Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

The R2P framework for robot prototyping: methodological approach, hardware modules, and software components

Tesi di dottorato di:
Martino Migliavacca

Relatore:
Prof. Andrea Bonarini
Correlatore:
Prof. Matteo Matteucci
Tutore:
Prof. Carlo Fiorini
Coordinatore del programma di dottorato:
Prof. Carlo Fiorini

2013 — XXV ciclo

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I 20133 — Milano

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

The R2P framework for robot prototyping: methodological approach, hardware modules, and software components

Doctoral Dissertation of:
Martino Migliavacca

Advisor:
Prof. Andrea Bonarini
Coadvisor:
Prof. Matteo Matteucci
Tutor:
Prof. Carlo Fiorini
Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2013 — XXV edition

Acknowledgements

This work has been supported by research grants within the following research projects:

"Robotics for the Masses" by ST Microelectronics and Regione Lombardia

"ROAMFREE: Robust Odometry Applying Multi-sensor Fusion to Reduce Estimation Errors" PRIN 2009 grant by the Italian Ministry of University and Research (MIUR)

"SINOPIAE" by Regione Lombardia

Abstract

A physical prototype is essential since early stages of development of robotic systems, as they involve mechanical, electronic, and software components and the overall success of robotic applications depends on the performance, and on the interplay, of all of them. Unfortunately, set up a prototype is a demanding process that often drains resources from research activities, and prevents results to be transferred to real-world applications. This is one of the main limiting factors in todays robotic research and a major reason explaining the still missing entry of robotics into the mass market with the expected impact.

This thesis presents the Rapid Robot Prototyping framework (R2P), which aims at dramatically reduce time and efforts required to build a prototype platform, allowing to focus on research aspects instead of struggling on implementation details. Modular, open source, development is the way to obtain this result: modules can be developed and consolidated individually, by research groups with specific competences, and then reused in multiple projects, sharing solutions and improving reliability. Modular approaches are common in several fields, and have been recently applied to robot software development; in this thesis, we bring the same approach to hardware and low-level control.

Starting from the identification of a set of common requirements for robotic platforms, we have implemented specific, standardized, hardware modules, featuring on-board computation, each focused on the satisfaction of a specific functional requirement. We designed a communication protocol to enable the modules interact in real-time on a bus, allowing to accomplish complex tasks by the cooperation of distributed devices. A lightweight publish/subscribe middleware has been developed to bring to embedded firmware development the programming techniques which are currently restricted to high-level software, thus extending modularity to low-level control software. R2P also provides native interfacing to ROS, currently the most adopted software framework for robotics, enabling the developed platforms to be easily integrated in a large range of projects. Complex systems can now be implemented by assembling off-the-shelf components and easily programming their interaction, without the need for domain-specific knowledge in electronics and low-level control. In the thesis the overall approach has been validated with some use cases to demonstrate the effectiveness of the proposed approach on real applications.

Contents

1. Introduction **1**
- 1.1. Main contributions . . . 3
- 1.2. Thesis outline . . . 5

2. Robot development: from the idea to the prototype **7**
- 2.1. A robot in every home? . . . 7
- 2.2. The problem of robot prototyping . . . 9
- 2.3. Case study: building tiltOne . . . 12
- 2.4. A call for modularity . . . 14

3. A framework for prototyping robotic applications **19**
- 3.1. Towards off-the-shelf robotics . . . 19
  - 3.1.1. Quick installation and upgrade . . . 21
  - 3.1.2. Massive hardware and software reuse . . . 22
  - 3.1.3. Flexibility . . . 22
  - 3.1.4. Performance . . . 23
  - 3.1.5. Integration with high-level software . . . 24
  - 3.1.6. Ease of use . . . 24
- 3.2. The approach . . . 25
  - 3.2.1. Identification of functional requirements . . . 25
  - 3.2.2. Modularity at the hardware level . . . 28
  - 3.2.3. Distribuited computation . . . 29
  - 3.2.4. Open source development . . . 30
- 3.3. Rapid Robot Prototyping: the big picture . . . 31
  - 3.3.1. Physical connection . . . 31
  - 3.3.2. Real-time communication . . . 33
  - 3.3.3. Middleware . . . 34
  - 3.3.4. Integration with ROS . . . 35
  - 3.3.5. Hardware modules . . . 35
  - 3.3.6. Open source licensing model . . . 37

4. Real-time Communication **39**
- 4.1. Communication requirements . . . 40
  - 4.1.1. Definitions . . . 40
  - 4.1.2. Time-triggered and event-triggered architectures . . . 40
  - 4.1.3. Communication in robotic systems . . . 41

4.2. The CAN bus . . . 43
4.2.1. Physical Layer . . . 44
4.2.2. Transfer Layer . . . 45
4.2.3. Application Protocols . . . 50
4.2.4. Mixing Time- and Event- Triggered Traffic on the CAN bus . . . 54
4.3. RTCAN: a CAN bus protocol for robotic applications . . . 58
4.3.1. Goals . . . 59
4.3.2. Message types . . . 59
4.3.3. The communication cycle . . . 61
4.3.4. HRT message scheduling . . . 62
4.3.5. SRT and NRT message arbitration . . . 64
4.4. Benchmarks . . . 67
4.4.1. Experimental setup . . . 67
4.4.2. Results . . . 69

5. Middleware **73**
5.1. Decoupling data flows . . . 73
5.1.1. Message passing . . . 75
5.1.2. Request/response . . . 75
5.1.3. Shared memory . . . 76
5.1.4. Notifications . . . 77
5.1.5. Publish/subscribe . . . 78
5.2. Publish/subscribe middlewares in robotics . . . 79
5.2.1. ROS . . . 79
5.2.2. LCM . . . 81
5.2.3. Famouso . . . 82
5.3. A lightweight publish/subscribe middleware . . . 82
5.3.1. Nomenclature . . . 83
5.3.2. Nodes . . . 83
5.3.3. Message type specification . . . 85
5.3.4. Topics . . . 86
5.3.5. Publishers and subscribers . . . 88
5.3.6. Real-time support . . . 92
5.3.7. Transports . . . 93
5.3.8. Loading and configuring nodes . . . 94
5.4. Benchmarks . . . 95
5.4.1. Memory footprint . . . 96
5.4.2. Messaging performance . . . 97

6. Integration with ROS **101**
6.1. Interfacing hardware devices to ROS . . . 102
6.1.1. Related work . . . 102
6.1.2. Toward native ROS hardware components . . . 103
6.2. ROS communication model . . . 104
6.2.1. ROS Master . . . 104
6.2.2. Remote procedure calls . . . 105
6.2.3. Data transport . . . 105
6.2.4. Type descriptor . . . 106
6.3. µROSnode . . . 106
6.3.1. Architecture . . . 107
6.3.2. XML-RPC . . . 108
6.3.3. TCPROS . . . 109
6.3.4. Integration with user application . . . 110
6.4. Benchmarks . . . 111
6.4.1. Memory footprint . . . 113
6.4.2. Communication performance . . . 113

7. Hardware modules **117**
7.1. Reference design . . . 118
7.2. Power supply . . . 120
7.2.1. Hardware design . . . 120
7.2.2. Embedded software . . . 121
7.3. DC motor controller . . . 122
7.3.1. Hardware design . . . 122
7.3.2. Embedded software . . . 123
7.4. Inertial measurement unit . . . 124
7.4.1. Hardware design . . . 124
7.4.2. Embedded software . . . 125
7.5. Proximity sensors . . . 126
7.5.1. Hardware design . . . 126
7.5.2. Embedded software . . . 126
7.6. Input/output . . . 127
7.6.1. Hardware design . . . 127
7.6.2. Embedded software . . . 128
7.7. Gateway . . . 128
7.7.1. Hardware design . . . 129
7.7.2. Embedded software . . . 129

8. Use cases **131**
8.1. Triskar2 omnidirectional platform . . . 132
8.1.1. Platform description . . . 133

8.1.2. Control architecture . . . 133
8.1.3. Discussion . . . 136
8.2. Robocom differential drive robot . . . 138
8.2.1. Platform description . . . 139
8.2.2. Control architecture . . . 140
8.2.3. Discussion . . . 142
8.3. tiltOne balancing robot . . . 143
8.3.1. Platform description . . . 144
8.3.2. Control architecture . . . 145
8.3.3. Discussion . . . 147

9. Conclusions **149**
9.1. Conclusions . . . 149
9.2. Future work . . . 151

Bibliography **153**

A. R2P module schematics **165**

Chapter 1

Introduction

> *" Imagine being present at the birth of a new industry. It is an industry based on groundbreaking new technologies, wherein a handful of well-established corporations sell highly specialized devices for business use and a fast-growing number of start-up companies produce innovative toys, gadgets for hobbyists and other interesting niche products. But it is also a highly fragmented industry with few common standards or platforms. Projects are complex, progress is slow, and practical applications are relatively rare. In fact, for all the excitement and promise, no one can say with any certainty when - or even if - this industry will achieve critical mass. If it does, though, it may well change the world. "*
>
> Bill Gates, January 2007

The previous excerpt is from the first paragraph of the article *"A robot in every home"* published on Scientific American [58]. Playing on the duality with the early ages of computer industry, Gates is actually referring to the emergence of the service robotics industry, highlighting how the lack of standards and basic building blocks, to easily turn an idea into a working system, might actually prevent its success. Today, indeed, robots are widely exploited in manufacturing processes, assembly lines, military applications, or to assist surgeons, but they still fail in

hitting the mass market and entering our everyday life.

The problems stressed by the above mentioned article were solved, in computer industry, by fostering modular approaches: hardware has been implemented by assembling interchangeable components, with standard interfaces, and software has been organized in libraries, packages, and frameworks. Reuse of software among different projects and by several developers has become the way to cut down costs, speed up development, and ease maintenance. The same happened for several other mass products (cars, cameras, appliances, etc.), as modularity enables the reuse of components in different products, thus enlarging the share set, reducing costs, in terms of both time and money, and improving reliability, too.

In recent years, also the robotics community started to push for modular, reusable, development approaches, and a variety of software frameworks (e.g., ROS [111], Orocos [37], OpenRDK [39], and many others) were proposed to assist researchers and to allow the sharing of results among different applications. Several projects did benefit from the availability of a variety of ready-to-use software packages, focusing on their specific goals and not wasting time on the preconditions needed to test solutions and evaluate results.

Despite evident progresses, developing real world robotic applications is still challenging, as they cover many domains, and require specific competences in different fields. Mechanical, electronic, and software components are involved in every robot and the overall success of a robotic application depends on the interplay of all of them. A physical prototype is thus essential for any robot development, both to validate research results or to develop a new product. Unfortunately, set up a prototype is a demanding process that often drains resources from research activities: no standardized components for the development of generic robots exists, and systems are often implemented from scratch, even if they share most of the requirements. Many unexpected issues often show up at early stages of development, delaying, or even preventing, the translation of interesting ideas into a working system.

Starting from the previous considerations, we propose in this thesis a modular approach to robot prototyping, which aims at solving the common problems faced in the development of robotic platforms. The result of the thesis is *Rapid Robot Prototyping* (R2P), a framework providing hardware devices, software components, and tools to easily implement robot platforms. R2P relies on the principle that the functionalities identified in a robotic application can be implemented by modules not only at software level, as it is common in most frameworks, but also at hardware level. Functional requirements can be easily identified through classic decomposition of a system into smaller subsystems, and some of them

can be recognized as pertaining to the physical platform, and should be embedded into it, while others are dependent on the specific application, and could rely on high-level software frameworks, as nowadays best practice.

R2P is focused on implementing basic platform functionalities (e.g., motor control, distance measurement, inertial navigation) by means of specific, standardized hardware modules, with corresponding firmware, each focused on a particular requirement, fostering separation of concerns. Modules are then plugged on a common bus and interact in real-time, allowing to accomplish complex tasks by the cooperation of distributed devices. R2P also brings to embedded firmware development the programming techniques which are currently restricted to high-level software, so that modularity is extended also to low-level control. Robot prototypes can then be implemented by assembling and easily programming off-the-shelf components, relieving researchers and designers from spending time and resources on activities which are far from their focus. Finally, a lightweight ROS communication library allows to seamlessly integrate the hardware platform within ROS to implement high-level functionalities.

1.1. Main contributions

In the following we briefly summarize the main contributions of this thesis and, whenever these contributions have been published as peer-reviewed publications, we address the interested reader to them.

Distributed hardware modules. By analyzing several common robotic systems, we identified a set of functional requirements which can be implemented by means of distributed devices, with on-board computation, each focused on a specific task. Interconnecting the basic modules on a real-time communication channel, robots can be implemented as distributed systems with off-the-shelf modules, enabling massive hardware/software reuse and fast prototyping.

Andrea Bonarini, Matteo Matteucci, Martino Migliavacca, Roberto Sannino, and Daniele Caltabiano. *"Modular low-cost robotics: What communication infrastructure?".* In Proceedings of 18th World Congress of the International Federation of Automatic Control (IFAC), pages 917-922, 2011.

RTCAN. We designed and implemented a novel real-time CAN-Bus protocol focused on robotic applications, which aims at combining

the advantages of different approaches to communication scheduling and supports the requirements of a distributed architecture for robots. RTCAN takes into account time-triggered communication, e.g., from control loops, which are handled with a pure TDMA approach to guarantee temporal determinism, as well as event-triggered transmission requests, e.g., from sensors, which are scheduled with an deadline-based policy to reduce delivery latency.

Martino Migliavacca, Andrea Bonarini, and Matteo Matteucci. *"RTCAN: a real-time can-bus protocol for robotic applications".* In Proceedings of 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO), pages 353-360, 2013.

Lightweight publish/subscribe middleware. To support robot designers in writing of modular, reusable, software components for robotic applications on embedded platforms, we developed a lightweight publish/subscribe middleware targeted at resource-constrained devices. The middleware enables to implement low-level control software by means of distributed nodes, following a programming approach and a syntax most robot developers might be familiar with.

Martino Migliavacca, Andrea Bonarini, and Matteo Matteucci. *"A Lightweight Publish/Subscribe Middleware for Embedded Modular Robotic Architectures".* (Submitted to Journal of Software Engineering for Robotics (JOSER)).

µROSnode. µROSnode is a lightweight ROS client which can run on low-cost microcontrollers, publishing native ROS messages over TCP/IP. With µROSnode, high-level software written in ROS can be easily interfaced with low-level control software embedded in hardware modules. More in general, µROSnode enables users, developers, and companies to develop ROS-compatible devices for robotics, which seamlessly integrate within ROS systems.

Martino Migliavacca, Andrea Zoppi, Andrea Bonarini, and Matteo Matteucci. *"µROSnode - running ROS on microcontrollers".* Presented at the 2013 ROS Developers Conference, May 11-12, Stuttgart, Germany.

Rapid Robot Prototyping. The concepts and the technologies presented above have been integrated in the Rapid Robot Prototyping framework (R2P), which is the main result of this Thesis. R2P provides all the components needed to prototype a robotic application: a set of hardware devices focused on robot requirements, a real-time

communication channel and a publish/subscribe middleware to implement distributed low-level control, native integration with ROS and tools to ease the development process.

The main concepts of the framework have been presented in:

Andrea Bonarini, Matteo Matteucci, Martino Migliavacca, and Davide Rizzi. *"R2P: an Open Source Modular Architecture for Rapid Prototyping of Robotics Applications"*. In Proceedings of 1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT'12), pages 68-73, 2012.

Andrea Bonarini, Matteo Matteucci, Martino Migliavacca, and Davide Rizzi. *"R2P: An open source hardware and software modular approach to robot prototyping"*. In Robotics and Autonomous Systems, (in press), 2013.

Examples of R2P applications are reported in:

Martino Migliavacca, Andrea Bonarini, and Matteo Matteucci. *"Modular Development of Mobile Robots with Open Source Hardware and Software Components"*. In Proceedings of 17th annual RoboCup International Symposium, (in press), 2013.

Davide A. Cucci, Martino Migliavacca, Andrea Bonarini, and Matteo Matteucci. *"Development of Mobile Robots Using Off-The-Shelf Open-Source Hardware and Software Components for Motion and Pose Tracking"*. (Submitted to 2014 IEEE International Conference on Robotics and Automation).

1.2. Thesis outline

A brief outline of the thesis structure follows.

Chapter 1 briefly introduces motivations, goals and approach of the presented work, and reports the main contributions.

Chapter 2 extends the motivations of this thesis, starting by the observation that robotics still failed to enter into our everyday life. The process of robot prototyping is identified as one of the main reasons currently limiting robotic research; to overcome this problem, we propose to apply to robot development a modular approach, as it did already boost progress in several other fields. A review of related work concludes the Chapter.

Chapter 3 introduces the R2P framework. The main goals of the framework are firstly detailed and motivated. Then the proposed approach is described, from the identification of robot functional requirements to their implementation through distributed, reusable,

hardware and software modules. An overview of the framework, outlining its main components, is finally presented for an easier reading of the rest of the thesis.

Chapter 4 presents RTCAN, a real-time CAN bus protocol focused on robotic applications. After highlighting communication needs of common robotic systems, an introduction to the CAN bus and a review of existing communication protocols are reported. Then RTCAN is introduced: its main objectives, the mixed approach to communication scheduling, as well as communication benchmarks, are presented.

Chapter 5 is focused on the middleware used by distributed R2P components to exchange information. The publish/subscribe communication paradigm is identified as effective solution to decouple data flows, and common middlewares for robotics are reviewed. R2P lightweight publish/subscribe middleware is then presented, with implementation details and communication benchmarks.

Chapter 6 introduces µROSnode, a lightweight ROS client for resource-constrained devices. µROSnode enables direct integration of R2P modules with ROS systems: topics published on the R2P network can be accessed from ROS nodes, and, at the same way, R2P modules can subscribe data published by ROS software.

Chapter 7 illustrates the hardware modules we already developed to satisfy some common requirements of robotic systems. In particular, we describe the power supply module, the motor controller, the inertial measurement unit, an interface to proximity sensors, the generic input/output board, and a gateway module providing Ethernet and USB connections.

Chapter 8 validates the proposed approach by reporting some use cases. We used R2P to prototype some robots, with different goals and requirements, showing how its modular architecture can be exploited to develop a variety of applications.

Chapter 9 concludes the thesis, reviewing obtained results and outlining planned future improvements.

Appendix A reports the schematics of the R2P modules presented in Chapter 7.

Chapter 2

Robot development: from the idea to the prototype

A prototype is a precondition for any robot development, aimed both at the validation of some research result or at the development of a new application. Robot prototyping is not a trivial process, requiring competences from many domains; developers with strong knowledge in their own field may have little or no experience about other domains, and building a prototype often drain much more resources than expected. For these reasons, we identify the robot prototyping process as one of the main limiting factors in today's robotic research, and a major reason explaining the still failed entry, apart from a few examples, of robotic applications into the mass market, contrary to many forecasts.

In this section we highlight the need of a prototype while developing robotic applications, and review several problems often faced in common approaches to robot prototyping.

2.1. A robot in every home?

Robotics has been expected to enter our everyday life not only by roboticists, but also by technology entrepreneurs, financial analysts, market experts, and even governments. The South Korean government, for example, announced in 2006 that personal robots will have 100% Korean

market coverage some time between 2010 and 2020 [53]. With a more conservative approach, the United States Government sponsored the *2013 Roadmap for U.S. Robotics* [42], drafted by leading U.S. roboticists, which also includes a section for service robotics. Similarly, the European Union invited researchers to contribute to the *Robotics2020 SRA Roadmap* [65], to collect proposals and opinions about several topics related to robotics, from technologies to ethical and legal issues, expecting the entry of robots in our society.

Despite such prominent expectations and efforts, it is evident, especially to people actually involved in robotics research, that we are still far from having robots ready to enter in every home. The only noticeable example of a robotic application successfully hitting the mass market are autonomous vacuum cleaner: the *Roomba* (pictured in Figure 2.1), for instance, was introduced by *iRobot* in 2002, and has been reported to be sold in over 8 million units by 2013 [74].

Roomba is a small differential drive base featuring a rotating brush spinning under the frame, capable to collect dust and light dirt, in order to help people in keeping their homes clean. The robot is equipped with bump sensors to avoid obstacles (actually, to *react* to the obstacles it bumps into), infrared proximity sensors to recognize walls and move along them, and a capacitive sensor allowing to recognize dust. It follows some preset motion patterns, using the sensors to prevent from getting stuck or falling down stairs, and it is able to locate the recharging station and return to it on low batteries. The robot itself is pretty simple, but very well engineered, targeted to the market, and capable of autonomous operations (but it still needs a human intervention to empty the dust tank). Although Roomba does not completely substitute a conventional cleaner, it has shown that robots can help in everyday life, and its very good reception by consumers demonstrated that people want robots in their homes. As a consequence, many manufacturers started offering their version of autonomous vacuum cleaners, actually creating a new, and growing, market.

Apart from this well known case, and a few other examples, there are almost no robots available on the mass market. This is in contrast not only with the expectation we mentioned, but also with the several demonstrations of achieved robot capabilities which have been presented, in last years, by robotic researchers. The main problem we see with the development of novel robotic applications is to transfer the results of research efforts into real-world applications. To this extent, an early requirement is the development of a working prototype to test an idea on.

Figure 2.1.: The Roomba autonomous robotic vacuum cleaner. It has been reported that more than 8 million units have been sold by 2013, making it the first robot successfully entering the mass-market.

2.2. The problem of robot prototyping

Developing a robotic system is a challenging and demanding task, involving multiple fields, such as mechanical design, electrical engineering, software development, signal processing, control theory and artificial intelligence. Researchers all over the world are focused on core requirements of robots, from autonomous navigation to computer vision, from machine learning to human-robot interaction. Advances in individual fields, such as computer vision, simultaneous localization and mapping (SLAM), or motion planning, are remarkable, and several very promising results have been obtained in recent years. If considered singularly, it is often possible to evaluate solutions to these problems by setting up simple systems (e.g., moving a camera by hand to benchmark a SLAM algorithm), or even relying on virtual setups (e.g., by simulating an environment to debug a motion planner). But, also having solved several relevant problems, the overall success of a robotic application depends on the interplay of all these components. As a consequence, it is almost always required to transfer research results on a real, working, robot to truly evaluate its performance.

The importance of having a prototype is evident, for example, in the field of autonomous, self-driving, cars: major advances have been achieved within the *Grand Challenge* competitions [4], which lead several teams to equip real vehicles with sensors and try to have them autonomously drive from a point to another, in real environments. In the first edition, in 2004, none of the robot vehicles was able to complete

a path through a desert area. In 2005, all but one of the 23 finalists in race surpassed the distance completed by the best vehicle in the 2004 race, and five vehicles successfully reached the target. The next edition, in 2007, was held in a urban environment, with traffic lights, cars driven by humans, and even pedestrians; despite such a much more challenging situation, 6 over the 11 finalists did success. Later, Google invested on one of the principal teams participating to the Grand Challenge, starting the *Google car* project, which developed the first autonomous car enabled to ride on public roads, and, hopefully, will lead in the next years to the production of the first commercial driverless car.

Prototyping a robotic system, though, is not a straightforward process. Several non trivial steps are involved: designing and building the mechanics, selecting the electronic devices to control it, integrate the several components, write the low-level control software and, finally, interface it with the high-level software. Moreover, to effectively assist robot designers and researchers, prototypes should also be reliable, a requirement not so often satisfied by custom setups worked out in research laboratories.

Mechanical designs are frequently application dependent, needing ad-hoc configurations which are hardly achievable by off-the-shelf robotic platforms. Robot frames can be realized by using standardized parts, such as aluminium profiles, with the respective joints, which allow to build relatively complex structures with simple tools. Several building systems are available on the market, and are exploited since years to build a variety of robots, from wheeled and legged mobile platforms to assistance and service robots [64, 108].

Having designed the frame,the process of selecting the electronic devices, e.g., sensor, actuators, and controllers, starts. Looking at today's possibilities, we can pick devices either from the automation market or from the hobby market. Components from automation market (e.g., Beckhoff automation systems [1]) are often expensive and offer overkilling performance with respect to the requirements of a robotic application prototype. Moreover, automation devices often need power supplies not suitable for battery powered systems like mobile robots. On the other hand, devices from the hobby market (e.g., Arduino [25]) are usually cheap, but they show poor performance, low reliability, and no real-time capabilities making it impossible to implement advanced robot requirements, such as distributed control loops. Another frequently adopted approach is to design custom devices from scratch, to satisfy the needs of a specific application. This is a process requiring technical competences which may be out of the scope of research goals, and, sure enough, greatly increases the development time. Custom so-

lutions also may prevent the reuse of developed components through different projects.

With no standardized components to use, also physically connecting devices becomes a problem, as different power requirements, communication media, and data protocols, are involved. It is common to have multiple power supplies and complex wiring, adding new points of failure to a system. Using heterogeneous devices also requires specialized drivers and interfaces, leading to design robots as centralized systems, where all the devices are connected to a single processing unit, e.g., a computer, thus preventing device-to-device communication. As we will see, many low-level control tasks may be implemented directly at hardware level, reducing latency, allowing real-time interactions, and improving reliability. If, instead, the robot hardware is controlled by an embedded processor, e.g., a microcontroller, specific skills are required, having to deal with registers and peripherals at hardware level and, often, with the execution of critical, real-time, tasks.

The last step often needed to prototype a robotic application is to write the high-level software controlling the robot. Facilitating developers in writing software for robotics has been a hot topic in the last years, and several frameworks have been introduced (see Section 3.2.1). Here, we mainly focus on the problem of interfacing a physical robotic platform with the high-level software. Widely adopted frameworks such as ROS, the *Robot Operating Systems*, provide drivers for several robotic platforms, such as the PR2 [56] service robot, the Turtlebot [57] mobile platform, the NAO humanoid robot [63], a few manipulators, and other robots. Although it is very easy to start using robots with off-the-shelf support, such platforms are often costly, and may not be suitable to investigate specific robotic applications needing particular mechanics. On the other hand, integrating hardware with high-level software may require, first of all, an adapter to physically interface a computer to the embedded devices (e.g., a CAN bus adapter) with the respective device drivers. Then, the manufacturer-dependent protocols have to be implemented on the computer, and interfaced to the high-level software, e.g., through an abstraction layer.

The issues reported so far are only the most common difficulties which need to be frequently faced to get a working prototype, and many other side problems often arise. Developing the *prerequisites* to start investigating a new application often takes more time than the development of the application itself, as we stress in the next section.

Figure 2.2.: TiltOne, a balancing robot we built in our lab in 2008.

2.3. Case study: building tiltOne

To point out the steps needed to get a working prototype and highlight the main issues, let's consider a robotic application we built back in 2008, named *tiltOne* (see Figure 2.2). It is a low-cost balancing robot supposed to self localize, recognize people, welcome them, and bring them to a given place, e.g., a booth in an open-air exhibition.

First of all, we designed and built the mechanical system, using standard aluminium profiles and common bicycle wheels. Three man months were sufficient to design and build the frame and the mechanics. Another activity was dedicated to select the devices needed to operate the robot: two motor controllers, an inertial measurement unit (IMU), a global positioning system (GPS) and a camera to recognize obstacles, people, and faces. Only at this stage, we realized that building a prototype to test our application would have been much more difficult than expected, as there were no ready-to-use standardized components that we could choose, connect together, and easily integrate in our project.

Having to rapidly develop a low-cost robot, we initially chose ready to use components from the hobby market, but they soon showed many issues. First of all the motor drivers were not suited for precise control, which is required to balance the robot, and we had to change them later.

At the same time, the inertial sensors used for tilt estimation were noisy and requested additional analog filtering circuits to get usable measures. At the end, a custom electronic board with a microcontroller, analog filters and interfaces for the different devices to control the robot was designed and built. To localize the robot and recognize obstacles, people, and faces, we used a commercial GPS receiver and a standard webcam, both with USB connection, which forced us to put a computer on board to run the GPS interface and the image processing software. The whole process of selecting, interfacing and testing these devices and the control board took as long as one man year.

The next step in prototyping our robotic system was the writing of the software that runs on the microcontroller and performs balance control. We started with the low level interface to the devices, then we added algorithms to estimate robot's inclination and to drive the motors. The software increased in complexity with time, and we decided to switch to a real time embedded operating system, which we had to choose after testing and evaluating some alternatives. The development of the whole balance control software took additional six man months.

Having the robot prototype ready, we finally started the development of our idea: a balancing robot that recognizes people and brings them to a given place. It took only three man months after the prototype was consolidated to have the robot balancing and driving around, while the GPS interface and image recognition software were reused from previous projects. Finally, the system was running.

Looking back to the development of this project, we realized that it took 24 man months to finish, of which only three have been spent on topics strictly related to the specific application. Most of the time has been spent in developing, testing and debugging the prototype we needed to validate the application. It is evident that this effort, in terms of money and time, is not compatible with the mass market requirements. Surely, we did mistakes, and time could be saved if different choices were taken, but we believe that the development process we walked through is, actually, very common when investigating new robotic applications. Besides the mechanical design, all other steps could have relied on standard modules aimed at robot prototyping. The devices we needed for the development of our idea are basic sensors and actuators common to almost every robotic application. If we had a set of ready-to-use modules, which we could just connect together in a plug and play fashion, we could have been able to develop the prototype in terms of weeks instead of years.

2.4. A call for modularity

Although robotic systems have been built for decades by integrating heterogeneous devices, or implementing custom solutions, we firmly believe that a modular approach based on off-the-shelf components would strongly help robot designers in developing new applications. For the vast majority of robotic applications, it is possible to identify a reasonably small set of common functionalities, which can be implemented in a reusable way by modular components. Standardized components have been widely recognized as fundamental in cost effective prototyping, design, and mass production. For instance, in the automotive field, the car platform is often designed to share mechanical and electronic parts among different models so that car manufacturer can reduce costs and leverage on platform sharing [100]. The same approach has been taken in the field of software engineering, leading to the development of new programming techniques which are now widely recognized as convenient to design and mantain complex systems. Software components, generally organized in libraries or frameworks, are re-used among different projects and by several software producers [68]. Also in the field of robotics, researchers fostered a modular approach since early years, to ease the implementation of complex requirements through the decomposition into simpler elements, which can then be shared and reused through many projects.

In recent years, major efforts have been put in modularization of high-level software for robotics; several development frameworks [37, 59, 71, 78, 111] have been proposed to assist researchers in the design of robotic applications. Perhaps one of the most successful projects is the Robot Operating System (ROS) [111], which aims at supporting robot development and research by providing a modular software framework, a communication infrastructure, and tools for debugging and inspecting robotic applications. Robot software implemented with ROS can benefit from a rich library of packages provided by other users, thanks to the open source development approach, to significantly reduce the development time of novel applications. ROS is becoming a *de-facto* standard in robotics research software development; its very good reception and its widespread adoption (see Figure 2.3) have shown that the community needed a framework to join research efforts, boosting the development of the high-level control software of robotic applications. Although we believe that ROS did succeed in its goal, the vertical development of autonomous robots, i.e., from mechanics to intelligence, still requires many low-level prerequisites that, as we stressed, are often implemented from scratch, leading to application-dependent, non reusable, solutions.

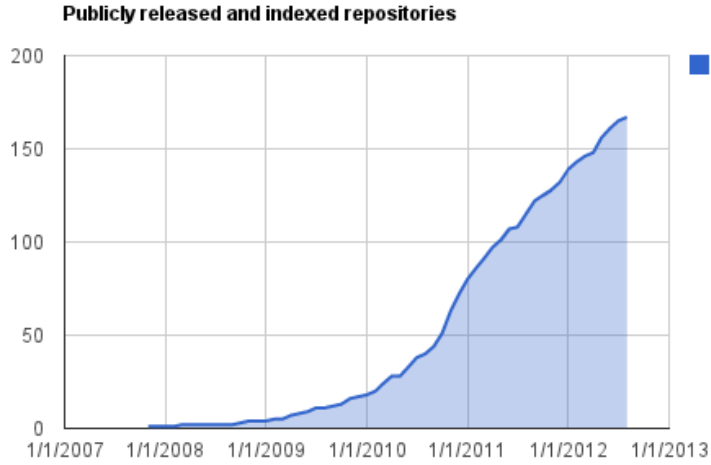


Figure 2.3.: More than 175 organizations or individuals publicly release ROS software in indexed repositories. Its widespread and fast adoption showed that robotics needed a modular approach to speed up the development of novel applications.

Moreover, ROS is designed to run on desktop-class computers, with demanding requirements in terms of memory and processing power, thus it cannot be used to write software for embedded targets. To the best of our knowledge, the only software frameworks focused on embedded development are FAMOUSO [120] and Aseba [91], which provide a communication infrastructure and development tools for the control of event-driven architectures. A more detailed review of these two frameworks will be presented in Chapter 5.

Efforts to extend modular approaches to hardware prototyping have been made mainly to produce educational and toy robotic construction kits, and in the field of swarm robotics. Several toys producers feature kits (e.g., Lego Mindstorm [82] and NXT [82], Fischertechnik Robot Construction Kits [52], or Modular Robotics roBlocks/Cubelets [115, 121]), which provide easy to connect mechanical parts and basic devices, such as small motors and simple proximity sensors. Despite these construction kits have been used also in academia within research projects [60, 123, 134], they are focused on teaching robotics to kids, and their application areas are very limited due to sensors and actuators performance, simple programming model, and limited processing power.

Ready to use modular mobile robotic platforms have been proposed

(a) e-puck

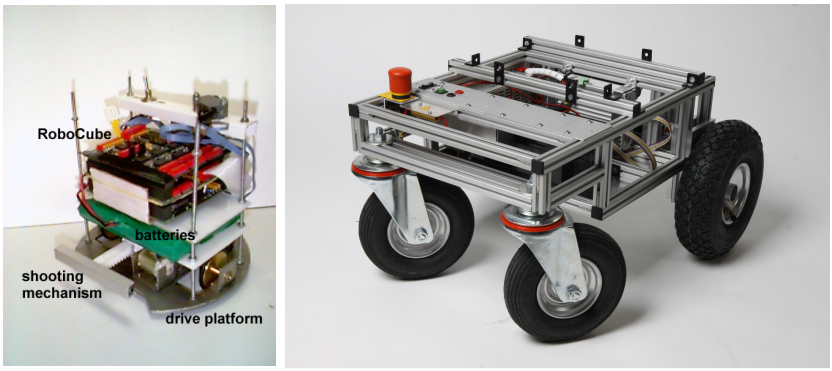
(b) Khepera-III

Figure 2.4.: Small mobile robotic platforms commonly used for educational and swarm robotics research purposes: the e-puck (a) is 70 mm in diameter, the Khepera-III (b) 130 mm.

for educational purposes and to facilitate research in the field of swarm robotics, such as the E-puck educational robot [41, 98] and the Kephera robot [66]; all of them offer a small mobile wheeled base to which accessories and custom boards can be added, providing additional actuators, sensors, cameras, and interfaces to allow interactions between the robots. Software tools such as graphical development environments, easy to use scripting languages, and debugging tools, enable short startup time while relieving the developers from working on the prerequisites of their novel application. These systems are widely used in academia, and their modular design did successfully facilitate advances in cooperative and swarm robotics research (e.g., at the moment of writing, a search on Google scholar for "Khepera" returns more than 6000 research papers).

More generic modular frameworks for robotics have been proposed too, such as the CubeSystem [28] and the VolksBot Construction Kit [136].

The CubeSystem is a collection of hardware and software components for fast robot prototyping, aiming at providing an open source collection of generic building blocks that can be freely combined into an application. It follows a tree-like architecture: a processor board is the central unit, to which additional boards are stacked to include sensors, actuators, or other devices. Programing is eased by RoboLib, a library providing common functions for robotics, and CubeOS, an operating system for the microcontroller employed on the processor board. The CubeSystem has been used to build some robots, from Robocup small



(a) CubeSystem (b) VolksBot

Figure 2.5.: Modular robotic systems. CubeSystem (a) is a set of stackable boards, with an embedded operating system and ready to use libraries. VolksBot (b) is a robot construction kit providing mechanic components, a motor controller board, and a few sensors; an on board computer runs the control software.

size league platforms, to a mobile surveillance robot and a humanoid torso [28].

VolksBot is a construction kit focused not only on electronics and control, but also on mechanical components. Standard aluminium construction extrusions are exploited to build robot frames, and ready to use mechanical components are provided, such as the *Universal Drive Unit*, a double layered aluminium frame including a 150 W DC motor, a drive coupler, and an air-filled tire. A motor controller board, capable of driving three motors up to 200 W, with encoder inputs and closed loop control, connects via serial interface to the on board computer. VolksBot sensors include an omnidirectional camera, a commercial range finder, together with machined plates to attach them to the robot frame. The robot requires an on board computer to operate, to which the motor controller board and the sensors are attached by RS232 serial ports. Software is designed using ICONNECT [124], a commercial framework for the visual composition of signal flow graphs. Several robots have been built with the VolksBot Construction Kit, such as RoboCup mid-size league and RoboCup@Home platforms, a robot to transport people, and a wheeled underwater vehicle.

Recently, the *Robotic Open Platform* (ROP) project [89] was started, aiming to make hardware designs of robots available under an open hardware license to the entire robotic community. Basically, it is a

website collecting CAD drawings, electric schemes, and the respective documentation, to build robot platforms open sourced by their original developers. This is obviously a very interesting initiative, allowing to easily share existing design avoiding to reinvent what have been already implemented by others. The listed robots, however, are based on heterogeneous architectures, each exploiting custom hardware devices and control architecture, preventing from sharing components and low-level control functionalities between different platforms. We aim to solve this problem, by proposing a framework that provides hardware devices, and low-level software components, which can be easily assembled and programmed to control a variety of robotic platforms.

Chapter 3

A framework for prototyping robotic applications

We can see robots as systems composed by both hardware devices, e.g., sensors and actuators, and software to operate them, e.g., filtering and control algorithms. The functional requirements needed to accomplish a task are then satisfied by the cooperation of hardware and software, as the hardware is necessary to physically interact with the environment while the software is needed to decide how this interaction has to be performed. With this vision in mind, we developed the *Rapid Robot Prototyping* framework (R2P), which pushes modularity concepts introduced by software engineers down to the hardware level: hardware modules are defined by analyzing common functional requirements of robotic systems, and the control software is written in a modular, loosely coupled, fashion, bringing common programming paradigms and patterns to resource-constrained devices.

3.1. Towards off-the-shelf robotics

The main goal of this thesis is to provide robot designers, researchers, and enthusiast in general, with off-the-shelf components to quickly set up generic robotic platforms, to enable them starting to work on their idea, allowing to avoid the many problems we commonly have to face in

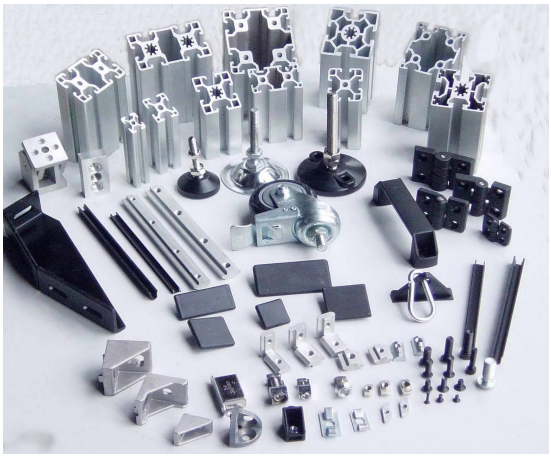


Figure 3.1.: Profiles, joints, and accessories of a modular mechanics building system.

the early stages of development of most robots.

We do not focus on the mechanical aspects of building a robot, mainly for two reasons. First of all, in our experience, mechanics is very application dependent, and providing basic mechanical components for generic applications is likely to reduce the flexibility and the possibilities of the framework, with the side effect of limiting its adoption to a specific class of robotic systems. Particular shapes are commonly needed, complying with strict weight and size constrains, which cannot be generalized. Moreover, for the range of robots which could take advantage from modular mechanics, we can already rely on multi-purpose building systems commonly used in the factory automation field. These systems provide several extruded aluminium profiles in various measures and shapes, and a huge set of accessories to join them and build a frame within hours (see Figure 3.1). Standard components are available from many distributors and can be reused through different projects, providing the modularity we need in most cases; for this reason, they are often used to build robot frames.

