R2P: An open source hardware and software modular approach to robot prototyping

Andrea Bonarini, Matteo Matteucci, Martino Migliavacca *, Davide Rizzi

Department of Electronics and Information, Politecnico di Milano, Italy

1. Introduction

The development of new robotic applications is challenging, as competences from many domains are required; mechanical, electronic, and software components are involved in every robot and the overall success of a robotic application depends on the interplay of all these components. Therefore, a robot prototype becomes a precondition for any robot development aimed both at the validation of some research result or at the development of a new robotic product. However, the time and effort needed to set up a prototype, together with the complexity of such a task for non-technical domain experts, often prevent the development of interesting ideas and an effective time to market.

On the other hand, 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 standard way by modular components. Modularity enables the re-use of components in different products and prototypes, thus enlarging the share set, reducing costs, in terms of both time and money, and improving reliability too. 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 [1]. In software engineering, software components, generally organized in libraries or frameworks, are re-used among different projects and by several software producers [2].

* Corresponding author.
E-mail addresses: bonarini@elet.polimi.it (A. Bonarini), matteucci@elet.polimi.it (M. Matteucci), migliavacca@elet.polimi.it, martino.migliavacca@gmail.com (M. Migliavacca), davide.rizzi@mail.polimi.it (D. Rizzi).

In recent years, several development frameworks [3–7] have been proposed to assist researchers in the design of robotic applications. However these frameworks are focused on high-level software development, with desktop-level computers as reference platforms, and they do not take into account embedded firmware development on resource-constrained platforms or the reuse of hardware components as a whole. To the best of our knowledge, the only middlewares targeted to embedded robotic development are COSMIC [8], which later evolved in FAMOUSO [9], and Aseba [10]. FAMOUSO is focused on asynchronous, event driven, communication, and it is mainly targeted at the development of multi-robot systems. Despite being a complete framework for such applications, it 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. Aseba is a development suite which includes a graphical IDE, an easy to use scripting language and deployment tools, but it focuses on the control of swarms of small robots rather than on the implementation of modular architectures for robots, and, like for FAMOUSO, it is not focused on real time applications [10]. Nevertheless, both projects are software frameworks, and they do not provide any hardware modularity.

On the other side, some modular hardware platforms, such as Arduino [11] or Lego NXT [12], are often used to build simple robots. These platforms feature a generic control board, running a low-cost microcontroller, and a set of simple sensors and actuators which can be attached to the corresponding expansion ports. The user is also provided with a sample code to interface with the supported devices, allowing an easier development with respect to bare-metal systems. Such platforms are generally targeted at toy-like robotics, where small actuators and simple sensors are involved. The centralized approach and the communication bus exploited make it impossible to connect devices farther than a few dozen centimeters. Moreover, the programming approach followed by most simple platforms, i.e., a single loop (the *superloop*), is not suitable for complex applications and it does not guarantee real-time execution.

Some modular mobile robotic platforms have been proposed too, such as the *E*-puck educational robot [13], the Khepera robot [14] and others [15,16]; all of them offer a mobile wheeled base to which accessories and custom boards can be added. Target applications are mainly related to swarm robotics, where many, small, robots are involved. However, such platforms aim only at controlling the mobile base they are designed for and do not allow to build different, general purpose, robots.

More powerful, modular solutions that provide hardware devices, software components, and tools for control and application development are adopted in the industrial automation field, e.g., Programmable Logic Controllers (PLC), but characteristics in terms of performance, costs and energy consumption are not compatible with requirements of most mobile robot applications and, in general, of affordable robot prototyping.

To fill the gap between toy-like, often closed, robot platforms and full-fledged industrial devices, we have developed Rapid Robot Prototyping (R2P [17]), an open source, modular, hardware and software framework, where off-the-shelf modules (such as sensors, actuators, and controllers) can be combined together in a plug-and-play way to implement complex robotic applications. R2P relies on the principle that the functionalities identified in a robot application can be implemented by modules not only at the software level, as it is common in most frameworks, but also at the hardware level. Basic functionalities such as motor control, distance measurement, inertial navigation are implemented by specific, standardized hardware modules, with corresponding firmware, that can be plugged on a common bus and interact in real-time. This makes it possible to develop and re-use modules which have been widely tested and can be replicated in different applications, bringing at a hardware level the principles of modularity that have become common practice in fields such as automotive and software development.

R2P is not designed for a specific domain: target platforms go from simple mobile robots, including toy-like ones, up to mobile and service robot platforms, as well as unmanned aerial or land vehicles. R2P does not provide mechanical components, while it is focused on the electronics needed by common robotic applications, and on the tools needed to develop the embedded software which controls the robot. As a consequence, R2P modules can also be exploited to control existing mechanics, e.g., a robotic arm or an autonomous wheelchair, relying on off-the-shelf hardware devices and a simple-to-use programming environment. The real limits of R2P, at the actual stage of development, are imposed by the modules already available; moreover, as R2P is an open source, modular, framework, it could be extended with additional modules to cover other application fields.

Section 2 of this paper describes the top down approach adopted to develop the prototype of a robotic application and underlines the typical issues we might have to face. In Section 3 we introduce the internals of R2P design together with some details about its implementation, while Section 4 describes some of the hardware modules we have developed and released in open source. In Section 5 a case study is presented using available modules to build an omnidirectional base, and some results about R2P performance are reported. Section 6 concludes this paper, summarizing the advantages of R2P in robot development.

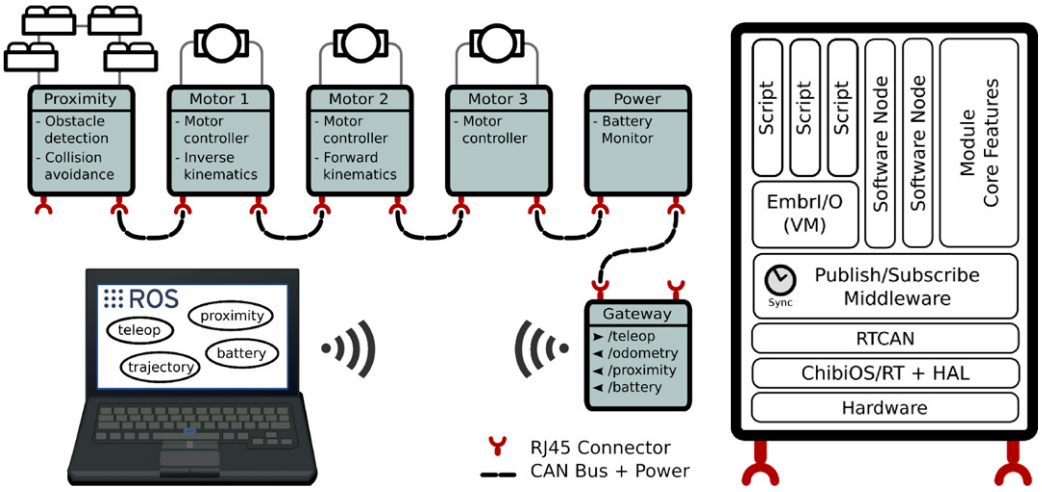
2. Robot development: from the idea to the prototype

The design of a new robotic application usually starts with the identification of its functional specifications: what do we need to have our robot achieving its tasks? Once required functionalities are identified, it starts an often painful process aimed at the identification of appropriate hardware and software solutions, their integration, and the set-up of a prototype to test the overall idea. Available components usually drive the design of the prototype and this is far away from the user-centered design approach [18], now common in most real market applications, where users and domain experts are deeply involved in the development process.

