Piscataway, NJ: IEEE.

Z. Shi, C. Beard, and K. Mitchell (2013). 'Analytical models for understanding space, backoff, and flow correlation in CSMA wireless networks.' *Wireless Networks*, 19(3):393–409.

A. R. Shirazi and Y. Jin (2017). 'A Strategy for Self-Organized Coordinated Motion of a Swarm of Minimalist Robots.' *IEEE Trans. Emer. Top. Comput. Intell.*, 1(5):326–338.

B. Siciliano and O. Khatib (2016). *Springer Handbook of Robotics*. Berlin, Germany: Springer.

J. M. Soares, I. Navarro, and A. Martinoli (2016). 'The Khepera IV mobile robot: performance evaluation, sensory data and software toolbox.' In 'Proc. 2nd Iberian Robot. Conf.', (pp. 767–781). Berlin, Germany: Springer.

W. Stallings (2014). *Operating Systems: Internals and Design Principles*. Upper Saddle River, NJ, USA: Prentice Hall Press, 8 edition.

B. Steux and O. E. Hamzaoui (2010). 'tinySLAM: A SLAM algorithm in less than 200 lines C-language program.' In 'Proc. 2010 IEEE Int. Conf. Contr. Autom. Robot. Vision,' (pp. 1975–1979). Piscataway, NJ, USA: IEEE.

D. P. Stormont (2005). 'Autonomous rescue robot swarms for first responders.' In 'Proc. 2005 IEEE Int. Conf. Compu. Intell. for Homeland Security and Pers. Safety (CIHSPS 2005),' (pp. 151–157). Piscataway, NJ, USA: IEEE.

K. Støy (2001). 'Using Situated Communication in Distributed Autonomous Mobile Robotics.' In 'Proc. 7th Scandinavian Conf. Artifi. Intelli. (SCAI 2001),' (pp. 44–52). Amsterdam, The Netherlands: IOS Press.

Y. Tan and Z. yang Zheng (2013). 'Research Advance in Swarm Robotics.' *Defence Technol.*, 9(1):18–39.

A. S. Tanenbaum (2009). *Modern operating systems*. Upper Saddle River, NJ, USA: Prentice Hall Press.

A. S. Tanenbaum and M. Van Steen (2007). *Distributed systems*, volume 2. Upper Saddle River, NJ, USA: Prentice Hall Press.

D. Tarapore, A. L. Christensen, and J. Timmis (2017). 'Generic, scalable and decentralized fault detection for robot swarms.' *PloS one*, 12(8):e0182058.

R. H. Taylor, A. Menciassi, G. Fichtinger, and P. Dario (2008). *Medical Robotics and Computer-Integrated Surgery*, (pp. 1199–1222). Berlin, Germany: Springer.

L. S. Terrissa, B. Radhia, and J.-F. Brethé (2015). 'Towards a new approach of Robot as a Service (RaaS) in Cloud Computing paradigm.' In 'Proc. 5th. Int. Symp. ISKO-Maghreb,' Hammamet , Tunisia: International Society for Knowledge Organization.

S. M. Trenkwalder (2019). 'Miniature Robots: Their Computational Resources, Classification, & Implications.' *IEEE Robot. Autom. Lett.*, 4(3):2722–2729.

S. M. Trenkwalder (2020*a*). 'Online supplementary material page of Miniature Robots: Their Computational Resources, Classification, & Implications.' URL https://trenkwalder.tech/pubs/robot-classification. Accessed on: 14-Feb-2020.

S. M. Trenkwalder (2020*b*). 'OpenSwarm - GitHub repository.' URL https://github.com/OpenSwarm/OpenSwarm.git. Accessed on: 14-Feb-2020.

S. M. Trenkwalder (2020*c*). 'OpenSwarm - Online Documentation.' URL http://openswarm.org/documentation. Accessed on: 14-Feb-2020.

S. M. Trenkwalder (2020*d*). 'OpenSwarm - Online supplementary material page.' URL http://naturalrobotics.group.shef.ac.uk/supp/2016-001. Accessed on: 14-Feb-2020.