We focus on the low-level control of generic robotic platforms, meaning all those components that allow a robot to move, to perceive surrounding objects, and to efficiently implement core functionalities like kinematics, trajectory following, obstacle avoidance, and reaction to events. Mobile robots are currently the main targets of the proposed framework, but, thanks to the distributed architecture, and to the open source development model, it can be easily extended to cover a wide range of robotic

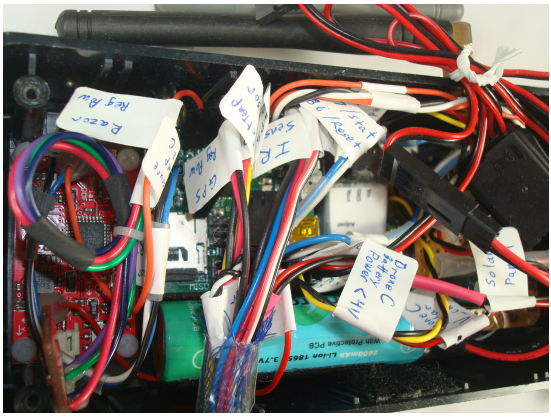


Figure 3.2.: The problem of robot wiring. Labels are mandatory to prevent mistakes. (Image from http://www.societyofrobots.com/)

systems. In the following, the main goals of the proposed framework are detailed and motivated.

3.1.1. Quick installation and upgrade

Every prototyping process of a new robotic system starts with the assembly of the frame and the selection and installation of the devices needed to operate the robot. With respect to the purposes of the proposed framework, we assume that the frame has been built, and the actuators have been selected and mounted. Commonly, here starts the often non trivial process of identifying the right motor controllers, sensors, and other needed devices, and the even less trivial task of wiring up everything, providing the right power supplies, connecting data cables, and, eventually, interfacing the embedded devices with an on board computer. This lead to complex wiring (see Figure 3.2), making the system unreliable and difficult to upgrade.

The first goal of an effective framework for robot prototyping is to solve these common problems. First of all, we aim at providing a set of off-the-shelf hardware devices which can be used to actuate a wide range of robotic platforms, eliminating the time consuming task of finding the right component, often approached by trial and errors. Then, we want to eliminate the complex wiring often found on robots, with different supply levels and data protocols within a single system, which adds several points of failure to the system. Aside from easing up the initial building of a platform, it should also be seamless to upgrade an existing

system by adding new devices, e.g., additional sensors, on the need; this sets an additional goal for the framework.

3.1.2. Massive hardware and software reuse

We introduced in Section 3.2.1 the benefits that a modular approach could bring to the development of new robots. Frameworks like ROS allow to implement new applications by simply integrating existing software packages, which are reused within many different projects. The advantage is not only in terms of development time, but also in terms of quality, as reusable modules receive contributions by several developers all around the world, refining the code and leading to better implementations. In our lab, as in many other research laboratories, both in academia and industry, we take daily advantage of using ROS to develop the high-level software for our robots, and it is common to discover that the functionality we just realized to need has been already implemented by someone else and it is available as ROS package.

As already stressed, we are firmly convinced that such a win-win approach should not be limited to the development of high-level robot software, but needs to be extended to the hardware level and to the development of low-level control components, which are too often reimplemented from scratch. Modular, reusable, hardware, with consistent interfaces and an easy to use firmware programming paradigm, could be shared among different projects and laboratories, with improvements in terms of performance, reliability, and development costs. The idea of developing a framework for the rapid prototyping of robotic platforms was born with exactly this goal in mind: bring the advantages that ROS provided for the development of high-level software down to hardware level, fostering massive reuse of developed components.

3.1.3. Flexibility

Research in robotics is advancing in several domains, like autonomous navigation, self localization and mapping in unknown environments, trajectory planning, motion control, robot-human interaction, edutainment and robot games, artificial intelligence, cognitive robotics, and many others. Robotic systems adopted in these fields are very diversified: from small, educational, robots, to autonomous mobile vehicles, from flying platforms, to manipulators and humanoid robots. Even if these platforms may be apparently different, the prototyping issues previously mentioned are common to many, if not all, of them, and solutions might be shared also through highly heterogeneous applications.

For this reason, we propose a framework as generic as possible, which can be exploited to speed up the development of a wide range of robots. Modular, effective, solutions dedicated to specific application fields are already available, as reviewed in Section 3.2.1, and they have been adopted by many research groups focused on that particular fields. However, it is common that research laboratories do not focus on a single topic, with people working on different, varied, projects, and a shared architecture would help transversely, allowing for a faster development in multiple projects. Moreover, the results of a single project could be tested on multiple platforms, in several environments, in order to evaluate and enhance their performance in different scenarios, still relying on the same interfaces, as the same building blocks are used.

3.1.4. Performance

Robots are complex systems, with several devices working together to accomplish tasks, eventually with real-time constraints. The overall result depends on how these devices work together, and the achieved quality is highly dependent on the performance of the hardware system. Reliability is important too: an unreliable robot slows down the development of any application, compromises results and often prevents from concluding a project on time. In our experience, available devices commonly show performance which may be too poor (e.g., when choosing to use hobby-market electronics), or very good, but at a high cost (e.g., industry automation devices). Custom devices are then often designed and built, increasing the development time and requiring deep technical skills. At the same time, control software, especially when working on embedded targets, needs to be carefully implemented to allow for good performance and prevent errors compromising the functionality of a system and leading to dangerous situations.

We need a framework which is not focused on building toy-like robots (*not only*, at least), but high-quality research platforms, where performance is a primary goal, both in terms of hardware and software components. Hardware devices must be robust and reliable, exploiting modern electronic components, and suitable for running robot platforms requiring precise motion control and accurate sensors. The same applies to embedded software, which needs to provide efficient implementations of common requirements (e.g., state of the art sensor filtering), and to support real-time interactions when needed.

3.1.5. Integration with high-level software

High-level robot requirements like reasoning, artificial intelligence, visual algorithms, or planners, received much attention by roboticists, and the need of an easy way to develop and share solutions to these problems led to the proposal of several framework for robot software development, as reported in Section 3.2.1. Those tasks are computational demanding, and the existing frameworks already offer effective programming environments and tools to perform research in those fields. We do not want to propose a framework that overlaps with already available solutions; on the contrary, we would like to take advantage of them and, in turn, bring some advantages to other projects focused on robot software development.

As integrating embedded devices with a full fledged computer, e.g., a desktop class computer, can be a non trivial process, often requiring hardware adapters and custom software; another major goal of the proposed framework is to offer an easy way to interface hardware platforms with high-level software. Benefits are evident from both point of views: platforms built with modular embedded components can take advantage of existing software packages to let the robot accomplish complex tasks, while high-level software frameworks can rely on the same interfaces to interact with a possibly wide set of robotic systems, without needing to write drivers or adapters for each platform.

3.1.6. Ease of use

A simple, intuitive, development approach is important to effectively reduce the effort while prototyping an application. This should not be ignored, since, besides the features it provides, a framework really speeds up the development process only if developers can easily adopt it, starting to be productive without needing a long practice time. Indeed, we believe that its simplicity and fast learning curve, which in turns lead to the composition of an extensive package repository, were the most relevant aspects determining ROS success over other proposed frameworks for robotic software. Ease of use is also important to support user-centered design approaches [104], that lead to many successfull applications, where users and domain experts are deeply involved in the development process. Even people with little or no expertise in robot development, or researchers that do not want to spend time in acquiring knowledge in fields far from their research interests, can easily build a working prototype of innovative applications, still relying on the technology provided by ready to use solutions.

This sets an additional goal for the prototyping framework: it has to be simple to use, in terms of physical assembly and low-level control development. At the hardware level, modules have to be easy to install, in a plug-and-play fashion, requiring no specific competences about electric characteristics, signal levels, or heat dissipation. At the embedded software level, we must provide users with an easy to use programming paradigm, tools for code development and debug, and a syntax most robot designers might be familiar with.

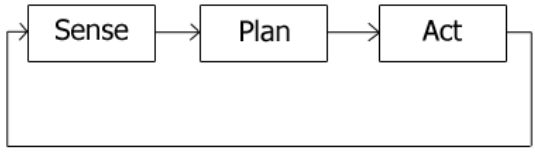
3.2. The approach

In this Section, the main concepts of the approach we propose are discussed: identification of requirements, implementation at hardware level, low-level control through a distributed architecture, and open source development.

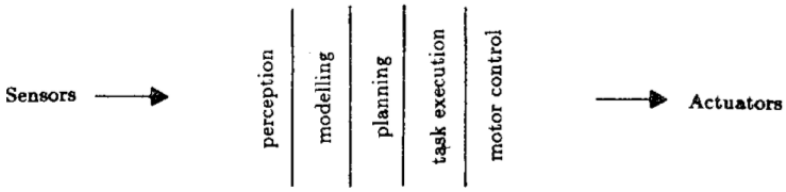
3.2.1. Identification of functional requirements

The design of a new robotic application usually starts with the identification of its functional requirements: what do we need to have our robot achieving its goal? To this extent, as robotic applications become more and more sophisticated, the complete system has to be decomposed into smaller sub-systems, to clearly separate concerns and to allow the development of each functional requirement as a stand-alone task. Although it may look as an obvious and trivial process, which has been discussed since years in the field of software engineering, the debate on how robotic systems should be decomposed lasted long, and it is still an hot topic. Since the very beginning of robotic research, several decomposition paradigms have been proposed; although they were mainly focused on defining a good control architecture of robots, the proposed approaches are also useful to identify functional requirements.

Historically, the traditional architecture is the so called *hierarchical/deliberative* paradigm, which follows a top-down approach: high-level goals are decomposed into several processing steps (i.e., *milestones* in the data flow from sensors to actuators) needed to achieve the desired goal. This architecture is also referred as *sense-plan-act* loop (see Figure 3.3). The result is the so-called *horizontal decomposition* of the problem in vertical slices, each identifying a processing unit which allows to accomplish a step in the data flow; Figure 3.3b reports the architecture of a mobile robot obtained by applying the hierarchical paradigm. To accomplish a task, the chain of processing steps is executed; maximum importance is given to high-level control, and low-level horizontal



(a) The sense-plan-act work flow schema.



(b) Classical hierarchical decomposition of a mobile robot control system into functional modules.

Figure 3.3.: The hierarchical paradigm.

communications are restricted as a consequence. Such serial execution flow introduces strong coupling between components: an instance of each of these components must be ready and running to operate the robot, and changes made to a component must not break the interaction with the adjacent ones, not to compromise the entire system.

A different approach has been proposed with the *behavioral/reactive* paradigm [21, 34]: rather than decompose a problem on the basis of data flow and internal processing, it is decomposed on the basis of desired external manifestations of the robot, called *behaviors*. Behaviors are defined as simple sense-act pairings, triggering actions through reactive control: by observing sensor values, each sub-system recognizes a stimulus and selects the corresponding output to produce (e.g., if the robot is approaching an obstacle, it should stop the motors). Goals are decomposed in tasks (i.e., sub-goals), instead of processing steps, each satisfying a requirement; this is referred as *vertical decomposition*, with tasks organized as horizontal slices, as shown in Figure 3.4. In this case, a bottom-up approach is followed: complex applications are composed by many independent, simple, behaviors executing concurrently; there is no predefined flow of information as in the hierarchical paradigm, and if a behavior does not work, this does not compromise the complete system. The behavioral paradigm does not prevent low level interactions; indeed, many behaviors interact directly with the hardware, e.g., keep balance or avoid obstacles.

With the purpose of identifying the functional requirements of a robotic

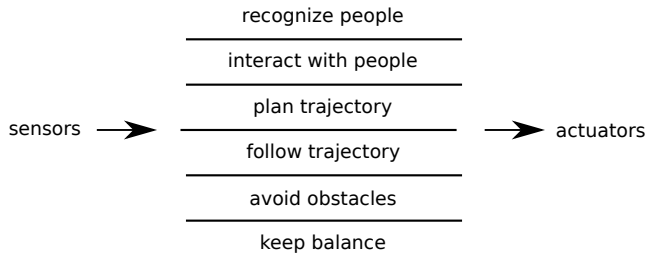


Figure 3.4.: Behavior-based decomposition of a balancing robot control system.

application, we can notice how the task-based analysis fostered by the behavioral paradigm enables an easier and clearer definition of single requirements, specially when moving down to the hardware level. Sensors and actuators involved in the execution of each task are precisely recognized, allowing to easily determine the required hardware devices. In the case of the application mentioned in Section 2.3, which involves a balancing robot interacting with people, functional requirements could be identified and detailed as follows.

- *Keep balance*: this task involves sensing the tilt of the robot, and compensating it by actuating the wheels. As a consequence, it requires an inclination sensor, two motor drivers, and a control algorithm to keep the center of gravity right over the wheels axis.

- *Avoid obstacles*: to this extent, the robot needs to sense surrounding objects and to inhibit the current motion command to prevent collisions. Thus, one or more proximity sensors have to be used, and a way to inspect and eventually override motion setpoints must be provided.

- *Follow a trajectory*, the robot needs to perceive its own movements and to issue motion commands, trying to be as close as possible to the given path. To track robot movements, we need and odometric system, i.e., encoders attached to the wheels (or, commonly, to the motors) and the forward kinematic model to translate encoder readings into motion paths; in outdoor environments, a GPS receiver could also be used to improve odometry. Then, motion setpoints are issued in order to realize the needed movements.

- *Trajectory planning*: the planner needs to know the current robot position and the destination. Current position is already provided

by the odometric system used by the trajectory follower, while the destination, in this application example, is fixed.

- *Look for people*: the robot can identify humans by running a face detection algorithm, which analyzes images from a camera, or even by using an RGB-D device such as the Microsoft Kinect [137].

- *Interact with people*: the robot may use audio or video messages, lights, movements, and so on, to communicate with a human. For this application, we choose to use only audio messages, which are synthesized by a software running on an on board computer.

To summarize, the development of the example application requires, at hardware level, two motor drivers, two wheel encoders, an inclination sensor, a proximity sensor, a camera, and an interface to receive motion commands from the on board computer. At software level, we require the balancing algorithm, a system to avoid collisions, a trajectory controller, a trajectory planner, human-robot interaction functionalities, and face detection.

3.2.2. Modularity at the hardware level

The task-based decomposition of robotic systems also allows to easily distinguish requirements strictly related to the application, which we call *application requirements*, i.e., high-level tasks, from *platform requirements*, i.e., the low-level components needed by the platform to operate, regardless of the particular application. In the considered examples, we can identify the three lower tasks as platform requirements: the balancing robot has to be able to keep balance, avoid obstacles, and follow a trajectory in most applications, e.g., either if it is autonomous or remotely operated by a human, while higher-level requirements, i.e., plan a trajectory, interact with people, and look for people, are needed by the specific application. We believe that platform requirements should be implemented within the platform, so they can be shared by several applications, providing simpler integration and allowing reuse within multiple research projects in an easier way. Embedding such requirements in the robot platform also improves reliability, as solutions to low level issues are immediately shared between applications. Unfortunately, today, platform requirements are not implemented with the same modular, reusable, approach followed for the development of higher level application requirements, which are often implemented within common software frameworks for robotics. For example, thinking again to the welcoming robot, ROS users have packaged several trajectory planners,

implementing state of the art algorithms; toolkits to perform human-robot interactions within the ROS are available [79, 88], as well as a ROS package to recognize human engagement [114]; face recognition is available out-of-the-box using the OpenCV [32] libraries supplied with ROS standard distribution.

Ready-to-use components to satisfy platform requirements, on the contrary, are much more rare, and, when available, their application is limited to a particular platform or research field. Although some of those requirements may be fairly trivial to implement, the lack of frameworks focused on platform prototyping and low level tasks leads to develop ad-hoc solutions, often from scratch. We extend the modular approach, which has been demonstrated to effectively speed up the development of application requirements, down to hardware level, providing ready to use hardware modules to prototype robot platforms. Controlling motors and computing odometry, measuring objects proximity, estimating the attitude of a robot, or providing an interface to a computer, are needs shared by most robotic platforms, and they should be implemented by off-the-shelf components. With hardware level modularity, robots can be rapidly built by simply assembling off-the-shelf components which implement the identified platform requirements.

3.2.3. Distribuited computation

Low-cost embedded processors available today, like modern microcontrollers, allow to embed computation directly into devices, so, in addition to providing the hardware needed to perform a task, they can also run the control software to accomplish it. This is the approach of the so-called *smart devices*, i.e., hardware components capable of accomplishing tasks with some autonomy, which are becoming more and more common in industries as automation and manufacturing, as, without needing an external controller, they show high reliability, improved flexibility, and their installation is much simpler.

Hardware modules for robot prototyping should be smart devices, each featuring the hardware components, an embedded microcontroller, and the low level control software, to fully implement common platform requirements. Attitude estimation, for example, is performed within a module, featuring inertial sensors and running an algorithm on the on board microcontroller to filter and fuse the readings. In the same way, the motor driver module sports the high power electronics to drive a motor, and ensures that it actually follows the setpoint by running a closed-loop controller.

On-board computation allows for a more comprehensive separation of

concerns, as all the components, both hardware and software, needed to perform a specific task, are embedded on a single device.

The on-board processor can also embed a communication protocol, for an easier integration of the components, and enable direct device-to-device communications, so that some low-level functionalities can be provided through distributed hardware modules without needing to interact with any external computer, improving performance and robustness. Indeed, although a single module running embedded control software can fully implement a specific task, several platform requirements involve more modules. For instance, to keep balance we need both a module capable of estimating attitude, to measure the inclination of the robot, and motor drivers to actuate the wheels. Then, the modules interact by exchanging information (i.e., the current angle and wheel speeds), and the balance keeping requirement is realized by the control loop closed through the two devices. It follows that robots are implemented as distributed systems, with several devices cooperating to satisfy the needed requirements and reach their goals.

An additional advantage of the distributed approach is an easier addition of functionalities, without the limits imposed by computational requirements. In a centralized architecture, as a matter of fact, the processor imposes a limit to the available computational power; even by leaving free resources when developing the system, at some point, such limit could be faced. In distributed architectures, each component provides the processing power it requires, and adding new features does not affect the functionalities of the existing ones. A limit can be introduced by the communication bandwidth required by the overall system to let the devices interact, but this can be eventually solved by splitting a system in multiple subsystems, each with their own communication channel. It follows that flexibility is increased, as improving an existing system does not require a complete upgrade of existing devices.

3.2.4. Open source development

Open source software has been around since decades, showing how software projects can take advantage of community-driven development. Everyone can contribute, adding features and fixing bugs, to actively improve open source projects. In the last years, users became more and more attracted by open source projects, especially hobbyists and people involved in education, having the possibility to see how the software works under the hood and to hack the code to suit their needs. Recently, the open source concept has been extended to hardware too, with successful projects like Arduino [24], dedicated fairs all over the world, and

the Open Source Hardware Association, founded in 2011 [8]. This approach then gained popularity as development strategy of big companies too, e.g., in 2012 Google chose Arduino as the platform to develop new Android compatible devices [72]. Open source hardware started to gain interest also as a business model, where earnings come from services and from selling ready to use products, while being open source allows to improve quality, gives visibility, and attracts new users [130] [15].

3.3. Rapid Robot Prototyping: the big picture

This thesis presents the Rapid Robot Prototyping framework (R2P), which aims at dramatically reduce time and efforts required to build a prototype platform, allowing to focus on research aspects instead of struggling on implementation details. The framework is composed by several components (see Figure 3.5 for reference): a set of distributed hardware modules, a physical bus to interconnect them, a real-time communication protocol, a publish/subscribe middleware, and a communication library to interface with ROS. Although these components were developed as part of the R2P project, they may also be used individually and integrated within other existing frameworks; robot designers may benefit of the hardware devices and program them with custom software, or integrate R2P communication libraries within existing hardware, thanks to the open source development approach we foster.

In the following, we describe the architecture of the framework, introducing its main concepts and features; more specific details will be reported in the next chapters.

3.3.1. Physical connection

The first element to be defined to connect hardware modules is the physical connection. We decided to use a single connector to transport both power and data, to make the prototype building process as easy and quick as possible. Power is provided by a dedicated module (see Section 7.2), and the bus is designed to handle up to 20 hardware modules over an up to two meter long cable. To reduce wires and connections in the system, a daisy chain wiring schema is exploited, where each module has two ports to connect to the previous and the next component, as shown in Figure 3.5.

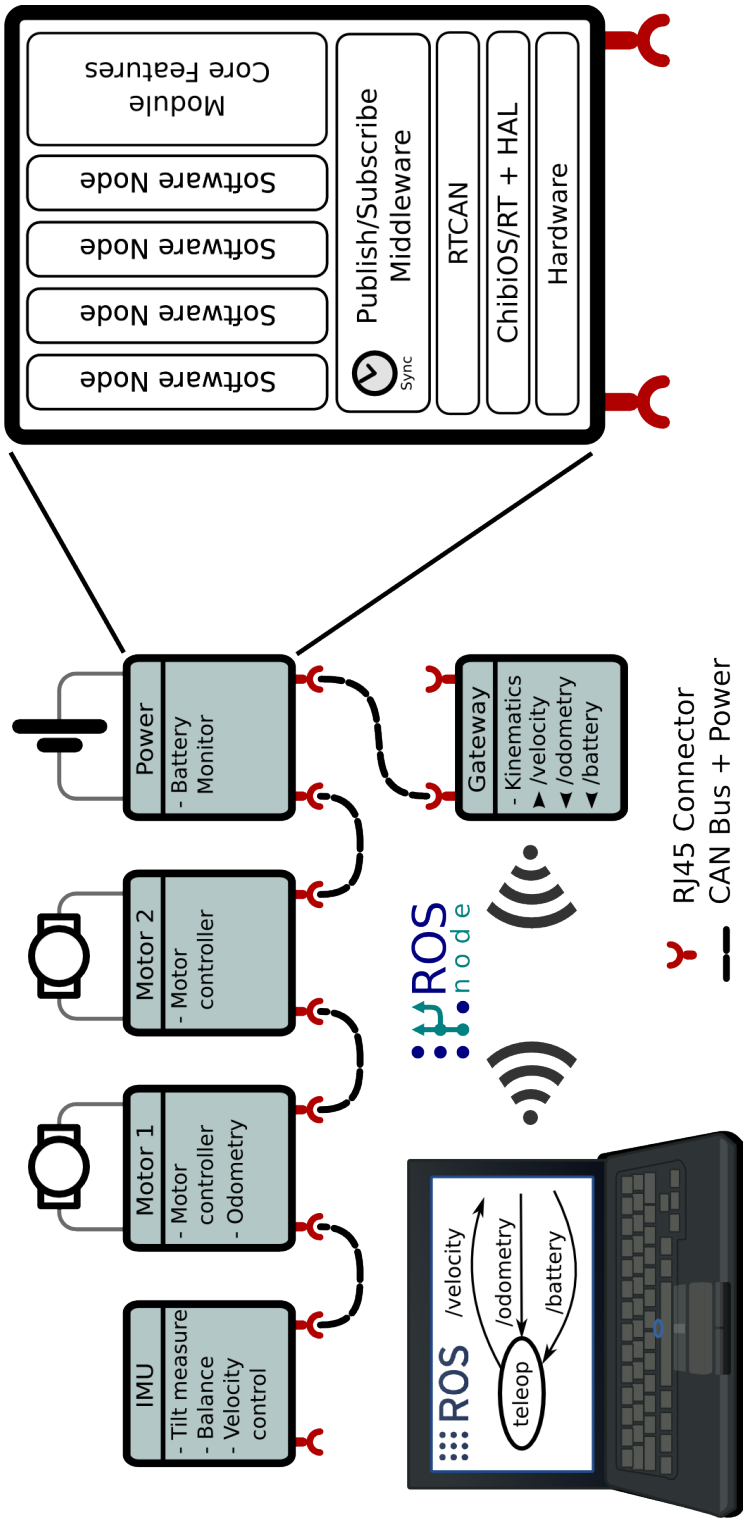


Figure 3.5.: Architecture of a balancing robot developed with the R2P framework.

We reviewed different communication standards [30], and we chosed the CAN bus to exchange data between modules, as it shows some features that best suit a modular architecture:

- it is a bus, so multiple devices can be connected on a single line;
- it is widely adopted in several fields, starting from the automotive field, so that most of today microcontrollers have an integrated CAN controller;
- CAN controllers can filter messages at hardware level, thus reducing processing requirements;
- it has been designed to work reliably in harsh environments;
- CAN transceivers are quite rugged, so they ideally suit the needs of fault tolerance intrinsic in prototyping.

The CAN bus has a maximum data rate of 1 Mbps, which is enough for a distributed system of smart devices where only pre-processed data is sent over the network, and raw data are processed locally [30]. Details about hardware characteristics of R2P modules are presented in Chapter 7.

3.3.2. Real-time communication

Distributed systems require reliable communication to let several components exchange information and perform a task. Moreover, robotic systems often involve tasks with real-time constrains, and the distributed architecture reflects such requirement to the communication channel. To define how communication occurrences have to be handled, a variety of issues must be considered: should transmission requests be triggered by time or by events? How priorities should be assigned to different data sources? Do we need a static or a dynamic network configuration? The constraints are hard real-time or soft-real time? There are many fields, e.g., factory automation, automotive networking, or sensor networks, where answers to these questions can be easily picked out, and one approach to communication scheduling can be recognized as preferable. This is not true for the flexible, generic framework we are proposing, and for robotic applications in general, where it is hard to define which communication paradigm is the best, as different requirements are needed by the different components of a complex robotic application.

The CAN bus presents several advanced built-in features suitable for real-time communication, such as priority-based bus access arbitration

implemented at hardware level, high determinism, and an efficient acknowledgement mechanism. A variety of communication protocols for the CAN bus are available, but, as we will discuss in Section 4.2, they are not focused on the requirements of robotic systems, and they do not provide the flexibility needed by the R2P framework. As a consequence, we developed RTCAN, a new CAN-Bus protocol focused on robotic applications, which aims at combining the best characteristics of different approaches to communication scheduling, providing time determinism, fast event response, and flexibility [96]. RTCAN schedules periodic communications with time-division multiple-access (TDMA) approach, which guarantees high determinism, low jitter, and improves reliability. Sporadic communications are handled by a best-fit approach based on Earliest Deadline First (EDF) scheduling, which increases priority of messages while they are approaching the transmission deadline, providing low latency and high flexibility. RTCAN is presented and detailed in Chapter 4.

3.3.3. Middleware

To support the development of modular software components on embedded targets, R2P features a lightweight communication middleware [95].

R2P middleware main goals are software reuse, real-time communication, efficient implementation, and ease of use. It follows the *publish/-subscribe* paradigm [49]: data producers *publish* messages on a *topic*, i.e., a communication channel, while data consumers *subscribe* the corresponding topic to receive messages. Identifying data by its content, i.e., the topic it is published on, instead of by its producer, promotes loosely-coupled software design and, thus, code reuse. The middleware provides concepts common to most robotics frameworks used on computer systems, such as software nodes, topics, publishers, subscribers, and message queues.

R2P middleware is written in a subset of C++, to take advantage of some object-oriented programming features without compromising performance on embedded targets. Its implementation is focused on code efficiency and messaging performance. Locally, messages are handled with memory sharing techniques, to achieve low delivery latency. Message queues, with the possibility to assign priorities to different nodes, guarantee that slow data consumers do not interfere with high-priority tasks. Software nodes can subscribe both local and remote publishers, with no difference from the user point of view. The middleware supports both periodic and sporadic publishers, which can specify real-time communication constraints, such as update period for time-triggered mes-

sages, and delivery deadline for event-triggered ones. Locally, processing nodes are implemented as threads, taking advantages of the underlaying RTOS to guarantee real-time execution; for remote messaging, real-time support is provided by the underlaying communication channel (i.e., RT-CAN). Finally, a simple API, which reminds the ROS syntax, enables developers to write distributed code, on embedded targets, as they are used to do on computer systems, fostering code reuse through different projects, and reducing the learning curve for robotics developers already used to ROS. The publish/subcribe communication paradigm, and its implementation provided by the R2P middleware, are detailed in Chapter 5.

3.3.4. Integration with ROS

While R2P supports rapid development of robotic systems using off-the-shelf hardware and software components, applications involving computation intensive tasks such as computer vision, localization, and complex planning, must also rely on a computer system and, eventually, a software framework. Among the many available development frameworks for robotics software, we chose to support ROS [111], as it currently is the most widely adopted in academia and research laboratories, and, recently, it is being considered also by industrial developers [118].

To natively integrate resource-constrained devices within ROS, we developed *µROSnode* [97], a lightweight, open source, ANSI C ROS client library publishing native ROS messages over a TCP connection. A R2P hardware module provides Ethernet connectivity and runs µROSnode, enabling direct integration of R2P modules with ROS systems. Topics published on the R2P network can be accessed from ROS nodes, and, at the same way, R2P modules can subscribe data published by ROS software. µROSnode is presented in Chapter 6.

3.3.5. Hardware modules

We have designed and built, as part of the R2P framework, a set of plug-and-play hardware modules that implement basic functionalities required by common robotics applications. R2P modules aim at filling the gap between hobby devices and automation components which are, currently, often used to prototype robot platforms. Modules are based on STM32 Cortex-M3 microcontrollers with $20Kb$ of RAM and $128Kb$ of Flash memory, running the ChibiOS/RT RTOS and the R2P middleware. Each module has two RJ45 ports for daisy-chain connection to the bus, a serial port for debugging purposes, and a JTAG header

for advanced users who want to directly access the microcontroller. A brief overview of the currently available modules follows. Notice that these are just few modules out of a possibly huge set that could be realized and shared with the R2P community, thanks to the open source approach followed for both software and hardware development. A detailed description of R2P hardware modules we developed is presented in Chapter 7.

Power supply. This is the power supply unit, which powers all the modules connected to the bus. Input voltage range is from $5.5V$ to $36V$ DC. A DC-DC converter produces a $5V$ regulated output with maximum current supply of $4A$ and short circuit protection. Both battery voltage and current drain can be published over the network to monitor power consumption and to estimate the residual battery life.

DC motor controller. This high-power motor controller board can drive DC motors up to $36V$, delivering a continuous $20A$ current. It features closed loop control, with position feedback from a quadrature encoder and current measurement from the on-board Hall-effect sensor. The DC motor module accepts position, speed, and current limit set points, and publishes position and speed messages, exploiting data from the encoder, and the measured output current.

Inertial measurement unit. A 10-DoF Inertial Measurement Unit featuring MEMS accelerometer, gyroscope, magnetometer and pressure sensor. An additional serial port to acquire GPS coordinates from an external GPS receiver is also provided on this module. The on-board sensor fusion algorithm computes heading and attitude information.

Proximity sensors. A module to interface with proximity sensors such as the Sharp IR rangers or *MaxBotix* [6] ultrasonic sensors. Each module connects to up to 4 sensors. Calibration and data filtering algorithms run on the microcontroller, which produces distance measurements.

Input/output. A generic module featuring several analog and digital inputs and outputs. It can be used to integrate existing devices, e.g., sensors and actuators, into a R2P system, or as a test platform while developing new R2P modules.

Gateway. This is a gateway module featuring an Ethernet port, an USB port, and a more powerful, Ethernet-enabled, microcontroller to handle the TCP/IP stack. R2P messages can be forwarded from the CAN-Bus to the IP network, or over an USB connection, and the other way around. By running µROSnode, it provides seamless integration of R2P-based systems within ROS.

3.3.6. Open source licensing model

R2P is fully open source, to be easily adaptable to specific application needs, and to take advantage of community-driven development to became a mature and widespread project. The schematics of R2P hardware modules, with the corresponding board designs, are distributed under Creative Commons Attribution-ShareAlike 2.0 (CC BY-SA 2.0) licensing [3]. The share alike model was chosen to foster the addition of hardware modules to the R2P ecosystem, allowing to provide more and more off-the-shelf devices satisfying common requirements of possibly diversified robotic platforms. The firmware running on the modules, instead, adopts the permissive BSD 2-clause license [13], to allow the use of low-level control software initially developed on R2P modules even in closed source, eventually commercial, projects with custom hardware. Board schematics and layouts, the corresponding firmwares, the real-time communication protocol, the middleware, together with some demonstration nodes, are available on R2P GitHub repository [112].

Chapter 4

Real-time Communication

A key aspect in distributed architectures is the way communication between the different components is handled. Several components are involved to run a distributed system, and they may show different, heterogeneous, communication requirements, which need to be taken into account by the communication layer. We chose the CAN bus as communication channel to connect R2P modules, as it allows for real-time communication, it is available on many modern microcontrollers and it is designed to work reliably in harsh environments. Existing CAN bus protocols, tough, are not suitable to handle communication in robotic systems, which are both event-triggered and time-triggered, and, do not provide the flexibility needed by a modular framework like R2P.

To overcome these problems, a novel communication protocol has been designed and implemented, named RTCAN (Real-Time CAN), which combines the advantages of different approaches to communication scheduling. This chapter introduces and details the RTCAN protocol, and it is organized as follows. Section 4.1 highlights the principal requirements of the distributed architecture proposed by R2P. Then, an introduction to the CAN bus is presented in in Section 4.2, with a review of existing protocols. In Section 4.3 RTCAN is presented, while Section 4.4 reports communication benchmarks.

4.1. Communication requirements

Robots developed with R2P are distributed real-time systems, with sensors, actuators and controllers working together to guarantee effective operation. It follows that the requirements of common tasks involved in robotic systems, such as timely execution of control loops or quick reaction to sensor readings, are propagated to the communication channel. In this section we briefly describe some of these tasks, highlighting the different paradigms which best handle them and, in turns, defining the communication requirements.

4.1.1. Definitions

A *real-time system* is a system where the correctness of its operation does not depend only on the value of its output, but also on the time when the output is produced [77]. Real-time systems are called *distributed* when they span across multiple nodes interconnected by a communication channel. To describe the temporal activity of a system, we call *instant* a particular point in the time line, and *event* an occurrence associated to a particular instant. The distance between two instants is called *interval*, and if event B should happen as a consequence of event A, the time delay between the observation of A and B is called *latency*. If a task is periodic, the interval between two executions is its *period*; if the period is not exactly the same at each repetition, the difference between the maximum and the minimum period is called *jitter*. The *deadline* is the instant within a task must be accomplished. Real-time tasks are classified as *hard real-time* if a missed deadline can imply severe consequences, or as *soft real-time* if a missed deadline is tolerable.

4.1.2. Time-triggered and event-triggered architectures

The choice between time-triggered and event-triggered paradigms to design real-time systems has been subject to a long debate [76, 105], which highlighted advantages and drawbacks of both approaches.

In event-triggered systems, every task is initiated by the occurrence of a significant change in the state of the system. For example, a two-states sensor, like a bump sensor or a switch, generates an event each time its value transits from one state to the other; in the same way, a generic transducer converting a physical quantity into a measure may generate an event every time the value changes, when it is above a threshold (i.e., a reference value), or when its changing ratio is above a threshold.

In the time-triggered paradigm, on the contrary, transmissions are

conducted at predefined instants, regardless of the state of the system. As a consequence, time-triggered designs lead to predictable temporal behavior, since each communication occurrence is planned at design time. This requires a detailed analysis of the overall system, which adds complexity to its design and limits further system expansions. On the other hand, a-priori scheduling allows to accurately test the system, e.g., to check messages schedulability, making it easier to guarantee quality of service. Time-triggered systems are generally static and defined at design time, but a centralized scheduler can be exploited to perform online admissibility control and to improve flexibility.

An event-triggered design does not need a-priori knowledge about the system to schedule communication, thus leading to a much more flexible design. But, as a consequence of flexibility, a much more extensive testing is required to verify that the system can handle communication requests under different load situations. Time-triggered communication shows high temporal determinism and, thus, low jitter. An advantage of event-triggered designs is, generally, a better resource exploitation with respect to time-triggered designs, with higher achievable throughputs [16].

4.1.3. Communication in robotic systems

A first source of communication in a distributed robotic system are control loops, which need to exchange data between sensors and actuators. These data transfers are, generally, periodic, deterministic, and known at design time. The update period is application dependent; generally, it may range from a few Hz to 1 Khz on common robotic systems. Data exchanged to run control loops is generally small in size, containing only a few values used to broadcast the system state or to set a command (i.e., a speed can often be expressed by a simple $16bit$ integer value), but it can also grow when representation of more complex variables is needed (i.e., the pose and attitude of a robot may be expressed by 6 32 bit floating point numbers, and other 6 may be needed to express also its linear and angular velocities, resulting in 384 bits transmitted for each update). Control loops often have hard-real time constrains, as missed deadlines may introduce error in the control action, compromising the operation of the system and even generating dangerous situations. Moreover, they are highly affected by the presence of jitter [110], which introduces variable delay and, thus, may induce overshoots of the control action and instability. Another common source of data transmission is the broadcast of system status, like a heartbeat signal, to suddenly halt the system if a failure happens. The system status could

be updated asynchronously on change, but updating it periodically is generally a safer solution: if the status is not received, every component of the system can enter a safe mode. Periodic communication is best handled in a time-triggered way, having transmission occurrences scheduled in a calendar with exclusive bus access, to prevent collisions and reduce jitter.

Besides the sensors used in control loops, in a robotic system, we find other kinds of data sources showing different requirements; for instance, proximity sensors and bumpers produce useful data only when triggered by some event, like approaching an obstacle, and the most important factor is to react to new readings as quickly as possible. As a consequence, delivery latency is, in many cases, the most important factor when dealing with event-triggered data. If sensor readings are transmitted periodically, the only way to reduce the worst case latency is to increase the update frequency, which leads to a waste of bandwidth when data are not relevant. Additionally, the worst case latency is still as long as the update period, and it is difficult to estimate the time elapsed between the occurrence of an event and its actual reception. It becomes clear that using event-triggered transmissions, for sporadic communication, saves bandwidth and, generally, reduces latencies. Depending on the particular task, transmission of event-triggered data may show different constraints; however, many event-triggered tasks can be considered as soft real-time, since the corresponding data sources will, generally, transmit again until the triggering condition is observed. Other sources of non periodic data are planners, which can be triggered by some event (e.g., the arrival of a new goal or a change in the environment) and their execution may be not constant in time; it follows that an event-triggered

Table 4.1.: Communication requirements of messages exchanged by an ideal balancing robot

	Time/event triggered	Update frequency [Hz]	Payload [bytes]	Real time
Motor torque setpoint	T	1000	2	hard
Motor speed setpoint	T	100	2	hard
Angle estimate	T	100	10	hard
Odometry	T	10	4	soft
Detected obstacles	E	-	8	soft
Trajectory waypoint	E	-	36	soft

messaging paradigm is preferable to update their output.

As an example, consider again the balancing robot presented in Section 2.3, which runs a control loop to keep its balance, another loop to execute motion commands, and sports several proximity sensors to avoid obstacles. Table 4.1 summarizes the requirements, in terms of data transfers, of its basic tasks, and the respective characteristics. Notice that some of these tasks may be not distributed in this specific application (i.e., they are executed within a single board), but, as an example of generic communication requirements in robotic systems, we report all of them. Impedance control of a robotic arm, for instance, may be distributed, as several joints contribute with their torque to the actual force applied by the end effector.

As a conclusion, in robotic systems both periodic, time-triggered, and sporadic, event-triggered, data transmissions are needed, thus a protocol which combines the two communication paradigms is desirable.

4.2. The CAN bus

The *CAN bus* (*Controller Area Network* bus) is a communication protocol designed to connect a network of independent controllers, usually resource-constrained devices, without the need of a host computer. The development of Controller Area Network bus was started in 1983 at Robert Bosch GmbH, as a vehicle bus protocol to connect the various subsystems, guaranteeing high reliability and security. Being adopted by the OBD-II and the EOBD vehicle diagnostics standards, which are mandatory for car producers respectively in North America and Europe, it rapidly gained popularity and it is now available as peripheral in most modern microcontrollers, or as stand-alone controller. In recent years, the CAN bus as also been adopted in different fields, such as industrial automation, and this led to the development of several CAN-enabled devices, such as Programmable Logic Controllers (PLC), as well as new communication protocols.