In the first phases of prototype development, components are chosen from a vast range of possibilities offered by markets that are still not directly oriented to robot development. From one side, we have the so-called industrial components, providing often expensive, yet robust, solutions; from the other side we have the low-end hobbyist components, often much cheaper, but less performing. Components are then selected considering both their characteristics and the possibility to put them together, facing problems that start from power supply and go up to physical and logical interfacing. In the following paragraphs, we present a review of classical issues we usually face in the process of prototyping a robotic application.

Wiring. Usually, to build a robot prototype, we integrate devices from different vendors, which need different power supplies and exploit different physical connectors. Additionally, all the devices rely on different data connections and adopt different communication protocols, so we need a separate data cable for each of them. This, in turn, means that we have a lot of wiring on the robot, adding complexity and points of failure to the system.

Firmware development. In order to manage the various electronic devices present on the robot, embedded boards are often considered. In general, they are equipped with a microcontroller that takes care of low level communication with other devices, processes data from sensors, runs on board control loops, and possibly communicates with an external computer. Working on



(a) The architecture of a robot prototype built exploiting R2P framework.

(b) Firmware structure of a basic R2P module.

Fig. 1. R2P prototyping environment.

embedded hardware is often a difficult task for non-experienced developers, as most people that still have a computer programming background are not familiar with firmware development (i.e., configuring hardware registers, handling interrupts, and other specific knowledge). The synchronization of the various tasks running on the robot is another critical challenge, which results in a huge additional effort for developers of robotic systems. To these extents, a Real Time Operating System (*RTOS*) can support developers to write firmware code easier, and in a more reliable way. An additional support comes from a Hardware Abstraction Layer (*HAL*), which abstracts the hardware implementation of different, low-level peripherals (e.g., analog to digital converters, timers, communication interfaces), relieving the developer from configuring the corresponding registers on the microcontroller and making easier the port of existing code to different platforms.

Communication protocol. Communication handling is another key aspect, as several devices need to exchange data in a reliable way to run a robotic system. Hardware components on a generic robot have different communication requirements which the communication protocol must consider. For instance, 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. Periodic communication are best handled in a time-triggered way, to schedule transmission occurrences, preventing collisions and reducing the jitter. Besides the sensors used in control loops, in a robotic system, we find other kinds of data sources; for instance, proximity sensors and bumpers produce useful data when triggered by some event, like approaching an obstacle, and the most important factor is to react to new readings as quickly as possible. 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. Using event-triggered transmissions saves bandwidth and, generally, reduces latency. Mixing time-triggered and event-triggered traffic on the same communication channel is not trivial, and communication errors can compromise the entire system functionality.

Software modularity. A modular implementation of software components supports massive reuse of existing code through different projects, which dramatically boosts the development time of new applications, as shown by many software development frameworks for robotics [3–6]. This effective approach is commonly followed to develop high-level software, but firmware running on resource-constrained, low-cost microcontrollers, which cannot run such frameworks, is often coded from scratch. As a consequence, low-level code reuse is difficult, requiring adaptations and customizations from project to project.

Development tools. The development of a robotic application from scratch, using components from different vendors, may require distinct tools, and this adds complexity to the process. Moreover, when a prototype is ready and needs testing, developers often need to write their own debugging tools, which may require as much resources as the prototype development itself. Also when the debugging phase is ended and only the high level behavior of the system needs to be tuned, common deployment techniques of embedded applications, like firmware flashing, slow down the refinement process, unless developers create specific tools, with additional efforts.

3. R2P: rapid robot prototyping

In R2P, we consider a robot as a distributed system of hardware devices (e.g., sensors, actuators, controllers) and software components (e.g., sensor filtering and conditioning, control loops, planning algorithms), implementing the basic functionalities required by its specific application. R2P provides the user with a set of hardware modules, and the corresponding firmware, that implement such basic functionalities. These modules can easily be combined as they are to develop a complex robotic system, and, at the same time, they are open to further developments, where improvements of the single module can be done without the need to reimplement the whole system.

3.1. Physical connection: single power and data link

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 fast as possible. Power supply requirements are usually not consistent among different electronic devices, then a compromise must be reached. To this extent, modules can be divided in two categories: modules requiring only little power, e.g., most of the sensors, and modules requiring much higher power, e.g., motor drivers. We decided to restrict the R2P bus to operate at 5 V, which suits most of the requirements of todays electronic

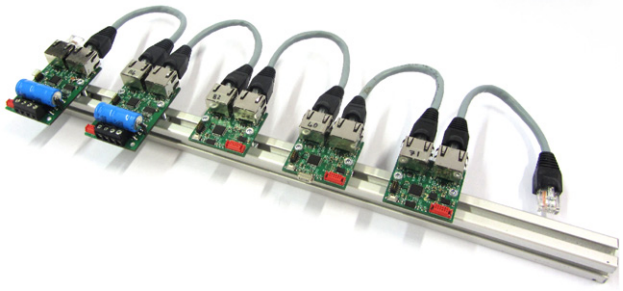


Fig. 2. A set of R2P modules connected together: the daisy-chain connection schema reduces wiring and eases the addition of new modules.

devices, while modules that require a higher power supply must rely on an auxiliary connection and power source. 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 (see Fig. 2). The bus is designed to handle up to 20 hardware modules, each consuming 200 mA maximum, over an up to two meter long cable. If more modules are present, or the bus length increases, an additional power supply module has to be added to the chain.

We have reviewed different communication standards to exchange data between modules [19], and we chose the CAN bus, which shows some features that best suit a modular architecture:

- it is a bus, so many devices can be connected on a single line;
- it is widely adopted in many fields, including automotive, so that most of today microcontrollers have an integrated CAN controller;
- it provides hardware bus arbitration, which makes the communication protocol simple;
- CAN controllers can filter messages at the hardware level, thus reducing processing requirements;
- CAN transceivers are quite rugged, so they ideally suit the needs of fault tolerance intrinsic in prototyping, and work reliably in harsh environments.

The CAN bus has a maximum data rate of 1Mbps, which is enough for a distributed system of smart devices where only high level processed data must be sent over the network, and raw data are processed locally on the modules [19]. Devices with higher traffic requirements, such as a camera streaming images for remote control, must rely on a different connection (e.g., Ethernet or USB). With reference to the last example, we would like to stress that the distributed philosophy of R2P would suggest the use of smart cameras that can elaborate images on board and share only high level information.

We chose a standard RJ45 Ethernet patch cable as physical connection, as it matches CAN bus specifications [20] 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.

3.2. RTOS and hardware abstraction layer

The use of an operating system, in addition to the real-time related aspects, greatly helps in writing software even for small embedded systems. A reliable RTOS relieves developers from a huge programming effort, so that they can concentrate on debugging and improving the high level code and functionalities, without worrying about low level core functionalities. In addition, software written to run above an OS is also more portable between different hardware and operating systems, as long as the OS provides a common interface, as it usually does.

We have evaluated several open source real-time operating systems and finally we have chosen ChibiOS/RT [21] for its portability, rich features set, extremely high efficiency and because it was the only offering a complete Hardware Abstraction Layer (*HAL*). Anyway, a deep review of the available open source real-time operating systems is out of the scope of this paper. ChibiOS/RT is designed for deeply-embedded, real-time applications where execution efficiency and compact code are important requirements. Its kernel size, with all features enabled, is about 8 Kb on ARM Cortex-M3 platforms. With a context switch time of about 1 uS on a 72 Mhz Cortex-M3 platform, it is also the best performing RTOS we have tried. ChibiOS/RT exposes a rich set of primitives, like threads, virtual timers, semaphores, mutexes, messages, mailboxes, event flags, queues and I/O streams. Writing software on top of these features is simple also for users without any embedded programming background, while writing firmware for a microcontroller from scratch requires experience and know how, besides a deep understanding of the specific hardware.