S. M. Trenkwalder, I. Esnaola, Y. K. Lopes, A. Kolling, and R. Groß (2019). 'SwarmCom: An Infra-Red-Based Mobile Ad-Hoc Network for Severely Constrained Robots.' *Auton. Robots*, (pp. 1–22). URL https://doi.org/10.1007/s10514-019-09873-0.

S. M. Trenkwalder, Y. K. Lopes, A. Kolling, A. L. Christensen, R. Prodan, and R. Groß (2016). 'OpenSwarm: An event-driven embedded operating system for miniature robots.' In 'Proc. 2016 IEEE/RSJ Int. Conf. Intell. Robots and Syst. (IROS 2016),' (pp. 4483–4490). Piscataway, NJ: IEEE.

V. Trianni, D. De Simone, A. Reina, and A. Baronchelli (2016). 'Emergence of consensus in a multi-robot network: from abstract models to empirical validation.' *IEEE Robot. Autom. Lett.*, 1(1):348–353.

B. Tribelhorn and Z. Dodds (2007). 'Evaluating the Roomba: A low-cost, ubiquitous platform for robotics research and education.' In 'Proc. 2007 IEEE Int. Conf. Robotics and Autom. (ICRA 2007),' (pp. 1393–1399). Piscataway, NJ: IEEE.

G. Tuna and V. C. Gungor (). 'A survey on deployment techniques, localization algorithms, and research challenges for underwater acoustic sensor networks.' *Int. J. Commun. Syst.*, (pp. 3350–3365).

A. E. Turgut, H. Çelikkanat, F. Gökçe, and E. Şahin (2008). 'Self-organized flocking in mobile robot swarms.' *Swarm Intell.*, 2(2-4):97–120.

B. Tutuko and S. Nurmaini (2014). 'Swarm Robots Communication-base Mobile Ad-Hoc Network (MANET).' *Proc. Elect. Eng. Comput. Sci. and Informatics*, 1(1):134–137.

H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar (2002). 'Miro-middleware for mobile robot applications.' *IEEE Trans. Robot. Autom.*, 18(4):493–497.

K. P. Valavanis and G. J. Vachtsevanos (2014). *Handbook of Unmanned Aerial Vehicles*. Berlin, Germany: Springer.

A. Van Deursen, P. Klint, and J. Visser (2000). 'Domain-specific languages: An annotated bibliography.' *ACM Sigplan Notices*, 35(6):26–36.

M. Waibel, M. Beetz, J. Civera, R. d'Andrea, J. Elfring, D. Galvez-Lopez, K. Häussermann, R. Janssen, J. Montiel, A. Perzylo, et al. (2011). 'RoboEarth.' *IEEE Robot. Autom. Mag.*, 18(2):69–82.

J. Wan, S. Tang, H. Yan, D. Li, S. Wang, and A. V. Vasilakos (2016). 'Cloud robotics: Current status and open issues.' *IEEE Access*, 4:2797–2807.

Q. Wang, K. Jaffrès-Runser, Y. Xu, J. L. Scharbarg, Z. An, and C. Fraboul (2017). 'TDMA Versus CSMA/CA for Wireless Multihop Communications: A Stochastic Worst-Case Delay Analysis.' *IEEE Trans. Ind. Informat.*, 13(2):877–887.

Z. Wang and M. Schwager (2016). *Multi-robot Manipulation Without Communication*, (pp. 135–149). Tokyo, Japan: Springer.

Wikimedia Foundation (2019*a*). 'List of common swarm robots.' https://en.wikipedia.org/wiki/Swarm_robotic_platforms. Accessed on: 29-March-2019.