The CAN bus is a multi-master, broadcast, half-duplex protocol, which covers the first two layers of the 7-layer OSI specification (see Figure 4.1). It is multi-master as there is no need of a coordinator node in the network, and all nodes which need to transmit data can act as master. The arbitration process to elect the current master is done by the hardware, thus relieving the processor from running software implementations of the protocol. When a transmitting node becomes the master, data is broadcasted to all other nodes, which are all receivers and passively listen to the bus. As there can be only one master at

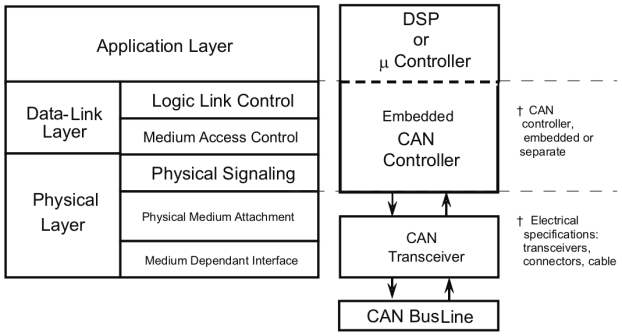


Figure 4.1.: The layered ISO 11898 CAN bus specification. (Image from Texas Instruments SLLA270 Application Report)

the time, communication is always from one single node to the others, implementing a half-duplex communication channel.

4.2.1. Physical Layer

Transmission Medium

Several physical mediums to interconnect CAN nodes are defined by different standards, such as the ISO 11898-2 (high speed), ISO 11898-3 (fault-tolerant), SAE J2411 (single wire) or ISO 11992 (point-to-point).

The most common standard is by far the high speed CAN bus, which specifies a two-wire differential bus (CAN_H and CAN_L signals) with a characteristic line impedance of 120 Ohm. The data rate is 1 Mbit/s over a theoretically possible bus length of 40 m, while for longer bus lengths the specified data rate is lower (allowing up to 1 km bus length at 50 Kbit/s); stubs, which are un-terminated, should not exceed 0.3 m in length. The maximum number of nodes is actually limited only by the electrical bus load (i.e., the overall load capacitance), but the high speed standard recommends to not attach more than 30 nodes on the same bus for 1 Mbit/s operation. The network topology is a single line, terminated by 120 Ohm resistors at both ends to avoid signal reflections, as shown in Figure 4.2; although not explicitly specified in the ISO standard, for high speed operation in harsh environment the use of a twisted-pair shielded cable is recommended [61].

The standard does not impose specific connectors, but they should not affect the bus operating parameters; higher layer protocols, such as CANopen [31] and DeviceNet [103] specify connectors required to implement compliant devices, which include 9-pin DSUB, RJ10, RJ45,

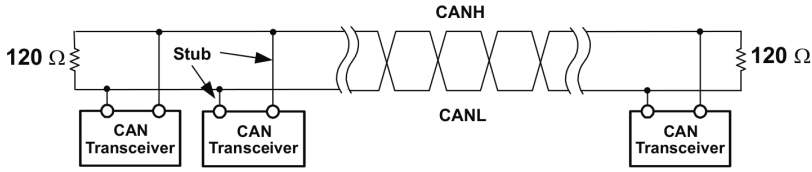


Figure 4.2.: A 120 Ohm terminated CAN bus. (Image from Texas Instruments SLLA270 Application Report)

M12 and others.

Signal Levels and Bit Representation

CAN specifies two logical states to represent bits: recessive and dominant. According to ISO-11898-2, recessive and dominant states are represented by a differential voltage, which is applied between the CAN_L and CAN_H signals of the bus. If the differential voltage is lower than the minimum threshold (1.5 V on the transmitter side, 0.5 V on the receiver side), the state is identified as recessive, which usually identifies a logic 1. Otherwise, a differential voltage greater than the threshold represents a dominant state, usually identifying a logic 0. The terms *dominant* and *recessive* refer to how the signals affect the input of CAN line drivers. In Figure 4.3 the block diagram of a typical CAN transceiver is reported: bot the high and low side are driven by open-drain (or open-collector) outputs, meaning that they can only impose a dominant state (a differential voltage over the threshold) on their output, while the differential voltage is driven low by the external terminator resistors. In other words, if at least one of the transceivers connected to the bus outputs a dominant bit, the state of the bus becomes dominant regardless of any other recessive bit outputs from the other nodes. This is the foundation of the nondestructive bitwise arbitration of CAN, which will be described in the following.

4.2.2. Transfer Layer

Bit Timing and Synchronization

The CAN bus encodes data with Non Return to Zero (NRZ) line code: the signal can be a logic 1 or a logic 0, and there is no neutral position. As the CAN bus is an asynchronous communication channel (i.e., there is no explicit clock), and NRZ is not a self-clocking representation, the bit time must be the same over all the network, and nodes are requested

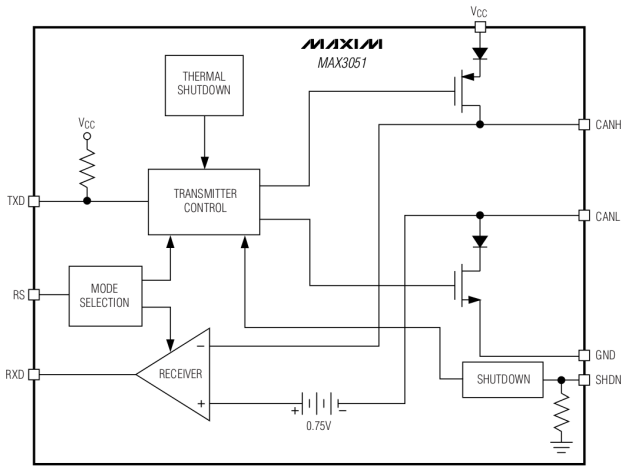


Figure 4.3.: The typical structure of a CAN transceiver. (Image from Maxim 3051 datasheet)

to be synchronized. To this extent, each CAN controllers runs a synchronization algorithm which adapts the receiver bit rate to the actual rate of the bits transmitted on the bus, aligning to the transition edges.

The CAN bit time is composed by four time segments, as illustrated in Figure 4.4:

- *synchronization segment*, used for synchronization purposes
- *propagation segment*, which is twice the delay introduced by the line driver and the signal propagation over the bus
- *phase 1 segment*, the interval before the bus level is sampled
- *phase 2 segment*, the remaining time after the sample point

Each segment is multiple of the interval t_q, the *time quantum*, which is generated by a free-running timer. The synchronization algorithm implemented in CAN controllers acts in two moments: at the beginning of each frame, when the bit time counter is restarted, and at each successive transition edge during the transmission, when the sample point is advanced or retarded depending on what is observed on the bus. This process guarantees, if there are no error conditions, the bit synchronization of all nodes, but works only if there are transitions between the two possible values of the bus. For this reason, the CAN bus imposes that there are no more than 5 consecutive equal values on the bus, and the

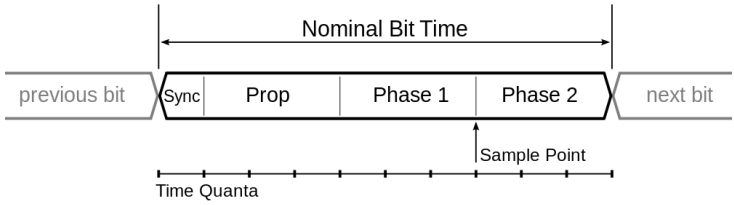


Figure 4.4.: CAN bit timing.

controllers apply a *bit stuffing* protocol introducing a transition every time the condition is not matched. The bit stuffing protocol implies that the length of frames is not constant, also if the payload does not change in size, as the amount of bits introduced (up to 26 in a single frame) depends on the particular data transmitted. This variable length needs to be taken into account for time-triggered protocols, as the frame time cannot be assumed fixed.

Message Frames

The CAN 2.0b specifications defines four types of frames which can be transmitted on the bus:

- *Data frames* carry data from a transmitter to the receivers
- *Remote frames* are sent by a node to request a Data frame with the same identifier
- *Error frames* are transmitted when bus errors are detected
- *Overload frames* can be transmitted to insert a delay between consecutive Data or Remote frames

Here we focus on the Data frame, which is relevant for the implementation of the protocol we are presenting in this chapter.

A CAN Data frame, as shown in Figure 4.5, is composed by several fields. The start of a new frame is identified by the *Start of Frame* (SOF) bit, a single dominant bit, used to synchronize the receiver clocks as explained in Section 4.2.2.

Then, the first field of the Data frame is the *Arbitration field*, containing the identifier of the message being transmitted. For the Arbitration field, two CAN bus specifications are available (see Figure 4.6): in the *Standard mode*, defined in the CAN 2.0a specification, the identifier is

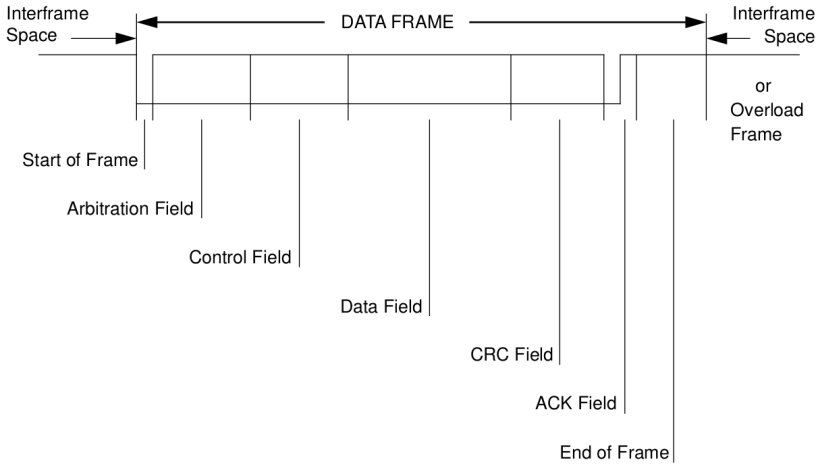


Figure 4.5.: The structure of a CAN Data frame.

composed by a 11bit, while in the *Extended format*, defined in CAN 2.0b specification, the Identifier is 29-bit long. To be compatible with some old CAN controllers, the 7 most dominant bits of the Arbitration field should not be all set to recessive, but, as all new controllers accept also those identifiers, the restriction is generally ignored on modern CAN systems. The identifier is followed by the *Remote Transmission Request* (RTR) bit, which identifies a Remote frame when set to dominant; the extended arbitration field also include two additional reserved bits. The overall length of the Arbitration field, thus, is 12 bit in Standard mode and 32 bit in Extended format.

The *Control field* is composed by 6 bits: the first one is used to distinguish Standard and Extended frames, the second one is reserved for future use, and the last 4 bits contain the *Data Length Code* (DLC), specifying the length of the payload.

Next there is the *Data field*, containing up to 8 bytes of payload, and the *CRC field*, containing a hardware generated 16-bit CRC sequence calculated on all the previous fields. The CRC is checked during the reception by all the receiver nodes, identifying correct or wrong messages on-the-fly.

The following field, the *ACK field*, is used to signal the reception of the message: the transmitter writes two recessive bits, while receivers which correctly get a correct message impose a dominant level as second bit. In this way, the transmitter node can check if the transmission was successfully completed; on the other way, if a single node did not

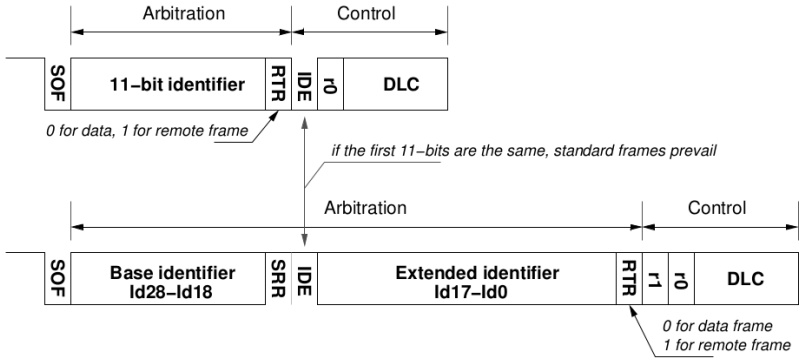


Figure 4.6.: The Standard mode and the Extended format Arbitration field.

correctly receive the message but the other nodes did, the transmitter is not signaled. As a consequence, protocols to guarantee the reception of data needs to be implemented by higher level layers, if needed.

Finally, the *End Of Frame* (EOF) field is inserted, which is composed by 7 recessive bits. This is the only condition where more than 5 consecutive bits with the same logic level are allowed to transit on the bus, violating the bit-stuffing rule. Frames are then separated by the inter-frame space, which consists of at least three recessive bits.

Arbitration

The CAN bus features a *Carrier-Sense Multiple-Access with Bitwise Arbitration* (CSMA/BA) *Media Access Control* (MAC), implemented at hardware level in the controllers. It is a priority-based arbitration protocol, which relies on the dominant/recessive bus levels and on the ability of CAN controllers to detect the bus status while transmitting. During the transmission of the arbitration field of a CAN frame, if the controller recognizes a dominant bit while it was trying to transmit a recessive one, it knows that the arbitration is lost and the node becomes a receiver. As a consequence, the higher priority Identifier is composed by all dominant bits (logic zeros), while the lower priority one is composed by all recessive bits (logical ones).

The CAN arbitration mechanism is *non-preemptive*: the arbitration process begins with the Start Of Frame and lasts for the duration of the Arbitration field, but the transmission of a message cannot be interrupted by a higher priority transmission request after it get access to

the bus.

Message Filtering

Another relevant feature of CAN controllers is the capability to filter incoming messages by their identifier at hardware level. Nodes can specify one or more *filters*, and then actually receive only the messages they are interested in. Furthermore, *bit masks* can also be specified, to apply a filter to a subset of the message arbitration field, allowing the reception of group of messages which shares a part of the identifier. Filters are generally associated to a specific set of receive registers (commonly called *Receive Objects*), thus the node not only avoids the reception of unwanted messages, but it also knows where to look for the messages it subscribed, speeding up the processing of incoming data.

4.2.3. Application Protocols

The CAN standard does not give any specification about device addressing, flow control, data fragmentation over multiple messages and the interface to the user application. These tasks are carried out by dedicated protocols, which implement the upper 5 layers of the 7-layer OSI specification. Several protocols have been developed, with application fields going from in-vehicle communications to aviation and military devices control. Car manufacturers generally adopt proprietary protocols to handle data for vehicle control, relying on standards such as OBD-II and EOBD only for diagnostics. Aviation and military targeted protocols are focused mainly on reliability, safety, and security, supporting strict hard real-time constraints due to their critical application contexts, but without taking into account flexibility.

In recent years, the CAN bus also started to be exploited for industrial automation, and several protocols have been proposed, such as SDS [80], CAN Kingdom [54], DeviceNet [103], CANopen [31], and others. All these protocols are based on the ISO 11898 CAN specification, and focus on the same problems: assigning CAN identifiers to messages matching the respective requirements and priorities, and provide an interface to application developers.

Early protocols

Smart Distributed System (SDS) protocol development was originally started by Honeywell in 1988 [80] to define a event-driven CAN application layer for industrial control applications in a Microsoft Windows control environment. It is optimized to be embedded with a very low

footprint in cost-effective sensors and actuators, and it is mainly used for point to point communication between a master and remote input/output devices.

CAN Kingdom has been developed by Kvaser AB since 1990, and it is mainly focused on distributed machine systems. The network is configured by a master controller, called *The King*, while all other devices do not know anything about the complete system. The idea behind the protocol is to give maximum flexibility to node developers, which can design nodes independently from a specific system: each node provides the network with some services, but it does not know how they will be used. As a consequence, the stack running on the single nodes is much simpler with respect to other protocols such as DeviceNET and CANopen. On the other hand, the system designer, who have complete knowledge of the specific application, configure everything through the King node, and can get the maximum performance as the network is set up ad-hoc to best satisfy his needs.

DeviceNet

DeviceNet was developed by Allen-Bradley (now Rockwell Automation) as application layer for the CAN fieldbus, adapting features of ControlNet, another industrial protocol developed by Allen-Bradley, to make it work on low-cost CAN devices. To promote the adoption of DeviceNet worldwide, Rockwell started the Open DeviceNet Vendors Association (ODVA); the association then defined a more generic technology, the Common Industrial Protocol (CIP), which is now used by DeviceNet, ControlNet and EtherNet/IP, fostering a simpler integration of different industrial control devices. DeviceNet specifications are not limited to the higher OSI layers, but they define also the physical layer, which follows a trunkline-dropline topology where a main line, carrying both the power supply and the communication bus, is deployed in the plant, and the devices can be easily attached to it, when needed, by means of droplines.

DeviceNet realizes a connection-based network: communication is always between exactly two nodes of the network which previously established a connection. Three kind of messages can be exchanged by DeviceNet nodes: *Explicit* and *Implicit* (also called *I/O*) messages can be exchanged between the master and one of the slaves, while *Peer* messages can be exchanged between two slaves. Explicit messaging follows the request/response communication paradigm: the DeviceNet master issues a request to a particular slave, which replies on the same Explicit message connection. Several fields are used to define the content of the

messages, i.e., if it is a set or get request, which node attribute is involved, and the data. This communication model is commonly used to write and read values to or from a node, e.g., during the configuration of a device. Explicit messages can be fragmented to support payloads bigger than 8 bytes, with the receiver sending reception acknowledgements back to the sender in order to allow the next fragment to be transmitted. Implicit or I/O messaging is used to exchange real-time data between the master and the slaves, or vice versa. Both nodes must be aware of the content of the messages, and transmission can be started in three different ways: it can be polled by the master, it can be cyclic, periodically triggered by the transmitting node, or it can be triggered by a state change in the system. Peer messaging is generally only supported between nodes of the same vendor, as no definition of data exchanged on Peer connections is provided by DeviceNet specifications, and they are rarely used.

DeviceNet is very popular in the industrial automation field, and many manufacturers produce devices compliant to DeviceNet. One of the major limit of this protocol is the connection-based communication paradigm, which does not support data broadcasting to more than one receiver. Moreover, besides DeviceNet supports cyclic messages, it does not provide any time-triggered bus access arbitration mechanism, and also periodic messages compete to gain bus control exploiting the standard CAN CSMA arbitration, which leads to transmission jitter.

CANopen

CANopen has been introduced by the CAN in Automation (CiA) organization in 1994, as CAN-based higher-layer protocol for embedded control systems. The main goal of CANopen is to provide standardized communication objects for real-time data, configuration data, and network management data, relieving the developer from dealing with low-level details such as the bit timing. The specifications comprise an addressing scheme, a set of small communication protocols, and the application layer, as well as application, device, and interface profiles.

Each CANopen device is composed by three main logical parts: the *protocol stack*, handling messaging over the CAN network, the *application software*, implementing the functionalities of the device and interfacing with the hardware, and the *object dictionary*, which interfaces the application software and the communication protocol. The object dictionary standardizes how to describe every object belonging to the CANopen device, such as configuration values, input data (i.e., setpoints), and output data (i.e., measured values). Through the com-

munication protocol, every entry in the object dictionary can be accessed and, eventually, modified. The description of object dictionary entries, as well as the communication behavior, are included in the *Electronic Data Sheet* (EDS) of the device.

Communication in CANopen is handled by different small protocols, depending on which type of data need to be transmitted. In all protocols, messages are broadcasted to every node in the network, which can consume or discard them depending on their configuration. The Network Management (NMT) protocols are used to change the state of the devices and to monitor their status through *heartbeat* messages. The Synchronization Object (SYNC) protocol provides the synchronization signal to trigger synchronous tasks, such as the execution of some control action or the transmission of synchronous messages. To read and write object dictionary entries, the Service Data Object (SDO) protocol is used: a client which wants to read or write the dictionary of another devices, which is called server, requests a SDO transfer specifying the entry index and, in write operations, the data to be updated. Entries with data longer than the CAN payload are automatically fragmented by the protocol. SDO transactions are always started by the client node, which waits for a reply from the server node, and are generally used to update the configuration of a device or to access auxiliary variables. Real-time data is exchanged with the Process Data Object (PDO) protocol, which handles Transmit Objects (TPDO), coming from the device, and Receive Objects (RPDO), going to the device. Each PDO can be mapped to more than one dictionary entry, but no fragmentation is provided by this protocol, so the maximum payload for PDO objects is 8 bytes. The transmission of PDO messages can be synchronous or asynchronous: in the former case, data transfer follows the SYNC messages, both periodically or after a particular event is observed; in the latter case, the message is sent after an internal or external trigger.

CANopen has been adopted by several manufacturers in applications going from medical devices to machine control, from vehicle and maritime electronics to building automation. The main limit with CANopen is the little payload of Process Data Objects, which limits their usage to small, rapid, transfers, and the request/response communication paradigm used to transmit Service Data Objects, which introduces overhead and makes them suitable only for configuration purposes.

4.2.4. Mixing Time- and Event- Triggered Traffic on the CAN bus

As discussed in Section 4.1, robotic systems involve both event-triggered and time-triggered tasks, imposing the same requirements to the communication channel. Due to its hardware bus arbitration feature (see Section 4.2.2), the CAN bus is well suited for fixed-priority communication, where each message is assigned with a priority and high priority messages only rely on winning arbitration against other competing ones. This is the approach followed by most CAN bus protocols, as the ones presented in the previous section, which rely only on the CAN bus CSMA arbitration, reserving priority ranges to different message types. Management messages generally have the higher priorities, to guarantee control of the system state also in heavy bus load situations. Also time triggered messages (i.e., periodic implicit messages in DeviceNet or cyclic PDOs in CANopen), are mapped to high priorities, to allow them win the arbitration over sporadic data transfers. Priorities are manually assigned by the system designer, or automatically calculated exploiting a scheduling algorithm, like the rate-monotonic scheduler [83, 122] which reserves higher priorities to transmissions occurring with shorter periods.

When mixing event-triggered and time-triggered traffic, the simple priority-based approach shows several drawbacks, which are summarized in the following. First of all, transmission requests can always be delayed by a higher priority request, introducing variable latency and preventing precise timing. This, obviously, applies also to time triggered messages, which can be delayed if messages with higher priority (e.g., a periodic message with lower period in the case of rate-monotonic scheduling) are transmitted. Moreover, the CSMA arbitration process starts with every start of frame bit, and ongoing transmissions cannot be preempted; as a consequence, if a time triggered transmission occurs while the bus is not idle, it will be delayed at least for the time needed to complete the current transfer.

Another source of variable delay, on the CAN bus, is the bit stuffing algorithm applied to the content of CAN frames (see Section 4.2.2). Transmission time is not constant, and it depends on the single bits of the particular CAN frame to be transmitted; in other words, the duration of the ongoing transfer may vary up to 26 bit times (the maximum number of bits inserted by the bit stuffing algorithm). For this reason, also a high priority message, if enqueued for transmission while another packet is on the bus, not only has to wait some time before the bus becomes idle, but this amount of time is not fixed and it cannot be known a priori,

as it depends on message content. As a consequence, in hard real-time applications, the recommended bus utilization factor for priority-based communication protocols is often limited; for rate-monotonic schedulers, for example, it has been proved that the utilization bound is 25% [19] of the bus capacity.

To efficiently communicate with precise timing, without sacrificing the bandwidth, a key aspect is to guarantee temporal isolation, so that the instant when a message is transmitted does not depend on bus load and higher priority transmissions. Temporal isolation can be enforced at two different levels; first of all, temporal isolation between time-triggered and event-triggered traffic should be provided, to avoid sporadic transmissions from degrading time sensitive messages. In other words, time-triggered and event-triggered messages are handled in distinct time slices, so that their transmission can never overlap. If interference between different time-triggered messages needs to be prevented too, temporal isolation can be applied with finer granularity, with time slices reserved to every single message occurrence, implementing a pure time-division multiple-access (TDMA) channel access method.

In order to extend CAN bus applications to distributed control systems, where temporal determinism and low jitter are mandatory, some protocols to schedule time-triggered traffic on the CAN bus have been presented [17,43], all of them provide temporal isolation between event-triggered and time-triggered traffic, allowing for a much higher temporal accuracy.

TTCAN

The time-triggered CAN (*TTCAN*) protocol has been developed to allow the use of CAN bus in hard real-time applications achieving a much higher bus utilization factor with respect to the recommended values defined by common CSMA-based CAN protocols, while providing a superior quality of service with better real-time scheduling guarantees [55, 81, 139]. The TTCAN protcol has also been standardized by the ISO Technical Commitee (ISO 11898-4), providing the first internationally accepted time-triggered specifications for any automotive protocol.

TTCAN, as every time-triggered protocol, requires a global time base to operate, and all scheduling actions are referred to that time base. To this extent, a master node periodically broadcasts a reference message, used by all nodes to adjust their local timer and to synchronize to the rest of the network. Two consecutive reference messages delimit the *basic cycle* (BC), which is composed of a fixed sequence of time slots, or

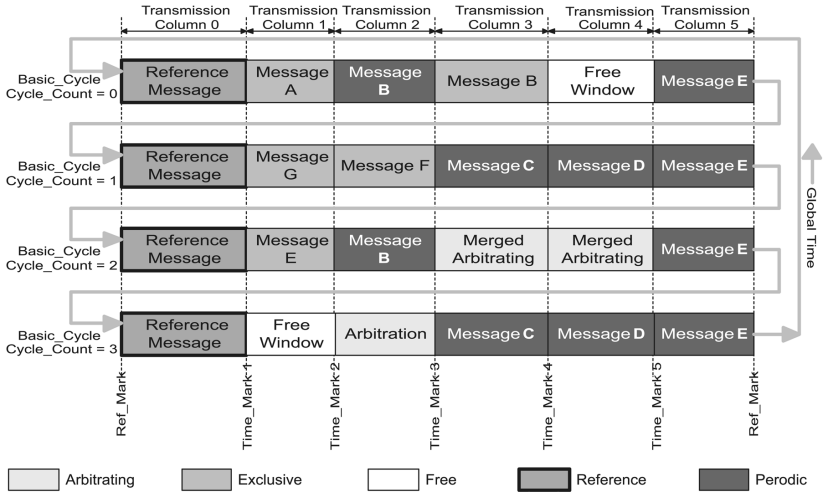


Figure 4.7.: The TTCAN system matrix.

time windows, as in common TDMA protocols. The complete schedule, called *matrix cycle* (MC) or system matrix, which is composed by several basic cycles (up to 64), repeats cyclically.

Each time window can be of one out of three different types: *exclusive time windows* are assigned to a single specific message, preventing other messages to access the bus; *arbitrating time windows* are assigned to more than one message, and competing nodes resolve transmission conflicts exploiting the native CAN arbitration mechanism; *free time windows* are not assigned to messages, providing room for future expansion of the network. In the schedule reported in Figure 4.7, for example, the matrix cycle is composed of 4 basic cycles, each containing 6 time slots. Windows are organized, within the matrix cycle, in columns, meaning that time slots in the same column have fixed length within all basic cycles (i.e., message A, G, and E in column 1 of Figure 4.7). Not to corrupt the time-triggered schedule, the CAN automatic re-transmission of messages is allowed only to reference messages, while for other messages an higher layer is required to handle transmission errors.

TTCAN allowed the adoption of CAN in hard real-time applications such as drive-by-wire in the automotive field, overcoming the limits of the native CSMA bus access arbitration mechanism when time-triggered messaging is needed. Many integrated circuit manufacturers have designed CAN controllers with hardware support for TTCAN, but, to the best of our knowledge, there are no open source implementations of the protocol available.

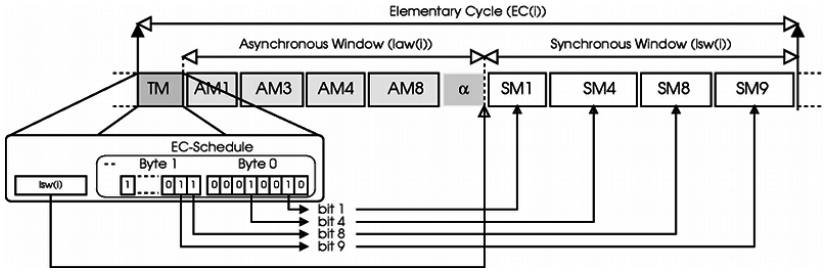


Figure 4.8.: FTT-CAN windowing approach.

The main drawbacks of TTCAN, which prevented us from adopting it as communication protocol in R2P, concerns the lack of flexibility, which is needed by the modular development approach of the framework. First of all, the system matrix is static and must be entirely known by all nodes at boot time, consuming memory and requiring to reprogram all nodes if a new message needs to be added. Moreover, the column-based organization of basic cycles, imposing the length of time windows, reduces flexibility even more and limits the maximum achievable throughput.

FTT-CAN

A different approach to allow time-triggered communication over the CAN bus has been proposed with FTT-CAN (Flexible Time-Triggered CAN) [17,109], which aims at mixing time-triggered and event-triggered messaging without losing flexibility. FTT-CAN exploits the native CAN bus CSMA arbitration, but provides temporal isolation between time-triggered and event-triggered traffic. Dynamic scheduling is also taken into account by FTT-CAN, allowing online admission control and centralized scheduling on the master node, which can be useful when working with dynamic network configurations.

As in TTCAN, also FTT-CAN needs a master node transmitting periodic trigger messages to align the local clocks on all nodes. The interval delimited by two trigger messages is called *elementary cycle* (EC) and defines a communication round. Within each round the protocol defines two consecutive, distinct, communication phases: the *asynchronous window* first, followed by the *syncronous window*. During the asynchronous window, event-triggered messages compete for the bus as soon as they request a transmission, using the CAN bus CSMA arbitration. The synchronous window is reserved for time-triggered messages, and its duration varies depending on the traffic scheduled within that EC. Within the synchronous window, time-triggered messages still use CSMA ar-

bitration, meaning that there is no predefined delivery order, but the synchronous window length guarantees that all scheduled messages will be delivered during the current EC. FTT-CAN is designed to allow different priority policies for CSMA arbitration (e.g., rate monotonic, deadline monotonic, earliest deadline first). To compute the length of the synchronous window, the master node runs a scheduler at the beginning of each EC and broadcasts the resulting duration in the payload of the trigger message. In this way, temporal isolation between the two communication phases is achieved.

FTT-CAN was one of the first CAN bus protocols to support time-triggered messaging, but later the research group responsible of its development mainly concentrated on the Ethernet versions of the protocol (FTT-Ethernet and FTT-SE).

A limit of FTT-CAN windowing approach is that the minimum period for time-triggered messages is the period of the trigger message, limiting the maximum frequency of periodic traffic. Reducing the period of trigger messages allows to transmit messages at higher frequencies, but also leads to higher protocol overhead and, thus, lower throughput. Moreover, using CSMA arbitration within the synchronous window, the jitter of time-triggered messages is bounded only by the duration of the synchronous windows they are scheduled in, which can be long and may vary at each communication cycle. The windowing mechanism increases the latency of event-triggered transmissions too, as grouping together the periodic messages means that all events occurred during a synchronous windows will be delayed to the next communication cycle.

4.3. RTCAN: a CAN bus protocol for robotic applications

From the analysis of TTCAN and FTT-CAN, we can conclude that TTCAN achieves better temporal determinism, handling time-triggered data with a pure TDMA approach, while FTT-CAN is much more flexible, with no shared and static schedule. On the other hand, TTCAN requires a rigid and static schedule which must be known by all nodes, sacrificing flexibility, while FTT-CAN is affected by some jitter and latency due to the windowing system.

To better match the requirement of distributed robotic systems, we developed RTCAN, a real-time communication protocol for the CAN bus which provides limited jitter for control loops, low latency to quickly react to events, and flexibility to easily add networked nodes to an existing system.

4.3.1. Goals

RTCAN goals were defined starting from the requirements of common robotic systems, which have been highlighted previously, and by analyzing existing protocols for the CAN bus. An outline of the main goals of the proposed protocol follows.

- Support both event-triggered and time-triggered traffic.
- Provide temporal determinism when delivering messages involved in distributed control loops, to allow for a precise robot control.
- Support high frequency messaging to enable effective closure of inner control loops, e.g., to implement current/torque control, without adding too much overhead.
- Guarantee low latency to quickly response to event when exchanging sporadic, event-triggered, messages.
- Allow flexible scheduling, to enable the connection of additional communication sources to the network without needing to reprogram every node.
- Avoid global calendars shared by all nodes, to support flexibility and to reduce memory footprint.

4.3.2. Message types

RTCAN mixes different approaches to communication handling, trying to effectively match the communication requirements of robotic systems. In RTCAN we define three distinct types of messages, each focused on the specific characteristics of its application area:

- Hard real-time messages (*HRT*) are periodic messages, e.g., from distributed control loops; they are deterministic, and their deadlines are absolute in time and should never be missed.
- Soft real-time messages (*SRT*) are triggered by events, e.g. new sensor readings; they are not periodic neither deterministic, but they need to be transmitted with the lowest possible latency. Deadlines are relative and if missed the system can still operate.
- Non real-time messages (*NRT*) do not expire in time, e.g., logging messages; they can be delivered without any latency constraints, exploiting free resources when available.

0 1 2 3 4 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28	
000000	HRT Message ID	Fragment #
Laxity	SRT Message ID	Fragment #
111111	NRT Message ID	Fragment #

Figure 4.9.: RTCAN priority and message ID encoded into the CAN bus 29-bit Extended ID

HRT transmission is time-triggered, in a pure TDMA approach, and bus access is reserved in a calendar for each message occurrence. Access to the bus is then exclusive, and no transmission conflicts need to be managed, preventing transmission delays and reducing the jitter. SRT messages are transmitted only when the bus is not reserved for HRT transmissions, so they do not interfere with high priority messages thanks to temporal isolation. They compete for bus access using dynamic priorities, so that messages gain chances to be transmitted while approaching the respective deadlines. NRT messages are handled as SRT ones, but they are assigned with a fixed priority, which is always lower than any other messages. RTCAN does not need a global schedule shared by all nodes, improving flexibility and reducing memory requirements, as detailed in next sections. As RTCAN is focused on real-time communication and not on fault tolerance, it does not handle transmission or receive errors (e.g., bus failures, overruns, or checksum mismatches): messages are just signed as corrupt on transmission failures, and error handling, eventually, has to be implemented by higher communication layers.

Figure 4.9 shows how RTCAN uses the CAN Extended ID field (i.e., the 29 bit version) for the different message types. HRT messages are identified by the first six bits set to dominant level, while, in the case of SRT messages, these bits are used to encode the laxity, which is used for bus arbitration as will be explained in Section 4.3.5; NRT messages have all recessive bits at the beginning of the ID, so they will always loose arbitration against SRT ones. The next 16 bits are used as message identifier, which is assigned by the application; they must be unique, as required by the CAN bus to perform bus access arbitration. Last 7 bits are used as fragment counter, to handle messages with payload bigger than the 8 byte CAN frame payload. Indeed, as it is common to need, even in control loops, transmission of variables which would not fit in the standard payload, RTCAN features a simple fragmentation protocol, and messages are fragmented at the origin and assembled back

on reception, in a transparent way for the application. As the CAN bus guarantees that packets are delivered in the same order they are transmitted, this is straightforward.

4.3.3. The communication cycle

To achieve fine-grained temporal isolation, and, as a consequence, high temporal determinism, RTCAN follows a TDMA approach: time is divided in time slots, organized in communication cycles, and each transmission must be completed inside the defined slot to avoid interferences with other transfers. The first requirement for TDMA transmission scheduling is that all nodes connected to the bus share a global time base. To this extent, each RTCAN network has a *master node*, which imposes the time to all other participants. The master node is statically defined at design time, or can be arbitrated during the bootstrap of the network by a simple protocol: each node is assigned by the system designer with a *master priority*, depending from its time accuracy or expected processor load. Then, during the bootstrap phase, all nodes broadcast their priority, together with a numeric identifier, which is unique within the network, and the higher value determines the master.

The master node sends periodic messages, named *sync messages*, which are used as reference to align the local clocks on all other nodes. Although this may seem trivial, and it is the approach followed by both TTCAN and FTT-CAN, we found out while developing RTCAN that a precise timestamping of CAN messages can be difficult. In the original RTCAN design, we aimed at using the SOF (see Section 4.2.2) timestamp as reference, which can be captured by CAN controllers at hardware level using the CAN bus bit time as time base. Unfortunately, we found out that CAN controllers, even if they provide the SOF timestamping feature, do not always give access to the hardware counter used to generate bit timing and, thus, to capture timestamps. As a consequence, the value can be used to precisely measure periods, but not to determine a global clock.

The impossibility of observing the counter which defines the time base also prevents the implementation of TTCAN *level 2* [67], which requires a global time reference. This problem affects several microcontrollers, including the one used by R2P modules, so we decided to timestamp sync messages within the reception interrupt handler. To provide a better estimate, the delay between the transmission of a message and the corresponding reception interrupt request is accurately measured at boot time, by issuing transmissions with the CAN controller configured in silent loop back mode (i.e., messages are not sent over the bus and

the node receives what it transmits). With a high priority assigned to CAN reception interrupts, and thanks to the fixed time interrupt handler invocation featured by ARM microcontrollers [138], the estimate is very accurate, allowing for time-triggered operations even without access to the CAN bus bit timing counter. It should be noticed that this is of primary importance for TDMA approaches, as the inaccuracy of local times requires dead times between transmissions to prevent overlapping, reducing the achievable throughput. Moreover, to prevent corruption of the TDMA schedule, the automatic retransmission feature of the CAN bus needs to be disabled.

Each sync message starts a new *communication cycle*, which is in turn divided in several *time slots*, which can be assigned to HRT transmission or left free for bus access arbitration of SRT and NRT messages. Time slots are triggered by an hardware timer running on each node, and issuing interrupts to send scheduled messages. The duration of a cycle, and the period of sync messages as a consequence, has to be chosen in order to prevent that the drift of local clocks lead to wrong slot delimitation. The error accumulated within a single communication cycle determines the needed inter-slot dead time and, as a consequence, the maximum bandwidth exploitation. The length of each time slot should be at least as long as an empty CAN 2.0B frame, and no longer than a full one (from 64 to 128 bit-times, plus the overhead of bit stuffing imposed by the CAN bus). The number of slots per cycle determines the maximum throughput: smaller slots give higher bandwidth if many small messages are sent on the bus, but a bigger reservation mask, described in the following, is needed.

4.3.4. HRT message scheduling

HRT messages are handled following a pure TDMA approach: they are scheduled for transmission by a centralized scheduler, and each transmission occurrence has exclusive access to the bus. In fault-free conditions, then, collisions are prevented, and transmission delays as a consequence; this allows to achieve high temporal determinism with low jitter (which is, ideally, removed).

The centralized scheduler runs on the master node, which keeps track of HRT transmission requests by all nodes. For each new request, admission control starts by checking that the needed transmission period is not relatively prime to other scheduled transmission periods, to guarantee that messages will never overlap. Then, an initial phase displacement, which determines the time slot for the first transmission, is assigned by the scheduler to the HRT message, and it is checked that all the sched-

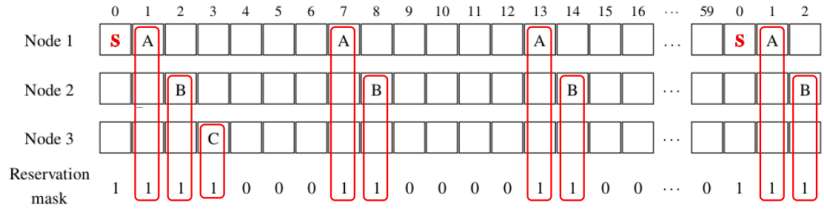


Figure 4.10.: RTCAN reservation mask: each slot reserved to HRT transmissions is identified by a bit set to 1 in the corresponding position.

uled messages fit within the available time slots. Messages sent at higher rate, i.e., with lower period, limit the number of available phase displacements, thus reducing the number of schedulable messages. This simple approach to HRT message scheduling restricts the range of concurrent scheduled periods, but removes transmission jitter (besides variable latency of the communication media, which is negligible on the CAN bus). Moreover, having control loops running at periods which are multiple of each other is common on robotic systems, then such a limitation is, generally, not a problem.

Given the initial phase displacement, since HRT messages are periodic, each node can compute the next time slot reserved to a HRT message simply adding its period. In other words, the scheduler and the admission control are centralized on the master node, but each node has a local reservation calendar with only its own scheduled messages. Using a centralized scheduler facilitates online dynamic scheduling, preserving flexibility, while the partial local calendars reduce memory requirements on the nodes. Only the master node needs to know all HRT message reservations to schedule new requests.