ChibiOS/RT also provides a Hardware Abstraction Layer (*HAL*) supporting a variety of abstract device drivers on various platforms. A Hardware Abstraction Layer (*HAL*), which gives consistent and reliable access to the peripherals of the microcontroller, can help most of the developers, so that they can interface with the hardware without needing specific knowledge. ChibiOS/RT HAL supports modular, portable code that relies on common low level functions and drivers, as it decouples the firmware code from the specific hardware, and simplifies software that needs to interface with external devices. In R2P, the HAL API can also be accessed from scripts running on the EmbrI/O Virtual Machine, described in Section 3.5.1, making hardware interfacing simple also for novice programmers.

3.3. Real-time communication protocol

To handle the communication between R2P hardware modules, we developed RTCAN [22], a real-time CAN bus protocol targeted at robotics applications. RTCAN design comes from the analysis of existing CAN bus protocols, to identify the advantages of different proposed approaches with respect to the requirements of robotic systems, which have been pointed out in Section 2; a detailed description of the motivation which lead us to the development of a new protocol, and of its internals, is provided in [22]. RTCAN supports both the time-triggered and the event-triggered transmission paradigms, exploiting the best of the two, to satisfy the needs of robot designers: limited transmission jitter for control loops, low delivery latency to quickly react to events, and scheduling flexibility to easily add features, i.e., networked nodes, to an existing systems.

3.3.1. RTCAN message types

To support the communication requirements of robotic systems, RTCAN defines three distinct message types:

- Hard real-time messages (*HRT*) are periodic messages, e.g., from distributed control loops; they are deterministic, their deadlines are absolute in time and should never be missed. Control loops are highly affected by the presence of jitter [23], which introduces variable delay and, thus, may induce overshoots of the control action and instability. To guarantee the correct execution of critical activities, critical messages need to be delivered on time and with low jitter.
- Soft real-time messages (*SRT*) are event-triggered messages, 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. SRT message transmission must not interfere with the exchange of HRT critical messages. To reduce the latency of non-deterministic messages, while optimizing the bus exploitation, they have to be scheduled with some best-effort policy.

	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	
HRT	000000	Message ID	Fragment #
SRT	Laxity	Message ID	Fragment #
NRT	111111	Message ID	Fragment #

Fig. 3. RTCAN priority and message ID encoded into the CAN bus 29-bit Extended ID.

- Non real-time messages (*NRT*), e.g., logging messages, do not expire in time; they can be delivered without any latency constraint, exploiting free resources when available.

3.3.2. Bus access arbitration

The CAN bus features a carrier-sense multiple-access (CSMA) media access control (MAC), which is based on the ability of CAN controllers to detect the bus status while transmitting. CAN transceivers have open collector outputs, and data is transmitted through a binary model of *dominant* and *recessive* bits where dominant is a logical 0 and recessive is a logical 1. The controller reads back the bus status while it transmits the arbitration field of a CAN frame, and, if a dominant bit is recognized while it was trying to transmit a recessive one, it knows that the arbitration is lost and the node becomes a receiver. This allows automatic bus access arbitration, given that all the competing messages have unique values in the arbitration field and that the transmissions start during the same bit time. As a consequence, the CAN bus is well suited for fixed-priority event-triggered communication, where high-priority messages win arbitration on low-priority ones; the drawback is that delivery latency depends on bus load, as the transmission of a message can always be preempted by a higher-priority request.

To extend CAN bus applications to distributed control systems, where temporal determinism and low jitter are mandatory, several protocols to handle time-triggered traffic on the CAN bus have been presented and reviews are available [24–26]. Temporal determinism is guaranteed by pure time-division multiple-access (TDMA) approaches: bus access is assigned by means of time slices associated with transmission occurrences. RTCAN follows a pure TDMA approach for HRT messages, which reserve bus access in a calendar, while SRT and NRT messages are transmitted exploiting the hardware CSMA arbitration of CAN controllers.

3.3.3. RTCAN communication cycle

The key aspect to effectively combine TDMA and CSMA approaches is temporal isolation: event-triggered transmission requests should not compromise time-triggered traffic. To this extent, RTCAN relies on periodic messages, named *sync messages*, sent by a master node to align the local clocks on all nodes. The interval between two sync messages defines a communication cycle, which is in turn divided in several time slots. Each time slot can be reserved for the time-triggered transmission of an HRT message (or a part of it, if fragmentation is needed) or can be available for SRT and NRT messages. Time slots are reserved by a centralized scheduler running on the master node, which communicates to all other nodes the reservation plan for the beginning cycle in the payload of the sync message. The reservation plan is a simple bit mask where a 1 denotes a slot reserved to transmit an HRT message, and a 0 means that the slot is free for the CSMA arbitration of other messages. An example of RTCAN communication cycle is reported in Fig. 4, with the corresponding reservation mask; in this example each node registers 3 HRT messages with different periods, which are inserted in the reservation calendar by the scheduler. Time slots length is application dependent: it 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 maximum number of time slots within each cycle is limited by the 8 bytes payload of CAN frames, which gives a temporal horizon of 64 time slots for the reservation mask. The choice of slots per cycle determines the maximum throughput: smaller slots give higher bandwidth if many small messages are sent on the bus, but, as the reservation mask is limited in size, the frequency of the sync messages must be increased wasting some bandwidth.

3.3.4. HRT message scheduling

HRT messages are time-triggered, they must always be delivered in time with the lowest possible jitter. In RTCAN, HRT messages are periodic, future transmission occurrences are known a-priori, and the scheduling process reduces to admission control and phase displacement, assigning the first transmission slot to each message, to avoid temporal overlaps.

Admission control is done by checking that the requested transmission period is not relatively prime to other scheduled transmission periods. Then, an initial phase displacement, which determines the time slot for the first transmission, is assigned by the scheduler to the HRT message. 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 messages 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 relatively multiple is common on robotic systems; then such a limitation is, generally, not a problem. Fig. 4 reports an example of RTCAN calendar, where slots have been reserved to three HRT messages with different periods.

Given the initial phase displacement, each node can compute the next reserved time slot for an HRT message simply adding the period. In other words, the scheduler and the admission control are centralized on the master node, but the reservation calendars are local. Only the master node needs to know all HRT message reservations, updating the global calendar to generate the reservation mask sent with the sync message; this is required to avoid event-triggered transmission attempts in reserved time slots. Using a centralized scheduler also facilitates online dynamic scheduling, preserving flexibility.

3.3.5. SRT and NRT message arbitration

SRT messages are triggered by events, their delivery latency should be low but they must not interfere with HRT communication. They are sent only in slots marked as available in the reservation mask, and compete for the bus using the CAN hardware arbitration. In order to reduce latency, while enhancing resource exploitation, deadline-based dynamic schedulers are preferable with respect to fixed priority schedulers or Rate Monotonic schedulers [27,28]. The scheduling policy we have adopted for soft real-time messages is inspired by Earliest Deadline First (*EDF*) and Least Laxity First (*LLF*) schedulers: an SRT message increases its priority while it is approaching its deadline [29,30]. 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 Fig. 3. In this way messages near their deadline have higher chances to be transmitted with respect to messages with far deadlines thanks to the hardware arbitration. Laxity can be encoded linearly or using a logarithmic scale, resulting in a finer resolution for nearer deadlines, and a coarser resolution for further deadlines [31]. The remaining bits of the arbitration field include an ID to identify RTCAN messages, and a fragment counter used to handle messages longer than the payload of a CAN frame, as explained in Section 3.3.6.