Wikimedia Foundation (2019*b*). 'List of embedded systems (WikiDevi Semantic Search).' https://wikidevi.com/w/index.php?title=Special:Ask&offset=0&limit=500&q=%5B%5BEmbedded+system+type%3A%3A %2A%5D%5D&p=format%3Dbroadtable%2Flink%3Dall%2Fheaders%3Dshow%2Fsearchlabel%3D...-20further-20results%2Fclass%3Dsortable-20wikitable-20smwtable&po=%3FEmbedded+system+type%0A%3FManuf%0A%3FCPU1+brand%0A%3FCPU1+model%0A%3FCPU1+clock+speed%0A%3FCPU1+cores%0A%3FFLA1+amount%0A%3FRAM1+amount%0A&eq=yes. Accessed on: 2-April-2019.

S. B. Williams, O. Pizarro, D. M. Steinberg, A. Friedman, and M. Bryson (2016). 'Reflections on a decade of autonomous underwater vehicles operations for marine survey at the Australian Centre for Field Robotics.' *Annu. Rev. Contr.*, 42:158–165.

Willow Garage (2017). 'PR2 Overview.' URL http://www.willowgarage.com/pages/pr2/overview. Accessed on: 10-Sept-2017.

S. Wilson, R. Gameros, M. Sheely, M. Lin, K. Dover, R. Gevorkyan, M. Haberland, A. Bertozzi, and S. Berman (2016). 'Pheeno, A Versatile Swarm Robotic Research and Education Platform.' *IEEE Robot. Autom. Lett.*, 1(2):884–891.

A. F. Winfield (2009). 'Foraging robots.' In 'Encyclopedia of complexity and systems science,' (pp. 3682–3700). Berlin, Germany: Springer.

A. F. T. Winfield, M. P. Franco, B. Brueggemann, A. Castro, M. C. Limon, G. Ferri, F. Ferreira, X. Liu, Y. Petillot, J. Roning, F. Schneider, E. Stengler, D. Sosa, and A. Viguria (2016). *euRathlon 2015: A Multi-domain Multi-robot Grand Challenge for Search and Rescue Robots*, (pp. 351–363). Berlin, Germany: Springer.

R. Wood, R. Nagpal, and G.-Y. Wei (2013). 'Flight of the Robobees.' *Scientific American*, 308(3):60–65.

S. Yaghoubi, N. A. Akbarzadeh, S. S. Bazargani, S. S. Bazargani, M. Bamizan, and M. I. Asl (2013). 'Autonomous robots for agricultural tasks and farm assignment and future trends in agro robots.' *Int. J. Mech. and Mechatr. Eng.*, 13(3):1–6.

Z. Yan, N. Jouandeau, and A. A. Cherif (2013). 'A Survey and Analysis of Multi-Robot Coordination.' *Int. J. Adv. Robot. Syst.*, 10(12):399.

M. Yin, M. Xie, and B. Yi (2013). 'Optimized algorithms for binary BCH codes.' In '2013 IEEE Int. Symp. Circuits and Syst. (ISCAS 2013),' (pp. 1552–1555). Piscataway, NJ: IEEE.

I. T. Young (1977). 'Proof without prejudice: use of the Kolmogorov-Smirnov test for the analysis of histograms from flow systems and other sources.' *J. Histochemistry & Cytochemistry*, 25(7):935–941.

J. Yu, S. D. Han, W. N. Tang, and D. Rus (2017). 'A portable, 3D-printing enabled multi-vehicle platform for robotics research and education.' In 'Proc. 2017 IEEE Int. Conf. Robot. and Autom. (ICRA 2017),' (pp. 1475–1480). Piscataway, NJ: IEEE.

J. Yuh (2000). 'Design and control of autonomous underwater robots: A survey.' *Auton. Robots*, 8(1):7–24.

J. Yuh, T. Ura, and G. Bekey (2012). *Underwater robots*. Berlin, Germany: Springer Science & Business Media.

M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. (2016). 'Apache spark: a unified engine for big data processing.' *Commun. ACM*, 59(11):56–65.

H. Zhang, A. Yang, L. Feng, and P. Guo (2018). 'Gb/s Real-Time Visible Light Communication System Based on White LEDs Using T-Bridge Cascaded Pre-Equalization Circuit.' *IEEE Photo. J.*, 10(2):1–7.