The local calendars, however, are not sufficient to know which time slots are free to transmit non-HRT messages, as nodes do not know reservations from other participants in the network. Only the master nodes, which tracks all HRT reservations, has full knowledge of the transmission schedule. At the beginning of each communication cycle, the master node updates the global calendar and fills a mask of bits, called *reservation mask*, with a number of bits equal to the number of time slots in the communication cycle. Bits set to 1 identify a reserved slot, while each bit set to 0 means that the corresponding slot is free for arbitration of SRT/NRT messages. The reservation mask is sent as payload of each sync message, and received by all nodes. Figure 4.10 reports an example of RTCAN communication cycle, and the corresponding reservation

mask. Nodes know only the time slots assigned to their HRT messages (e.g., message A for node 1), but they can distinguish free slots from reserved ones by simply checking for the corresponding bit in the reservation mask. As soon as a new time slot is triggered by the internal timer, nodes check for reservation of the current slot. If the slot has been marked as reserved, and the local calendar has a corresponding reservation, the node sends the HRT message associated to that slot. Otherwise, the node remains silent as it knows that other nodes are going to transmit an HRT message. If the slot has been marked as free for arbitration, each node checks its queued SRT/NRT messages and, if any, it tries to win access to the bus using the CAN CSMA arbitration process.

4.3.5. SRT and NRT message arbitration

Event-triggered messages cannot be scheduled a-priori, and multiple transmission requests can be issued at the same time. To handle concurrent requests, aiming at favor the most important ones, a priority has to be assigned to each competing message. Priorities can be assigned to data sources and mapped in the CAN ID field, to exploit the hardware CSMA arbitration provided by CAN controllers which automatically gives bus access to the message with higher priority. This is the approach followed by most CAN protocols, but priorities have to be assigned at design time, which is not suitable for a dynamic system as a network of R2P modules: the architecture of a robot developed with R2P can vary, by adding both modules or software nodes, and we do not want to define priorities for all messages at each change in configuration.

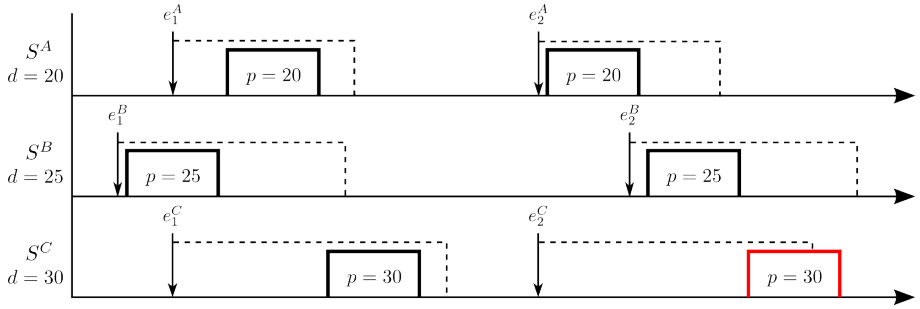
Events, in general, need to be handled within a certain amount of time, for two main reasons: because the system has to quickly react to the event (i.e., to avoid hitting an obstacle), or because the data has to be considered valid only for a certain period (i.e., the measured value changes rapidly). Without a-priory knowledge of the complete system, we can still define how quickly the event has to be managed, or how long a transmission can be considered useful, and we can assign a deadline to messages. Fixed priorities can be assigned with the same approach of rate-monotonic schedulers [83], but considering messages deadlines instead of their periods, as it is done by deadline-monotonic (DM) schedulers [22, 23]: messages with shorter deadlines have higher transmission priority. This scheduling policy is simple to implement, as priorities are directly mapped to deadlines, but in some cases the obtained schedule is not optimal and deadlines which could be met are actually missed. When the bus is loaded, the highest priority transmis-

sion will always meet its deadline, while lower priority messages often miss deadlines. Moreover, the maximum resource utilization factor to guarantee deadlines for this class of schedulers, in the general case, as been demonstrated to be $ln(2) = 0.69$ [83].

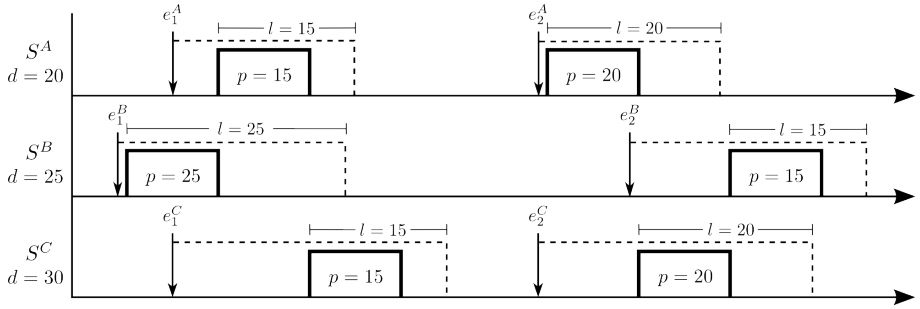
Different approaches are possible considering dynamic schedulers, where priorities are not static, but are computed during the execution of the system. One of the most common dynamic schedulers used to serve non periodic tasks is the Earliest Deadline First (EDF) [126]: at each scheduling request (i.e., the previous task has finished or yielded the resource), the scheduler looks for the task closer to its deadline and assigns the resource to it. EDF scheduling has been recently adopted in a variety of scenarios, including real-time systems, as it shows high flexibility and can reach a resource utilization factor equal to 1; although it may look much more complex to implement, it has been claimed that such believes are questionable under many aspects [38]. EDF based approaches have been proposed to schedule communication on the CAN bus [86,101] but, to the best of our knowledge, no implementation actually exists.

Going back to the problem of assigning priorities to event-triggered transmissions, let now compare how the DM and the EDF scheduling approaches apply. To be more precise, as a representation of absolute deadlines would not fit in the ID of a CAN frame (i.e., too many bits would be used to express an absolute deadline), instead of an EDF scheduler, we focus on a variant of it, which assigns priorities on the basis of the residual time before the deadline is missed, which is called *laxity*; this scheduling policy is called *Least Laxity First* (LLF) [107]. As transmission time is relative to the configured CAN bit rate, and also the laxity encoded in the CAN ID field as well, as explained in the following, we use here pure numbers. Figure 4.11 reports a simple scenario with 3 event sources: S^A emits events with a deadline $d = 20$, S^B is assigned with a deadline $d = 25$ and events from source S^C have deadline $d = 30$. Priorities are represented as they would be processed by the CAN bus: a lower number, thus CAN ID starting with dominant bits, is equal to a higher transmission priority, as explained in Section 4.2.2. For simplicity, each message transmission is assumed to take the same amount of time, which is equal to 10.

Applying a DM scheduler, the priority of each message is static and equal to its relative deadline, so messages from source S^A have priority $p = 20$, S^B have priority $p = 25$, and so on, as shown in Figure 4.11a. The first 3 transmission requests are correctly handled: e_1^B is transmitted as soon as the event triggers, since there are no other messages; then, two concurrent requests, triggered by event e_1^A and e_1^C, compete for bus access when the ongoing communication is concluded, and e_1^A is



(a) DM scheduling: transmission of event e_2^C misses its delivery deadline.



(b) LLF scheduling: event e_2^C is transmitted before event e_2^B, and both deadlines are met.

Figure 4.11.: Comparison of Deadline-monotonic (DM) and Least Laxity First (LLF) schedulers applied to transmission scheduling.

transmitted first as its deadline is shorter, thus its transmission priority is higher. Then, the message triggered by event e_1^C is transmitted, and all communication requests did meet their deadlines. The following requests are issued in a different order: events e_2^A and e_2^C are triggered at the same time, and e_2^A wins arbitration due to its shorter deadline. In the meanwhile, also event e_2^B triggers, and as soon as the first transmission ends, it gains access to the bus, delaying again the communication of event e_2^C which misses its deadline.

Consider now the application of the LLF scheduler, as represented in Figure 4.11b. Here the transmission priority is recomputed each time there is a transmission chance (i.e., when the ongoing transmission concludes and the bus becomes idle) by looking at the laxity. The first transmissions are handled in the same order as in the previous example: when the transmission of e_1^B ends, e_1^A wins the arbitration over e_1^C as, at that moment, its laxity is $l = 15$, while event e_1^C still has a laxity of $l = 25$ before its deadline. The second round, instead, is handled differ-

ently with respect to the DM scheduler: when transmission of event e_2^A concludes, event e_2^B has just triggered, and it still has a laxity of $l = 25$, while event e_2^C is nearer to its deadline, with a laxity of $l = 20$, and it gets transmitted first. Then, e_2^B gets transmitted too, and all deadlines are met.

RTCAN uses LLF scheduling for transmission of event-driven SRT messages, taking advantage of its better handling of sporadic tasks and its higher resource utilization. To exploit the CAN bus carrier-sense multiple-arbitration, the laxity of the message is encoded in the first bits of the CAN frame arbitration field, as shown in Figure 4.9. Laxity can be encoded linearly or using a logarithmic scale, resulting in a finer resolution for nearer deadlines, and a coarser resolution for remote deadlines [101]. NRT messages are handled as SRT ones, but their transmission priority is always lower (the laxity bits are all recessive) and it is never increased, thus they always lose the arbitration against SRT messages.

4.4. Benchmarks

RTCAN have been benchmarked to evaluate its communication performance in terms of transmission jitter and latency. In the following sections, the experimental setup is described and benchmark results are reported.

4.4.1. Experimental setup

RTCAN tests have been conducted a set of R2P hardware modules (see Chapter 7 for specifications). To evaluate the performance of the protocol, four nodes have been used: three of them communicate using RTCAN, while an extra module has been used to capture and timestamp with microsecond resolution all transmitted packets. During the tests the cycle period is set to 10 ms and the number of slots per cycle is 60. The CAN controllers are configured for 1 Mbit data rate. As a consequence, each time slot lasts 166 µs, which can contain a 8 byte CAN frame with worst case bit stuffing (158 bit times), plus the CAN intermission field (3 bit times) and 5 µs of separation between slots. A fixed slot length of 166 µs introduces a discretization of admissible frequencies, thus the actual frequency obtained could be slightly different from the required one (e.g., 1000 Hz is translated to a period of 6 time slots, which leads to an actual frequency of 996 Hz). Anyway, differences are, generally, small, and we are more interested in guaranteeing temporal determinism and low jitter than precise transmission frequencies.

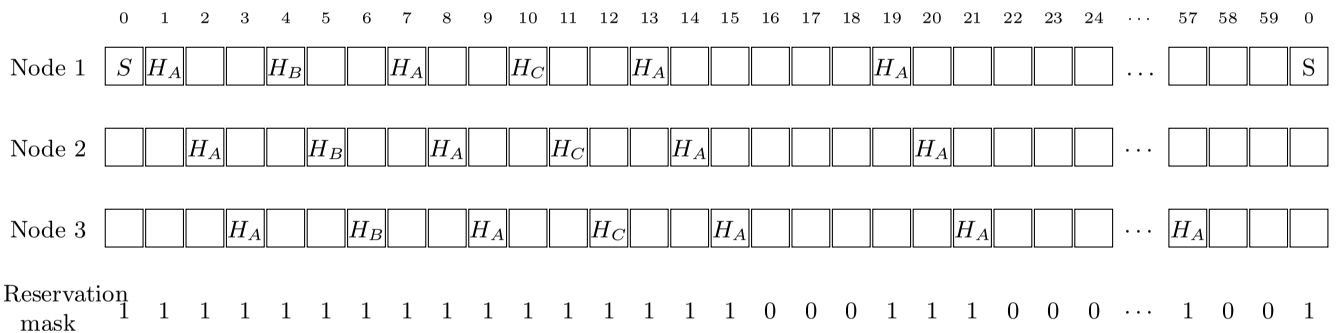


Figure 4.12.: The HRT calendar returned by the scheduler with messages used in the benchmarks and the respective reservation mask. H_A messages have 1 ms transmission period, H_B 5 ms and H_C 10 ms. S is the sync message from the master node.

ID	#	Period [ms]	Payload [bytes]	Throughput [kbps]
H_A	1	1	8	64
H_B	1	5	8	12.8
H_C	1	10	8	6.4

(a) HRT time-triggered messages scheduled on each node

ID	#	Deadline [ms]	Event Ratio [ms]	Payload [bytes]
S_A	5	10	10–20	8
S_B	5	10	20–50	16
S_C	5	50	50–100	20

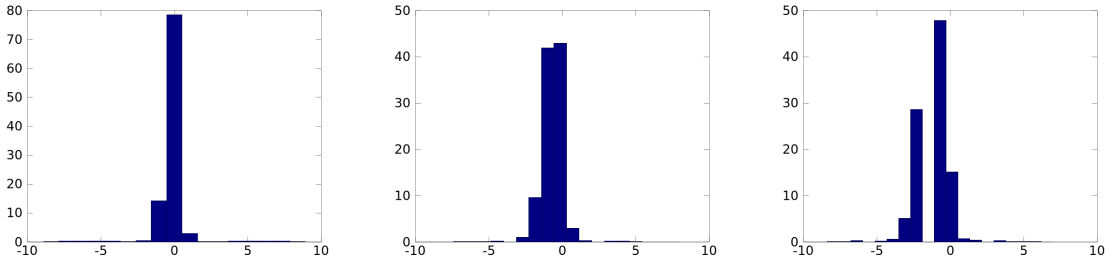
(b) SRT event-triggered message activity on each node

With the test configuration, the maximum throughput is 378 kbps, which means only a little overhead with respect to the maximum data rate of CAN 2.0B specifications, which is 398 kbps (again, in worst-case bit stuffing situation). Having 60 time slots per cycle, the maximum frequency for HRT messages is 3 KHz, due to the fact that slot 0 is reserved to the sync message. To benchmark RTCAN, each of the nodes schedules 3 HRT messages according to the frequencies reported in Table 4.13a; this gives a total time-triggered traffic of 249.6 kbps leaving some resources to SRT and NRT messages. Given the HRT requirements of the nodes, the centralized scheduler produces a reservation mask to reserve bus access to all time-triggered messages occurrences. The resulting communication cycle, which in this configuration is periodic, due to the requested frequencies, is presented in Figure 4.12.

To saturate the remaining bandwidth, several SRT messages are transmitted by each node, simulating event-triggered traffic. The trigger period varies, while the relative deadline is fixed for each message. The test configuration for SRT messages is shown in Table 4.13b; the resulting average traffic is 150.9 kbps. As a consequence, the total requested throughput is about 400 kbps, higher than the maximum admissible value of 378 kbps.

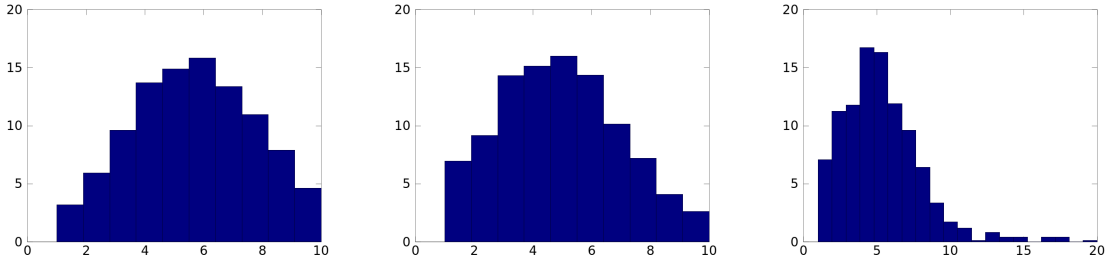
4.4.2. Results

During the benchmarks about 250 k HRT and 100 k SRT messages were transmitted. With respect to HRT messages, we evaluated the actual update period, its standard deviation and the distribution of transmission jitter. Results are reported in Table 4.2, while Figure 4.13 shows the transmission jitter. The results show that HRT messages never missed a deadline and the jitter is bounded to ± 3 µs. Moreover, due to temporal separation, HRT performances are not influenced by the actual bus load. About the measured periods, we believe that some unavoidable jitter is introduced by the CAN itself, which adapts the bit time to other nodes, in order to have consistent bit timing over the bus. This feature cannot be deactivated on our CAN controllers and it is not measurable without specific network analyzers.



(c) 1 ms period (d) 5 ms period (e) 10 ms period

Figure 4.13.: Distribution of HRT messages transmission jitter, expressed in µs.



(a) 10–20 ms period, 10 ms deadline (b) 20–50 ms period, 10 ms deadline (c) 50–100 ms period, 20 ms deadline

Figure 4.14.: Distribution of SRT messages transmission latency expressed in ms.

ID	Message count	Measured period [µs]	Standard devation [µs]
H_A	61136	995.9	0.86
H_B	12427	4979.4	0.93
H_C	6213	9958.8	1.05

Table 4.2.: HRT messages mean period and its standard deviation.

ID	Message count	Missed ratio	Mean latency [ms]
S_A	4121	1.60%	4.33
S_B	1594	5.52%	5.13
S_C	784	5.21%	14.94

Table 4.3.: SRT messages missed deadline ratio and mean transmission latency.

For SRT messages, we evaluated the missed deadline ratio and the delivery latency. Missed deadline ratio and mean latency are reported in Table 4.3, while latency distribution is shown in Figure 4.14. The results show that S_C messages, which have longer deadlines, thus a lower initial transmission priority, rarely win arbitration when the laxity is still high due to the EDF scheduling, and higher priority messages, i.e., with shorter deadlines, have less delivery latency. It can be noticed also how S_B messages, which have the same deadline of S_A messages, have a higher miss ratio, due to their fragmentation: two frames must win arbitration for a successful delivery.

The test shows that RTCAN can handle high frequency HRT messages with low jitter and a very little overhead, given by the sync message, which occupies 1/60 of the bandwidth. As a comparison, consider that to transmit a 1 Khz message in FTT-CAN, a 1 Khz trigger message is needed, wasting 1/6 of the available bandwidth. The jitter is lower too, as in FTT-CAN it is only bounded by the length of the synchronous window. Compared to TTCAN, the centralized scheduler used in RT-CAN, with only local calendars on the nodes, gives the same temporal determinism of pure TDMA protocols, but without affecting flexibility as using a static system matrix.

Chapter 5

Middleware

RTCAN, presented in Section 4.3, provides low level communications with real-time capabilities between R2P modules, but higher level interfaces are needed to support developers in writing the distributed control software and to foster the development of modular, reusable, code. To this extent, R2P includes a lightweight publish/subscribe middleware, which is presented in this Chapter. In Section 5.1 we focus on the problem of decoupling data flows, a prerequisite for flexible, modular, architectures; in Section 5.2 a quick overview of common middleware for robotics is given, highlighting the lack of available solutions targeted at resource constrained devices. Section 5.3 introduces the middleware of R2P, its architecture, and some implementation details, while in Section 5.4 communication benchmarks are reported.

5.1. Decoupling data flows

A generic data flow is a transfer of information from a sender, the data *producer*, to a receiver, the data *consumer*, through a communication channel. Information can be carried between several configurations of participants, i.e., one-to-one (one producer and one consumer), one-to-many (one producer and several consumers), and many-to-many (many producers and many consumers). Different time relationship between source and destination are also possible: the consumer may wait for

new data within its execution, in a synchronous way, or may be asynchronously signaled by the producer whenever new data has been transmitted, i.e., through a callback; finally, messages can be queued at both the sender and the receiver sides, to be consumed later.

Distributed environments may change in time, with devices being added and removed. Components may be interested in data in different ways (e.g., to log a sensor measure or to close a control loop on that measure), and generally their behavior is not fully deterministic. For these reasons, decoupling the production and consumption of information is recommended, as it increases scalability and removes explicit constraints between the interacting components. According to [49], the decoupling that the communication model provides between producers and consumers can be described by three factors, which are summarized in the following.

- *Space decoupling* defines if the interacting parties need to know each other. In space decoupled data flows, producers can send data through the communication channel, and consumers get data indirectly through it. Producers do not need to hold references of the consumers, or to know how many of them are taking part into the interaction, and can send data even if there are no consumers at all. In the same way, consumers do not hold references of the producers, do not know how many producers are active and which one produced the data they are receiving.

- *Time decoupling* defines if the data exchange requires both the producer and the consumer to be active at the same time. With time decoupling, producers can send data even if the consumer is disconnected at that moment, and consumers may receive data generated by a sender which is not active any more.

- *Synchronization decoupling* specifies if consumer and producer activities are suspended while sending and receiving data. If synchronization decoupling is provided, a producer is not blocked while sending data, even if receivers are slowly consuming information. On the other side, consumers are asynchronously notified, i.e., through a callback, of the availability of new data while performing some concurrent activity.

A rapid review of popular communication paradigms follows, highlighting the decoupling level they provide and how they would apply to the modular architecture for robotic applications we are proposing.

5.1.1. Message passing

Message passing is one of the very first techniques developed to allow distributed computation. In its simplest implementation, the data source fills a message and it is responsible for delivering it to each of its recipients, on a given communication channel. The message passing model is rarely used directly to develop distributed systems, since it exposes many issues, such as recipient addressing and flow control, to the application. On the top of this simple paradigm, however, many complex interaction schemes have been built in the years, implementing much more comprehensive features. Modern inter-process communication systems like D-Bus [87] and, in general, message-oriented middlewares, are based on the concepts from message passing, and used in many popular software projects as the Gnome window manager [132].

Despite being widely used in software engineering, the message passing model does not satisfy the decoupling requirements needed by distributed systems. Strict time coupling is imposed, as the consumer is required to be ready to handle incoming data when the producer sends a message, or the transfer will fail. Without message queuing techniques, the reception of messages is synchronous, which gives synchronization coupling on the consumer side. Moreover, as the data source must hold references to receivers, which are explicitly specified, producers and consumers are also space coupled.

5.1.2. Request/response

In the request/response model, information exchange occurs by issuing requests and waiting for corresponding responses; examples of this kind are client/server transactions and remote calls. Client/server interactions are asymmetric requests commonly used to implement master/slave type of queries. The client acts as a master, making a request and waiting for the response from the server, which is a normally idle slave waiting for requests from clients. As soon as the request is received, the server elaborates the reply and sends it back to the client. This kind of interaction is very popular and is often used to implement complex software projects, with one server, or more, dispatching requests coming from other components of the application. The use of client/server transactions is also very common when many distinct peers need to access the same resource, e.g., a file system or a communication channel, with a server directly managing the resource to satisfy all the requests.

Client/server interactions are not generally suitable for distributed real-time interactions due to some drawbacks of this technique. First

of all, each data transfer needs two transactions, i.e., the request and the corresponding responses, and occurs between exactly two peers, the client and the server. Moreover, when a client is waiting for a response by the server, it is blocked and cannot execute concurrent operations, leading to strict synchronization coupling. Nevertheless, the server must be running and ready to handle incoming requests, imposing time coupling. As the client has to know the server address to issue a request, space coupling is also present.

The request/response model has also been used to implement *remote procedure calls* (RPC) [29, 128] and, in object oriented languages, *remote method invocations* (RMI). These techniques allow an application to call a function which is executed on a remote system, receiving the corresponding return value. The RPC/RMI technique gained high popularity within distributed computing, since, from the point of view of the application, remote and local interactions work in the same way. Such an easy to use solution presents many advantages, enabling any developer to quickly write distributed applications, but also presents some drawbacks, since issues which are not locally detectable (e.g., a communication failure) have to be handled explicitly by the application. *Distributed object middlewares* (DOM) have been developed on the top of the RPC/RMI model, e.g., DCOM [36], CORBA [99, 133] and ICE [69]. These technologies have also been adopted by several of the first communication middlewares for robotics, e.g., Orocos [37], Orca [93], RT-Middleware [20], and Miro [131].

RPC/RMI techniques suffer from the same problems of the client/server interactions: each data exchange is composed by two transactions, adding overhead, and there are time and space couplings as caller entities must hold remote references to each of their callees, which synchronously handle requests. For the mentioned reason, the request/response model, and its derivatives, is not the best choice for modern robotics applications where most of the data flow is not peer-to-peer but one-to-many or even many-to-many, and decoupling is needed due to the frequent changes in the network of data producers and consumers.

5.1.3. Shared memory

The concept of shared memory is a common programming technique, which allows multiple programs simultaneously access the same memory area to exchange information avoiding redundant copies, in the same way they were accessing their own local memory. This technique has been extended to distributed computing systems, i.e., with processes running on different machines which cannot access the same physical memory,

with the introduction of the *distributed shared memory* paradigm [18,85]. When an application accesses a shared memory address, the operating system takes care of mapping the virtual address to the actual, physical, memory area, in a way completely transparent to the application. Communication occurs by simply writing into and reading from locations of the shared memory; changes are immediately available to all the peers, which must read the updated values promptly, before new data overwrites it.

The shared memory model offers time and space decoupling, because data producers and consumers do not know each other, they simply access the same memory area to exchange information, remaining anonymous. Producers write data into memory asynchronously, while consumers read new values from the shared space in a synchronous way; thus, synchronization decoupling is provided only at producer side.

The shared memory paradigm is very popular and widely adopted, allowing applications to be designed as they were not distributed, thanks to the underlying address mapping system. Despite its ease of use, the simple shared memory paradigm, with consumers required to synchronously access the memory to not loose consistency, is not appropriate for applications needing to communicate events and react to data change like robotic systems.

5.1.4. Notifications

In order to achieve synchronization decoupling, a synchronous remote request can be split in two asynchronous transactions: a first one from the consumer to the producer to specify how any occurrence of new data can be signaled, and a second one back from the producer to the consumer with the actual data. In the notifications model, the consumer first sends a request to the producer with the details of the transaction and a reference to the callback function used to actually perform the transaction; then, the producer notifies the consumer returning its reply, as soon as new data is ready, through the callback. This paradigm, with consumers showing their interest on a particular data flow by directly registering with producers, corresponds to the well known observer design pattern [75], where an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. The mechanism can be extended allowing the producer to return multiple replies, by calling multiple callbacks to the consumer, implementing basic publish/subscribe transactions.

Although notifications are asynchronous, providing synchronization

decoupling, the strict relationship between producers and consumers introduced by the registration process still forces strong space and time coupling.

5.1.5. Publish/subscribe

The publish/subscribe model has been proposed to overcome the coupling problems shown by common communication paradigms. In this model, consumers express their interest in a data flow by a issuing a subscription to that specific flow, in order to receive any data update matching their interest. On the other hand, producers can publish information on a data flow, causing all subscribed consumers to be notified whenever new data have been published. The core component of this paradigm is the dispatcher, which acts as a mediator in charge of keeping trace of active subscriptions and efficiently handling the notification process.

Several flavors of publish/subscribe have been implemented, with differences about how data flows are identified and, thus, subscribed. The earliest publish/subscribe scheme was based on the concept of *topics*, or *subjects*, identified by keywords, to which participants publish and subscribe information. Topics can be seen as communication groups: subscribers of a topic are somehow members of a group, to which publishers broadcast heterogeneous data. Another variant is referred as content-based publish/subscribe, where subscribers specify the actual content of the data they are interested in through a subscription language. In this way, publisher-subscriber relationships are not defined through a static criterion (e.g., the topic name), but are evaluated at run time analyzing the content of each data transfer. A different addressing schema has been developed observing that data grouped in the same topic show similarities not only in terms of content, but also in terms of structure [50]. This led to the proposal of the type-based publish/subscribe model, which allows a closer integration between the middleware and the application code, enabling type safety through different components of the system: data flows are identified by the topic name, which also defines the type of published messages.

This communication paradigm introduces space decoupling between producers and consumers through the fact that publishers and subscribers do not know each other: publish/subscribe operations, and in particular the delivery of notification to all the subscribers, are mediated by the dispatcher, whose architecture can be either centralized or distributed. Time and synchronization decoupling are also provided, as the only synchronous invocations are issued between the publisher and

the dispatcher, and, in a second moment, between the dispatcher and the subscriber. This model offers significant advantages in situations where transferred data correspond essentially to time changing values of an otherwise continuous signal (e.g., sensed data, control signals, etc.), as a single subscription replaces a continuous stream of requests, allowing for data transfer with minimum delay since the exchange is one way and asynchronous. Being notification based, the publish/subscribe model is very beneficial when a system needs to monitor a great number of perhaps infrequent events, while in a client/server model or shared memory model, the monitoring process should continuously poll for possible changes. One-to-many and many-to-many communications are supported by the publish/subscribe model in a natural way, and its implementation can often take advantage of multicast and broadcast mechanisms at the network level to improve efficiency of event notification.

5.2. Publish/subscribe middlewares in robotics

The publish/subscribe communication model is very common in robotics, as it is normal to have several sources of information of the same type (i.e., multiple proximity sensors), and multiple consumers of the same data (i.e., path planners and low-level collision avoidance). Moreover, thanks to the complete decoupling provided by this paradigm, it supports the development of highly modular architectures. Indeed, several development frameworks for robotic applications provide support to publish/subscribe interaction.

5.2.1. ROS

Probably, the most noticeable publish/subscribe middleware for robotics is ROS, which provides the user with three different communication patterns to exchange data between peers: topics, services and actions.

Topics are named buses over which nodes exchange messages following the publish/subscribe paradigm; they are intended for unidirectional, continuous communication such as data streams from sensors. ROS topics are strongly typed: publishers define the data type while advertising a topic, specifying the ROS *message type* to refer to, and subscriber can only receive messages with a matching type.

Messages are simple data structures, comprising several typed fields; they are defined through *message files* and can include standard primitive types (e.g., integers, floating point numbers, boolean values, etc.) as well as previously specified ROS messages. The building system then

parses the message files and generates the code needed to handle each type of message, i.e., marshalling and unmarshalling functions.

Currently ROS supports two data transport to actually transfer data over a network of nodes: the *TCPROS* transport, which streams message data over persistent TCP/IP connections, and the *UDPROS* transport, which broadcasts messages as UDP packets.

Pairing between subscribers and corresponding publishers is managed by a centralized service, the *ROS Master*, acting as a name server keeping track of publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate each other, then communication is established on a transport and the nodes can start communicating.

ROS services are intended for request/response interactions, such as remote procedure calls, that are not well handled by the publish/subscribe paradigm. Transaction contents are described by *service description files*, composed of two ROS message type specifications: one for the request and the other for the corresponding reply. Services should only be used for remote procedure calls that terminate quickly, as they implement synchronous, blocking, calls.

Asynchronous requests can be issued through actions, the ROS provider of client-server interactions. Action specifications include three message definitions used to describe an action: the goal, sent by the client to describe the requested action, the feedback, used to tell a client about the incremental progress of a goal, and the result, sent by the server upon goal completion. ROS actions are used to create servers executing long-running goals that can be preempted, e.g., moving the robot base to a target location or applying a chain of filtering algorithms on sensor data to extract interesting features.

The ROS feature-rich communication model has been proved to be very effective to develop software for robotics, allowing users to easily integrate software components implementing complex applications with a distributed approach. Unfortunately, ROS was developed with desktop-class computers as target, and its implementation is not suitable for the resources of low-cost embedded processors such as the microcontrollers used in R2P modules. First of all, the ROS master rely on a XML-RPC based protocol to setup nodes and connections, which ensures maximum portability among different platforms but presents heavy requirements in terms of both memory and computation. Moreover, ROS C++ implementation depends on complex software libraries and advanced language features (such as exceptions and RTTI) which cannot be enabled on microcontroller targets due to their memory requirements. Currently supported transports rely on communication protocols needing dedicated

hardware and heavy software stacks (i.e., TCP and UDP). Finally, ROS does not support any real-time constraints, being developed to mainly design high level behaviors such as reasoning and planning. Messages can be queued for delivery at both producer and consumer sides, but no specifications about message priority or delivery deadline can be defined. Despite a ROS client-only implementation for low-cost microcontroller is feasible, and has been implemented to interface R2P modules to a ROS network (see Chapter 6), R2P needs a much more lightweight and real-time oriented middleware to enable communication among software nodes as well as hardware modules.

5.2.2. LCM

The *Lightweight Communication and Marshalling* (LCM) framework provides a publish/subscribe messaging system, offering very high performance, and enabling data exchange between processes with high throughput and low latency. As in ROS, topics are strongly typed: each topic carries only one type of message, with its data content specified through message descriptor files. To prevent type inconsistency, e.g., due to mismatching topics or wrong message descriptions, each LCM message header contains a 64 bit hash code of its type descriptor, and type checking is performed on every transmission.

LCM relies on UDP multicast messaging, so messages are sent once for the whole network, scaling well with the number of nodes and achieving low delivery latency. UDP does not provide control flow and, as a consequence, dropped packets are not resent, reducing the latency of successive packets, but requiring the application to deal with lost messages, if needed.

LCM is very simple and efficient, and has been used also to perform soft real-time tasks on real robots [35, 84, 129]. On the other side, as we observed with ROS, having desktop-class computers as target platforms, some LCM design choices make it not suitable for resource constrained devices. The transport is not actually abstracted, so the strict dependency from the UDP protocol requires an Ethernet-capable platform. Performing run-time type checking prevents several problems from occurring and sure enough has many advantages on powerful platforms, but it introduces an overhead we need to avoid on embedded targets. As an example, the 64 bit hash included into the LCM header would fill an entire CAN packet for each message sent, which is off curse a cost too high for a feature we do not strictly need. Moreover, LCM message dispatching is worked out by comparing topic name strings at each reception, adding overhead to the delivery process. Finally, no real-time

constrain can be specified in LCM, preventing its use to implement hard real-time distributed tasks.

5.2.3. Famouso

To the best of our knowledge, the only publish/subscribe middleware supporting embedded devices is FAMOUSO, which claims to be able to run on platforms ranging from 8-bit microcontroller to desktop-class computers. FAMOUSO is focused on seamless integration of heterogeneous devices, enabling them to exchange data over a broad variety of communication media like CAN, ZigBee, AWDS and UDP. FAMOUSO messaging system is strictly event-driven: communication is always asynchronous, avoiding control flow dependencies and enabling the independence of communication participants. Each FAMOUSO *event* is specified by three elements: a *subject*, optional *attributes*, and the *content*. Subjects represent topics, and are defined by the applications; each subject is then mapped to a network-dependent address, building a global address space spanning across all networks. This feature is exploited by gateways, to enable and manage communication between different networks. Optional attributes could be context attributes, which deliver additional information about the event, such as origin or timestamp. The event content is specified by the user and is application dependent. Data is actually carried by *event channels*, which are abstract channels for event dissemination. Event channels support the specification of transmission requirements like deadline, jitter, omission degree, and the reservation of the needed network resources to enforce delivery guarantees. Despite being a powerful framework for event-driven applications, FAMOUSO does not suit the requirements of a distributed architecture where also periodic, hard real-time, communication is needed, e.g., to implement distributed control loops.

5.3. A lightweight publish/subscribe middleware

To let R2P modules exchange information, and to foster the development of modular, reusable, embedded software, we have designed and developed a lightweight middleware with the goal of bringing to resource constrained devices the same programming techniques often exploited on desktop-class processors.

R2P middleware follows the type-based publish/subscribe paradigm, and it is focused on limited resource requirements, efficiency, thread-safety, and real-time communication. The architecture is decentralized, meaning that there is not a *master node*, but all participants have the

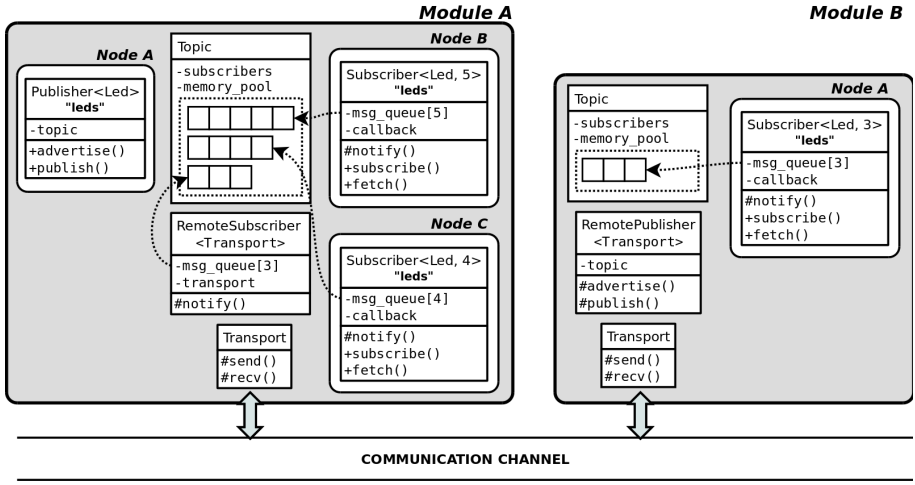


Figure 5.1.: R2P middleware main components.

same role in the network. Each R2P module runs an instance of the middleware, and network configuration is carried out through *lazy*, asynchronous, interactions (i.e., a subscriber to a topic which has no publishers yet, periodically polls for new publishing nodes).

5.3.1. Nomenclature

A quick recap about the nomenclature used in R2P follows; see Figure 5.1 for reference. Hardware boards are called *modules*, each hosting a microcontroller and connected to other modules by a communication channel. Modules run several *nodes*, the software components which perform computation. Data is exchanged among nodes by *messages*, which are simple data structures encoding some information. Nodes send messages by *advertising* a *publisher* about a specific *topic*, which identifies the data content. On the other way, nodes can declare a *subscriber* on the same topic, and the middleware associates it to the corresponding publisher.

5.3.2. Nodes

Nodes are software components performing simple tasks, with the goal of satisfying a specific functional requirement or a part of it. An application is then implemented as a network of nodes, running on the same module or distributed on several modules. Nodes provide modularity from the software point of view: jobs, e.g., complex activities to

achieve a goal, are split in multiple, smaller, tasks solving a well defined and bounded problem. Then, nodes can be reused in different projects showing common needs, speeding up the development and improving maintainability.

Each node is assigned a string, the *node name*, which is used to identify it, and which is unique within the module. The node name is then composed with *module name*, building a hierarchical path string in the form of */module name/node name*, to uniquely address nodes within the R2P network.

The life cycle of a node is composed by three phases: initialization, loop, and termination. In the initialization phase, nodes set up the elements they need to operate and wait for everything to be ready. Publishers and subscribers are declared and advertised to the middleware, which will associate them to the respective topics. If data coming from specific topics is required to correctly run the node, it can wait for the corresponding publisher to be instantiated. Nodes working with hardware also initialize the corresponding peripherals before going through the next phase.

When a node is operational, it enters a loop: here computation is performed, decisions are made and messages carrying the results are published to be consumed by other nodes. Incoming messages are processed through the execution of callbacks or, eventually, directly getting them from the subscriber, as shown in Section 5.3.5. Loop execution is managed by calling the *spin()* method, which suspends the execution, until new messages are available, and invokes callbacks to handle incoming data. Eventually, a timeout can be specified as argument of the *spin()* method, and the thread is awaked if no messages have been received within the timeout.

Nodes can be requested to exit the loop, and the termination phase begins. Depending on the specific application, a node may terminate directly or execute some instructions before exiting (e.g., a node controlling a motor may halt it or place it to a safe position to avoid unexpected situations).

A node is actually implemented as a thread of the underlaying real-time operating system, and thus it inherits thread features. Execution priorities can be assigned to nodes, which then will be scheduled for execution giving precedence to mandatory tasks. A motion control node, for example, can be prioritized with respect to a node acting as a logger, and its operation will not be preempted by nodes implementing less important tasks. Node execution can also be synchronized, if needed, relying on the synchronization primitives offered by the OS like signals, semaphores, mutexes, and timers, imposing an execution flow and guar-

```
class Timestamp : public r2p::Message {
  uint32_t sec;
  uint32_t nsec;
} R2P_PACKED;

class IMUAttitude : public r2p::Message {
  float roll;
  float pitch;
  float yaw;
} R2P_PACKED;
```

Listing 5.1: Message content is specified by C++ classes

anteeing system consistence. Using the global time base provided by RTCAN (see Section 4.3), threads can also be synchronized through different R2P modules, coordinating distributed operations.

5.3.3. Message type specification

R2P follows the type-based publish/subscribe paradigm: a topic does not only identify the subject of messages being exchanged, but also their data type. The strong typed definition of data flows improves the decoupling between producers and consumers, as referring to the same topic guarantees that the content of exchanged messages is consistent. Nodes can then be updated, or even replaced, with any other implementation which publishes and subscribes the same topics. With the assumption that R2P modules share the same architecture, the binary representation of message content is guaranteed to be consistent too.

The content of R2P messages is specified by defining standard C++ classes, composed by typed fields (see Listing 5.1), and such specifications are common to all nodes. Message definitions must inherit from the *r2p::Message* class, which provides an additional header used for memory management, as explained in Section 5.3.4. They are also packed, i.e., without added padding, to both save memory and avoid the need for type-dependent marshalling and unmarshalling functions.

An abstract message definition language could have been used to specify messages, as it is done in ROS and LCM, improving flexibility and type-safety. We did not adopt that solutions for two main reasons: first of all, R2P modules share the same microcontroller architecture, so machine-dependent problems such as binary representation of data and endianness are not a problem; moreover, marshalling and unmarshalling functions add execution overhead and require to copy the payload from a data structure to another, needing to allocate more memory. R2P

performs zero-copy message local delivery, and even the payload of messages coming from other nodes through a transport (i.e., a communication channel) are written directly into memory to reduce allocations and improve performance, as detailed in a while.