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.

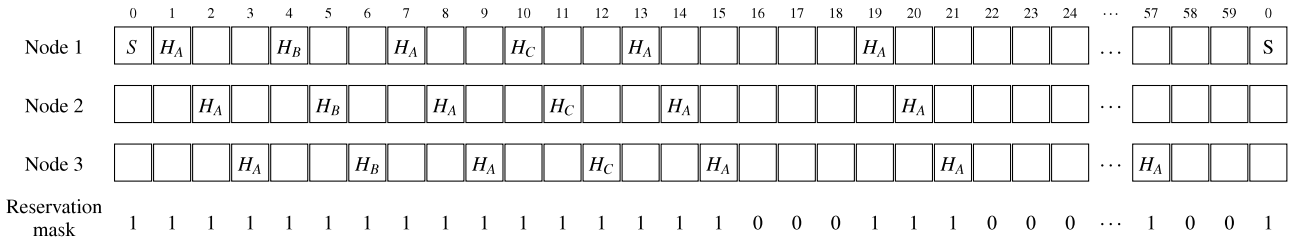


Fig. 4. The HRT calendar returned by the scheduler and the respective reservation mask. H_A messages have a 1 ms transmission period, H_B 5 ms and H_C 10 ms. S is the sync message from the master node. Empty slots are available to transmit SRT and NRT messages.

3.3.6. Fragmentation

Fragmentation is handled by RTCAN as well; payloads longer than 8 bytes, which is the maximum payload of CAN frames, are fragmented and transmitted in sequence. A fragment counter in the CAN arbitration field identifies fragmented messages, and they are concatenated during the reception. This simple approach exploits the CAN bus guarantee that packets are received in the same order as they are transmitted. All RTCAN messages can be fragmented: HRT ones will span over more reserved time slots, while SRT ones must deliver the last fragment before their deadline.

RTCAN is focused on real-time communication and not on fault tolerance; receive errors (e.g., overruns or checksum mismatches) are not handled by RTCAN: the message is just signed as corrupt and retransmission requests are eventually handled by higher layer protocols.

3.4. Publish/subscribe middleware

The modular, distributed, approach to robot prototyping proposed by R2P extends also to the firmware running on the hardware modules. Most software development frameworks foster separation of concerns and software decoupling, to support an easier design and maintenance of applications and to enable code reuse through different projects. R2P brings these concepts to embedded software development, featuring a lightweight publish/subscribe middleware which is in charge of handling message passing between both local and remote software components. Fig. 5 reports an example of firmware developed with R2P and its middleware; please refer to the use case presented in Section 5 for details about the specific application.

3.4.1. Nomenclature

R2P hardware boards are called *modules*, while *nodes* are 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 side, nodes can declare a *subscriber* on the same topic, and the middleware manges the association with the corresponding publisher.

3.4.2. Software nodes

Nodes perform simple tasks, each of which satisfies a specific functional requirement. They are identified by a string, which is unique for nodes within a single module. The overall system is then composed of several nodes, running on the same module or distributed on a network of modules. Each node subscribes the topics it needs to operate and defines the topic on which it will publish data, as explained in the following section. To support safe operation of robots, nodes may implement also special methods to handle system status changes (e.g., *run, stop, fail*). Thanks to this loosely-coupled communication paradigm, nodes can easily be modified, updated, and replaced without affecting the rest of the system: the only requirement is to subscribe and publish the same topics.

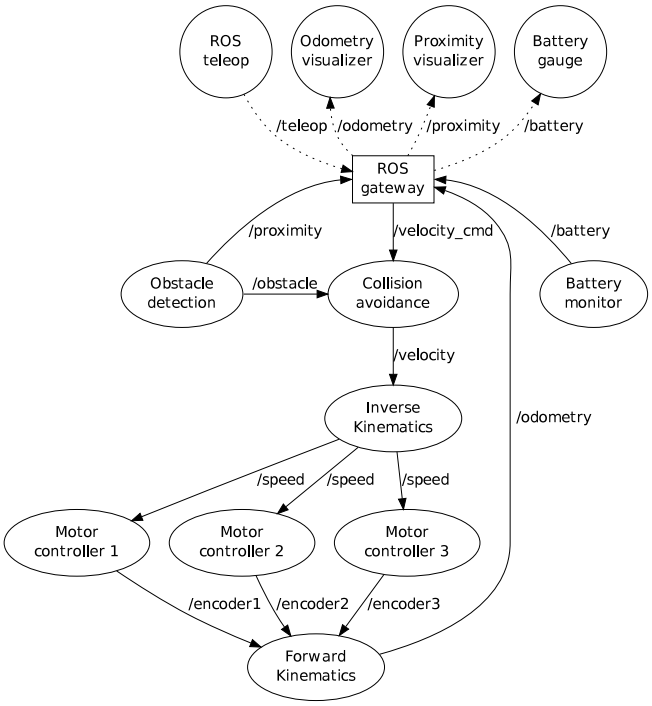


Fig. 5. Example of firmware architecture. Ellipses represent R2P nodes, while arrows show the topics. Circles are ROS nodes running on the remote PC.

3.4.3. Message topics

Nodes exchange data by sending and receiving messages on a given topic. The content of messages is specified by standard C data structures, composed of typed fields, and their definition is common to all nodes. Topics are declared by publishers, which also specify the data type of published messages. Each topic is strictly related to a single message type: defining a topic also determines the data type published on it. In other terms, following the publish/subscribe paradigm, communication is addressed by data content, not by source or destination. Topics are identified by simple strings, and they are uniquely identified on the network as *module_name/node_name/topic*.

3.4.4. Publishers and subscribers

Publishers and subscribers handle message passing between nodes producing data and nodes consuming data. When a producer node advertises a publisher on a given topic, the middleware broadcasts the advertisement to all nodes. In the same way, consumer nodes register their subscribers to the middleware, which associates them with the respective publishers. To obtain asynchronous operations, subscriptions can be queued, to be decoupled from the advertisement, and they will become active when a publisher on the same topic is advertised. After at least one subscription has been registered, producers can start publishing messages.

Incoming messages can be handled by subscribers both declaring callbacks, which are invoked on each message reception, or through polling. To avoid continuous polling, a node can suspend its execution, possibly specifying a timeout.

Publishers and subscribers are also in charge of allocating memory to store messages, as, on resource-constrained devices, dynamic memory allocation functions (e.g., *malloc()* and *new()*) are generally not supported. To this extent, each publisher in the R2P middleware manages a *memory pool*. Subscribers declare a queue for incoming messages, and the memory reserved to the corresponding buffer is yield to the publisher, which adds it to the memory pool. Messages are then stored in the shared memory area: they are passed by reference and the allocated memory is available again when all subscribers consumed and released the message. This mechanism has two advantages: distinct subscriber queues guarantee that slow, low-priority, consumers do not prevent publishers from delivering messages to other nodes, while the shared memory pool avoids message copying, which would introduce latency.

```
node.advertise(&proximity_pub);
node.advertise(&obstacle_pub);

while (node.run()) {
  measure = adcConvert(...);

  proximity = proximity_pub.alloc();
  if (proximity) {
    proximity->distance = measure;
    proximity_pub.publish(distance);
  }

  obstacle = obstacle_pub.alloc();
  if (obstacle && (measure < TRESHOLD)) {
    obstacle->distance = measure;
    obstacle_pub.publish(obstacle);
  }
}
```

Listing 1: Publisher code sample

In a modular, distributed, hardware and software architecture like R2P, publishers and subscribers can be deployed either on a single module or across a network of modules, depending on the application. From the user point of view, there is no difference between local and remote messaging. If a local publisher has remote subscriptions, a *remote subscriber* is created, in charge of transmitting messages on a specific *transport*, e.g., the CAN bus. In the same way, the subscribing module initializes a *remote publisher* that locally publishes messages received on the transport. The transmission queue on the remote publisher is still defined by the subscriber; if more remote subscriptions are instantiated, the longer queue determines the actual buffer length.

