Journal of Software Engineering for Robotics 3(1), March 2012, 13-27
ISSN: 2035-3928

Design Choices for Modular and Flexible Robotic Software Development: the OpenRDK viewpoint

Daniele CALISI[1] Andrea CENSI[2] Luca IOCCHI[1] Daniele NARDI[1]

[1] Department of Computer and Systems Science, "Sapienza" University of Rome, Via Ariosto, 25 - Rome 00185, Italy
[2] Control & Dynamical Systems Department, California Institute of Technology, 1200 E. California Blvd., 91125, Pasadena, CA, USA

Abstract—Developing reliable robotics applications is a difficult and resource-consuming task. The scientific community is undertaking several initiatives to devise standard design techniques, and the deployment of reusable and interoperable components. At this point in time, a variety of approaches for robotic software development have been proposed, due to the wide range of domains where robots are used, the many forms and functions that a robot can have, and the diversity of people involved in robotics. Specifically, we focus on modularity and rapid development of distributed robotic systems. First, we survey the main issues in developing software frameworks for robotics, and we briefly discuss existent approaches, highlighting their goals, advantages and weaknesses. Then, we address the most significant design choices that arise in the implementation of robotic software and motivate the specific approach taken in OpenRDK, a software framework developed in our laboratory. Finally, we describe how the flexibility of OpenRDK allows to develop robotic applications using different paradigms. In conclusion, we argue that the specific point in design space provided by OpenRDK can be successful for a significant class of robotic software development efforts.

Index Terms—Robotics framework, software development, information sharing, interoperability.

1 Introduction

Developing reliable *robotic applications*, i.e., the software that is needed to realize robotic systems, is a challenge. For example, developing complex tasks in challenging environments for a team of heterogeneous robots leads to a large distributed software system. Consequently, robotic applications become complex software systems.

In this context, formal methods used in automation for developing distributed control systems, such as the IEC 61499 standard [1], [2], are not designed to handle complex robotics applications, because they are focused on relatively simple control systems. At the same time, conventional approaches to software development might not be effective, for a variety of reasons. First of all, they have to cope with several *technical challenges arising from the deployment environment*: issues such as network unreliability, real-world uncertainty and unpredictability are typically not present in conventional software applications. Moreover, network unreliability may also have a severe impact on overall performance of the system. Finally, the computational units that belong to the distributed system may have different capabilities, can be combined in several settings and can interact in a variety of ways. The resulting system is a *highly heterogeneous distributed system*, that requires a palette of solutions and patterns for concurrent processing and information sharing.

Robotic applications range over a *very wide spectrum*: robots are employed in an increasing number of domains, and can have a variety of forms and functions, that must be achieved through new forms of modularization and reuse of existing components. *Standardization* is not yet fully reached, in particular for applications that go beyond industrial robotics, making difficult to realize an effective interoperability of solutions developed for different problems. Finally, the *human resources* that are involved in robotic application development are rather diverse [3], often lacking skills and experience to deal with large software systems.

The scientific community is undertaking several initiatives to improve and rationalize the development of robotic applications. There is a need to develop standard design techniques, deploy reusable and interoperable components [4], [5], that can effectively support robotic software development for a variety of applications and development teams. These efforts have different goals and therefore typically address different aspects of the overall enterprise. For example, the Robot Operating System (ROS [6], [7]), which is nowadays

Regular paper – *Manuscript received April 14, 2011; revised January 27, 2012.*

commonly used by many research groups, has a specific focus on collaborative development. ROS is both a common layer of drivers that access low-level hardware interfaces, expose their services and data using common interfaces and provides a peer-to-peer communication infrastructure, and a repository of software components specifically targeted at robot software development. While ROS provides an invaluable support to the development, our goal is to address the application layer, possibly in a way that is independent of the underlying operating system or middleware.

The goal of the present paper is to discuss what we consider major design choices in the implementation of a software framework for robotics, through our experience in designing OpenRDK[1], a software framework for robotics, that we have been developing in the last ten years and used in several applications, on a variety of robots and computational devices.