5.3.4. Topics

Topics represent the channel used by nodes to actually exchange messages about a particular subject. Each topic is identified by a string, the *topic_name*, and they are uniquely identified on the network as *module_name/node_name/topic*. Topics are instantiated when they are first referred by an advertisement (the creation of a publisher), or by a subscription (the creation of a subscriber). With distributed publishers and subscribers, an instance of the corresponding topic exists on all producer and consumer modules, and they exchange data through a communication channel, as explained in the next Section.

In R2P, topics are also in charge of managing the memory where message content is stored. The implementation of memory management mechanisms depends greatly upon operating system, architecture and available resources; on resource-constrained devices, such as the microcontrollers used in R2P modules, standard dynamic memory allocation functions (e.g., *malloc()* and *free()*), are often not supported and, generally, their use is not recommended. The main reason is that dynamic memory management is prone to fragmentation: after some allocation and deallocation requests, there will be sections of used and unused memory, and, if successive allocations require chunks bigger than the unused fragments, some memory is actually wasted. Many techniques have been developed to efficiently use the available memory [135], trying to minimize the wasted space and keeping track of unused areas. Functions provided by standard libraries, like the GNU C library (glibc), introduce execution and memory overhead, to achieve high utilization factors keeping track of unused memory areas, which are not compatible with embedded targets. Moreover, performance of most dynamic allocators is affected by non-constant execution time, due to the search for free memory chunks and to the optimization of unused areas, and this prevents their use in critical sections of real-time applications. Although alternative allocation algorithms have been proposed for real-time systems [94,106], most RTOS targeted at resource-constrained processors provide dynamic memory management primitives implementing very simple allocation mechanisms, like first-fit algorithms (the first free block large enough to satisfy the request is returned), affected by memory fragmentation and allocation time issues.

Topics provide the user with functions to allocate memory for messages to be published based on memory pools. Memory pools, also called fixed-size-blocks allocators, allow dynamic memory allocation, with constant execution time, preventing fragmentation and with little memory overhead. A number of memory blocks, all with the same size, are preallocated to constitute the memory pool; then, the application can allocate, access and free memory at run time as with standard allocators. The limit of managing only memory blocks with the same size is not a problem for our purposes, as all messages exchanged on a topic are of the same type and, thus, have the same memory footprint.

Within the creation of a new topic, an empty memory pool is initialized and assigned to it. Memory blocks to fill the pool are provided on subscription: each subscriber statically allocates memory for its message queue (see Section 5.3.5]), and yields it to the memory pool, which grows in size as a consequence. In other words, each local subscriber to the topic participates to form the memory pool by donating the memory it allocated. As a consequence, if several nodes subscribe to the topic, each one with its message buffer length, the overall size of the memory pool will be the sum of all buffers donated by subscribers. This is possible because chunks managed by the memory pool do not need to have consecutive locations, but can be sparse in memory, as each free chunk holds a pointer to the next available memory area. Publishers allocate memory for new messages from the memory pool and publish them; when all subscribers did consume the message, the memory can be released and put back into the memory pool for future allocations.

When a producer publishes a message on a topic, the corresponding topic handler cycles its subscribers and tests if they are ready to receive a message (i.e., their message queue is not full) and, for ready subscribers, the message is delivered into their queue. Then, the middleware notifies nodes of new messages to consume, and schedules them for execution, requesting the underlaying RTOS to awake them.

Message delivery and notification are important operations which need to be optimized, as they affect performance of the whole middleware. Firstly, this is the step in which data is actually transfered from producers to consumers, thus the maximum throughput mainly depends on how message delivery and notification are implemented. Moreover, these operations involve critical sections of code (e.g., to work with message queues and with the scheduler), requiring to be executed atomically by locking the system or by acquiring a mutex. As a consequence, they are source of jitter and other tasks can be delayed while messages are being delivered. In R2P, these operations have been carefully implemented, trying to reduce their execution time and overhead.

To minimize the notification time, the middleware implements a *zero-copy* message delivery mechanism: messages are filled by the publisher and then notified to all subscribers by passing a reference to them. Message queues, thus, do not contain messages with their content, but only pointers to the memory areas where the messages to be consumed are actually stored. On message publication the topic handler cycles each subscriber and checks if its message queue is full; if not, the pointer to the message is put into the queue, without needing to perform a copy of its content. For each successful delivery, a reference counter associated to the message is increased; then, when a subscriber finishes consuming the message, it is released and the corresponding reference counter is decremented. When the reference counter is equal to zero, it means that all consumers did fetch and release the message, and the corresponding memory can be freed. In fact, the delivery mechanism works together with the memory pools responsible of allocating memory for messages: allocation requests are issued by publishers, and the memory is freed and put back in the pool after all subscribers dereferenced the message. The zero-copy technique reduces delivery latency and jitter too, minimizing the contributions dependent on the number of ready subscribers, which is conditioned by the status of the system and, thus, it is not constant.

Notification of new enqueued messages is accomplished by performing two actions: first, the thread is marked as ready for execution, so that it gets scheduled by the operating system and executed again; then, a bit mask is used to identify which subscriptions have messages to be consumed, allowing to directly invoke callbacks without polling each subscriber. Callbacks are invoked within the execution context of the node, so the order they get called follows the priority of the corresponding node.

5.3.5. Publishers and subscribers

Publishers and subscribers are the mediators used by nodes to communicate. They are declared within nodes, defining the data they are going to produce and the data they need to operate. In other words, by declaring its publishers and its subscribers, a node specifies the *output* and *input* ports to connect with other nodes. The data type of messages must be specified as parameter of a C++ template, within the declaration of publishers and subscribers, as shown by the code excerpts reported in Listing 5.2 and 5.3. The data type is also used internally to correctly instantiate type-dependent structures, such as the memory pools used to store messages and to handle memory allocations as described in Section 5.3.4.

```
void ledpub_node(void *arg) {
  Node node("ledpub");
  Publisher<LedMsg> led_pub;
  bool toggle = 0;

  node.advertise(led_pub, "leds");

  while (r2p::ok()) {
    LedMsg *msgp;
    if (led_pub.alloc(msgp)) {
      msgp->led = led;
      msgp->value = toggle;
      toggle ^= 1;

      led_pub.publish(*msgp);
    }

    Node::sleepMs(500);
  }
}
```

Listing 5.2: Publisher node code sample

Depending on the application, it might be needed to queue messages when a subscriber is not ready to consume available data. To this extent, a subscriber declaration expects as a parameter, together with the message data type, the length of the queue to initialize. Each subscriber holds its own queue, and messages are delivered to subscribers with free slots in their queue, while messages are lost only by subscribers with a full queue. In this way, decoupling between subscribers is provided, and slow consumers are prevented from blocking the delivery of new messages to faster subscribers. Message queuing is realized without involving copies of messages, to avoid data replication and execution overhead, and messages are passed by reference. Details about message notification and delivery are provided in Section 5.3.4.

Advertisement and subscription

Nodes communicate to the network their intention to produce data about a particular subject by advertising a publisher on the corresponding topic. On the other side, subscriptions are issued by consumer nodes to associate subscribers to the topics they are interested in.

Advertisement and subscription actions are triggered by nodes, calling the advertise() or the subscribe() operations on a publisher or on a subscriber instance, respectively. The node is required only to specify

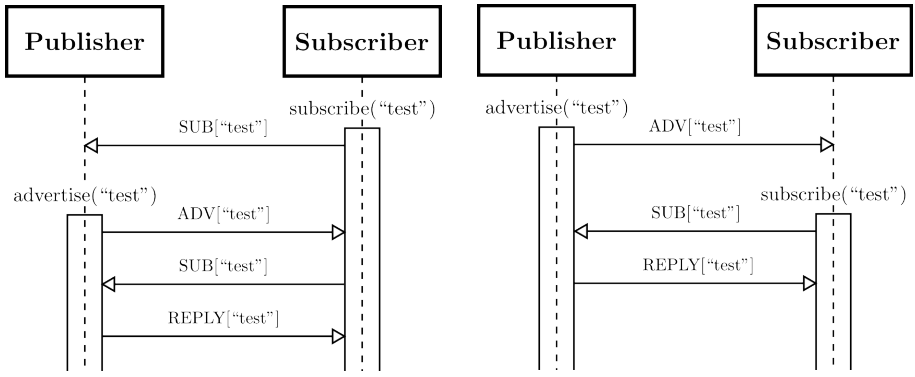
```
void callback(const LedMsg &msg) {
  setLed(msg.led, msg.value);
}

void ledsub_node(void * args) {
  Node node("ledsub");
    Subscriber<LedMsg, 5> sub(callback);

  node.subscribe(sub, "leds");

  while (r2p::ok()) {
    if (!node.spinMs(1000)) {
      /* Timeout! */
      handle_timeout();
    }
  }
}
```

Listing 5.3: Subscriber node code sample

the string representing the subject of the data flow (i.e., the *topic name*), to identify the topic. These operations request the middleware to find a local instance of the corresponding topic; if it is not found (i.e., it is the first reference to that topic), a new instance is created. Local advertisements and subscriptions are carried out by simple operations: as soon as the node advertises a publisher, the middleware associates it to an existing, or a new, topic instance. The same applies to subscribers, which can be declared after or before respective publishers: the first to be instantiated will cause the middleware to create the corresponding topic.

Advertisements and subscriptions are also broadcasted to all the registered transports, to find remote instances of the topic and correctly associate them. The protocol, which is sketched in Figure 5.2, is asynchronous and stateless. Advertise and subscribe messages are exchanged on the */r2p* topic, which is dedicated to network management and publication of debug messages. The */r2p* topic is subscribed by a special node, running on each module, responsible for managing node requests, creating new topic instances, dispatching debug messages and, eventually, in charge of handling the node loading process that will be described in Section 5.3.8.

Whenever a publisher is advertised, an advertise message is sent on every transport, containing the name of the topic; if a module connected to the network has already a local instance of the same topic (the case in Figure 5.2a), it replies with a subscription request. The request is



(a) The publisher is instantiated after the subscriber

(b) The publisher exists on subscription request

Figure 5.2.: Advertisement and subscription through a generic transport; note that notifications are broadcasted to all modules, if supported by the communication channel

received by the management node running on the publisher side, which looks for a local instance of the corresponding topic (i.e., being stateless, there is no knowledge about the previous advertisement); if it is found, a *remote subscriber*, specialized for the given transport, is instantiated on that topic. Finally, a reply is broadcasted containing again the name of the topic and, eventually, transport dependent parameters, like the RTCAN ID on which messages on that topic will be published, and the subscriber module creates a *remote publisher* on the corresponding local topic instance. The same process applies when subscribers are declared after the corresponding publishers: the initial advertisement message is ignored, as there were no local instances of the topic, but on subscribe operations the request is always broadcasted through transports, and if publishers exist on any other module, a reply is transmitted.

From the topic point of view, local and remote publishers, or subscribers, act at the same way: remote publishers receive messages through the transport they were created on, and publish them on the local topic instance, as local publishers do. In the same way, messages published by local nodes are notified to the remote subscriber and, thus, sent the corresponding transport.

Remote publishers and subscribers are specialized on a transport, and may exploit specific features like the ID filtering capability of CAN controllers to receive only messages on subscribed topics (see Section 4.2.2), thus reducing processor load. The arbitration of transport-dependent

parameters, such as the identifier associated to each topic, is left to the implementation of the transport itself.

Producing and consuming messages

To produce a message, the application first tries to allocate memory to hold the message content, by calling the *alloc* method on the publisher; the request is actually forwarded to the corresponding topic, which manages a memory pool, as already mentioned. Then, the message is broadcasted by the publisher through the *publish()* operation, informing the topic instance that new data needs to be delivered to subscribed nodes.

Messages can be consumed by nodes both in a synchronous and an asynchronous way. Synchronous reception is achieved by calling the *fetch()* operation on the subscriber, which returns the older message in the queue, if available, or puts the node thread in wait state; a timeout can be eventually specified to awake the node even if no message was received. It should be noted that only the reception is synchronous, as the message was already published and put in the queue; publishers are not blocked and their execution is not affected by how messages are consumed. To enable asynchronous reception, the node can specify a callback function within the declaration of the subscriber and wait for messages by calling the *spin()* method; then, for every message successfully delivered to the node (i.e., its queue was not full), the callback is invoked to handle incoming data. An example of asynchronous message handling is reported in Listing 5.3. Callbacks are invoked with the same priority of the node, so that the execution of slow callbacks is prevented from affecting the operations of other nodes.

5.3.6. Real-time support

R2P middleware is designed to run over a real-time operating system (RTOS) featuring synchronization mechanisms and multi-threading. Dependency from the operating system is abstracted by classes acting as interface between the software and the actual implementation of common RTOS primitives (e.g., threads, signals, semaphores, mutexes, etc.), for an easier porting of the middleware to different architectures, also independently from the R2P framework.

In the current implementation, the middleware supports ChibiOS/RT [46], a compact, fast, open source RTOS focused on embedded real-time applications. ChibiOS/RT provides both cooperative and round-robin scheduling, which helps developers with little or no experience in RT systems to write concurrent software nodes. It also features a complete

Hardware Abstraction Layer (HAL), easing access to microcontroller peripherals.

As nodes are threads, they are scheduled for execution by the RTOS. Priorities are assigned to nodes in order to prevent critical tasks from being blocked by lower priority ones. Developers can take advantage of the many features offered by the underlaying RTOS to synchronize local nodes, manage mutual access on hardware resources, and so on.

To support real-time interactions, the middleware defines 3 classes of publishers:

- Periodic real-time publishers, which are time-triggered and publish periodic messages, e.g., from distributed control loops. In this case, the user specifies the update rate and provides a callback to get the content of the message, which is then automatically published. If the memory pool is empty, thus the message cannot be published, the publishing node is signaled.

- Sporadic real-time publishers, which are event-triggered. These publishers have a deadline, which specifies a time limit by which messages must be inserted in the subscribers queue. If a deadline is missed, the publishing node is signaled.

- Non real-time publishers, which do not expire in time, like logging messages. If the message queue is full, the node is suspended, waiting for an available slot.

Local and remote subscribers do not differentiate between the type of publishers; anyway, as nodes can specify a timeout when waiting for messages, they can notice missed deadlines and handle the condition. Real-time interactions between distributed nodes, running on different modules, depend on the features provided by the underlaying transport.

5.3.7. Transports

Transport are responsible of delivering messages to distributed nodes, by using remote publishers and remote subscribers as previously explained. The preferred transport used to connect R2P modules is RTCAN, presented in Chapter 4; it provides time-triggered communication with low jitter for periodic real-time publishers, event-triggered messaging with deadline-based priorities to support sporadic real-time publishers, and low priority transmissions for non critical data transfers.

RTCAN allows to realize distributed real-time control loops, and reduces processing overhead thanks to the hardware filtering capabilities

of CAN, so that each node automatically discards messages it is not interested in, but cannot be implemented on an host computer without a real-time operating system due to its strict timing requirements. For this reason, R2P middleware is designed to support a variety of transports, which can be implemented by providing basic functions such as *send()* and *receive()*, eventually wrapped into a thread of the RTOS acting as dispatcher. Transports are also responsible of marshalling and unmarshalling data, meeting the characteristics of the specific communication channel, and must provide functions to build advertisement/subscription messages, as they may contain transport-dependent fields on which topics and messages are mapped.

At the time of writing, the middleware includes a debug transport over a serial connection, which can be used to inspect the status of a node or of the whole R2P network, as well as publishing and subscribing messages. It uses a human-readable format to exchange data (i.e., plain strings to identify topics and ASCII hexadecimal encoding of binary data), and a python script can be used to marshal and unmarshal messages. The debug transport can be used also on a TCP network stream, so that easy access to the R2P network is available also through Ethernet and WiFi connections, thanks to the R2P Gateway module (see Chapter 7).

5.3.8. Loading and configuring nodes

The firmware running on R2P modules can be compiled and deployed following two approaches: by flashing a full and static image, or by writing only a basic image and load nodes at run time. Static flash programming is the common approach with microcontrollers: the developer compiles the whole code (i.e., the operating system, the middleware, the transport, and the application-dependent nodes) on a host machine, and rewrites the flash memory every time an update is needed. Flashing the firmware with standard tools, however, is often inconvenient, especially when working on robots, as direct access to the devices is needed, as well as specialized hardware programmers (e.g., a JTAG adapter) and a complete development environment for the specific target.

To ease the deployment of low-level software, R2P features a *node loader*, i.e., a *Python* tool to dynamically load nodes on modules by transmitting the binary image via the middleware, without requiring special tools or direct access to the robot. The node loader tool exchanges information with the target module on the */r2p* management topic to correctly prepare the binary image. Program and data memory requirements of the node are communicated to the module (i.e., the sizes of the *.data*, *.bss*, and *.text* sections of the ELF image [102]), which

Component	RAM usage [bytes]
Node	36
Topic	56
Publisher	16
Subscriber	48
Remote Publisher	16
Remote Subscriber	28

Table 5.1.: RAM footprint of main middleware classes

replies with the addresses where the corresponding sections will be written into the flash memory. The tool relocates the code to the correct addresses and delivers the binary image to the module using the standard Intel HEX format [44]. The module can now launch the new node, allowing to test new functionalities without needing direct access to the board.

Nodes configuration, e.g., the name of the topics, or the update period or the parameters of a control algorithm, are also stored into a dedicated area of the flash memory, and they can be remotely read and updated. This allows for a much easier and faster tuning of the control software, and makes code reuse straightforward: in several scenarios, the firmware does not even need to be modified, and the only needed operation is to configure a few parameters.

Using these tools, new nodes can be deployed, configured, and run, over any transport supported by the middleware, allowing to test and tune functionalities without needing physical access to the robot. Starting from these tools, we plan to develop a graphical development environment to easily inspect the R2P network, write and deploy nodes, and configure them.

5.4. Benchmarks

We have run some benchmarks to evaluate R2P middleware footprint and its messaging performance. Tests have been run on R2P hardware modules, sporting STM32 ARM Cortex-M3 microcontrollers with $72Mhz$ clock, $20KB$ of RAM and $128KB$ of Flash memory.

Section	Used [bytes]	Available [kbytes]	Ratio [%]
.text	23524	128	18.3
.data	1324	20	6.6
.bss	7172	20	35.9

Table 5.2.: ELF sections size of a firmware with 2 publishers and 2 subscribers

Section	Used [bytes]	Available [kbytes]	Ratio [%]
.text	41988	128	32.8
.data	1328	20	6.6
.bss	5588	20	29.3

Table 5.3.: ELF sections size of the static firmware without symbols removal.

5.4.1. Memory footprint

The R2P middleware is written in a subset of C++, to take advantage of some object-oriented programming features without compromising performance on embedded targets. Some advanced features of the language have been disabled (e.g., RTTI and exceptions), and template metaprogramming techniques have been exploited for efficient execution on resource-constrained HW architectures. Table 5.1 reports the memory requirements, in terms of RAM, of the main middleware components; on the low-cost microcontrollers used in R2P modules, which feature 20 kbyte of RAM, tens of nodes, publishers, and subscribers can be instantiated.

Table 5.2 reports the requirements in terms of flash and RAM memory in case of static firmware. Data refer to an example node running the complete RTOS and HAL, the middleware, the RTCAN protocol and two nodes, each featuring a publisher and a subscriber, with the corresponding remote counterparts statically instantiated (i.e., to give a real usage example).

Dynamically loading nodes means that the module needs to be firstly flashed with a base firmware, which includes all functions that may be eventually needed by loaded nodes. This requires that no functionalities are excluded from the base firmware, and some compiler optimizations, such as the removal of unused symbols, have to be disabled. The resulting firmware is then bigger in terms of program memory, as shown in

Table 5.3.

5.4.2. Messaging performance

A first benchmark we executed aims at measuring message delivery latency and jitter for periodic publishers, i.e., from memory allocation to message reception. The purpose of this test is to evaluate the influence of low-priority subscribers over high-priority tasks. To run the test, a publisher has been set to transmit 8 bytes messages every 10 ms, with one high-priority local node subscribed to the topic; then, several low-priority nodes subscribe the same topic. Low-priority nodes used in the benchmark are also slow consumers: half of them can consume queued messages every 100 ms, the other half every 1 s. As a consequence, they start loosing messages as soon as their receive queue fills, while the high-priority node receives all messages. We run the test with 0, 2, 5 and 20 low-priority subscribers on the same module where the high-priority one is. The same test has been also repeated with the high-priority subscriber deployed on a remote node. Each test lasts 60 seconds, corresponding to 6000 published messages.

Table 5.4 reports the results we collected with the different configurations. The latency increases linearly with the number of subscribers, as it is due to subscription cycling required by message notification. No delay is introduced with the slow consumption of messages by low-priority nodes, thanks to priority handling and message queuing. We observed a latency jitter, bounded to less than 1 µs per subscriber, which comes from the different time of notification required by subscribers with a full queue with respect to ready ones. Although the measured jitter is small, and negligible for most applications, we are investigating different approaches to notify subscribers involved in periodic, hard real-time, tasks. The drawbacks in doing this comes from the need for differentiating sporadic and periodic subscribers, which originally we choose to keep identical to decouple them from the source.

Table 5.4.: Message delivery latency and jitter.

Low-priority	**Latency [μs]**		**Jitter [μs]**	
Subscribers	Local	Remote	Local	Remote
0	6.6	114.6	±0.5	±3.2
2	7.4	117.4	±2.1	±4.8
5	10.2	120.2	±4.2	±6.1
20	23.5	133.5	±16.2	±18.9

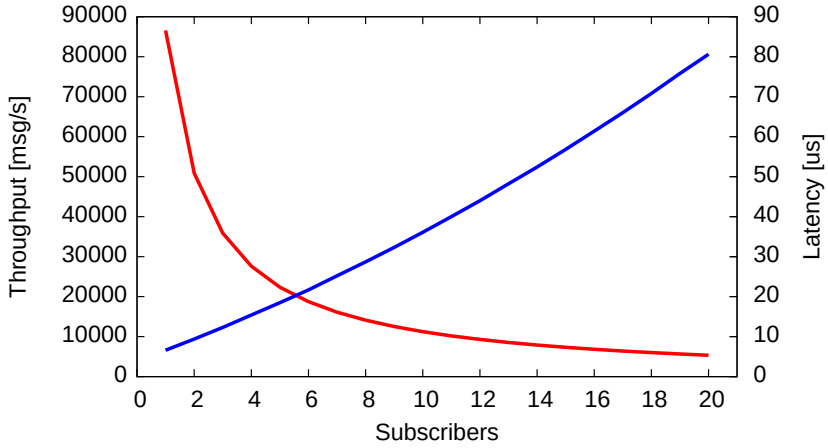


Figure 5.3.: Maximum message throughput, in red, and delivery latency, in blue, with respect to the number of subscribers.

Remote performance is mainly affected by the transport layer, as the middleware makes no difference between local and remote subscriptions. During the benchmark we used RTCAN as transport, which introduces an unavoidable delay due to the transmit time over the CAN-Bus. The jitter for periodic messages in RTCAN is bounded to less than 3 µs (see Section 4.4), which is added to the jitter introduced by the middleware in the figures of Table 5.4.

The second benchmark we run aims at measuring the maximum throughput, in terms of messages per second obtainable with the R2P middleware. In this case, a publisher node publishes messages to several subscribers (from 1 to 20), with the same priority, as soon as they are all ready to receive it. For each configuration, 1 million messages have been transmitted. The graph shown in Figure 5.3 reports the measured throughput and the delivery latency, with respect to the number of subscribers. We measured a maximum throughput of 86600 msg/s, which lowers increasing the number of subscribers in an exponential way; this is due to the publishing time which increases linearly with the number of subscribers. It should be noticed that, in this case, all subscribers are always ready to receive messages, which motivates the higher delivery latency with respect to the previous benchmark. The graph shows that each successful message delivery costs, in terms of latency, less than 4 µs. For remote subscriptions, the maximum throughput is limited by the bandwidth of the communication bus used, which for RTCAN is 384 kbps out of ideal 400 kbps of CAN (i.e., this is the maximum throughput in terms of payload, net of unavoidable overhead).

To summarize, benchmarks showed that the R2P middleware is able to handle a high message throughput, and that high-priority tasks are not influenced by slow consumers. The measured delivery latency and jitter match the requirements of most robotics applications.

Chapter 6

Integration with ROS

R2P is focused on robot platform prototyping and low-level control, while high-level, computation-intensive, functionalities are implemented with ROS, which has become a *de facto* standard. ROS is a framework which typically runs on a desktop-class computer, and hardware devices are currently interfaced to ROS with approaches in contrast to the distributed architecture it fosters. For this reason, we developed µROSnode, a lightweight ROS communication library targeted to resource-constrained devices, allowing to exchange native ROS messages. µROSnode has been developed within this thesis, and it is exploited by the R2P framework to interface robot platforms with ROS, nevertheless it is a stand alone library that may also be used independently from R2P.

In this chapter, we review the common approaches adopted to interface hardware devices to ROS, and briefly describe the ROS communication model. Then we introduce µROSnode, detailing how it was implemented to allow its execution on resource-constrained devices. Finally, benchmarks of its footprint and its communication performance are presented.

6.1. Interfacing hardware devices to ROS

Robots developed with ROS usually feature an on-board computer, which runs several software modules providing the needed requirements, like vision, planning, and navigation. The distributed, loosely coupled, architecture fostered by ROS has turned out to be quite effective for the development of high-level functionalities, but interfacing hardware devices to a ROS system is still an open problem. Most devices used on robots (e.g., generic sensors and actuators), as well as R2P modules, are based on low-cost microcontrollers, which cannot run the standard ROS stack, as we stressed in Section 5.2.1.

6.1.1. Related work

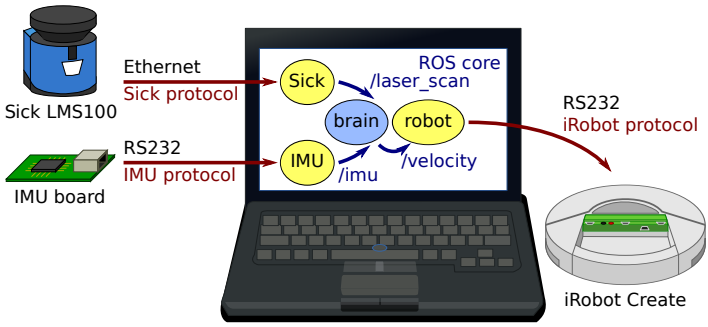
The common solution to interface hardware devices to a ROS system is to write a ROS node, directly communicating with the device, which implements the required communication protocol and acts as a proxy by publishing, or subscribing, messages on a ROS topic. Exchanged data transit through the proxy node, introducing latency and avoiding any direct device-to-device communication, in contrast with the distributed development approach promoted by ROS.

To overcome this problem, the ROS community has proposed three solutions to make ROS available on resource-constrained embedded systems: *eros* [127], *rosc* [62], and *rosserial* [51].

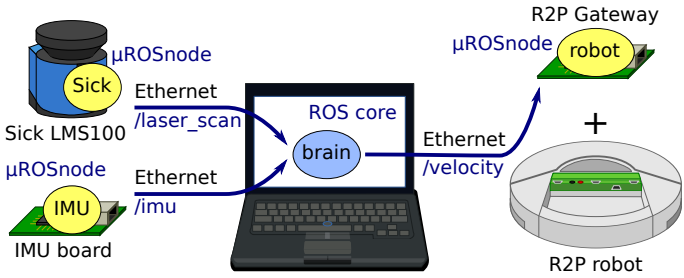
The *eros* projec proposes the creation of a cross-compiling tool-chain to port standard ROS to embedded systems. It aims at running a complete ROS system on embedded devices such as smart phones or industrial computers. These systems are clearly much more powerful than microcontrollers generally used in sensors and actuators. Moreover, *eros* is, currently, limited to a sketch on the ROS documentation wiki, with apparently no active development ongoing.

The *rosc* project, instead, aims at implementing a complete ROS communication library, like the *roscpp* or *rospy* libraries, using pure ANSI C language. This would enable to handle ROS communication on resource-constrained devices, which is our goal. Unfortunately, at the moment of writing, it is under heavy development and no working implementation is available yet.

Presently, what is really available in terms of code and documentation, among the different proposals for embedding ROS, is *rosserial*, a general protocol for point-to-point ROS communications over a serial transmission line. It simply serializes and de-serializes standard ROS messages, and adds a packet header which allows multiple topics to



(a) Devices with legacy ROS proxy nodes



(b) ROS-enabled devices using µROSnode

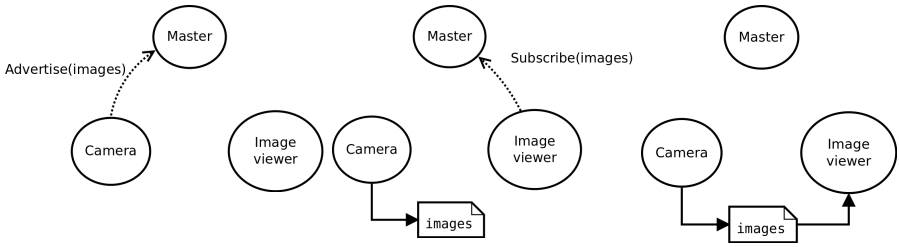
Figure 6.1.: A simple robot running ROS, the legacy approach and the µROSnode approach

share a common serial link. Its simplicity makes *rosserial* suitable for simple architectures such as 8-bit microcontrollers used on *Arduino*, but there is no natural integration with the ROS ecosystem: a proxy node on the host machine is still required, although hidden to the user, to bridge the connection from the serial protocol to the ROS network.

6.1.2. Toward native ROS hardware components

µROSnode proposes a different approach to integrate hardware components within ROS. Using µROSnode, robotic devices register as native ROS nodes, without the need of specific drivers, or software proxies, running on the host computer. In this way, robot designers can seamlessly connect sensors and actuators to ROS systems through an Ethernet connection.

Consider, for instance, a generic robot, as in Figure 6.1a, made of an *iRobot Create* [73] moving base controlled via RS232 connection, a *Sick LMS100* [125] laser scanner, and a custom IMU board. These devices



(a) Advertisement (b) Subscription (c) Communication

Figure 6.2.: The role of ROS Master

are connected to a host computer running a ROS application which uses them. As they do not communicate with ROS directly, the computer must run custom ROS nodes which interface with their low-level drivers to ROS itself.

Instead, consider the configuration in Figure 6.1b. The laser scanner could be a ROS-enabled device publishing native ROS messages, as it already features an Ethernet port and an on-board processor, and the same applies to the custom IMU board, thanks to modern microcontrollers and µROSnode; a robot platform developed with R2P components can simply add a Gateway module, which provides Ethernet connectivity and runs µROSnode (see Section 7.7), to communicate with ROS. With this design, the user only has to connect the Ethernet cables, launch ROS, and start developing the application, without worrying about low-level protocols or device drivers.

6.2. ROS communication model

ROS has been already reviewed in Section 5.2.1; basically, it provides several communication patterns to exchange data between peers: topics, which follow the publish/subscribe paradigm, services, to perform remote procedure calls, and actions, to issue asynchronous requests. Here we mainly focus on how communication is set up and handled by ROS, before describing how µROSnode has been implemented to support native ROS communication.

6.2.1. ROS Master

Each ROS systems requires to run a ROS *Master*, which is unique in the network and is generally instantiated by the *roscore* application. The Master provides naming and registration services to all instanti-

ated ROS nodes, and tracks publishers and subscribers to topics as well as services. The main role of the Master is to enable individual ROS nodes to locate the needed data providers. Figure 6.2 shows a typical association of a publisher and subscriber node: the producer advertises a topic, which is registered on the master, so that the consumer is correctly associated to it on subscription; once the nodes have located each other they communicate by peer-to-peer connections. The Master node also provides the Parameter Server, which is used by nodes to store and retrieve parameters at runtime.

Communication with the Master node is accomplished by *remote procedure calls*, which are implemented by exchanging *XML-RPC* messages.

6.2.2. Remote procedure calls

ROS nodes are capable of calling *remote procedures*, available through the API exposed by the node objects themselves. Remote procedure calls are executed by using the *XML-RPC* protocol, which uses XML to encode calls and HTTP as a transport. These calls are only used to manage the status of the computation graph and some global settings, and they are not intended for actual data streams. XML-RPC was chosen by the designers of ROS because of the many available implementations, in several languages, and because the use of the XML markup language makes debugging simpler. Moreover, XML-RPC calls are *stateless*, a property which simplifies the control logic, since there is no state to keep track of. These characteristics make XML-RPC parsing much easier than bare XML, while keeping most of its advantages.

Three different XML-RPC APIs are defined by ROS:

- the *Master API* exposes the methods implemented by the unique Master, e.g., to allow nodes to register as publishers, subscribers, or service providers, and to perform name resolution;
- the *Slave API*, exposed by every ROS node, provides methods to receive callbacks from the Master (i.e., to notify published messages), and to negotiate connections with other nodes;
- the *Parameter Server API* exposes the methods implemented by the unique Parameter Server, to manage the centralized directory of shared global parameters.

6.2.3. Data transport

Messages exchanged by nodes, on topics as well as through services, are transferred by a custom protocol, named *TCPROS*, over a TCP

connection. For lower latency, but less reliable, transfers, UDP streams are provided by an alternative transport, *UDPROS*, but this is currently not supported by all the components of the ROS framework, so it is rarely used. Within the same process, messages can also be exchanged with a zero-copy transport, by using *nodlets*, which are a variant of standard ROS nodes.

TCPROS uses standard TCP/IP sockets to transfer message data. To instantiate a connection, a *TCPROS Header*, containing message specifications and routing information, has to be sent to a TCP socket server. For instance, to perform a subscription the consumer sends a header containing its own name, the name of the topic, the complete message definition (see Section 6.2.4), and a few other fields. The header is routed by the socket server to the corresponding publisher, which replies by sending back the message type: if everything matches, the connection is finally established and messages start to be exchanged.

6.2.4. Type descriptor

Data streams are based on the exchange of messages of a particular type, specified by the topic/service at registration time. A message is a sequence of values, which type and organization is defined by a type descriptor. In ROS, two kinds of descriptors are used: the *topic descriptor*, and the *service descriptor*.

A ROS topic message descriptor is written as a plain text file, which specifies message content. Every line can define a strong-typed variable, where the type is either a primitive type, or the name of another message type descriptor, or even an array (fixed or variable) of any of such types. Descriptors can also declare constant names of primitive types. Listing 6.1 shows the content of a sample descriptor containing other ROS messages, and Listing 6.1 reports the corresponding expanded structure.

Service type descriptors follow the same syntax of message types, with a separator line: the first part specifies the request message descriptor, and the second part represents the response message descriptor.

Since descriptor content may be different among nodes, even if their names coincide, a MD5 sum is used to determine if two nodes actually share the same type.

6.3. µROSnode

µROSnode is a lightweight implementation of the ROS communication stack, targeted to resource-constrained devices, which emulates a stan-

```
# A representation of pose in free space.
Point position
Quaternion orientation
```

Listing 6.1: A ROS message descriptor

```
geometry_msgs/Point position
  float64 x
  float64 y
  float64 z
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
```

Listing 6.2: The expanded structure of a ROS message

dard ROS node supporting topics, services, and parameters. In this section we present the µROSnode architecture, and detail its implementation, reporting the design choices we made to implement all the functionalities ROS expects from a compliant node.

6.3.1. Architecture

The main goals of µROSnode are portability and low memory footprint, while being fully compliant with the ROS communication protocol. To this extent we decided to develop the whole library in ANSI C89, which is supported by almost all C compilers for embedded systems. Being targeted to hardware devices for robotics, which often involve the execution of tasks with real-time constraints, µROSnode is designed to run on the top of a RTOS providing common synchronization primitives. Software portability has been obtained through an appropriate hardware abstraction layer, which acts as interface to required low-level functionalities such as communication, threading, synchronization, and memory management. In its current implementation, µROSnode supplies bindings the ChibiOS/RT RTOS [46] and the LWIP lightweight TCP/IP stack [47], together with a port for *Posix* operating systems.

The architecture of µROSnode is organized in modules, grouped in three main categories.