R2P middleware syntax follows the coding style common to many publish/subscribe frameworks, helping users in writing embedded software in the same way they are used to do on computer systems. The code snippets in Listings 1 and 2 are relative to the use case presented in Section 5. Listing 1 shows the code run by a publisher node, which publishes proximity measures on the */proximity* topic, and only the measures which are under a given threshold on the *obstacle* topic. Memory is allocated to the new message, if available, which is then filled and published. As an example of subscriber, refer to Listing 2, which shows a node subscribing to two different topics. The node waits for messages from any of the topic it is subscribed to, and its execution will be restored by the middleware on new receptions or, eventually, after a timeout. Whenever a message is received on the */obstacle* topic, a routine checks if it is on the current path of the robot and, eventually, the velocity set point is overridden. The resulting velocity is then published to the other nodes. When the node has consumed a message, the message is released and the memory is available again for future allocation.

```
node.subscribe(&obstacle_sub);
node.subscribe(&velocity_sub);
node.advertise(&velocity_pub);

while (node.run()) {
  node.wait();

  new_cmd = velocity_sub.get();
  if (new_cmd) {
    current_cmd->release();
    current_cmd = new_cmd;
  }

  obstacle = obstacle_sub.get();
  if (obstacle) {
    avoidCollision(current_cmd, obstacle);
  }

  velocity = velocity_pub.alloc();
  if (velocity) {
    velocity->setpoint = current_cmd->setpoint;
    velocity_pub.publish(velocity);
  }
}
```

Listing 2: Subscriber code sample

3.4.5. Real-time support

R2P software nodes are implemented by threads, and they are scheduled for execution by the underlying RTOS. Execution priorities are assigned to nodes, in order to prevent critical tasks from being blocked by less important ones. Developers can take advantage of the many features offered by the RTOS to synchronize local nodes, manage mutual access on hardware resources, and so on.

To extend real-time support to distributed communication, the middleware defines 3 classes of publishers:

- Synchronous real-time publishers are time-triggered and they publish periodic messages, e.g., from distributed control loops; users specify the update rate and provide a callback to get the content of the message, which is automatically updated and broadcast. If the memory pool is empty, then messages cannot be published, the publishing node is signaled.
- Asynchronous real-time publishers are sporadic, i.e., event-triggered; a deadline can be specified, defining the time limit by which messages must be delivered to subscribers. If a deadline is missed, the publishing node is signaled.
- Non real-time publisher, which do not expire in time, e.g., logging messages. If the message queue is full, the node is suspended, waiting for the next available slot.

Local and remote subscribers do not differentiate between synchronous, asynchronous and non real-time publishers, but a node can specify a timeout when waiting for messages, thus noticing missed deadlines.

3.4.6. Software deployment

An easy software deployment process is mandatory when multiple hardware modules are distributed through the system, each running its own firmware. R2P provides users with a dynamic loader for software nodes: each board is flashed with a *static* firmware, composed of the RTOS, RTCAN, the middleware and the core libraries, while the user application can be dynamically loaded and updated at run-time by means of uploading middleware nodes. Nodes are firstly compiled on the host machine and

partially linked to object files common to all R2P modules; at the moment of loading on the target module, code sections are relocated to the correct addresses and the linking is finalized. All the linking and relocation process is implemented with common GNU ELF tools, avoiding any specific, non-standard, techniques. The dynamic loader allows users to easily deploy, configure and update their applications, without needing physical access to the hardware modules. Advanced developers can still access the on-board standard programming interface to directly flash custom firmwares.

3.5. Development tools

R2P also offers tools to support the development of the software that runs the robot. These tools support writing the firmware code in an easy and fast way, and speed up the deployment and debugging phases.

3.5.1. EmbrI/O scripting language

An optional component, integrated in our framework to make the development of embedded firmware even easier, is EmbrI/O, a small virtual machine that can run scripts on the embedded target. EmbrI/O is a port of the Embryo Virtual Machine [32] to embedded platforms, with custom additions to interface with hardware devices and events. Scripts are written in the PAWN language [33], which is a typeless, 32-bit extension language with a C-like syntax focused on fast execution speed, stability, simplicity, and a small footprint.

```
#include <hw>
#include <mw>

@message("SPEED_TOPIC") {
  new speed;

  speed = getMessage("SPEED_TOPIC");
  setSpeed(MOTOR_LEFT, speed);
  setSpeed(MOTOR_RIGHT, speed);
}

@message("STEER_TOPIC") {
  new steer, left_speed, right_speed;

  steer = getMessage("STEER_TOPIC");
  left_speed = getSpeed(MOTOR_LEFT);
  right_speed = getSpeed(MOTOR_RIGHT);
  speed = (left_speed + right_speed) / 2;
  setSpeed(MOTOR_LEFT, speed - steer);
  setSpeed(MOTOR_RIGHT, speed + steer);
}

@inputPin(EMERGENCY_BUTTON) {
  setSpeed(MOTOR_LEFT, 0);
  setSpeed(MOTOR_RIGHT, 0);
}

main() {
  print("Hello world from EmbrI/O script!\n");

  while(1) {
    togglePin(LED1);
    delayMs(500);
  }
}
```

Listing 1: Sample EmbrI/O script to control a differential drive robot

In EmbrI/O we have extended the syntax of the original language adding a custom notation for the identification of event handlers functions. Users can thus specify how to react to events such as hardware interrupts (e.g., a new reading from the A/D converter) or software notifications (reception of a new message from the middleware). To improve execution speed, EmbrI/O scripts are not interpreted on the target device, but compiled to EmbrI/O opcodes on a host computer. On the virtual machine each opcode is an index in a table that contains a jump address for every instruction, maximizing execution speed. Compiled scripts are then dynamically loaded to the target module through the R2P network and executed on the system without any need to reprogram the current firmware. To save RAM memory, the script code is saved to the flash memory of the microcontroller in a reserved area.

Each virtual machine is a ChibiOS/RT thread, so multiple scripts can run at the same time, while task scheduling still relies on the underlaying RTOS. The execution overhead of EmbrI/O scripts is significantly small compared to other scripting solution, e.g., Embedded LUA [34] and Python-on-a-Chip [35], as the virtual machine is register based and optimized for performance, resulting in smaller code size and faster execution [36]. Nevertheless, EmbrI/O should be used only to write the high-level code that controls the behavior of the robot, while resource consuming tasks should rely on software modules which can be called by scripts, but are implemented in native language. A simple script to control a differential drive robot is reported in Listing 1.

3.5.2. uROSnode

Among the many available software frameworks to develop robotic applications [4–7], the *Robot Operating System* (*ROS*) is currently the most widely adopted in academia and research laboratories. ROS is an open source project which provides communication primitives, libraries, visualizers, package management, and more, to help software developers create robot applications. To support the integration of R2P modules within an ROS system, we developed μROSnode, a lightweight, open source, ROS client library targeted at resource-constrained devices. Thanks to μROSnode, R2P nodes and topics are listed as native ROS components, and the other way around, allowing ROS users to write their applications as they are used to do in standard ROS systems. On the other hand, the network of resource-constrained R2P devices can easily be interfaced to a computer running ROS and it can exploit its resources for complex, resource-consuming tasks which cannot be executed on embedded microcontrollers.