Our aim, however, is not to introduce yet another framework for the development of software for robotic applications; rather, we focus on key design choices by discussing their consequences in terms of both their impact on software development and of the challenges in implementing them. Our specific goals in the development of robotic software put special emphasis on *distributed/decoupled* development, *component re-use* and *rapid prototyping*. Consequently, we identified the following key features that a framework for robotic software development must address: concurrency model, information sharing model, support tools, and interoperability. Although other relevant design criteria and implementation features for robotic software development can also be considered (for example, component management, parameter configuration, etc.), we believe that the ones considered in this article have a significant impact on the software development cycle.

In this paper, we first survey the state-of-the-art development environments with respect to the above mentioned features. Then, we provide a detailed discussion of specific solutions that are supported by OpenRDK, including examples that highlight their advantages.

Two remarks are necessary in order to put the analysis carried out in the paper in a correct perspective. First, we are not concerned with the issues that arise in real-time software design, since we assume that the real-time behavior of the system can be achieved through specialized components, designed with known tools and methodologies. Instead, we focus on the complexity of designing anything else that is needed for the robotic application in addition to the real-time components. Second, any design problem is centered on the needs of a development group, whose aim is to create innovative technical solutions for complex mobile robotic systems, trying to capitalize on the software infrastructure. Our reference development team is composed by members with heterogeneous skills and experience, working for a short time on a complex project (e.g., multi-robot heterogeneous exploration in a disaster scenario), usually with different background, skills and experience in developing robotic applications. Thisnotwithstanding, our aims have a significant overlapping with a large class of efforts and the analysis and solutions presented are relevant to several robotic software development enterprises.

The paper builds on a preliminary analysis started in [8] and is organized as follows. In Section 2 we analyze the main characteristics of a software framework for robotics, reviewing other existing frameworks. Section 3 describes the main choices that impact on the design of a software framework for robotics, and discuss in detail the specific solutions that are provided by OpenRDK. Section 4 shows how the flexibility achieved by OpenRDK allows to develop applications according to multiple paradigms. Section 5 concludes the article with a discussion of future directions.

1. http://openrdk.sourceforge.net

2 ISSUES IN ROBOTICS SOFTWARE DEVELOPMENT

Generally speaking, one of the goals of engineering is to reduce the complexity of a problem by dividing it into smaller problems (divide et impera). This has many benefits in software design and development, especially when dealing with complex architectures, such as software for robotics. In software, this practice leads to dividing an application in smaller software modules: mutually decoupled software units with well-defined interfaces.

In this article, we focus on those frameworks that deal only with the software development and that do not force a specific conceptual architecture (i.e., component arrangement, organization and responsibilities).

This section describes the main issues that software frameworks for robotics should tackle. In particular, we describe two important design criteria: concurrency model and information sharing model, as well as two important development capabilities: tools and interoperability. Since these choices are often strictly inter-related, we analyze in detail the direct and indirect effects of these choices. Moreover, we describe the rationale followed by some existing software frameworks for robotics (summarized in Figure 1) and the consequences on their features and drawbacks.

2.1 Concurrency model

In every modular architecture, there are three main approaches to address concurrent module execution.

- Modules are *processes* distributed over one or more machines. In this case, developers have the greatest freedom (including using different languages and operating systems), with respect to the other possible choices. The major drawback of the process approach is the need of some communication infrastructure that allows for inter-process communication. Performance becomes an issue when not using shared memory and accessing inter-process shared memory requires special attention to avoid deadlocks.

- Modules are *threads* inside a single process. With multi-threading, sharing information is easily accomplished using shared memory, using some mechanism for concurrent data access and thread synchronization. The thread-based paradigm seems to be a reasonable compromise between efficiency and ease of use, but a pure thread-based approach, where the whole application resides on a single process can be excessively restrictive, for example in particular situations that require to distribute the computation over different processes, or over different machines.
- Modules are built as *call-back functions* and there is a single process (i.e., a scheduler) that repeatedly calls them either in response to some event or periodically. The call-back functions allows for a more accurate execution control and thus are preferable for low-level device interaction; however, the control cycle must be carefully designed by the programmer and should adapt to satisfy the time constraints, also by splitting long computation in multiple calls, which is not easy to achieve.

There are two issues to be considered when considering concurrency models. The first one is the *ease of development*: on this axis, the callback-function paradigm is the less user-friendly, because the programmer must deal with distributing the computation over several calls, while, at the other end of the spectrum, the process-based architecture allows for the largest freedom in development, because different processes are independent. The second issue is *inter-module communication*: on this axis, the preference order is reversed, since the call-back function mechanism allows to exchange information using shared memory, without any need for data integrity management; using processes, an inter-process communication mechanism is needed to ensure data integrity.