- *Core modules*, which are platform-independent, provide the main features of µROSnode: internal management of topic and ser-

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 309

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><i4>1</i4></value>
            <value></value>
            <value><i4>6544</i4></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodResponse>
```

Listing 6.3: Example of XML-RPC response

vice registries, node state machine, abstraction layers, parsers and streamers of XML-RPC and ROS-specific protocols (e.g., TCPROS).

- *Low-level-driver modules* are responsible of binding the core abstraction layers to the specific target platform; they are the components which need to be ported to support different hardware architectures and different operating systems.

- *User modules* are application-dependent, providing functions for topic and service handlers, as well as marshalling/unmarshalling functions for the corresponding message types.

6.3.2. XML-RPC

The first feature we need, to be a compliant ROS node, is the XML-RPC protocol, and the related handlers. Encoding remote calls with the XML markup language surely shows several advantages, as we mentioned in Section 6.2.2, but XML-RPC is not trivial to implement on resource-constrained devices, mainly because of its verbosity and of the parsing process necessary to extract the needed information.

Take for instance the example reported in Listing 6.3, which shows the response to a simple remote call supposed to return 3 values (i.e., the Slave API *getPid()* method [117]). The size of the complete message

is 309 bytes of paylod, as reported by the header, plus the length of the header itself, 66 bytes in this case. The actual content is composed by two integers (i.e., 8 bytes), plus a string with variable length, which is empty in the considered example. It follows that there is an overhead of 367 bytes to transfer the 8 bytes we are interested in. Moreover, received messages need to be parsed: in this case, the first useful data is nested into 9 XML tags, and this depends on the specific message, so we cannot simply skip a part of the payload to extract the actual content.

Common XML parser allocate a buffer to store the full message; this is not applicable to resource-constrained devices, as RAM is limited, and, additionally, the length of a message varies and it is not known a-priori, requiring dynamic allocations. µROSnode provides an optimized XML parser exploiting the properties of deterministic top-down grammars [119]. It performs single pass parsing with zero-copy, directly accessing the memory already allocated by the underlaying communication stack (e.g., LWIP). A single preallocated buffer is still needed, to address a well known flaw of XML-RPC: the protocol allows to store strings not only inside the dedicated *string* tags, but even into generic *value* tags; in this case, the length of the payload is not specified, which is an exception to the mentioned grammars, and a buffer is mandatory, with a size big enough to handle the maximum possible string length.

About actual procedures, all ROS *Slave API* methods are implemented by µROSnode, as well as calls to the *Master API* and to the *Parameter Server API* implemented by the Master.

6.3.3. TCPROS

Full support to TCPROS is provided by µROSnode. Headers to establish topic and service connections with other ROS nodes can be filled by using a set of macros, which are expanded to corresponding marshalling and unmarshalling statements, along with error checking. Once the connection is active, the TCPROS protocol is very close to the binary representation in memory, so µROSnode merely performs raw data transmission and reception.

Transmissions are performed without buffering: message fields are sent on-the-fly, without filling the entire message, to reduce memory requirements as well as latency.

Inbound data can be handled in two ways: a first approach is to allocate memory from the heap, buffer the entire message upon reception, and finally deliver it to the user application; this is easer for application development, as messages can be slowly consumed and then released, but, on constrained platforms, heap fragmentation becomes soon a prob-

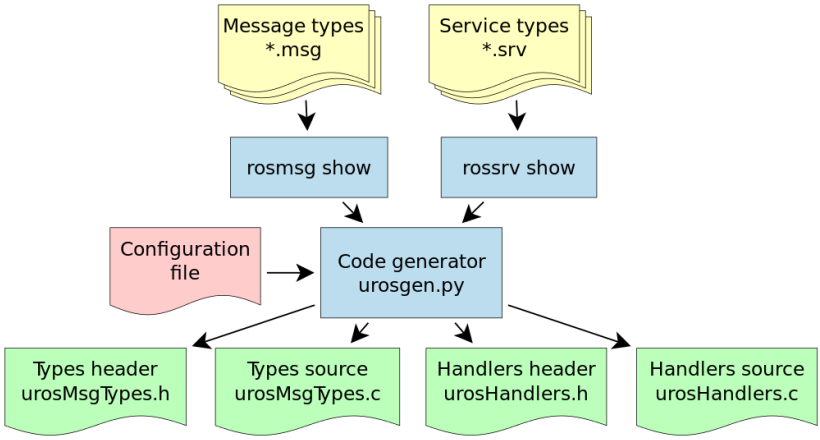


Figure 6.3.: Code generator flowchart

lem. A second approach is to directly access the buffers allocated by the network stack, with zero-copy, but memory needs to be released soon to prevent loosing successive packets. This increases the throughput, as we will see in Section 6.4, and it is the way we forward messages from ROS to R2P nodes: data is just transfered into the corresponding R2P message, and memory is quickly released.

Routines to marshal and unmarshal message content belong to the User modules, and should be implemented by application-specific code. As writing these procedures by hand, as well as topic/service handler routines, is a repetitive and error-prone task, they are actually generated by an automated tool provided by µROSnode, which is presented in the next Section.

6.3.4. Integration with user application

Having low footprint and high efficiency as main goals, µROSnode exposes an API, and follows a programming model, which is quite different from ROS. To ease the integration of user applications within µROSnode, we developed a code generator tool which takes care of generating the needed marshalling and unmarshalling functions, the ROS headers, and the handling routines. The tool, *urosgen*, is a python script which extracts information from standard ROS utilities, so that it should be robust to future changes in the ROS framework. The workflow of *urosgen* is sketched in Figure 6.3.

It is only required to fill a configuration file, specifying the name of the node, its publishers, subscribers, and services, with the corresponding ROS type descriptors. Listing 6.4 reports the configuration file to

```
[Options]
nodeName                        = turtlesim

[PubTopics]
rosout                          = rosgraph_msgs/Log
turtleX/pose                    = turtlesim/Pose
turtleX/color_sensor            = turtlesim/Color

[SubTopics]
turtleX/command_velocity        = turtlesim/Velocity

[PubServices]
clear                           = std_srvs/Empty
kill                            = turtlesim/Kill
spawn                           = turtlesim/Spawn
turtleX/set_pen                 = turtlesim/SetPen
turtleX/teleport_absolute       = turtlesim/TeleportAbsolute
turtleX/teleport_relative       = turtlesim/TeleportRelative

[CallServices]
```

Listing 6.4: *urosgen* configuration file for the *turtlesim* demo

implement the *turtlesim* demo [116], a ROS sample application most ROS developers may be familiar with. Once the involved message type names are known, the tool extracts their description, by running the *rosmsg* and *rossrv* ROS utilities, from the corresponding *.msg* and *.srv* files. Finally, the message types are processed to produce needed marshalling and unmarshalling functions.

Topic and service handler stubs are generated for each configuration file entry. Listing 6.5 shows the generated publisher handler for the */turtleX/pose* topic, while Listing 6.6 reports the code subscribing the */turtleX/velocity* topic: the only needed instructions are to fill, or to read, message content. When using µROSnode to interface R2P and ROS, messages are just forwarded, inside the corresponding handlers. The tool also generates detailed self-documentation for *Doxygen* (not reported in listing examples, as well as most of the comments).

6.4. Benchmarks

In this section we present µROSnode benchmarks, reporting the memory occupation and the maximum achievable throughput, and providing figures about the overhead introduced on common embedded systems. These benchmarks are rather simple, since they have been designed

```
uros_err_t pub_tpc__turtleX__pose(
    UrosTcpRosStatus *tcpstp) {

  UROS_TPC_INIT_H(msg__turtlesim__Pose);

  while (!urosTcpRosStatusCheckExit(tcpstp)) {

    /* TODO: Generate the contents of the message.*/

    UROS_MSG_SEND_LENGTH(msgp, msg__turtlesim__Pose);
    UROS_MSG_SEND_BODY(msgp, msg__turtlesim__Pose);

    clean_msg__turtlesim__Pose(msgp);
  }

  tcpstp->err = UROS_OK;

_finally:
  UROS_TPC_UNINIT_H(msg__turtlesim__Pose);
  return tcpstp->err;
}
```

Listing 6.5: Generated publisher handler

```
uros_err_t sub_tpc__turtleX__command_velocity(
    UrosTcpRosStatus *tcpstp) {

  UROS_TPC_INIT_H(msg__turtlesim__Velocity);

  while (!urosTcpRosStatusCheckExit(tcpstp)) {
    UROS_MSG_RECV_LENGTH();
    UROS_MSG_RECV_BODY(msgp, msg__turtlesim__Velocity);

    /* TODO: Process the received message.*/

    clean_msg__turtlesim__Velocity(msgp);
  }

  tcpstp->err = UROS_OK;

_finally:
  UROS_TPC_UNINIT_H(msg__turtlesim__Velocity);
  return tcpstp->err;
}
```

Listing 6.6: Generated subscriber handler

to test the performance of µROSnode only; more complex benchmarks would introduce a significant bias due to multi-threading and networking subsystems and would not be appropriate to evaluate µROSnode.

Benchmarks were run on the R2P gateway module; hardware details are presented in Section 7.7, here we just anticipate it sports an ARM Cortex-M4 STM32F407 microcontroller by ST Microelectronics, featuring $112Kb$ of available RAM and clocked at $168MHz$. µROSnode should be able to run on any Ethernet-enabled microcontroller with similar characteristics. Tests were performed by running µROSnode on top of ChibiOS/RT and the *LWIP* network stack.

6.4.1. Memory footprint

The first benchmark is aimed at computing µROSnode footprint in terms of RAM and code size. To this extent, we implemented with µROSnode the *turtlesim* application [116], which is used in common ROS tutorials. The code was compiled with different *gcc* compiler optimization levels, and with µROSnode and ChibiOS/RT assertions, and runtime checks, enabled or disabled.

To compute the memory footprint, we started from a plain firmware with ChibiOS/RT and related features (USB, Ethernet, shell emulator, and self-tests), then incrementally included LWIP, µROSnode, and the *turtlesim* user application. According to this benchmark, µROSnode footprint is less than 40 Kbyte, comparable with the LWIP stack or the operating system.

Table 6.1 summarizes the maximum stack depth reached by the principal components of the *turtlesim* demo, with *-O3* optimization enabled and all checks disabled. Given these results, a few tens of µROSnode publisher, subscribers, or services, could run on the test platform, which can be considered satisfactory for most embedded robotic devices.

6.4.2. Communication performance

Communication performance has been measured by connecting the R2P Gateway board to a standard notebook host, featuring an *Intel T6500* processor (dual core, 2.1 GHz), through a 100 Mb/s Ethernet connection. The firmware was compiled with maximum optimization level (i.e., *-O3*), disabling both assertions and error messages. Within these benchmarks, a single TCPROS topic is continuously streamed from the µROSnode to the laptop, in the transmission case, and from the laptop to the µROSnode, in the reception case. The topic is of string type (i.e., ROS *std_msgs/String* message type), so that it is possible to easily assign

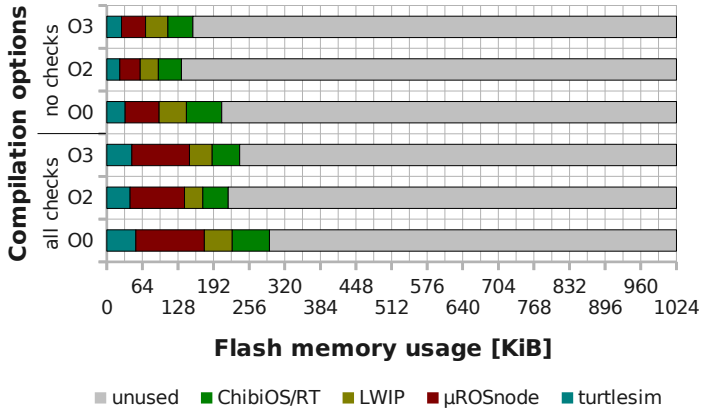


Figure 6.4.: Code footprint of the *turtlesim* application

µROSnode component	Max stack depth [B]
`pub_srv__spawn`	904
`pub_tpc__rosout`	432
`pub_tpc__turtleX__pose`	408
`sub_tpc__turtleX__command_velocity`	464
`urosNodeThread`	888
`urosRpcSlaveListenerThread`	476
`urosRpcSlaveServerThread`	616
`urosRpcSlaveServerThread`	476
`urosThreadPoolWorkerThread`	148

Table 6.1.: Stack usages of the principal *turtlesim* components.

content with different length to published messages.

In the transmission benchmark, the R2P Gateway publishes messages subscribed by the host computer via the ROS *rostopic* utility. Results are reported in Figure 6.5, which shows the CPU usage of the involved components (i.e., the µROSnode TCPROS server, and the two threads instantiated by LWIP), the CPU idle time, and the maximum reachable throughput in msgs/s, while varying the message size. When sending messages smaller than 100 bytes, the board actually saturates the subscriber on the laptop, at $\approx$ 20000 msg/s, with still some CPU idle time left. By increasing the message size up to 2000 bytes, the LWIP processing thread takes most of the processing, leaving no CPU idle time, and limiting the overall throughput. Beyond 2000 bytes per message, the

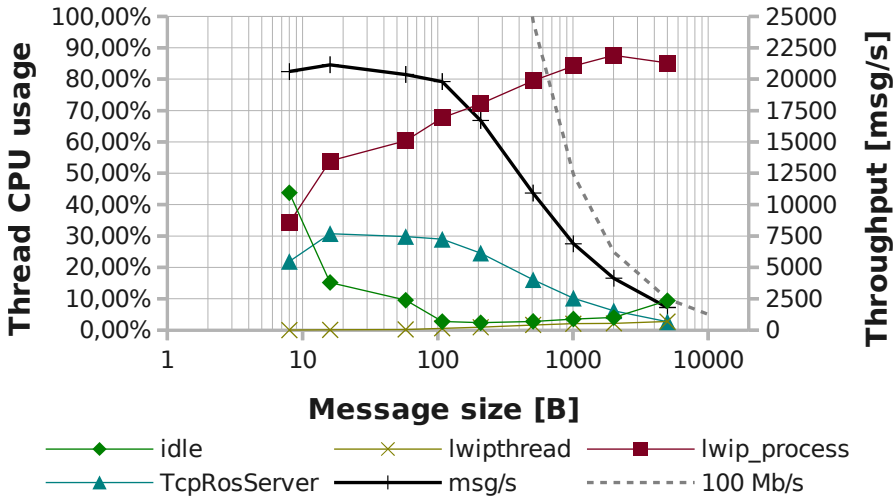


Figure 6.5.: Transmission benchmark results

throughput approaches the 100 Mb/s limit of the Ethernet connection, which is reached with messages over 5000 bytes.

These results demonstrate that µROSnode can handle, on the target platform, really high throughputs with small messages, which is the most common situation when communicating with hardware components of a robot; when message size increases, the TCP/IP stack limits the performance, probably mainly due to LWIP output buffer handling, which is performed within the *lwip_process* thread.

The reception benchmark involves a publisher on the laptop, again instantiated by using the ROS *rostopic* utility, which publishes messages at the maximum possible rate, while the R2P gateway subscribes the corresponding topic. In this case, the throughput limit of the *rostopic* publisher on the host machine is of $\approx$ 10000 msg/s.

The plot in Figure 6.6 reports the benchmark results, showing the same figures of the transmission benchmark. With a size of up to 500 bytes per message, the board is able to receive messages at the target rate, i.e., 10000 msg/s, still maintaining an idle time for the CPU higher than 50%, and a µROSnode topic handler load below 20%. Between 1000 bytes and 2000 bytes per message, the LWIP threads require $\approx$ 70% of the CPU time, while the topic handler settles at $\approx$ 30% making the CPU almost at full load. Beyond 2000 bytes, the 100 Mb/s limit is reached, as also highlighted by the increase of CPU idle time. Results from the reception benchmark shows that both µROSnode and LWIP saturate the Ethernet bandwidth, while with small messages a limit is

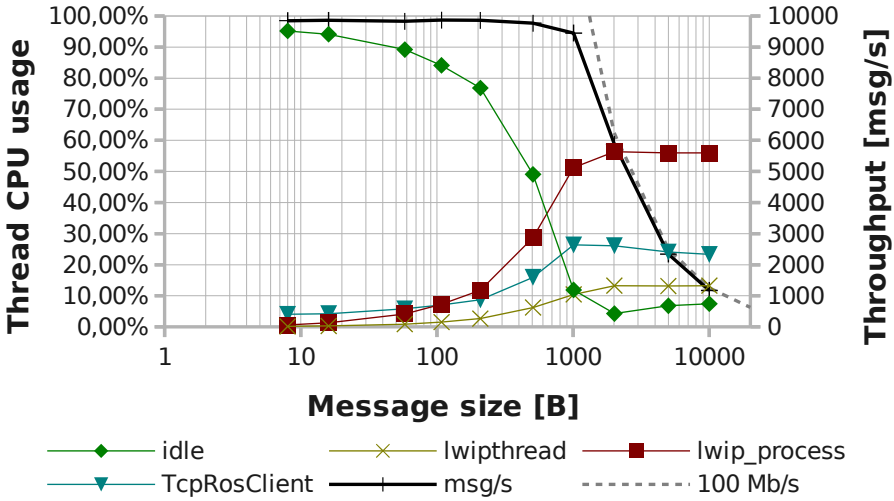


Figure 6.6.: Reception benchmark results

imposed by the ROS *rostopic* utility.

Overall, the benchmarks show that modern microcontrollers can handle a high throughput on Ethernet connection, and that µROSnode can fit also resource-constrained devices, allowing to natively communicate with a ROS systems. These results demonstrate the feasibility of the proposed approach: it is possible to develop ROS-enabled hardware devices for robotics, which do not need specific drivers or software proxies to be interfaced to ROS. Within the R2P framework, this enables to seamlessly integrate R2P-based robotic platforms, which still rely on real-time communication to provide low-level control, with high-level software developed in ROS.

Chapter 7

Hardware modules

The last building blocks offered by the proposed framework to enable robot designers rapidly prototype their applications are a set of off-the-shelf hardware modules, each focused on solving a specific functional requirement of common robot platforms. Modules use RTCAN as real-time communication channel, and run embedded software implemented through nodes of the publish/subscribe middleware. Having identified the platform requirements, developers and researchers can select the needed hardware modules, or develop new ones starting from the reference design, easily connect them, and deploy the low-level control software with a distributed approach.

In this Chapter, we firstly introduce, in Section 7.1, the main characteristics of R2P modules and the reference design they are based on. Then, a review of the hardware devices we have already developed, to allow to test the framework some real robots, is presented. Notice that the software nodes reported are just an example of the possible nodes one could develop end run on R2P modules, and that parameters as topic names are fully configurable. Module schematics are reported in Appendix A.

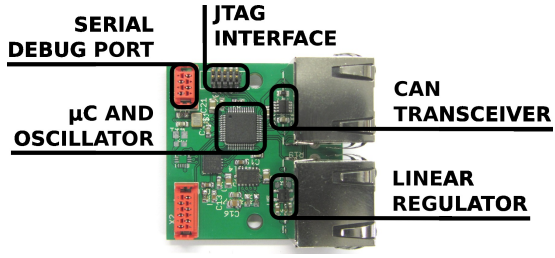


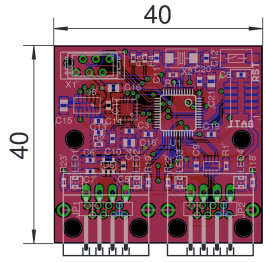
Figure 7.1.: A R2P hardware module, with components from the reference design highlighted.

7.1. Reference design

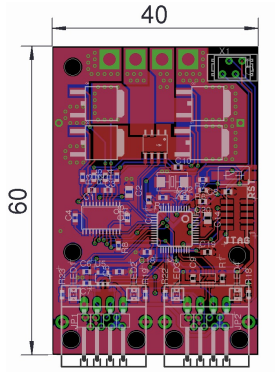
All R2P modules are based on the same basic components, and new modules can be easily developed by starting from the open source reference designs. Figure 7.1 shows one of the R2P modules, with shared components highlighted.

Each module sports a STM32F103C8T6 ARM microcontroller by ST Microelectronics as processing unit, which features a Cortex-M3 core running at 72 Mhz, 20 Kbytes of RAM and 128 Kbytes of Flash memory in a 48 pin LQFP package. We chose the STM32 family as it provides a wide set of peripherals (e.g., 4 hardware timers with 4 channel each, a 12 bit A/D converter with 10 inputs, 2 SPI, 2 I2C and 3 USART interfaces, CAN and USB controllers, and 36 general purpose input/outputs), and it is extremely low cost, with a target price of about 3 USD for 1000 units. Moreover, being based on the ARM Cortex-M processor, it is supported by many compliers and toolchains (e.g., the open source *GNU Tools for ARM Embedded Processors* [5]) and by several open and commercial real-time operating systems (e.g., ChibiOS/RT [46], FreeRTOS [113], eCos [48]), so that R2P hardware modules may eventually be used as ready to use platform also outside the proposed framework. The reference design also includes a 3.3 Volt linear regulator with a resettable fuse on its input, the CAN transceiver (a MAX3051 from Maxim Integrated), the JTAG interface, and a serial port for debugging purposes. Actually, the two debug interface footprints can be left unpopulated if nodes are developed with the tools provided by the R2P framework (see Section 5.3.8).

We defined two standard sizes for R2P modules: smaller ones are 40×40 mm in size, while bigger ones (i.e., power modules) are 60×40 mm. Both versions share the same mounting holes, for an easier attachment to robot frames and to adopt similar housings.



(a) 40 mm × 40 mm



(b) 60 mm × 40 mm

Figure 7.2.: R2P hardware modules size expressed in millimeters.

Modules are connected using a single cable to transport both power and data, fostering for a *plug-and-play* approach. Each module has two ports to be connected to the previous and the next component, following a daisy-chain connection schema (see Figure 7.3); this reduces wiring and allows to easily add modules to existing systems. We chose a standard RJ45 Ethernet patch cable as physical connection, as it matches CAN bus specifications [61] to guarantee 1 Mbps operation in harsh environments and it is easily available. A pin header footprint is also present on each board, as it provides space saving and a simple connection alternative when modules are close and RJ45 jack/plug pairing space is a problem.

Power supply requirements are usually not consistent among different electronic devices, then a compromise must be reached. According to power requirements, robotic modules can be divided in two categories: modules needing only little power, i.e., less than 200 mA, like most of the sensors, and modules requiring much higher power, like motor drivers. The R2P bus has been designed to operate at 5 V, which suits most of the requirements of todays electronic devices, while modules that require a higher power supply must rely on an auxiliary connection and power source. Of the 8 available conductors in standard Ethernet cables, one pair is used for the CAN bus, while the other 6 are used for power supply (3 for the positive and 3 for the negative). Common CAT5e UTP cables use AWG-24 copper conductors (0.5 mm in diameter), and using three of them gives a voltage drop of about 50 mV/A for each meter [14]. It follows that a 2 meters long bus with 10 modules attached exhibits a drop of about 200 mV, so power dissipation is limited and voltage at

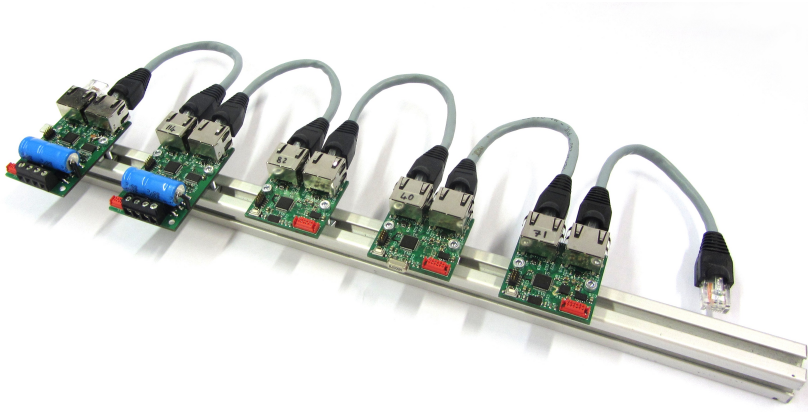


Figure 7.3.: A set of R2P modules connected together: the daisy-chain connection schema reduces wiring and eases the addition of new modules.

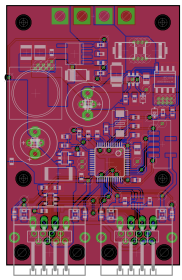
bus end is far above linear regulator requirements. The bus can safely handle up to 20 hardware modules, each consuming 200 mA maximum, over an up to 4 meter long cable.

7.2. Power supply

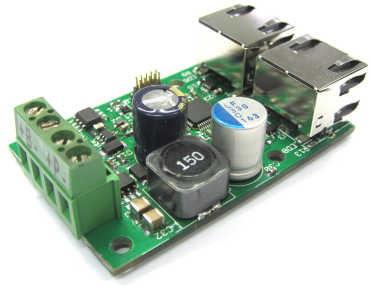
The power supply module (PS) is responsible of powering the R2P bus and, as a consequence, all connected modules. The bus can also be powered in other ways, if a 5 V supply is already present on the robot, but this module also provides some useful features. First of all, it sports a switching DC/DC converter, with a wide range of power inputs and a stable 5 V output, so that it can be used with a variety of batteries; moreover, it monitors battery voltage and can be used to estimate remaining battery life. This module is 60×40 mm in size. The layout of the module and an assembled board are visible in Figure 7.4.

7.2.1. Hardware design

The main role of the PS module is to provide power to the bus. To this extent, we designed a switching DC/DC regulator based on the TPS5450 step-down converter by Texas Instruments. The regulator accepts any input voltage between 5.5 V and 36 V and provides a stable 5 V output with a maximum current of 4 A, thus able to power up to 20 R2P standard modules, with an efficiency in the range of $88 - 93\%$. The PS



(a) PCB layout



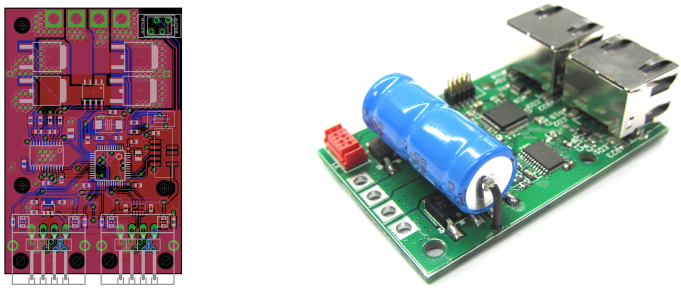
(b) Assembled board

Figure 7.4.: Power supply module.

module has two inputs, so that it can be connected to both robot batteries and an auxiliary input (i.e., a wall adapter to be used while charging batteries). A power ORing circuit automatically switches between the two inputs, and guarantees very low voltage drop on the battery path by driving a N channel MOSFET, to reduce power losses and improve efficiency and battery life. Inputs are also protected from voltage spikes (e.g., when connecting and disconnecting the power cables) by a pair of TVS diodes, improving reliability and protecting the connected modules. A voltage divider, with a Zener diode protection, is used to acquire the voltage level on battery input, which can be monitored to estimate remaining battery life. On the output path, a low value resistor (i.e., 30 mOhm), and the INA193 high-side measurement current shunt monitor by Texas Instruments, allow to monitor the current flowing to the bus.

7.2.2. Embedded software

The core libraries of the PS module expose interfaces to acquire the battery voltage and instant current consumption through the internal A/D converter. Analog watchdogs can also be configured, i.e., high and low thresholds to raise events when the acquired value is out of the preset range. A ready-to-use R2P node tracks battery voltage and, with proper calibration, can publish the remaining battery life on the */battery* topic, so that the robot can take actions as a consequence (e.g., return to the charging station when running out of power). If a system only drains power from the R2P bus, current consumption can be used to profile power requirements too, providing better estimates.



(a) PCB layout

(b) Assembled board

Figure 7.5.: DC motor controller module.

7.3. DC motor controller

One of the first R2P modules we developed is a DC motor controller board (DCM), to satisfy one of the most common primary platform requirements: drive a motor and let the robot move. The DCM module features all the electronics needed to drive motors from 7 V up to 36 V, with a maximum current of 20 A without heat sink, thanks to the state-of-the-art components employed. This module is 60 × 40 mm in size. The layout of the module and an assembled board are visible in Figure 7.5.

7.3.1. Hardware design

The high-power H bridge uses an Allegro A4940 automotive full bridge MOSFET driver, which includes a charge pump to drive high-side N Channel MOSFETs for better efficiency. Indeed, we use 4 discrete MOSFET with a Drain-Source resistance of only 3 mOhm, reducing power dissipation to only 1.2 Watt, on each active transistor, with 20 Ampere of current drawn. This allows to dissipate heat on the PCB copper, removing the need of an heat sink in most applications. The MOSFET driver inputs are independent, i.e., the transistors are driven directly by the STM32 advanced timer TIM1, to precisely configure the behavior of the H-bridge, including the dead time to prevent cross conduction, for accurate motor driving. Additional cross conduction protection is provided at hardware level, imposing a minimum dead time of 1 µS, to prevent hardware damages. An incremental optical encoder (e.g., mounted on motor shaft) can be connected to the encoder input, to acquire motion feedback and perform closed loop control. Finally, a ACS711 Hall-effect current sensor by Allegro Microsystems is used to measure the current

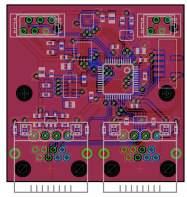
flowing into motor windings, keeping a low-resistence path, in the range of ±25 Ampere.

7.3.2. Embedded software

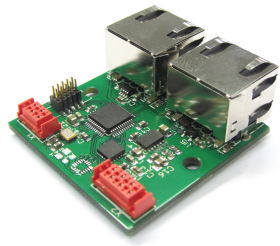
The firmware of the DCM module provides core functionalities to realize different PWM control strategies, i.e., sign/magnitude or locked-antiphase, and with fast or slow current decay mode. Frequency and resolution of PWM can be adjusted: the maximum frequency is mainly limited by the dead time, and by the turn-on and turn-off times which are under 100 nS; STM32 timers can be configured to use up to 16 bits to synthesize the PWM wave, with a counter clock of 72 Mhz. Obviously, a compromise has to be reached, as higher PWM resolution lead to lower realizable periods. By default, the module is configured for sign/magnitude control, with a PWM frequency of 17.5 Khz, and fast decay mode. Quadrature encoder readings are performed by another hardware timer, and a function to get the number of ticks from the last reading is available. Closed loop position or velocity control can be performed by using the provided PID controller implementation, which features saturation filters and conditional integrator for anti-windup; to be application-agnostic, the motor is controlled in terms of ticks, and conversion to actual measurement units is left to the kinematics. Additionally, a maximum current value can be expressed for current-chopping, to impose current/torque limits or to realize constant current control. Dynamic current/torque control is under development.

A node is run by default on the DC module, subscribing the */setpoint* topic, and publishing on the */encoder* topic, to easily drive motors through the R2P middleware. It can be configured for position or speed control, and a timeout can be also specified: if the module does not receive a new setpoint for a certain amount of time, the motor is actually stopped for safety. PID parameters (i.e., k_P, k_I, k_D, and saturation limits) can be tuned by updating the node configuration structure, and saved to the Flash memory when the control law is satisfactory.

On the motor boards we often run also the nodes responsible of computing forward and inverse kinematics, as will be shown in Chapter 8. Currently, we have developed ready to use nodes to drive differential robots (see Section 8.2) as well as three wheels holonomic omnidirectional platforms (see Section 8.1).



(a) PCB layout



(b) Assembled board

Figure 7.6.: Inertial Measurement Unit module.

7.4. Inertial measurement unit

The inertial measurement unit module (IMU) allows the measurement of acceleration, angular velocity, and bearing with respect to the heart magnetic field, together with the altitude of a robot. By applying filtering algorithms, 6 DoF pose and attitude can be estimated; moreover, an external GPS receiver can be attached to integrate its measures in a R2P embedded architecture. Inertial measurements can be a functional requirement of a robotic platform, e.g., a two wheeled balancing robot (see Section 8.3), or can be used to improve performance of control algorithms or odometry estimates. The IMU module is equipped with state-of-the-art MEMS sensors and on board filtering algorithm to provide reliable estimates. This module is 40 × 40 mm in size. The layout of the module and an assembled board are visible in Figure 7.6.

7.4.1. Hardware design

The IMU module features several MEMS sensors to measure the different physical quantities. Linear accelerations and magnetic field measurements are acquired by a LSM303DLHC 3 axis accelerometer and magnetometer sensor by ST Microelectronics, which embeds in a very small package state of the art MEMS devices. Acceleration with full scale range going from ±2 g up to ±16 g can be measured, with a sensitivity down to 1 mg/LSB and an acceleration noise density of 220 µg/$\sqrt{Hz}$. The sensor has a digital output over I2C interface, providing 12 bit representation of the measured value and a configurable Output Data Rate (ODR) from 1 Hz up to 1.344 Khz. Magnetic field full-scale ranges from ±1.3 to ±8.1 gauss, represented by a 12 bit digital value with a sensitivity as small as 0.9 mgauss and ODR up to 220 Hz. A MEMS gyroscope (the ST Microelectronics L3G4200D) provides direct measurement of

angular velocities, with a full-scale range from 250 dps to 2000 dps. It is also a digital output sensor, via SPI interface, with a 16 bit on board A/D converter providing up to 800 samples/s; its best sensitivity is 8.75 mdps/LSB and the declared noise density is of 0.03 dps/$\sqrt{Hz}$. Finally, pressure measurements are performed by a LPS331AP MEMS sensors from ST Microelectronics, a 24 bit digital barometer which has a sensitivity of 0.020 mbar RMS (i.e., less than 20 cm altitude difference) and an ODR from 1 Hz to 25 Hz. An additional connector is present, to interface an external GPS receiver, exposing a TTL serial port and 3.3 V power supply.

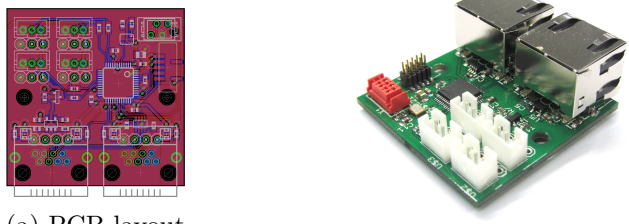
7.4.2. Embedded software

The IMU module provides interfaces to configure and run the on board sensors, and to get the respective readings. However, inertial measurement are noisy, and filters need to be applied to extract useful information such as attitude and heading (AHRS) estimates. A variety of filtering algorithms have been proposed in the literature, even recently, as modern MEMS sensors enable to perform inertial measurements with low cost devices. The IMU board provides, within its core functionalities, the implementation of two AHRS algorithms which have been recognized to provide effective estimates with low cost inertial sensors: one based on nonlinear complementary filters [92], and the other performing gradient descent optimization [90].

An interface to the MEMS pressure sensor provides pressure, thus altitude, measurements, which can be conditioned, eventually, by activating a low pass filter. The GPS NMEA parser, which is derived from an open source project [12], can be activated to receive GPS absolute position measures by the optional external receiver.

Nodes are available to publish orientation data using one of the available filtering algorithms, with tunable parameters and configurable stream period. The full representation can be published (i.e., roll, pitch, yaw, and the respective angular velocities), by default on the */imu* topic, or only inclination and rate among a single axis, on the */tilt* topic. A node to stream unfiltered readings is also provided if raw data are needed, as in the case of the application presented in Section 8.2. Barometric and GPS readings can be published through two dedicated nodes, on the */pressure* and */gps* topics respectively.

As MEMS sensors characteristics can vary from device to device, and their orientation depends on the soldering process, we also developed a set of calibration routines, to run on a host computer, which allow to easily calibrate sensor parameters with common tools, i.e., a cube to



(a) PCB layout

(b) Assembled board

Figure 7.7.: Proximity sensors module.

attach the IMU module to, and a rotating plane with known angular velocity, i.e., a turntable [70]. The routines are available as python scripts on the IMU open source repository.

7.5. Proximity sensors

The proximity module can be used to interface common proximity sensors, e.g., infrared and ultrasonic rangefinders, with the R2P architecture. It can be used to sense surrounding objects and avoid collisions with obstacles, a primary functional requirement of most mobile robots.

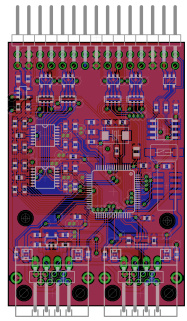
This module is 40×40 mm in size. The layout of the module and an assembled board are visible in Figure 7.7.

7.5.1. Hardware design

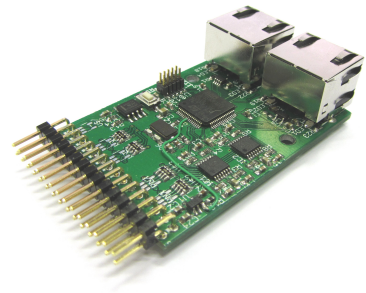
The proximity module has a quite simple hardware design, featuring 4 sensor inputs, with dual connector footprint, i.e., the JST connector employed in common Sharp GP2Y series IR sensors [11], and a standard 0.1 inches pin header. The module is also compatible with 3 wires ultrasonic proximity sensors, such as MaxBotix rangefinders [6]. An on board mosfet allows to power the sensors on demand. Analog sensor values are routed to microcontroller pins which can be configured as inputs of the internal A/D converter, or as capture inputs of an hardware timer. A current limiting resistor and a protection Zener diode are inserted in the signal path to prevent damages to the microcontroller.

7.5.2. Embedded software

The firmware running on this module exposes functions to configure microcontroller peripherals in order to acquire the connected sensors. Analog sensors readings, e.g., from Sharp IR rangers, are acquired through



(a) PCB layout



(b) Assembled board

Figure 7.8.: Input/output module.

the A/D converter; it is possible to calibrate the maximum and minimum values, and to filter readings to compensate for the non linearity of such sensors. Sample curves for common sensors are provided. Sensors with digital outputs, i.e., a PWM signal, are acquired by hardware timers, configured in capture-compare mode; these sensors generally output a value with linear dependency to the measure, and no additional conditioning is needed.

Proximity readings from the 4 sensors are converted to millimeters and published, by default, on the */proximity* topic.

7.6. Input/output

The input/output module (IO) is a generic module featuring several analog and digital inputs and outputs. It can be used to integrate existing devices, e.g., sensors and actuators, into a R2P system, without the need of designing a complete board. Hardware designers can also use the IO module as a test platform while developing a new R2P module, verifying its operation before finalizing the layout. This module is 60×40 mm in size. The layout of the module and an assembled board are visible in Figure 7.8.

7.6.1. Hardware design

The IO module features a variety of connections for both analog and digital signals to interface with external devices. It features up to 8 analog inputs, 2 analog outputs, up to 16 digital inputs/outputs, and a 3.3 V supply output. To increase the number of connections, this module sports a 64 pin version of the STM32 microcontroller from the

high-density family (an STM32F103RET6). All the connections are protected from electrostatic discharge transients and from shorts.

Analog inputs are buffered by operation amplifiers, to prevent damages to the microcontroller, and connected to the internal A/D converter, which features 12-bit resolution and a maximum sample rate of 1 Msps. The 2 analog outputs are based on the 2 D/A converters provided by the STM32, with an additional output buffer capable to source and sink currents up to 50 mA.

Digital inputs and outputs are routed to different microcontroller peripherals, exposing, depending on configuration (i.e., some peripheral pins are mutually exclusive):

- 8 timer channels for PWM output and input, quadrature encoder reading, etc.;
- 3 UART/USART ports, also usable as SPI interfaces
- 1 SPI/I2S interface
- 1 I2C bus
- 1 4-bit SDIO port to interface memory cards

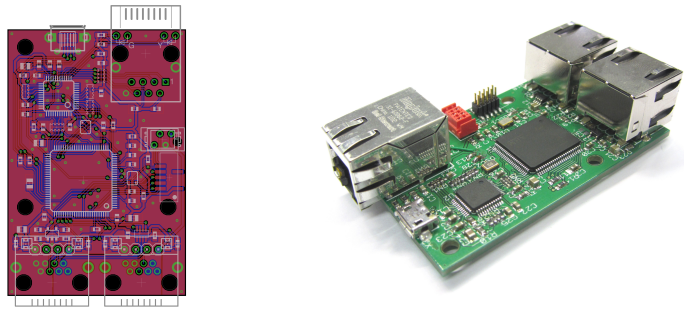
Finally, a dedicated linear regulator provides 3.3 V supply which can be enabled and disabled by software.

7.6.2. Embedded software

The IO module provides access to the peripherals through ChibiOS/RT HAL, but does not run any node by default, being designed for custom applications. Nodes can be easily developed to interface external components and, by publishing and subscribing R2P topics, to integrate additional devices in a R2P network. As use case, we interfaced an Avago ADNS-9500 mouse sensor to the IO module, through the SPI bus. A node configures the sensor to acquire motion values from it, and publishes the readings through the middleware, allowing odometry on systems that are not equipped with encoders. This is only an example of the many possible applications the IO module can be used for.

7.7. Gateway

The gateway module (GW) is used to connect the real-time R2P network to an external computer, by means of USB or Ethernet connection. Thanks to this module, interfacing low-level hardware with high-level



(a) PCB layout

(b) Assembled board

Figure 7.9.: Gateway module.

software is seamless, as it removes any needs of specific adapters, bus controllers, and software drivers. Plugging in a USB cable or an Ethernet cable is sufficient to control a robot from a computer, through the Python port of the R2P middleware or by communicating on standard ROS topics. This module is 60 × 40 mm in size. The layout of the module and an assembled board are visible in Figure 7.9.

7.7.1. Hardware design

This module sports a more powerful microcontroller with respect to other R2P modules, as it needs to handle the TCP/IP stack for Ethernet communication as well as the µROSnode native ROS client. We used a STM32F407 microcontroller by ST Microelectronics, with 192 Kbytes of RAM and 1 Mbyte of Flash memory, and featuring an ARM Cortex-M4 core clocked at 168 MHz. In addition to the standard CAN bus connectors, the GW module exposes a USB micro-B connector, and a 100 Mb/s Ethernet port with integrated magnetics. The Ethernet connection is driven by a DP83848 PHY by Texas Instruments.

7.7.2. Embedded software

The firmware that runs on the GW board provides all the functionalities to interface R2P modules with an host computer, both through USB or Ethernet connection. USB communication is provided by means of a USB CDC serial connection, i.e., a *virtual* UART interface, which is handled out of the box by any modern operating system. The USB driver is part of ChibiOS/RT operating system, and can be seen as a standard serial connection also on the microcontroller side. By default, the gateway instantiates a DebugTransport on the USB serial port (see

Section 5.3.7), so that topics on the real-time CAN network can be subscribed through the USB connection, or, in the other way, messages can be published on existing topics from the connected computer.

Ethernet connectivity is handled by the LWIP library, an open source, lightweight, implementation of the TCP/IP stack [47]. The default firmware provides two ways to interface through Ethernet: by running a DebugTransport over a TCP connection, or by using the µROSnode native ROS client. The DebugTransport can be used by opening a connection to a TCP port on the GW module address, e.g., with a *telnet* client, and it can be used as it was a serial connection, allowing to remotely inspect and debug the R2P network.

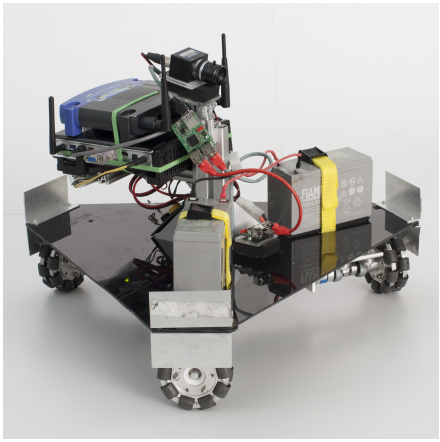
Additionally, thanks to µROSnode, the GW module can be easy configured to expose R2P topics into a ROS network, and vice versa, to provide seamless integration between high-level software deveoped in ROS and the robotic platform. Demo applications have been developed, which translates R2P messages, which are generally more lightweight than ROS counterparts, in standard ROS messages. As an example, data from the IMU module can be formatted as ROS *sensor_msgs/Imu* message and consumed by any ROS node, as shown by the use case presented in Section 8.2.

Chapter 8

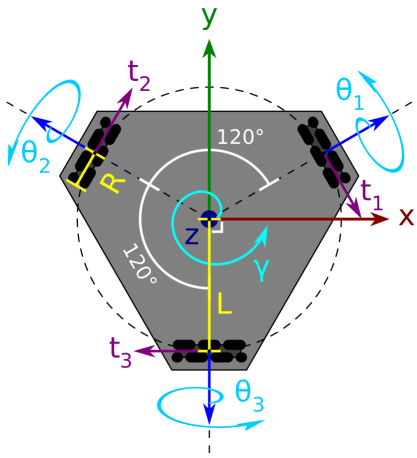
Use cases

We have built some robot platforms exploiting the R2P framework, to verify the several components (i.e., hardware modules, middleware, and real-time communication) in real applications and to test R2P effectiveness in reducing prototyping time. In this chapter, we present 3 of the robots we have built, reviewing their functional requirement, highlighting how they are satisfied by means of R2P components, and describing the low-level control we realized with the R2P architecture. Although these platforms are all mobile wheeled robots, they have fairly different mechanics, and they were built for varied applications, which needed different functional requirements. Here we show that by using the R2P framework the different platforms have been built following the same process, and sharing most of the components. This allowed us to build and control a robot in hours, instead of weeks or even months as we were used to.

In Section 8.1 we describe *Triskar2*, an omnidirectional platform we use in applications focused on interaction between humans and robots; Section 8.2 presents *Robocom*, a heavy duty differential drive robot built for sensor fusion research; finally, Section 8.3 reports the development of the new low level control of tiltOne, the two-wheeled balancing platform for service robotics applications we already introduced in Chapter 2.



(a) The platform



(b) Kinematics

Figure 8.1.: The triskar2 omnidirectional robot: a picture of the realized platform (a) and its kinematics diagram (b).

8.1. Triskar2 omnidirectional platform

Triskar2 is the first robot we equipped with R2P modules. We needed an holonomic omnidirectional platform to develop some human-robot interaction applications, like robotic games. The main goal of this platform is to provide a simple to use interface in ROS, so that our computer engineering students can easily develop the high-level software, and implement their games, without getting stuck on problems related to mechanics, electronics, and low-level control. We also wanted to embed kinematic calculations in the low-level control software, to allow developers simply issue motion commands in terms of forward, lateral, and angular speeds. Finally, for safer operations, we wanted obstacle avoidance within the low-level control loop, to prevent the robot from hitting things, or people, during tests, if wrong commands were issued.

The platform requirements needed for the Triskar2 robot are summarized in the following:

- follow motion setpoints;
- compute odometry;
- provide low-level obstacle avoidance;
- publish and subscribe standard ROS topics.

8.1.1. Platform description

Triskar2 is a three-wheeled robot, with wheels spread at 120 deg on a circle, having their axes crossing at the center of the body frame; the actual platform is shown in Figure 8.1a. The frame is built out of standard aluminium profiles, which we cut in our lab and assembled using the standard joints provided by the manufacturer. Wheels are of omni type, with several rollers on their perimeter, to give traction on the component orthogonal to their axis while allowing free movements in the axis direction, thus enabling movements with three degrees of freedom. Each wheel is actuated by a 70 W DC motor by Maxon Motor [7], with a planetary gear head providing 14 : 1 reduction and an incremental optical encoder with 2000 ticks/turn mounted on the rear shaft. The robot weights about 20 Kg and can move at a maximum speed of about 2 m/s.

The kinematics diagram of the robot is depicted in Figure 8.1b. The body frame, which has origin in the middle of the robot, is aligned so that the y axis points forwards, the x axis rightwards, and the z axis upwards. The angle between the x axes of body and world frames, with respect to the z axis, is named γ. Wheel angular positions are θ_1, θ_2, and θ_3. Having fixed the angle on which the wheels are distributed, the only parameters used by the kinematic model become the wheel radius, R, and the distance of the wheel origin from the body frame origin, L. Forward kinematics equations are reported in 8.1, while inverse kinematics equation are presented in 8.2.

$$\begin{cases} \dot{x} = & R\cos\left(\frac{\pi}{3}\right) & \dot{\theta}_1 & +R\cos\left(\frac{\pi}{3}\right) & \dot{\theta}_2 & -R & \dot{\theta}_3 \\ \dot{y} = & -R\cos\left(\frac{\pi}{6}\right) & \dot{\theta}_1 & +R\cos\left(\frac{\pi}{6}\right) & \dot{\theta}_2 & & \\ \dot{\gamma} = \frac{1}{L}(& -R & \dot{\theta}_1 & -R & \dot{\theta}_2 & -R & \dot{\theta}_3) \end{cases} \tag{8.1}$$

$$\begin{cases} \dot{\theta}_1 = \frac{1}{R}(& \cos\left(\frac{\pi}{3}\right) & \dot{x} & -\cos\left(\frac{\pi}{6}\right) & \dot{y} & -L & \dot{\gamma}) \\ \dot{\theta}_2 = \frac{1}{R}(& \cos\left(\frac{\pi}{3}\right) & \dot{x} & +\cos\left(\frac{\pi}{6}\right) & \dot{y} & -L & \dot{\gamma}) \\ \dot{\theta}_3 = \frac{1}{R}(& -1 & \dot{x} & & & -L & \dot{\gamma}) \end{cases} \tag{8.2}$$

8.1.2. Control architecture

Triskar2 sports several R2P hardware modules, each running various nodes communicating through the middleware. In particular, the following modules have been used:

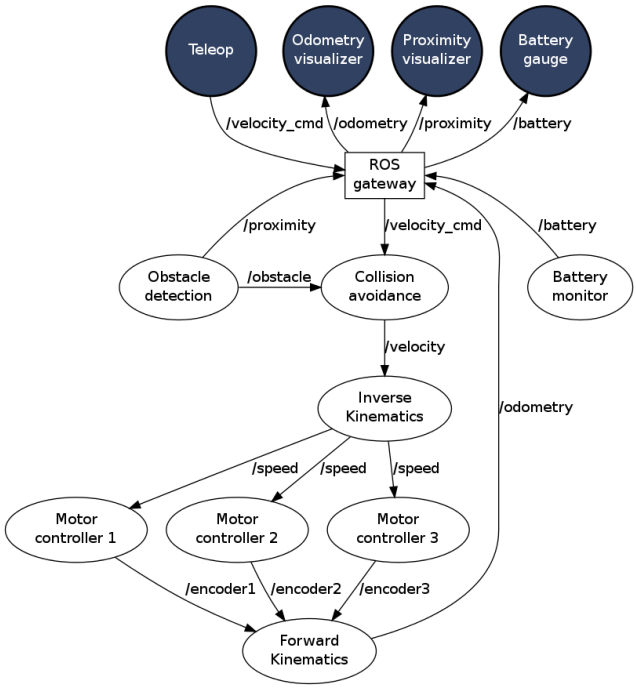


Figure 8.2.: Triskar2 software architecture: white ellipses are embedded nodes running on R2P modules, while blue circles are ROS nodes on a host computer.

- 3 DC motor controllers to drive the 3 wheels;
- 2 proximity modules for sensing obstacles, with 4 IR sensors on the front of the robot and 4 on its rear;
- 1 GW module to interface the R2P network with ROS;
- 1 PS module to power the R2P bus.

The overall distributed architecture of the software is reported in Figure 8.2. Looking at the node graph, people familiar with ROS may recognize a very similar approach to implement a robotic application, although all the low-level control software runs on tiny, low-cost, microcontrollers. The needed functionalities are implemented by distributed nodes, which can be easily modified, replaced or shared through different projects thanks to the loosely-coupled design. Nodes are deployed on the hardware modules as sketched in Figure 8.3.

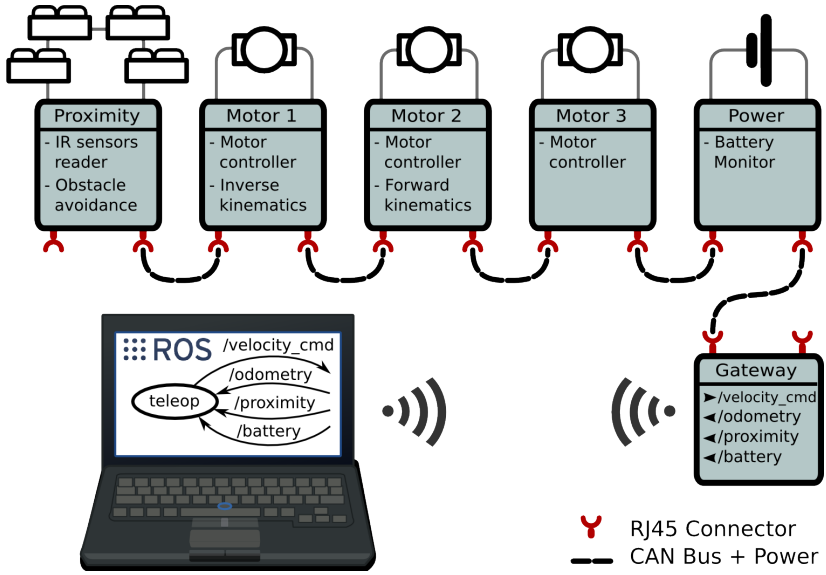


Figure 8.3.: Triskar2 architecture, with the R2P modules we used, the corresponding control nodes, and the ROS application.

Velocity setpoints are published by a generic ROS application and subscribed by the GW module through µROSnode and forwarded in the R2P network on the */velocity_cmd* topic. Here they are subscribed by an obstacle avoidance node, running on the proximity module, which filters them whenever an incoming obstacle is detected: if the issued motion command is in a direction where an obstacle has been detected, it is overridden to avoid contact. Then, another node, running on one of the motor boards, is in charge of computing the inverse kinematics to get the actual wheel speed setpoints, published on the */speed* topic and subscribed by the nodes responsible of running the PID controllers on the motor boards. Encoder readings are published by each motor board and subscribed by a node applying the forward kinematic model to estimate robot movements, which are published on the */odometry* topic, subscribed by the GW module and routed to the computer as standard ROS messages. The gateway also forwards proximity readings, and the estimated remaining battery life which is monitored by an additional node running on the PS module.

Triskar2 has an on board computer, to accomplish computational intensive tasks, like vision and path planning, and to run ROS; an alternative setup has the R2P gateway connected to a WiFi router, with the high-level control software running on an external computer.