3.5.3. Graphical IDE

An easy-to-use, but powerful, graphical integrated development and debugging environment is the last tool we need in a rapid prototyping framework for robotic applications.

We are developing an Eclipse-based graphical IDE that covers all the development steps of a complex system: writing firmware for the hardware components, deploying it to the modules, setup the publish/subscribe network, scripting high level behavior and debugging the whole system. The IDE runs on a computer that is connected to the R2P network through a gateway module (see Section 4), and installed hardware modules are displayed on the user screen. By means of point and click operations, the user can inspect and debug the system, configure module parameters, and set up the network by connecting publishers and subscribers in a simple way such as drawing a line from a module to another.

The IDE also offers tools to modify and update firmware on a module, to set up the network that defines topics, and to connect publishers and subscribers, or to write high level scripts and inject them into modules. In this way, it is possible to change high level control policies modifying a simple high-level script and test it with a mouse click, directly observing how the real system reacts and thus speeding up the definition of the robot behavior.

3.6. 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 the project. 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 [37], dedicated fairs all over the world, and the Open Source Hardware Association, founded in 2011 [38]. This approach then gained popularity as a development strategy of big companies too, e.g., Google chose Arduino as the platform to develop new Android compatible devices [39]. 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 gives visibility and attracts users [40,41].

R2P is fully open source, both in hardware and in software, and it wants to take advantage of community-driven development to became a mature and widespread project. R2P repository is currently hosted on Github.[1] Besides from releasing the sources of our hardware modules and their firmware code, we share also software components, like filtering algorithms and control loop implementations, which can easily be integrated in user's projects thanks to the approach presented in Section 3.4. Common topic definitions will be hosted on a website, in order to encourage users to use the same conventions, which is mandatory if we want to decouple software implementations from their functionalities exploiting the middleware. Hardware modules are flashed by default with a set of commonly used software components, but users can connect to the community website and download algorithms shared by others to access new functionalities.

4. R2P hardware modules

A brief overview of the main R2P modules follows; notice that these are just few modules out of a possibly huge catalog that could be realized and shared with the R2P community, thanks to the open source approach proposed by R2P both for software and hardware development. All R2P modules feature an STM32 ARM Cortex-M3 microcontroller with 72 Mhz clock, 20 KB of RAM and 128 KB of Flash memory, a CAN transceiver and a voltage regulator.

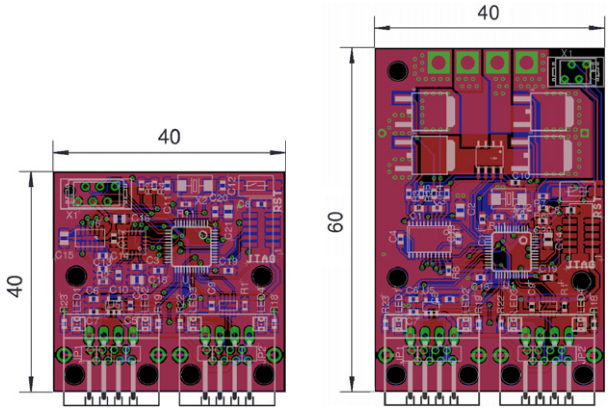
PSU module. The PSU module powers all the modules connected to the bus. Input voltage ranges from 5.5 to 36 V DC. A DC–DC converter produces a 5 V regulated output with a maximum current supply of 4 A and short circuit protection. Both battery voltage and current drain can be published over the network to monitor power consumption and to estimate remaining battery life.

DC Motor Controller module. The high-power DC Motor Controller module can drive DC motors up to 36 V, delivering a continuous 20 A 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 Controller module accepts position, speed, and torque setpoints, and publishes position and speed messages, with data from the encoder, or the measured current drawn. The PCB layout of the DC motor controller module is shown in Fig. 6(b).

IMU module. The IMU module is 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. The on-board sensor fusion algorithm produces heading, attitude and position messages. The PCB layout of the IMU module is shown in Fig. 6(a).

Proximity module. The proximity module interfaces with proximity sensors such as the Sharp IR rangers or *MaxBotix* ultrasonic sensors. Each module connects to up to 4 sensors. Calibration and data filtering algorithms run on the microcontroller, which publishes distance measurements.

[1] http://github.com/openrobots-dev.



(a) R2P IMU module. (b) R2P DC motor controller module.

Fig. 6. PCB layout of two R2P modules with dimension expressed in millimeters.

Analog and Digital Input/Output module. A generic analog and digital input/output module has been designed, too: it can be used to interface the R2P network with generic sensors and switches or to connect existing electronic devices. It features input protections and implements most common integrated circuits buses (e.g., SPI and I^2C), as well as analog input and outputs, to easily integrate custom devices into the framework.

Smart Camera module. The smart camera module has a VGA CMOS sensor and a powerful ARM microcontroller with DSP capabilities to run vision algorithms on board. At present, it can run algorithms to recognize colored blobs and human faces, and to implement image segmentation and optical flow [42]. It can publish the image analysis results on the CAN bus.

Optical Odometry module. To add position feedback to existing robots, we have designed an optical odometry module that exploits cheap mice sensors pointed to the ground to measure movements [43,44]. With this board, precise trajectory control can be added without intrusive modification to the hardware, exploiting movement estimations published on the CAN network.

Gateway module. Finally, the gateway module features an Ethernet port and a more powerful, Ethernet-enabled, microcontroller to handle the TCP/IP stack. R2P messages are forwarded from the CAN bus to the IP network, and the other way around. The gateway module runs μROSnode which enables a direct integration of R2P modules with ROS systems.

5. Case study: the Triskar2 robot

We used the Rapid Robot Prototyping framework to actuate and control the mobile robot *Triskar2*, shown in Fig. 7. Triskar2 is an omnidirectional robot, with three *omni* wheels driven by three 70 W DC motors. It is built from standard, modular, aluminum profiles, allowing an easy attachment of additional elements. The platform is 60 cm in diameter, weights about 20 Kg, with batteries, and is designed to carry a payload up to 50 Kg at a maximum speed of 2 m/s. Triskar2 is built with R2P modules, which also run the low-level embedded software for motion control; high level behavior comes from an ROS application, which directly interfaces with the mobile platform. We designed Triskar2 as versatile and fast-to-implement mobile base for assistive robots, robots able to interact with people, robogames, and, in general, indoor robots.

5.1. Hardware modules

The architecture of Triskar2, with the R2P modules used to drive the robot, is shown in Fig. 1(a). We started from the functional requirements of the platform: it must provide precise motion, with

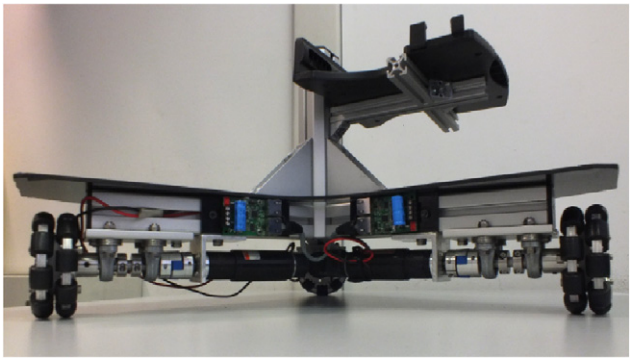


Fig. 7. The Triskar2 omnidirectional robot.

low-level obstacle avoidance, accepting commands from an ROS application running on the on-board computer.