```
msg_t inversek_node(void *arg) {
  Node node("inversek");
  Publisher<Speed3Msg> speed_pub;
  Subscriber<VelocityMsg, 5> velocity_sub;
  Speed3Msg *speedmp;
  VelocityMsg *velmp;

  node.advertise(speed_pub, "speed");
  node.subscribe(velocity_sub, "velocity");

  while (r2p::ok()) {
    if(sub.fetch(velmp)) {
      if ((speedmp = pub.alloc()) == NULL) {
        sub.release(velmp);
        continue;
      }

      speedmp.wheel1 = (1/_R) * (_COS60 * velmp->forward
        - _COS30 * velmp->lateral - _L * velmp->angular);
      speedmp.wheel2 = (1/_R) * (_COS60 * velmp->forward
        + _COS30 * velmp->lateral - _L * velmp->angular);
      speedmp.wheel3 = (1/_R) * (-velmp->forward
        - _L * velmp->angular);

      pub.publish(speedmp);
    }
  }
}
```

Listing 8.1: The Triskar2 inverse kinematics node source code.

8.1.3. Discussion

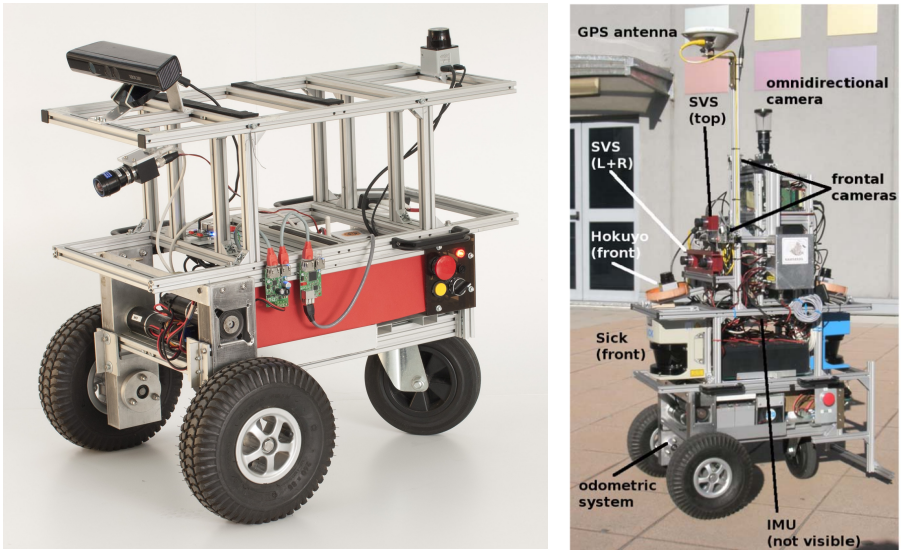
Triskar2 was not the first omnidirectional robot we developed, as we used to participate to the RoboCup Middle Size League (MSL) some years ago, and Triskar2 mechanics is very similar, although it is bigger in size, to common RoboCup MSL robots. Having always built our RoboCup platforms out of custom components in the past, and having experienced all the related problems, we could appreciate, while assembling Triskar2 electronics, the plug&play nature of the R2P framework, which saved us a lot of time. Indeed, it took only a couple of hours to attach the modules on the robot frame, wire them, and have the wheels spinning, encoder readings working, and synchronization between the 3 distributed motor drivers.

The only component we had to implement from scratch was the node to apply the inverse kinematics and compute required wheel speeds. Knowing the kinematic model formulas, it took a few minutes to write

the node, which is shown in Listing 8.1: a publisher and a subscriber are instantiated, then, for each velocity command received, a message with the corresponding wheels setpoints is filled and published back. All motor boards will receive the message, as they subscribe the */velocity* topic, and apply the respective setpoints by running a PID controller; if the mechanics changes, and thus also the kinematic model, only the node calculating it needs to be updated. Now, as kinematics is calculated within a node, this becomes an additional ready to use component we may reuse in several other projects.

The gateway, thanks to µROSnode, allowed us to rapidly interface the robot with ROS, without needing to write specific drivers and to employ adapters: we ran the *uroscfg* script (see Section 6.3.4) to generate handlers for incoming ROS messages, and declared the corresponding R2P publishers to forward messages to the hardware modules. To expose R2P data to ROS, we proceeded in the same way, resulting in a native ROS interface to drive the robot as well as native ROS messages to get feedback from the platform.

Although while building Triskar2 we also found, and had to fix, some bugs in the R2P framework, as this was the very first real robot built out of R2P modules, we can claim that, using off-the-shelf R2P components, we can build a platform like this, and have it moving by issuing commands from a ROS node, in less than one day. Triskar2 was initially tested by driving it from a remote node with a gaming-like interface, i.e., accepting inputs from mouse and keyboard. The platform has also been already used in a hide-and-seek robotic game involving a remote controlled AR.drone quadricopter [33] and Triskar2 moving autonomously; all the game logic was written in ROS, relying on the hardware platform controlled by R2P.



(a) The platform (b) Loaded of sensors

Figure 8.4.: Robocom, a differential drive robot for research in sensor fusion: the new platform (a) and a setup with several sensors attached during a Rawseeds acquisition (b).

8.2. Robocom differential drive robot

Robocom is an heavy duty, differential drive, indoor robot, which can carry a high payload. This platform has been built for sensor fusion experiments, where sensory data from disparate sources are combined to achieve a better estimation of the observed entities. For this reason, we needed a mobile, modular, platform which could be easily equipped with a variety of sensors. Robocom has been built some years ago within the Rawseeds project [40], which was focused on providing benchmarking tools for mobile robotic systems by publishing high-quality multisensor datasets with corresponding ground truth. Recently, its custom hardware has been replaced by R2P modules to better satisfy its requirements while improving reliability.

The main requirement for any sensor-fusion application is to provide reliable data from sensors, with accurate timestamps, to test filtering algorithms on. In particular, we need, from the platform, encoder readings and inertial measurements. Using the components commonly used in research robots, i.e., industrial or hobby market motor drivers and sensors, precise timestamping is not trivial to satisfy. Indeed, the several devices would be connected to a on board computer, possibly dealing with dif-

ferent buses and protocols exploited by different vendors, and data is timestamped at software level. This approach does not allow for precise timing, as delays introduced by the different communication channels is not taken into account, and, above all, software running on a computer is prone to latency and jitter, if particular care is not paid, e.g., by running a real-time operating system. Timestamp misalignments can be estimated and corrected, but this can be done only by off-line analysis (e.g., by looking at cross-correlation between sensor readings), and it may even falsify acquired data. Obviously, the robot also needs to move around, and we also need interface to ROS as it is the framework we adopt to develop sensor fusion algorithms.

The platform requirements we identified for Robocom are:

- follow motion setpoints;
- provide encoder readings;
- provide inertial measurement readings;
- accurately timestamp sensor data;
- publish and subscribe standard ROS topics.

8.2.1. Platform description

Robocom is a differential drive robot, with two actuated wheels on the front and an additional caster wheel on the rear; the complete platform is shown in Figure 8.4a. As for the previous use case, the frame is made of standard aluminium profiles, which can be easily assembled to realize the required structure. Robocom is actually composed by two frames: the lower one is the mobile base, which sports all the hardware needed to actuate the robot, the batteries, and a computer; the upper part is a detachable structure to mount the sensors, as shown in Figure 8.4b. This allows to easy swap a set of sensors with another, or to use the mobile base in different projects, by simply removing the 4 screws that connect the two parts.

Motion is provided by two high power 150 W DC motor by Maxon Motor [7], with ceramic gear heads ($74:1$ reduction ratio), and optical encoders with a resolution of 2000 ticks/turn. To provide mechanical decoupling, the wheels are driven by a synchronous belt, so that hits on the wheels are prevented from damaging the motors; each motor and wheel pair is assembled in an aluminium profile and can be substituted in case of need. Front wheels are 24 cm in diameter, and offer an high grip also with a full payload, which can be as high as 80 Kg.

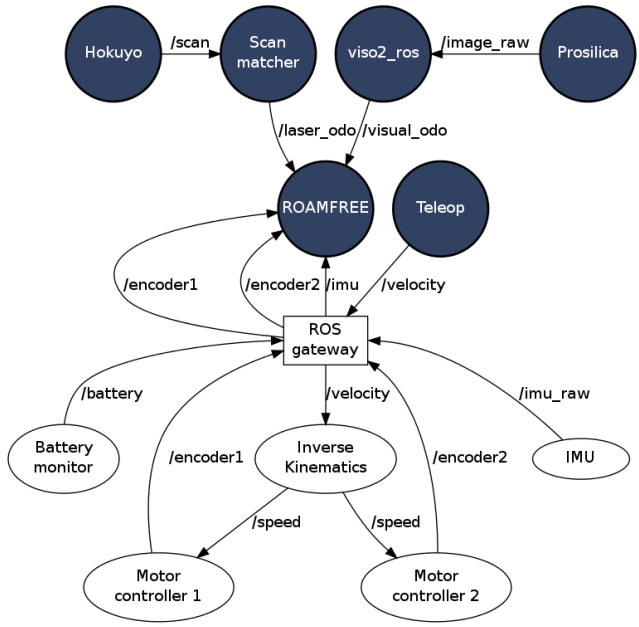


Figure 8.5.: Robocom software architecture: white ellipses are embedded nodes running on R2P modules, while blue circles are ROS nodes on a host computer.

A computer is mounted on the lower frame, running ROS, to provide the computational power needed to acquire data from high bandwidth sensors, e.g., the several cameras used to collect visual information. The lower frame also accommodates 6 lead-acid batteries, for a total 24 V 21 Ah capacity, to power the motors and supply the equipment.

8.2.2. Control architecture

Robocom custom control electronics has been completely replaced by R2P modules, which feature much better specifications (i.e., the maximum continuous current of R2P motor controllers is 20 A with respect to the 3 A supported by the old drivers), in a smaller footprint (i.e., the old drivers were inside a case with $300 \times 100 \times 100$ mm size, while the new drivers are only $60 \times 40 \times 15$ mm each). We also replaced the original commercial IMU with the R2P IMU module.

Summarizing, the following modules have been used on Robocom:

- 2 DC motor controllers to actuate the wheels;
- 1 IMU module to collect inertial measurement;

- 1 GW module to interface the R2P network with ROS;
- 1 PS module to power the R2P bus.

Figure 8.5 shows the distributed embedded control software of the robot, in the lower part, and the ROS nodes used in one of the application developed with Robocom, in the upper part. Motion setpoints are produced by the *Teleop* ROS node, which receives commands by a human using a traditional joypad. The GW module subscribes the ROS */velocity* topic and forwards messages to the corresponding R2P topic, which is then subscribed by the node responsible of running the inverse kinematics model. Motor controller nodes running on the 2 DCM modules subscribe the computed speeds and, by running the on board PID controller, drive the motors following the received setpoints.

DCM modules are also in charge of publishing readings from the motor encoders, with precise timing. For this purpose, encoder data sampling is synchronized to the reception of RTCAN sync message, and messages are timestamped using the global time provided by RTCAN. This results in almost simultaneous reading of the two encoders (some latency, and thus jitter, may be introduced depending on threads running on each module, and their priorities), and data is timestamped with 1 µs resolution and an accuracy comparable to RTCAN time-triggered messages jitter (i.e., ± 3 µs). Inertial measurement acquired by the IMU module are also timestamped using RTCAN global time, and readings by each sensor are published on the */imu_raw* topic. With the receiver connected to the IMU, also data from GPS is timestamped with the same time base.

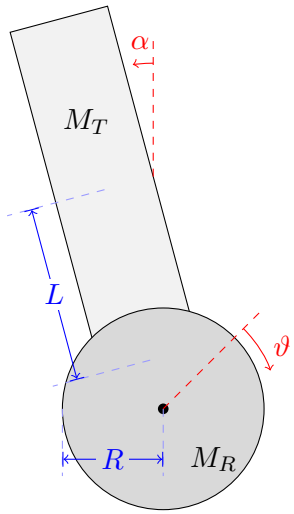
The GW module, while forwarding R2P messages to the ROS network, fills the ROS standard header (i.e., *std_msgs/Header*) with timestamps provided by R2P, so that all data produced by R2P modules have precise timing also within ROS. To provide online synchronization between ROS and R2P clocks, the GW module can read the headers of messages coming from ROS (e.g., on the */velocity* topic), and track the offset between the two time bases. This only allows for a coarse synchronization, but it is still better than timestamping data at reception time on the computer. A much accurate approach whould be having a PTP client running on the GW module, so that the clocks are aligned with very high precision. Currently, we are working to port the PPSi PTP client [9], an enhanced and lightweight PTP implementation developed at CERN, to the R2P framework, aiming at providing clock synchronization as feature of the GW module.

8.2.3. Discussion

The issues showed by the old Robocom platform, within the Rawseeds project, were mainly due to difficulties in aligning sensor readings, as the robot sported several devices with no common time base and drifts up to 150 ms were noticed [10]. For example, it used a custom motor driver board, which also provided encoder readings, connected by a serial interface, a commercial IMU was used to collect inertial measurement, with a USB connection to the on board computer, and a GPS receiver, producing position measures as well as the reference time, connected via USB too. Some of the cameras were connected through Ethernet, with a Precision Time Protocol (PTP) client to align their clocks, while others had Firewire interfaces and required timestamping on the computer. This resulted in misalignment in sensor timestamps and, moreover, the different clocks showed variable drift, worsening the accuracy over time. A lot of work has been dedicated to realign sensor data, and cross-correlation analysis was used to estimate respective offsets and drift ratios.

By using R2P modules, most of these problems do not occur, as data is timestamped at the source of information, and using a consistent time base over the whole network. This allows to perform research relying on high quality data from the platform, instead of trying to correct misaligned data with the risk of introducing additional errors and wasting time and efforts. Robocom has been used within the ROAMFREE (Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors) project [45], which aims at providing a comprehensive, high-performance, easy-to-use software library for sensor fusion and robust odometry.

(a) The platform



(b) Notation used to describe the motion model

Figure 8.6.: tiltOne omnidirectional robot: a picture of the realized platform (a) and its kinematics diagram (b).

8.3. tiltOne balancing robot

Recently, we developed a second version of tiltOne, the two-wheeled balancing robot introduced in Section 2.3. This platform has been designed for service applications which need a tall robot (e.g., to interact with humans) while keeping a small footprint to move in complex environments such as apartments and offices. Balancing robots satisfy these needs: the footprint is reduced with respect to platforms with more wheels, they can easily move in narrow spaces, being able to turn in place, and they can carry loads much more efficiently with respect to legged solutions. The main drawback of these robots is that they are inherently unstable, needing to be actively balanced in order to remain upright.

The original version of tiltOne showed some issues mainly related to its reliability, as it was built by assembling hobby market components, and it was hard to upgrade with additional features due to the centralized, ad-hoc, architecture. For this reason, we updated the robot with R2P modules, aiming at satisfying with off-the-shelf components the following

platform requirements:

- keep balance;
- follow motion setpoints;
- provide odometry;
- publish and subscribe standard ROS topics.

8.3.1. Platform description

The new version of tiltOne shares only the motors and the wheels with the original design, while the frame has been rebuilt to improve modularity and to allow easy implementation of different applications with the same platform; the current robot is shown in Figure 8.6a. The aluminium frame is composed by two parts: the vertical shaft, to which application-dependent devices can be attached, and the base, housing all the electronic components, the motors, and a computer. The shaft and the base can be separated in few minutes, by removing four screws and disconnecting the cables providing power and communication. A battery holder is attached to the frame, at an adjustable height, to better distribute the weights and allow for precise balance and motion control.

The robot sports two modified bicycle wheels, actuated by two 150 W DC motor by Maxon Motor [7], with a 26 : 1 reduction ratio, and optical encoders providing 2000 ticks/turn. A synchronous belt transmission is used to decouple the wheels from the motors, so that motor axis are not directly loaded with robot and payload weight; the pulleys introduce an additional 4 : 1 reduction to the drive path. Motors parameters have been defined to allow the robot carry a quite high payload (up to 50 kg) travelling at speeds of about 1 m/s.

Balancing robots are similar to classic inverted pendulum on a cart systems, with an additional constraint as the pivot point of the pole is not free. The motion model of a balancing robot can be obtained by decomposing the forces acting on the system in their horizontal and vertical components. Equations 8.3 report a simplified motion model, where M_R and M_T are the masses of the wheels and of the frame, θ represents the wheel rotation, α is the angle of the frame with respect to the equilibrium position, L is the distance between the frame center of mass and the wheel axis, and R is the wheel radius, as reported in Figure 8.6b. Additionally, J_T represents the moment of inertia of the frame, g the gravity acceleration and C the torque applied to the wheels.

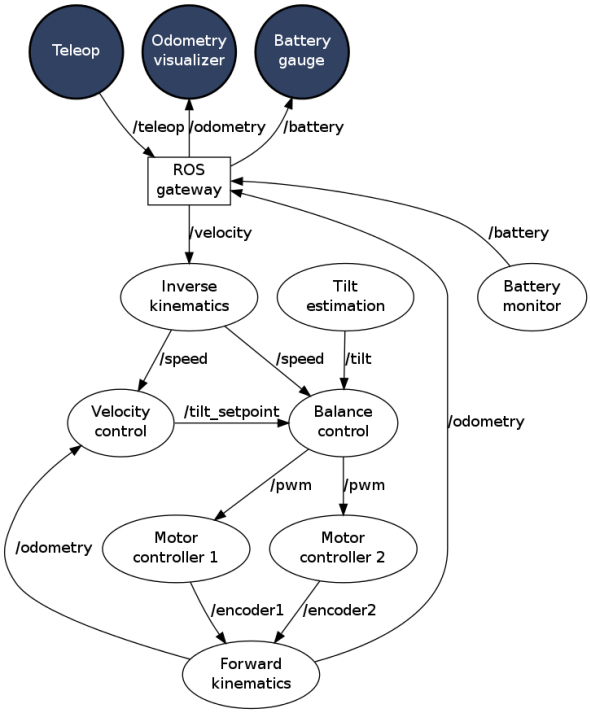


Figure 8.7.: tiltOne software architecture: white ellipses are embedded nodes running on R2P modules, blue circles are ROS nodes on a host computer.

$$\begin{cases} (M_R + M_T)\ddot{\theta}R - M_T l\dot{\alpha}^2 \sin\alpha + M_T l\ddot{\alpha}\cos\alpha &= \frac{C}{R} \\ (J_T + M_T L^2)\ddot{\alpha} - M_T gl \sin\alpha + M_T R\ddot{\theta} l\cos\alpha &= 0 \end{cases} \tag{8.3}$$

8.3.2. Control architecture

To control tiltOne, we need to precisely estimate the current tilt angle, which needs to be controlled to both keep balance and follow motion setpoints. The control action is then applied by driving the two wheels. These requirements can be satisfied by using the following R2P modules, which replaced the original electronics:

- 2 DC motor controllers to actuate the wheels;
- 1 IMU module to estimate the tilt;

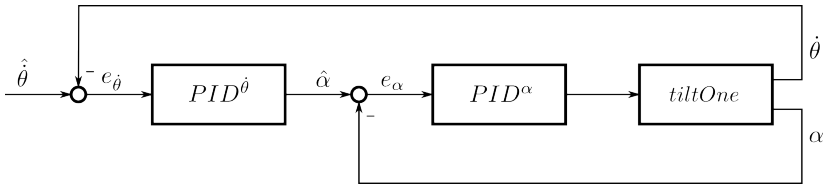


Figure 8.8.: The nested controllers used to keep balance and follow motion setpoints on tiltOne.

- 1 GW module to interface the R2P network with ROS;
- 1 PS module to power the R2P bus.

Figure 8.7 shows the distributed embedded control software of tiltOne and a simple ROS application used to drive the robot. Motion setpoints are issued by a ROS node, as usual, and forwarded on the R2P network by the Gateway module on the */velocity* topic. The *Inverse kinematics* node applies the differential drive equations to the requested setpoint and computes the corresponding wheel speeds, as tiltOne, from a planar motion point of view, is a differential drive platform.

As tiltOne is a balancing robot, we cannot directly drive the wheels at the requested speed, as this would cause the robot to suddenly fall down. To move the robot forward, we first need to tilt in the corresponding direction, then we can apply torque to the wheels to gain speed. When the robot reaches the requested setpoint, it must stand vertical again, which is the configuration to move with constant speed; to stop, it has to tilt backward and then break the wheels. The control law we used to drive tiltOne is realized by two nested PID controllers (see Figure 8.8): the inner controller, PID^{α}, follows the tilt setpoint, while the outer controller $PID^{\dot{\theta}}$ modifies such setpoint to accelerate or decelerate the robot reaching the target speed. These controllers are run inside the corresponding nodes, i.e., *Velocity control*, which subscribes the speed setpoint, compares it with the current speed, and publishes the tilt setpoint with an update rate of 20 Hz, and *Balance control*, which compares the setpoint with the actual tilt measured by the IMU module and computes the actual motor commands with an update rate of 100 Hz. The overall control is than realized by a distributed control loop: the IMU module runs the control algorithms, while the motor drivers actuate the motors and close the loop publishing back the actual wheel speeds.

To let the robot turn, both the control nodes subscribe the motion setpoints: in *Velocity control* we only take into account the motion component $\dot{x}$ (i.e., the forward velocity), and compute the tilt needed to

reach that velocity; in the *Balance control* node we apply a correction to the actions computed by the controller, by adding and subtracting a value proportional to the requested angular velocity (i.e., the ratio of turn) to the two wheels.

Finally, the direct kinematic model is applied to compute odometry, while another node is in charge of monitoring the battery level and communicating it to the user application.

8.3.3. Discussion

The development of the first version of tiltOne took as long as one and half man years, as reported in Section 2.3, and most of the time was spent in looking for the right hardware devices, integrating them, and learning how to correctly implement low-level software on the custom control board we built. Developing the new version of the robot was, with respect to that experience, totally straightforward. It took about a week to build the new frame, assemble the motors and the wheels, attach and wire the battery, and complete the mechanics. Then we mounted the R2P modules we needed, simply connected them, and we wrote the entire low-level control software in only one day. The *Forward kinematics*, *Inverse kinematics*, *Motor controller*, and *Battery monitor* nodes were the same developed to drive Robocom (see Section 8.2.2). Thanks to the loosely coupled software architecture, we were able to reuse the nodes without requiring any modifications. Tilt measurement was already provided by the IMU module, and we simply extracted the pitch component of its attitude estimate.

The only nodes we needed to implement were the ones focused on balance and motion control. For this purpose we used the already implemented PID controllers (i.e., the PID we use for motor control on the DCM module), which we simply instantiated within two nodes and tuned for these specific purpose. First we worked on balance control, starting from the constants we were using on the old version of the robot and tuning them by modifying the configuration of the *Balance control* node, which runs the PID^{α} controller. When the balance control was satisfactory, we followed the same approach with the *Velocity control* node to tune the $PID^{\dot{x}}$ controller.

At that point, we were ready to drive the robot by a ROS node, as it subscribes standard ROS messages; the platform we realized can then be used for a variety of applications, and easily upgraded to satisfy additional requirements. The architecture of the new tiltOne is, actually, the architecture we presented in Figure **??** (see Section 3.3) to show how a robot could be built out of off-the-shelf modular components.

For instance, if we need obstacle avoidance, we can add a couple of R2P Proximity modules and deploy the obstacle avoidance node from the Triskar2 robot. If we are interested in studying alternative control algorithms, we can just replace the controller nodes with the new ones. By comparing the process of building the two versions of tiltOne, and the resulting platforms we obtained, we were finally convinced of the effectiveness of the approach we foster with the R2P framework.

Chapter 9

Conclusions

This Chapter concludes the thesis: in Section 9.1 the results of this work are reviewed, while Section 9.2 outlines future improvements.

9.1. Conclusions

Starting from the observation that robotics still fails to enter the mass market, we identify in the problem of robot prototyping one of the main limiting factors in today's robotic research. Robotic applications need to be prototyped at an early stage of development, as their success depends on the interplay of a variety of components, involving several, different, competencies. Today, the prototyping process can be even more demanding than developing the application itself: no standardized components for the development of generic robots exists, and systems are often implemented from scratch, even if they share most of the requirements. Many unexpected issues often show up at early stages of development, slowing down, or even preventing, the translation of interesting ideas into working systems.

In this thesis, we presented a novel approach to robot prototyping based on modular development techniques which have been recognized effective in many other fields. We extended the modular, component-based, approach to hardware level, developing hardware devices and programming tools focused on robot requirements, to enable massive

hardware/software reuse and fast prototyping. By analyzing common robotic systems, we identified a set of functional requirements that we implemented at hardware level as distributed modules with on-board computation, each focused on a specific task. Modules share a standard physical interface, with a simple connection schema, so that robot prototypes can be developed by selecting the modules that satisfy the identified requirements and by assembling them in a plug-and-play fashion.

Real-time interaction between the different devices is provided by RT-CAN, a novel protocol for the CAN bus we developed starting from the requirements of distributed architectures for robots. It combines the advantages of different approaches to communication scheduling: time-triggered communication, e.g., from control loops, is handled with a pure TDMA approach to guarantee high temporal determinism, while event-triggered communication requests, e.g., from sensors, are handled with EDF scheduling, to be delivered with low latency.

The publish/subscribe middleware extends modular development also to low-level control software, which is implemented by means of distributed, loosely coupled, nodes executed on the hardware modules. In this way software components running on resource-constrained devices can be reused and shared through different projects, with a programming model most robot developers might be familiar with.

To easily integrate robotic platforms with high-level control software, we developed µROSnode, a lightweight ROS client that can run on low-cost microcontrollers, publishing native ROS messages over TCP/IP. µROSnode enables directly interfacing with the low-level control network, without the need for specific adapters and device drivers; in this way, the same interfaces are shared among different robot platforms, allowing to adopt the same ROS software in different projects.

The contributions of this thesis may be used independently and integrated in existing systems, or within the Rapid Robot Prototyping (R2P) project, an open source hardware and software framework aiming at reducing time and effort needed to develop robotic systems. With R2P, common requirements are satisfied by off-the-shelf components, allowing to quickly obtain working robot platforms to evaluate, develop, and benchmark novel ideas and applications. We used R2P to prototype some robots, with different goals and requirements, showing how the proposed architecture can be exploited to develop a variety of applications.

9.2. Future work

R2P is a quite mature project, but it is still under active development and several improvements have been already planned.

First of all, the current set of hardware and software components, although can already help in the development of a number of platforms, should be enlarged to allow more robot topologies to be implemented with the R2P framework. We are already developing some additional modules, e.g., a brushless DC motor driver and a module to drive standard hobby servos, while others have been drafted. For instance, it would be interesting to integrate vision capabilities to allow robot platforms perform simple, real-time, vision tasks without needing an on-board computer, e.g., to perform visual odometry; for this purpose, we are evaluating the integration of the CMUcam [2] embedded vision sensor as a R2P module. Further software components are becoming available while developing new robots and applications, as requirements that are not already satisfied are implemented as middleware nodes, and they can be shared and reused in other projects.

Building new, different, robot platforms is also planned to evaluate and benchmark the framework in a variety of situations with different functional or performance requirements. We plan to realize the low-level control of a multi-rotor flying drone with distributed R2P modules, aiming at developing an open platform for research in control theory, aerial photogrammetry, and digital terrain mapping. Another platform we are evaluating to equip with R2P is an autonomous all-terrain vehicle used for complex autonomous operations [26,27]. Currently, its low-level control is performed with closed, industrial, controllers, which only allow to tune a few parameters; with an open, easy to use, architecture, we aim at significantly improve its performance, being able to implement novel control techniques.

To shorten the learning curve of the framework, we are also developing a graphical IDE, which helps in inspecting the network of modules, updating and deploying software nodes, and configuring their parameters. An interesting scenario we are evaluating is to exploit R2P as low-level interface for model-driven robot development, which is an active research field to structure and formalize the robot development process.

Bibliography

[1] Beckhoff automation gmbh. http://www.beckhoff.com.

[2] CMUcam: Open Source Programmable Embedded Color Vision Sensors. http://www.cmucam.org.

[3] Creative Commons Attribution-ShareAlike 2.0 Generic License. http://creativecommons.org/licenses/by-sa/2.0/.

[4] Darpa Grand Challenge. http://archive.darpa.mil/grandchallenge/.

[5] Gnu tools for arm embedded processors. http://launchpad.net/gcc-arm-embedded.

[6] Maxbotix high performance ultrasonic. http://www.maxbotix.com.

[7] Maxon Motor Ag. http://www.maxonmotor.com/.

[8] Open Source Hardware Association. http://www.oshwa.org.

[9] PPSi - PTP Ported to Silicon. http://www.ohwr.org/projects/ppsi.

[10] Rawseeds WP3 Deliverable D3.2 - Final Data Certification. http://www.rawseeds.org/home/wp-content/uploads/2009/03/rawseeds_d32_v10.pdf.

[11] Sharp gp2y0a21yk distance measurement sensor. http://www.sharpsma.com/webfm_send/1208.

[12] A small nmea0183 parser. https://github.com/jrcutler/NMEA0183.

[13] The BSD 2-Clause License. http://opensource.org/licenses/BSD-2-Clause.

[14] Wire gauge and current limits including skin depth and strength. http://www.powerstream.com/Wire_Size.htm.

[15] Remo Giovanni Abbondandolo. Open source hardware fostering user entrepreneurship: Empirical evidence from Arduino users. Master Thesis, Economics and Management of Innovation and Technology, Bocconi University, 2011.

[16] Amos Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. *Embedded World*, 2004:235–252, 2004.

[17] L. Almeida, P. Pedreiras, and J. A. G. Fonseca. The FTT-CAN protocol: why and how. In *IEEE Transactions on Industrial Electronics*, pages 1189–1201. IEEE Press, 2002.

[18] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[19] Björn Andersson and Eduardo Tovar. The utilization bound of non-preemptive rate-monotonic scheduling in controller area networks is 25%. In *Industrial Embedded Systems, 2009. SIES'09. IEEE International Symposium on*, pages 11–18. IEEE, 2009.

[20] Noriaki Ando, Takashi Suehiro, Kosei Kitagaki, Tetsuo Kotoku, and Woo-Keun Yoon. Rt-middleware: distributed component middleware for rt (robot technology). In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 3933–3938. IEEE, 2005.

[21] Ronald C Arkin. *Behavior-based robotics*. MIT press, 1998.

[22] NC Audsley, Alan Burns, and AJ Wellings. Deadline monotonic scheduling theory and application. *Control Engineering Practice*, 1(1):71–78, 1993.

[23] Neil C Audsley, Alan Burns, Mike F Richardson, and Andy J Wellings. Real-time scheduling: the deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*. Citeseer, 1991.

[24] Massimo Banzi. *Getting started with Arduino*. Make:Books, Sebastopol, CA, 2008.

[25] Massimo Banzi. *Getting Started with arduino*. O'Reilly Media, Inc., 2009.

[26] Luca Bascetta, Davide Cucci, Gianantonio Magnani, Matteo Matteucci, Dinko Osmankovic, and Adnan Tahirovic. Towards the implementation of a mpc-based planner on an autonomous all-terrain vehicle. In *In Proceedings of Workshop on Robot Motion Planning: Online, Reactive, and in Real-time (IEEE/RJS IROS 2012)*, pages 1–7, 2012.

[27] Luca Bascetta, GianAntonio Magnani, Paolo Rocco, and Andrea Maria Zanchettin. Design and implementation of the low-level control system of an all-terrain mobile robot. In *Advanced Robotics, 2009. ICAR 2009. International Conference on*, pages 1–6. IEEE, 2009.

[28] Andreas Birk. Fast robot prototyping with the cubesystem. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 5, pages 5177–5182. IEEE, 2004.

[29] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[30] Andrea Bonarini, Matteo Matteucci, Martino Migliavacca, Roberto Sannino, and Daniele Caltabiano. Modular low-cost robotics: What communication infrastructure? In *In proceedings of 18th World Congress of the*

International Federation of Automatic Control (IFAC), pages 917–922, 2011.

[31] H Boterenbrood. Canopen high-level protocol for can-bus. *NIKEF, Amsterdam*, 20, 2000.

[32] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'reilly, 2008.

[33] Pierre-Jean Bristeau, François Callou, David Vissière, Nicolas Petit, et al. The navigation and control technology inside the ar. drone micro uav. In *18th IFAC World Congress*, pages 1477–1484, 2011.

[34] Rodney Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, 1986.

[35] Hunter C Brown, Ayoung Kim, and Ryan M Eustice. An overview of autonomous underwater vehicle research and testbed at perl. *Marine Technology Society Journal*, 43(2):33–47, 2009.

[36] Nat Brown and Charlie Kindel. Distributed component object model protocol–dcom/1.0. *Online, November*, 1998.

[37] Herman Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation*, pages 2523–2528. IEEE Press, 2001.

[38] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29:5–26, 2005.

[39] Daniele Calisi, Andrea Censi, Luca Iocchi, and Daniele Nardi. Openrdk: a modular framework for robotic software development. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 1872–1877. IEEE, 2008.

[40] Simone Ceriani, Giulio Fontana, Alessandro Giusti, Daniele Marzorati, Matteo Matteucci, Davide Migliore, Davide Rizzi, Domenico G Sorrenti, and Pierluigi Taddei. Rawseeds ground truth collection systems for indoor self-localization and mapping. *Autonomous Robots*, 27(4):353–371, 2009.

[41] Christopher M Cianci, Xavier Raemy, Jim Pugh, and Alcherio Martinoli. Communication in a swarm of miniature robots: The e-puck as an educational tool for swarm robotics. In *Swarm Robotics*, pages 103–115. Springer, 2007.

[42] U.S. Robotics Roadmap Consortium. A roadmap for U.S. robotics: From internet to robotics. http://robotics-vo.us/sites/default/files/2013%20Robotics%20Roadmap-rs.pdf.

[43] J.O. Coronel, F. Blanes, G. Benet, J.E. Simó, P. Pèrez, and M. Albero. CAN-based distributed control architecture using the SCoCAN communication protocol. In *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2005.

[44] Intel Corporation. Intel HEX format specification. http://microsym.com/editor/assets/intelhex.pdf.

[45] Davide Antonio Cucci and Matteo Matteucci. A flexible framework for mobile robot pose estimation and multi-sensor self-calibration. In *Proc. of 11th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*. Springer, 2013.

[46] Giovanni Di-Sirio. ChibiOS/RT Real Time Operating System. http://www.chibios.org, 2007.

[47] Adam Dunkels. Design and implementation of the lwip tcp/ip stack. *Swedish Institute of Computer Science*, 2:77, 2001.

[48] eCosCentric Ltd. eCos. http://ecos.sourceware.org/.

[49] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[50] Patrick Th Eugster and Rachid Guerraoui. Content-based publish/subscribe with structural reflection. In *COOTS*, volume 1, pages 10–10, 2001.

[51] Michael Ferguson. rosserial. http://www.ros.org/wiki/rosserial.

[52] Fischertechnik. Robot construction kits. http://www.fischertechnik.de, 2009.

[53] Stefan Lovgren for National Geographic News. A robot in every home by 2020, south korea says. http://news.nationalgeographic.com/news/2006/09/060906-robots.html.

[54] Lars-Berno Fredriksson. A can kingdom. *Kvaser edition, Revision*, 2:921231, 1995.

[55] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, and Michael Walther. Time triggered communication on can (time triggered can-ttcan). In *7th international CAN Conference*, 2000.

[56] Willow Garage. Overview of the pr2 robot. http://www.willowgarage.com/pages/pr2/overview, 2009.

[57] Willow Garage. The turtlebot. http://www.turtlebot.com, 2009.

[58] Bill Gates. A robot in every home. *Scientific American*, 296(1):58–65, 2007.

[59] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[60] Christopher M Gifford, Russell Webb, James Bley, Daniel Leung, Mark Calnon, Joe Makarewicz, Bryan Banz, and Arvin Agah. Low-cost multi-robot exploration and mapping. In *Technologies for Practical Robot Applications, 2008. TePRA 2008. IEEE International Conference on*, pages 74–79. IEEE, 2008.

[61] Robert Bosch GmbH. Can specification 2.0b. http://www.semiconductors.bosch.de/media/pdf/canliteratur/can2spec.pdf, 1991.

[62] Synapticon GmbH. rosc. https://github.com/synapticon/rosc.

[63] David Gouaillier, Vincent Hugel, Pierre Blazevic, Chris Kilner, Jérôme Monceaux, Pascal Lafourcade, Brice Marnier, Julien Serre, and Bruno Maisonnier. Mechatronic design of nao humanoid. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 769–774. IEEE, 2009.

[64] Juan Carlos Grieco, Manuel Prieto, Manuel Armada, and P Gonzalez de Santos. A six-legged climbing robot for high payloads. In *Control Applications, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 1, pages 446–450. IEEE, 1998.

[65] EU Roadmapping Task Group. Robotics2020 SRA roadmap. http://www.eurobotics-project.eu/ppp-in-robotics/roadmapping/roadmapping.html.

[66] Robert M Harlan, David B Levine, and Shelley McClarigan. The khepera robot and the krobot class: a platform for introducing robotics in the undergraduate curriculum. In *ACM SIGCSE Bulletin*, volume 33, pages 105–109. ACM, 2001.

[67] Florian Hartwich, Bernd Müller, Thomas Führer, Robert Hugel, et al. Timing in the ttcan network. In *Proceedings of the 8th International CAN Conference (iCC02)*, 2002.

[68] G.T. Heineman and W.T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley Professional, 2001.

[69] Michi Henning. A new approach to object-oriented middleware. *Internet Computing, IEEE*, 8(1):66–75, 2004.

[70] Nguyen Thi Thao HO. Activity recognition using smartphone-based sensors. Master thesis, Politecnico di Milano, 2013.

[71] A.S. Huang, E. Olson, and D.C. Moore. LCM: Lightweight communications and marshalling. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4057–4062. IEEE Press, 2010.

[72] Google Inc. Android Accessory Development Kit 2012. http://developer.android.com/tools/adk/adk2.html.

[73] iRobot Corporation. iRobot create. www.irobot.com/create.

[74] iRobot Corporation. iRobot launches new indoor and outdoor home robots. http://www.irobot.com/us/Company/Press_Center/Press_Releases/Press_Release.aspx?n=081412.

[75] Ralph Johnson, Richard Helm, John Vlissides, and Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[76] Hermann Kopetz. Should responsive systems be event-triggered or time-triggered ? *Institute of Electronics, Information, and Communications Engineers Transactions on Information and Systems*, E76-D(11):1325–1332, 1993.

[77] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[78] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132, February 2007.

[79] Ian Lane, Vinay Prasad, Gaurav Sinha, Arlette Umuhoza, Shangyu Luo, Akshay Chandrashekaran, and Antoine Raux. Hritk: the human-robot interaction toolkit rapid development of speech-centric interactive systems in ros. In *NAACL-HLT Workshop on Future Directions and Needs in the Spoken Dialog Community: Tools and Data*, pages 41–44. Association for Computational Linguistics, 2012.

[80] Wolfhard Lawrenz. *CAN system engineering: from theory to practical applications*, volume 1. Springer, 1997.

[81] G. Leen and D. Heffernan. TTCAN: a new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77–94, 2002.

[82] Lego. Mindstorm nxt. http://mindstorms.lego.com, 2009.

[83] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171. IEEE, 1989.

[84] John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, et al. A perception-driven autonomous urban vehicle. *Journal of Field Robotics*, 25(10):727–774, 2008.

[85] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[86] Mohammad Ali Livani and Jörg Kaiser. EDF Consensus on CAN Bus Access for Dynamic Real-Time Applications. In *Proceedings of 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing*, volume 1388, pages 1088–1097. Springer, 1998.

[87] Robert Love. Get on the d-bus. *Linux Journal*, 2005(130):3, 2005.

[88] David V. Lu and William D. Smart. Polonius: A wizard of oz interface for hri experiments. In *Proceeding of the 6th ACM/IEEE international conference on Human-robot interaction*, HRI '11, New York, NY, USA, 2011. ACM.

[89] Janno Lunenburg, Robin Soetens, Ferry Schoenmakers, Paul Metsemakers, René van de Molengraft, and Maarten Steinbuch. Sharing open hardware through rop, the robotic open platform. In *Proceedings of 17th annual RoboCup International Symposium*, Lecture Notes in Artificial Intelligence (LNAI), page In Press. Springer-Verlag, 2013.

[90] Sebastian OH Madgwick, Andrew JL Harrison, and Ravi Vaidyanathan. Estimation of imu and marg orientation using a gradient descent algorithm. In *Rehabilitation Robotics (ICORR), 2011 IEEE International Conference on*, pages 1–7. IEEE, 2011.

[91] Stphane Magnenat, Philippe Rtornaz, Michael Bonani, Valentin Longchamp, and Francesco Mondada. ASEBA: A modular architecture for event-based control of complex robots. In *PIEEE/ASME Transactions on Mechatronics*, pages 321–329, 2011.

[92] Robert Mahony, Tarek Hamel, and J-M Pflimlin. Nonlinear complementary filters on the special orthogonal group. *Automatic Control, IEEE Transactions on*, 53(5):1203–1218, 2008.

[93] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. Orca: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168, 2006.

[94] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE, 2004.

[95] Martino Migliavacca, Andrea Bonarini, and Matteo Matteucci. An Open Source Lightweight Publish/Subscribe Middleware for Distributed Robotics Embedded Components. In *(Submitted to 2014 IEEE International Conference on Robotics and Automation)*.

[96] Martino Migliavacca, Andrea Bonarini, and Matteo Matteucci. RTCAN: a real-time CAN bus protocol for robotic applications. In *Proc. of 11^{th} International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, 2013.

[97] Martino Migliavacca, Andrea Zoppi, Andrea Bonarini, and Matteo Matteucci. uROSnode: a lightweight ANSI C ROS client for microcontrollers. In *(Submitted to 2014 IEEE International Conference on Robotics and Automation)*, 2014.

[98] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stephane Magnenat, Jean-Christophe Zufferey, Dario Floreano, and Alcherio Martinoli. The e-puck, a robot designed for education in engineering. In *Proceedings of the 9th conference on autonomous robot systems and competitions*, volume 1, pages 59–65, 2009.

[99] Thomas J Mowbray and Ron Zahavi. *The essential CORBA: systems integration using distributed objects*. John Wiley & Sons, Inc., 1995.

[100] Moreno Muffatto. Introducing a platform strategy in product development. *International Journal of Production Economics*, 6061:145–153, 1999.

[101] Marco Di Natale. Scheduling the can bus with earliest deadline techniques. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 259–268, 2000.

[102] Mary Lou Nohr. *UNIX System V: understanding ELF object files and debugging tools*. Prentice Hall PTR, 1993.

[103] Dan Noonen, Stuart Siegel, and Pat Maloney. DeviceNet Application Protocol. In *1st International CAN Conference*, 1994.

[104] Donald A Norman. *The design of everyday things*. Basic books, 2002.

[105] Roman Obermaisser. *Event-triggered and time-triggered control paradigms*, volume 22 of *Real-Time Systems*. Springer, 2004.

[106] Takeshi Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *Real-Time Computing Systems and Applications, 1995. Proceedings., Second International Workshop on*, pages 21–25. IEEE, 1995.

[107] Sung-Heun Oh and Seung-Min Yang. A modified least-laxity-first scheduling algorithm for real-time tasks. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, pages 31–36. IEEE, 1998.

[108] Hyun Keun Park, Hyun Seok Hong, Han Jo Kwon, and Myung Jin Chung. A nursing robot system for the elderly and the disabled. *International Journal of Human-friendly Welfare Robotic Systems (HWRS)*, 2(4):11–16, 2001.

[109] Paulo Pedreiras and Luis Almeida. Combining event-triggered and time-triggered traffic in FTT-CAN: Analysis of the asynchronous messaging system. In *IEEE International Workshop on Factory Communication Systems*, pages 67–75, 2000.

[110] P. Pèrez, G. Benet, F. Blanes, and J. Simó. Communication jitter influence on control loops using protocols for distributed real-time systems on CAN bus. In *Proceedings of 5th IFAC International symposium SICICA*, pages 237–243. Springer, 2003.

[111] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*. IEEE Press, 2009.

[112] Rapid Robot Prototyping. R2P repository. https://github.com/openrobots-dev.

[113] Real Time Engineers Ltd. FreeRTOS. http://www.freertos.org/.

[114] Charles Rich, Brett Ponsler, Aaron Holroyd, and Candace L Sidner. Recognizing engagement in human-robot interaction. In *Human-Robot Interaction (HRI), 2010 5th ACM/IEEE International Conference on*, pages 375–382. IEEE, 2010.

[115] Modular Robotics. Cublets. http://www.modrobotics.com/cubelets.

[116] ROS. Turtlesim. http://wiki.ros.org/turtlesim.

[117] ROS. XML-RPC Slave API. http://wiki.ros.org/ROS/Slave_API.

[118] ROS-Industrial Consortium. ROS-Industrial. http://rosindustrial.org/.

[119] Daniel J Rosenkrantz and Richard Edwin Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226–256, 1970.

[120] Michael Schulze and Sebastian Zug. A Middleware based Framework for Multi-Robot Development. In *Proceedings of the 3rd IEEE European Conference on Smart Sensing and Context (EuroSSC)*, pages 29–31, Zurich, Switzerland, October 29-31 2008.

[121] Eric Schweikardt and Mark D Gross. roblocks: a robotic construction kit for mathematics and science education. In *Proceedings of the 8th international conference on Multimodal interfaces*, pages 72–75. ACM, 2006.

[122] Lui Sha, Ragunathan Rajkumar, and Shirish S Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, 1994.

[123] Bih-Yaw Shih, Chen-Yuan Chen, Chung-Wei Chen, and I Hsin. Using lego nxt to explore scientific literacy in disaster prevention and rescue systems. *Natural hazards*, 64(1):153–171, 2012.

[124] A Sicheneder, A Bender, E Fuchs, R Mandl, and B Sick. A framework for the graphical specification and execution of complex signal processing applications. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1757–1760. IEEE, 1998.

[125] SICK AG. Sick lms100. http://www.sick.com/.

[126] Marco Spuri and Giorgio C Buttazzo. Efficient aperiodic service under earliest deadline scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11. IEEE, 1994.

[127] Daniel Stonier and Morten Kjærgaard. eros. http://www.ros.org/wiki/eros.

[128] Beng Hang Tay and Akkihebbal L Ananda. A survey of remote procedure calls. *ACM SIGOPS Operating Systems Review*, 24(3):68–79, 1990.

[129] Seth Teller, Matthew R Walter, Matthew Antone, Andrew Correa, Randall Davis, Luke Fletcher, Emilio Frazzoli, Jim Glass, Jonathan P How, Albert S Huang, et al. A voice-commandable robotic forklift working alongside humans in minimally-prepared outdoor environments. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 526–533. IEEE, 2010.

[130] Clive Thompson. Build it. share it. profit. can open source hardware work? *Wired Magazine*, 16.11, 2008.

[131] Hans Utz, Stefan Sablatnog, Stefan Enderle, and Gerhard Kraetzschmar. Miro-middleware for mobile robot applications. *Robotics and Automation, IEEE Transactions on*, 18(4):493–497, 2002.

[132] Koen Vervloesem. Control your linux desktop with d-bus. *Linux Journal*, 2010(199):3, 2010.

[133] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[134] FENG Wei and YANG Yang. A research and realization on the gait of the fischertechnik hexapod bionitc robot [j]. *Machine Design and Research*, 3:010, 2005.

[135] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, pages 1–116. Springer, 1995.

[136] Thomas Wisspeintner, Walter Nowak, and Ansgar Bredenfeld. Volksbot–a flexible component-based mobile robot system. In *RoboCup 2005: Robot Soccer World Cup IX*, pages 716–723. Springer, 2006.

[137] Lu Xia, Chia-Chih Chen, and JK Aggarwal. Human detection using depth information by kinect. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 15–22. IEEE, 2011.

[138] Joseph Yiu. *The definitive guide to the ARM Cortex-M3*. Access Online via Elsevier, 2009.

[139] Holger Zeltwanger. Time-Triggered communication on CAN. In *SAE 2002 World Congress & Exhibition*, pages 01–0437, 2002.

Appendices

Appendix A

R2P module schematics

In this appendix we report the schematics of the R2P modules presented in Chapter 7. Board designs are open source, distributed under the Creative Commons Attribution-ShareAlike 2.0 (CC BY-SA 2.0) license [3], and available on the R2P repository [112].

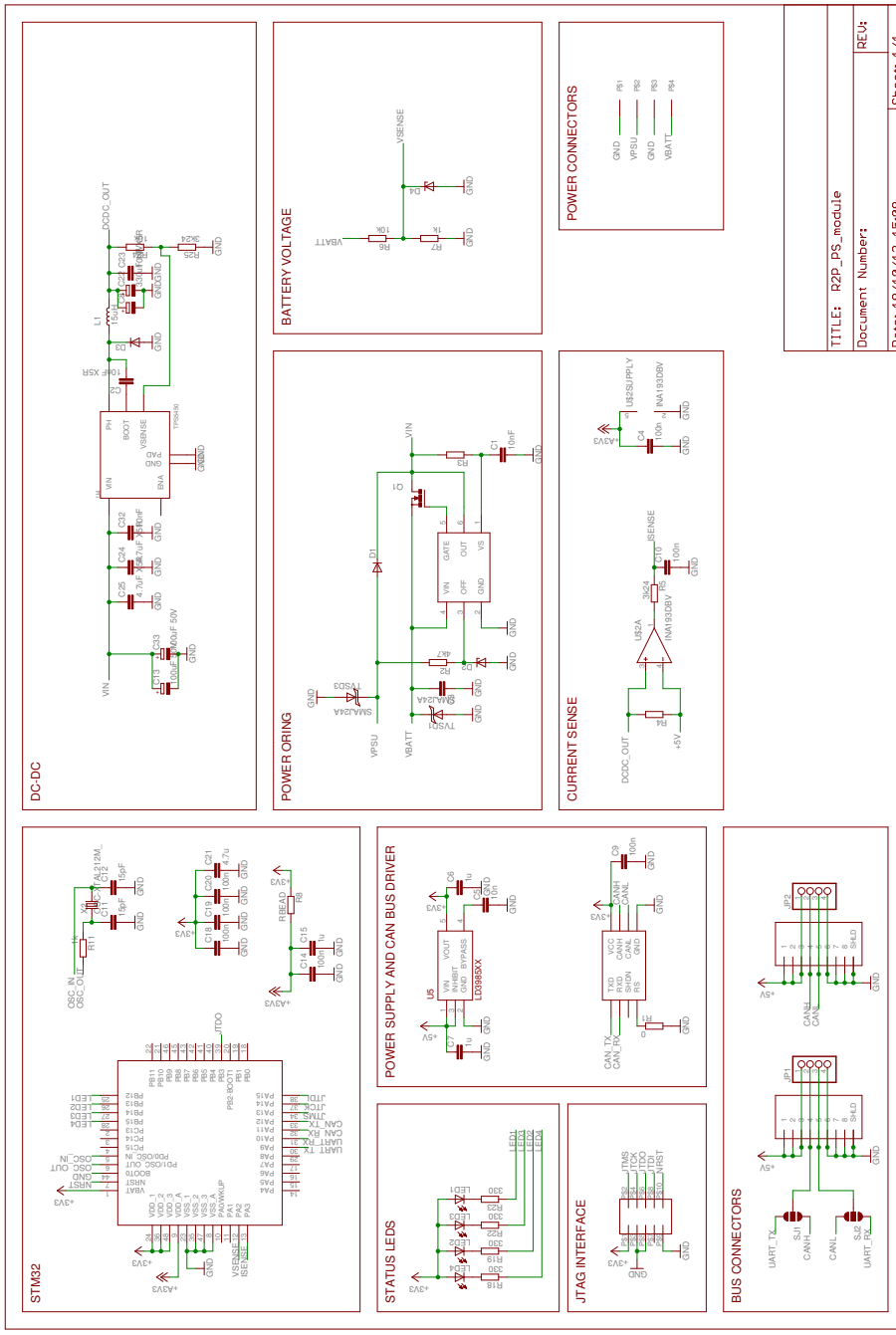


Figure A.1.: R2P power supply module

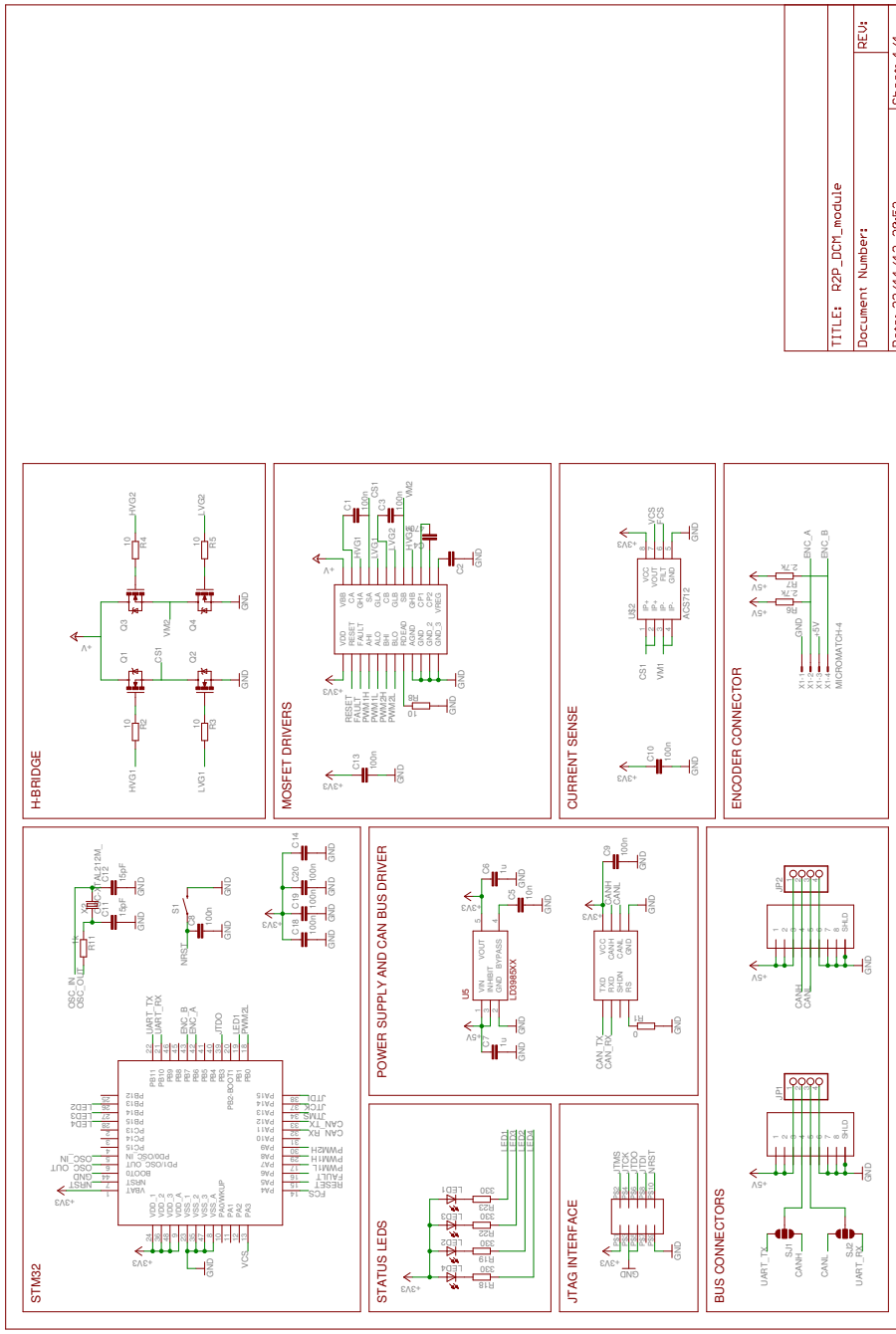


Figure A.2.: R2P DC motor controller module

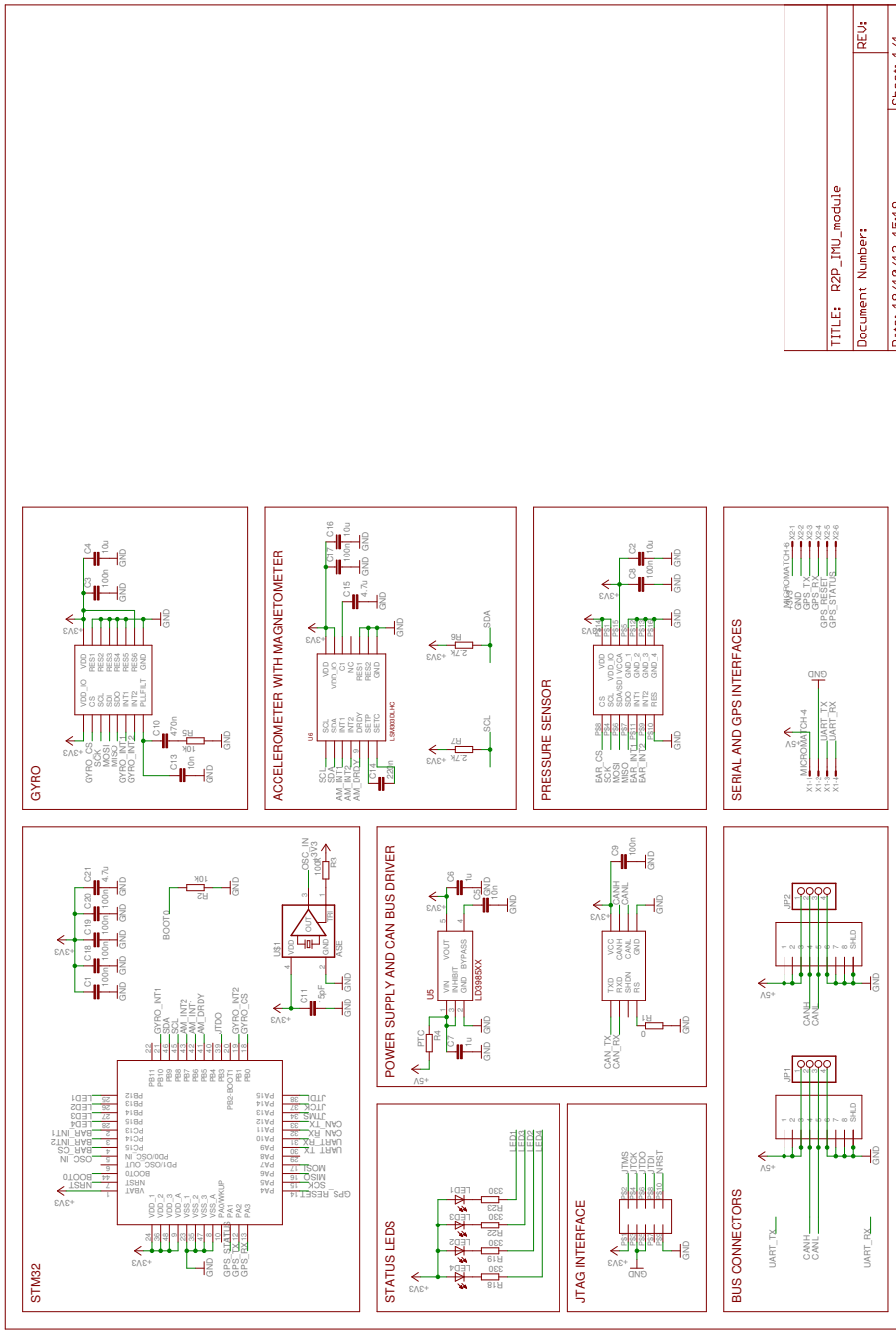


Figure A.3.: R2P inertial measurement unit module

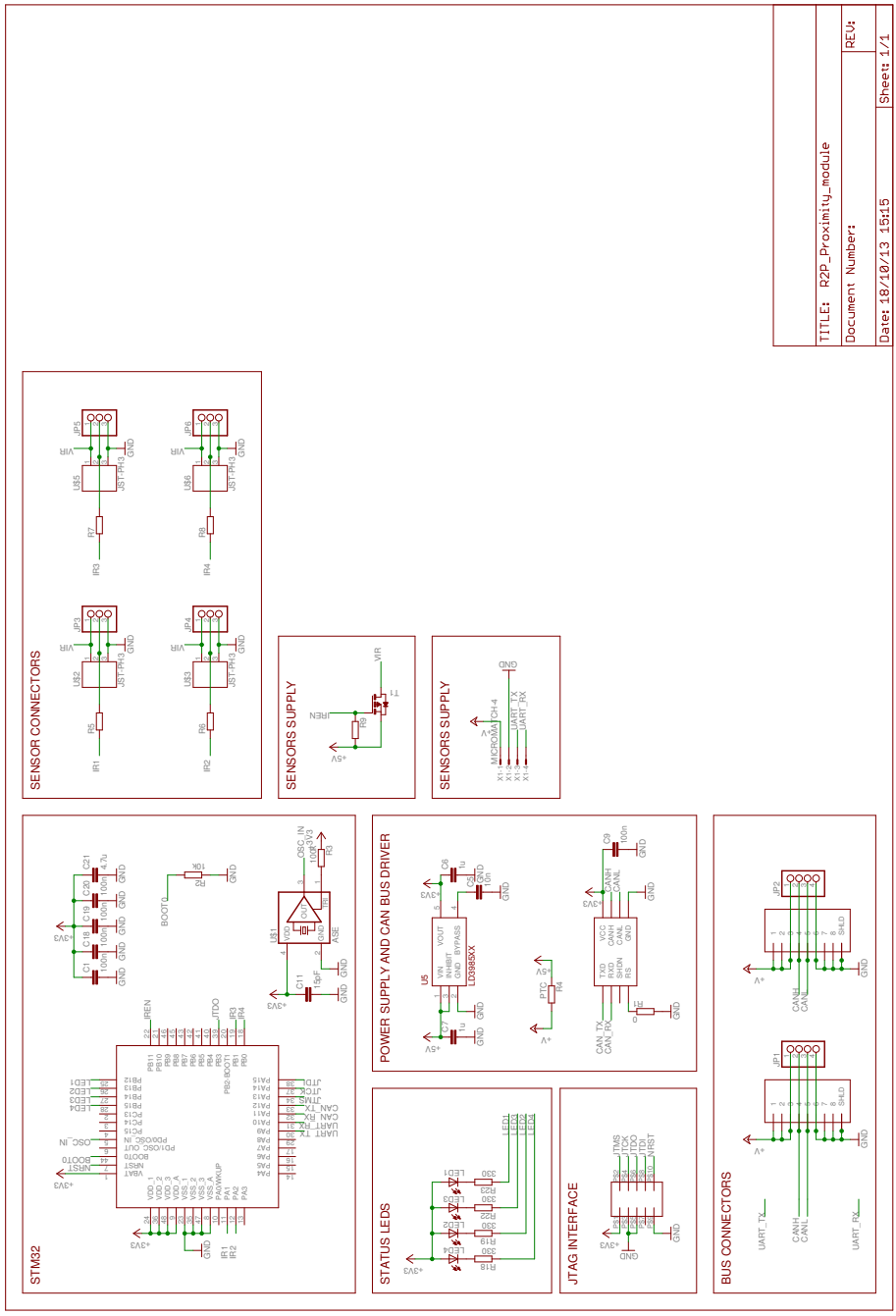


Figure A.4.: R2P proximity sensors module

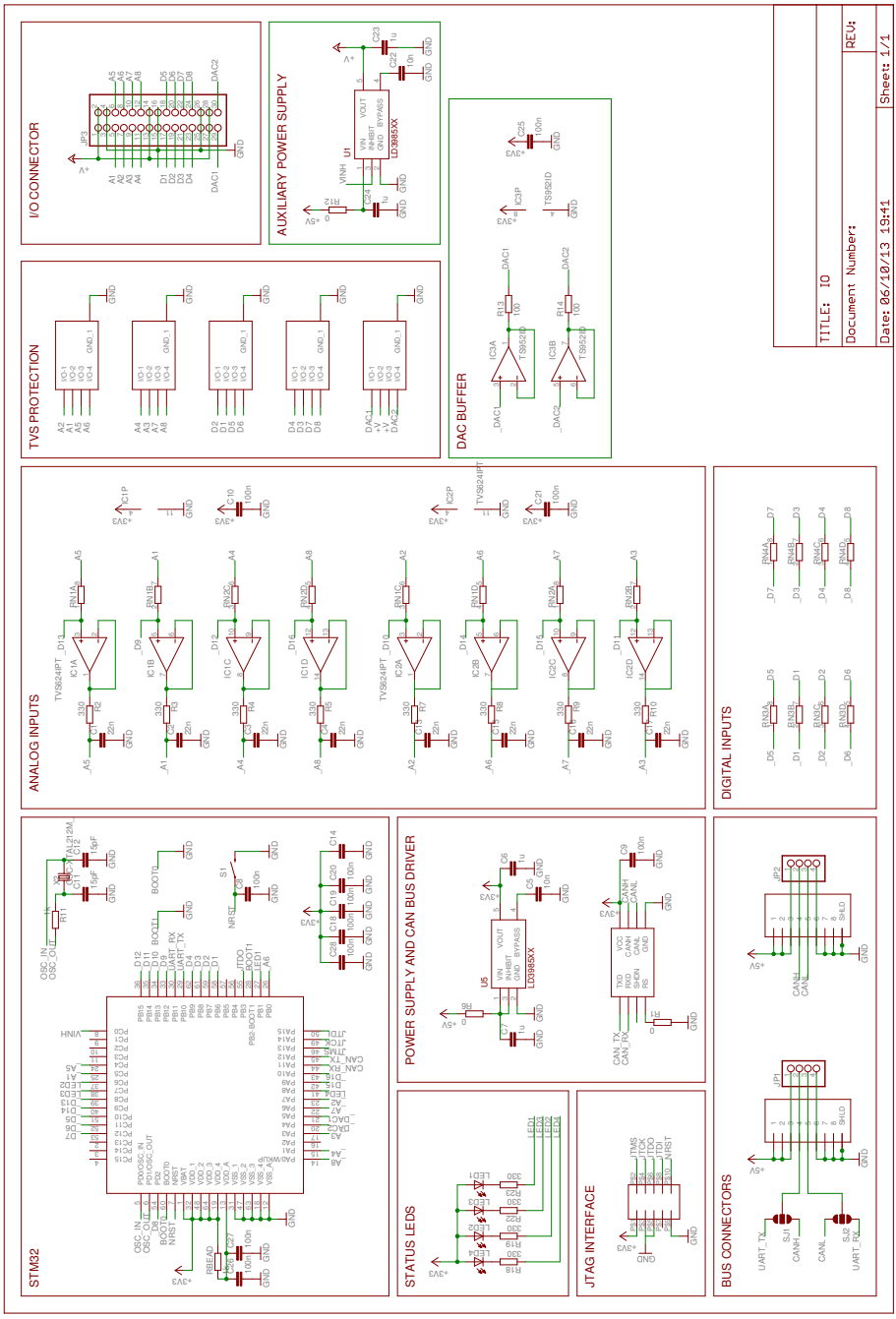


Figure A.5.: R2P generic Input/output module

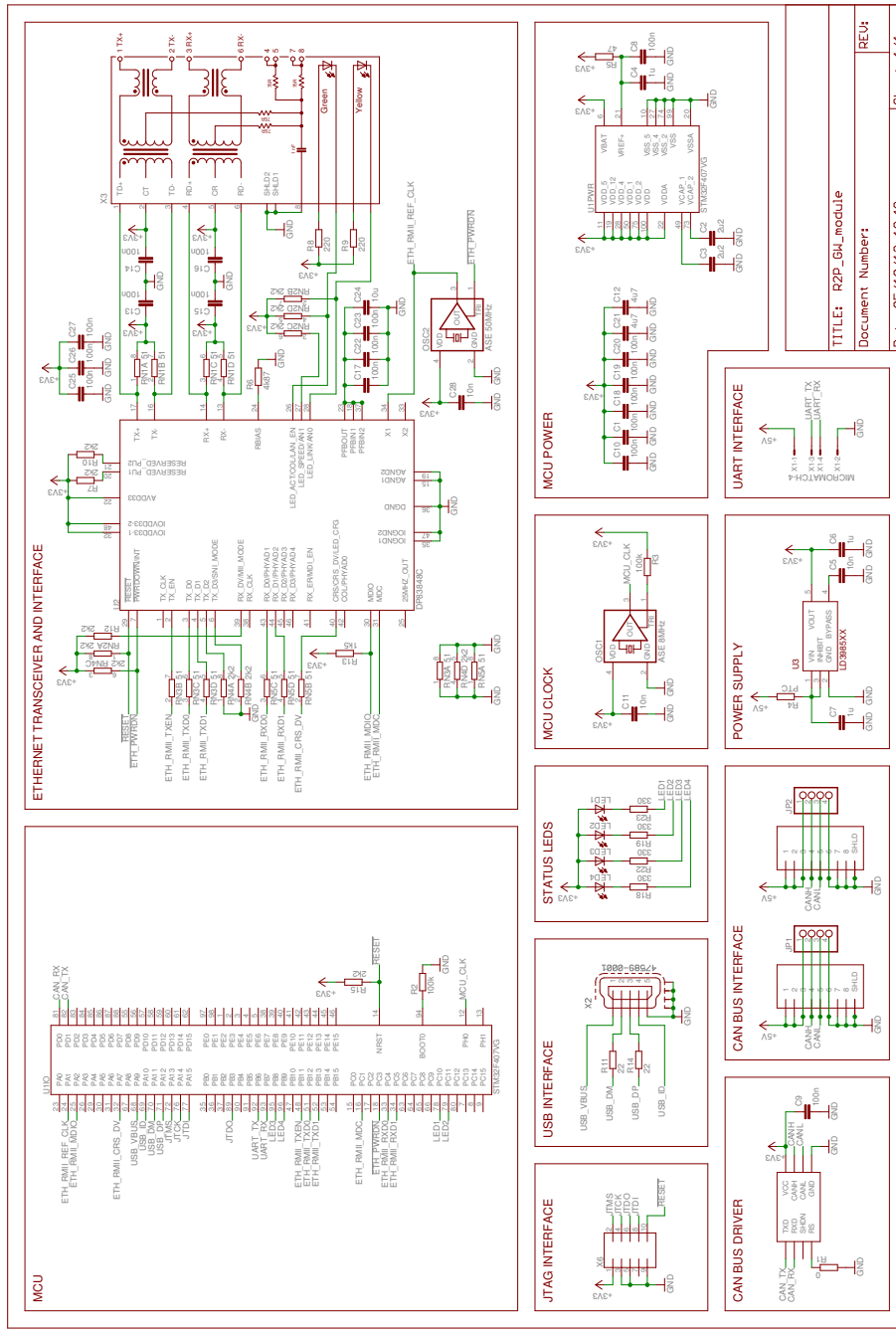


Figure A.6.: R2P gateway module