First of all, we used a PSU module to power the other R2P modules from the 24 V batteries. To drive the electrical motors we used three DC Motor Control modules, which also acquire actual motor speeds by the optical encoders attached to motor shafts. To detect approaching obstacles, four Sharp IR proximity sensors are attached to a R2P Proximity module, which reads the output of the sensors and converts it to distance measurements. Finally, a Gateway module is in charge of proxying messages from the ROS application to the R2P network, and the other way around. Thanks to their simple wiring and daisy-chain connections, mounting the R2P modules on the robot frame was a straightforward process that required few hours.

5.2. Software architecture

The architecture of the embedded control software is reported in Fig. 5. Motion commands are sent by an ROS node running on the on-board computer on the */teleop* topic, which is subscribed by the Gateway module running *uROSnode* and forwards messages on the corresponding R2P topic. For low-level collision avoidance, the obstacle detection node subscribes for both proximity measures and velocity commands; if an obstacle is detected on the requested path, the command is overridden and the filtered setpoint is published on the */velocity* topic. At this point, the kinematics node receives the requested motion, applies the inverse kinematics model, and computes wheel speed set points in a 100 Hz loop. Finally, each motor board executes a *motor controller* node, which drives the motors by the closed-loop PID controller running at 200 Hz. The actual wheel speed is published by the motor controllers and subscribed by the odometry node, which computes the robot trajectory by applying forward kinematics. An additional node monitors the battery level, to estimate the remaining battery life. The */proximity*, */trajectory* and */battery* topics are subscribed by the gateway module, which forwards them to the ROS network.

Software nodes have been deployed on the modules as shown in Fig. 1(a). Some nodes have to be on a particular board (e.g., those that are directly connected to the hardware like motor controller nodes), while others can run on any connected module. For example, in our tests, the inverse kinematics model to compute wheel speeds was run on the *Motor* 1 module, while the odometry node was deployed on *Motor* 2. In this way, we can balance processor load and reduce latency, easily moving nodes from a hardware module to another.

5.3. Benchmarks

We have run some benchmarks to evaluate communication performance, which is a key element in a distributed architecture like R2P. For a more detailed analysis of the results please refer to [22].

First of all, we evaluated RTCAN performance in terms of jitter of HRT messages and of delivery latency of SRT messages. Fig. 8(a) reports the distribution of transmission jitter for HRT messages from the inverse kinematics node, which have a period of 10 ms. Results show that the jitter is bounded to ± 3 μS, enabling to run distributed control loops. For SRT messages, we evaluated the delivery latency of messages from the obstacle detection node, which are event-triggered and have a 10 ms deadline. Fig. 8(b) reports the distribution of delivery latency: 95 of messages are received within 5 mS, allowing for fast event response.

As R2P fosters the development of modular, reusable, embedded software, where basic functionalities are implemented by nodes exchanging messages through the middleware, also locally, we evaluated its messaging performances. The benchmark considers the complete life of a message, from its allocation by the publisher node to the consumption by all the subscribed nodes. The graph shown in Fig. 9 reports the measured throughput and the delivery latency, with respect to the number of subscribers, for local subscriptions. We measured a maximum throughput of 86,600 msg/s, which lowers increasing the number of subscribers with a quadratic shape; this is due to the publishing time which increases linearly with the number of subscribers. The overhead introduced by the middleware is less than 7 μS, while the remaining time is due to thread scheduling and context switch by the RTOS. Locally, thanks to memory sharing, the maximum throughput does not depend on message size. For remote subscribers, performance highly depends on the communication channel; using RTCAN, periodic messages are sent as HRT RTCAN messages, and thus the jitter is almost the same as for local subscriptions, while the delivery latency is degraded by the message transmission time over the CAN bus.

6. Conclusion

Robotics is an active research and engineering field that, against all the expectations and forecasts (e.g., [45]), and unlike other recently wide spread technologies (e.g., hand held computers, smart phones, digital cameras) is struggling to generate many products for the real life market. One of the main problems is the wasteful effort required to build working hardware prototypes, which subtracts precious resources, in terms of time and money, for the true development of new ideas.

In this paper, we have presented R2P, an open source modular hardware and software architecture for rapid prototyping of robots, which consists of a set of easy-to-use hardware components, a software development IDE, and deployment tools. R2P aims at supporting the development of robotics applications, by reducing the time and efforts needed to build a working prototype. R2P includes features which make it interesting for a wide range of users, such as researchers, designers, students and robotic enthusiasts in general. Research groups, like university laboratories, and R&D departments, can take advantage of such a framework, reducing the time to implement a working robot prototype which is necessary to validate any innovative idea. Working with a toolkit of robotic components that can be assembled in hours, instead of weeks, leaves much more time to work on specific application issues. With R2P, robot designers can face the implementation of a robot prototype also if they do not master all the technical knowledge otherwise needed. This makes it possible to include in the robot application development team, with an operative role, also users and domain experts, otherwise excluded, with evident positive effects on the generation of new, interesting applications. Students and people involved in education can exploit plug-and-play hardware modules and easy-to-use software tools, like a scripting language and a visual development environment, which enable

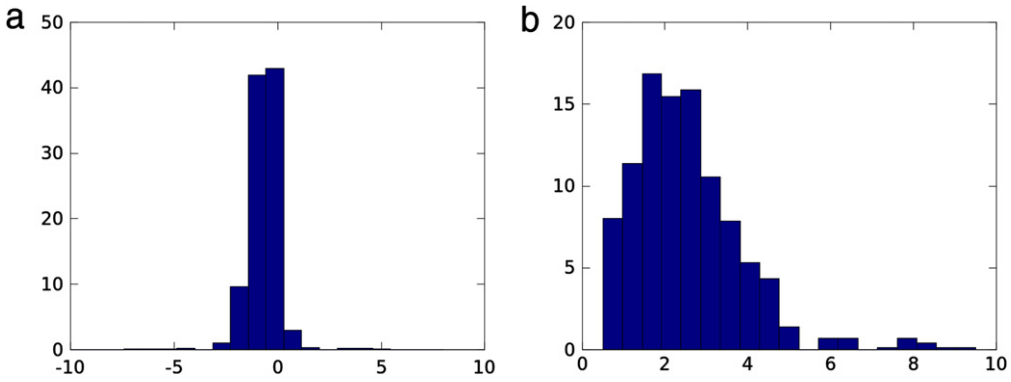


Fig. 8. RTCAN performance: distribution of HRT messages transmission jitter, expressed in μs (a), and distribution of SRT messages transmission latency, expressed in ms (b).

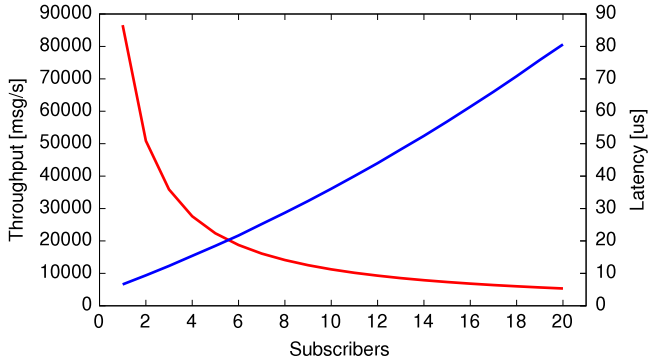


Fig. 9. R2P middleware performance: maximum message throughput, in red, and delivery latency, in blue, with respect to the number of subscribers. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

robotics-based activities at school. Finally, robotic enthusiasts can rely on an open source low-cost framework that can boost the development of new ideas and foster users entrepreneurship, leading to innovative products as shown by success business stories like Arduino [40,37].

Acknowledgments

This work has been partially supported by the research grant "Robotics for the Masses" from ST Microelectronics and Regione Lombardia, and by the Italian Ministry of University and Research (MIUR) through the PRIN 2009 grant "ROAMFREE: Robust Odometry Applying Multi-sensor Fusion to Reduce Estimation Errors".

References

[1] M. Muffatto, Introducing a platform strategy in product development, International Journal of Production Economics 6061 (1999) 145–153.
[2] G. Heineman, W. Councill, Component-Based Software Engineering: Putting the Pieces Together, Addison Wesley Professional, 2001.
[3] B.P. Gerkey, R.T. Vaughan, A. Howard, The player/stage project: tools for multi-robot and distributed sensor systems, in: Proceedings of the 11th International Conference on Advanced Robotics, 2003, pp. 317–323.
[4] H. Bruyninckx, Open robot control software: the OROCOS project, in: Proceedings 2001 ICRA, IEEE International Conference on Robotics and Automation, 2001, pp. 2523–2528.
[5] M. Quigley, K. Conley, B.P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, ROS: an open-source robot operating system, in: ICRA Workshop on Open Source Software, 2009.
[6] A. Huang, E. Olson, D. Moore, LCM: lightweight communications and marshalling, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2010, pp. 4057–4062.
[7] J. Kramer, M. Scheutz, Development environments for autonomous mobile robots: a survey, Autonomous Robots 22 (2) (2007) 101–132.
[8] J. Kaiser, C. Mitidieri, C. Brudna, C.E. Pereira, COSMIC: a middleware for event-based interaction on CAN, in: 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2003, pp. 669–676.
[9] M. Schulze, S. Zug, A middleware based framework for multi-robot development, in: Proceedings of the 3rd IEEE European Conference on Smart Sensing and Context (EuroSSC), Zurich, Switzerland, 2008, pp. 29–31.
[10] S. Magnenat, P. Rtornaz, M. Bonani, V. Longchamp, F. Mondada, ASEBA: a modular architecture for event-based control of complex robots, PIEEE/ASME Transactions on Mechatronics (2011) 321–329.
[11] Arduino, http://www.arduino.cc.
[12] Lego NXT, http://mindstorms.lego.com.
[13] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, A. Martinoli, The *e*-puck, a robot designed for education in engineering, in: Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions, vol. 1, 2009, pp. 59–65.
[14] R.M. Harlan, D.B. Levine, S. McClarigan, The khepera robot and the krobot class: a platform for introducing robotics in the undergraduate curriculum, in: ACM SIGCSE Bulletin, vol. 33, ACM, 2001, pp. 105–109.
[15] A.B. Holger, H. Kenn, T. Walle, Robocube a "universal" "special-purpose" hardware for the robocup small robots league, in: 4th International Symposium on Distributed Autonomous Robotic Systems, Springer, 1998.
[16] Lego NXT, http://www.arexx.com/rp6/.
[17] A. Bonarini, M. Matteucci, M. Migliavacca, D. 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), 2012.
[18] D.A. Norman, The Design of Everyday Things, Basic Books, New York, NY, 2002.
[19] A. Bonarini, M. Matteucci, M. Migliavacca, R. Sannino, D. Caltabiano, Modular low-cost robotics: what communication infrastructure?, in: Proceedings of 18th World Congress of the International Federation of Automatic Control (IFAC), 2011, pp. 917–922.
[20] Robert Bosch GmbH, CAN Specification 2.0B, http://www.semiconductors.bosch.de/media/pdf/canliteratur/can2spec.pdf, 1991.
[21] ChibiOS/RT Real Time Operating System, http://www.chibios.org.
[22] M. Migliavacca, A. Bonarini, M. Matteucci, Rtcan: a real-time can-bus protocol for robotic applications, in: Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics (ICINCO 2013), 2013.
[23] P. Pèrez, G. Benet, F. Blanes, 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, 2003, pp. 237–243.
[24] L. Almeida, P. Pedreiras, J.A.G. Fonseca, The FTT-CAN protocol: why and how, in: IEEE Transactions on Industrial Electronics, IEEE Press, 2002, pp. 1189–1201.
[25] T. Nolte, M. Nolin, H. Hansson, Server-based scheduling of the can bus, in: Proceedings of the 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'03), IEEE Press, 2003, pp. 169–176.
[26] J. Coronel, F. Blanes, G. Benet, J. Simó, P. Pèrez, M. Albero, CAN-based distributed control architecture using the SCoCAN communication protocol, in: 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2005.
[27] G.C. Buttazzo, Rate monotonic vs. EDF: judgment day, Real-Time Systems 29 (2005) 5–26.
[28] C. Lu, J. Stankovic, S. Son, G. Tao, Feedback control real-time scheduling: framework, modeling, and algorithms, Real-Time Systems 23 (2002) 85–126.
[29] M.A. Livani, J. Kaiser, EDF consensus on CAN bus access for dynamic real-time applications, in: IPPS/SPDP Workshops, 1998, pp. 1088–1097.
[30] J. Kaiser, M.A. Livani, Invocation of real-time objects in a can bus-system, in: Proceedings of the The 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE Press, 1998, pp. 298–307.
[31] M.D. Natale, Scheduling the can bus with earliest deadline techniques, in: Real-Time Systems Symposium, 2000, Proceedings, The 21st IEEE, 2000, pp. 259–268.
[32] Embryo Virtual Machine, http://trac.enlightenment.org/e/wiki/Embryo.
[33] Pawn Scripting Language, http://www.compuphase.com/pawn/pawn.htm.
[34] Embedded LUA, http://www.eluaproject.net/.
[35] Python-on-a-Chip, http://code.google.com/p/python-on-a-chip/.

[36] Y. Shi, K. Casey, M.A. Ertl, D. Gregg, Virtual machine showdown: Stack versus registers, ACM Transactions on Architecture and Code Optimization 4 (2008) 1–36.
[37] M. Banzi, Getting Started with Arduino, Make:Books, Sebastopol, CA, 2008.
[38] Open Source Hardware Association, http://www.oshwa.org.
[39] Android Accessory Development Kit 2012, http://developer.android.com/tools/adk/adk2.html.
[40] C. Thompson, Build it. share it. profit. can open source hardware work?, Wired Magazine 16.11.
[41] R.G. Abbondandolo, Open source hardware fostering user entrepreneurship: empirical evidence from Arduino users, Master Thesis, Economics and Management of Innovation and Technology, Bocconi University, 2011.
[42] V. Rana, M. Matteucci, D. Caltabiano, R. Sannino, A. Bonarini, Low cost smartcam design, in: Proceedings of 6th IEEE Workshop on Embedded Systems for Real-tme Multimedia (ESTIMedia 2008), IEEE Computer Press, 2008, pp. 27–32.
[43] A. Bonarini, M. Matteucci, M. Restelli, A kinematic-independent dead-reckoning sensor for indoor mobile robotics, in: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2004, pp. 3750–3755.
[44] B. A., M. Matteucci, M. Restelli, Automatic error detection and reduction for an odometric sensor based on two optical mice, in: Proceedings of the International Conference on Robotics and Automation (ICRA2005), IEEE Computer Press, 2005, pp. 1675–1680.
[45] B. Gates, A robot in every home, Scientific American (1) (2007) 58–65.