Nevertheless, these choices are not mutually exclusive, and some frameworks make use of a hybrid solution, allowing different models to be used at the same time and to be integrated in a single system. Another important issue, from the point of view of the developer, is if the framework can be realized in such a way that the actual concurrency model used is transparent to the user. This introduces additional complexity in the design and implementation of the framerwork and it is not yet fully achieved in existing frameworks.

The distributed process model is a very common choice: most existing frameworks use this solution (e.g., Orca [9], [10], ROS [6], [7], SPICA [11], etc.), since they choose to give as much freedom as possible to module development, so that developers have a high degree of design and implementation freedom. For example, ROS nodes are independent heterogeneous (i.e., developed using different programming languages) components that exchange data through pre-defined messages and services. Similarly, SPICA allows for developing loosely coupled heterogeneous components and provides an ontology-based modelling language used as a common basis for information exchange.

OROCOS [12], [13] is focused on low-level robot and machine control and implements a concurrency model that has several variants: a module (called "activity" at run-time in OROCOS) is executed in a separate thread and may be event-driven (i.e., a function is called every time a particular event is fired) or implemented as a call-back function that may be requested to satisfy real-time constraints. OpenRTM-aist [14]

Framework	**Concurrency model**	**Information sharing**	**Tools**	**Focus**
OROCOS	call-backs, threads	lock-free data ports (CORBA)	remote inspection, logging	low-level interaction with devices
Orca	processes	ICE	remote inspection, logging	mobile robots
IPC/TDL	processes	IPC	none	generic event-driven architectures
CARMEN	processes	IPC	logging, visualization	mapping and navigation
Cognitive Map	none specified (user developed)	Whiteboards	none (?)	humanoid robots
OpenRTM-aist	threads or callbacks	CORBA	configuration GUI	general robotics
Microsoft Robotics Studio	processes	HTTP/DSSP via DSS	3D simulator	general robotics
Player	threads (server), processes (clients)	client/server, proprietary over TCP	2D and 3D simulators	low-level device drivers
MOOS	processes	centralized, proprietary over TCP	logging, viewers	mobile robots
CLARAty	relies on ACE (threads, processes)	relies on ACE	none (?)	real-world systems
MARIE	processes	many (3rd party)	configuration GUI	connecting different frameworks
MOAST	processes	NML	logging, visualization	USARSim, mobile robots
MIRO	processes	CORBA	logging	mobile robots
ROS	processes (call-backs via nodelets)	proprietary protocol	many	general robotics
SPICA	process based	proprietary protocol	code-generation tools	mobile robots
SPQR-RDK	call-backs, threads	proprietary over TCP	remote inspection GUI	mobile robots
OpenRDK	threads	shared memory, proprietary over TCP/UDP	remote inspection GUI, logging	mobile robots

Fig. 1. Summary of existing software robotic frameworks: architecture choices, tools and focus

(a RT-Middleware [15] implementation) and SPQR-RDK [16] use a very similar architecture. CLARAty [17], [18] relies on ACE (Adaptive Communication Environment[2]) for modules spawning at run-time: its modules can be either processes or threads.

In Player [19], [20] there are two types of modules: they are either threads inside the main Player process (they are usually low-level device drivers), or they are distinct processes connecting through a client/server mechanism.

OpenRDK modules are mapped to threads, and the framework provides all concurrency management primitives (e.g., memory locks) for a multi-threading environment. Moreover, as shown in Section 3, an OpenRDK application can be formed by several processes each one including different modules. From the user point of view, communication among the modules is uniform independently of whether they are threads of the same process or they are threads on different processes (possibly running on different machines). The use of threads is a good compromise between efficiency and ease of development, given our goals of rapid prototyping, modularity and code-reusability. In fact, the callback-based mechanism allows for the highest scheduling control (e.g., for real-time purposes), but it requires skilled developers to implement modules. On the other hand, in process-based solutions the communication between modules must be carefully designed.

2.2 Information sharing

Another key issue in a software framework, strictly related to the concurrency model, is how to share information among modules. This aspect can be addressed from two different points of view: the first one is the semantic "metaphor"; the second one is how this mechanism is actually realized.

In the following we revise the main communication metaphors and then we summarize how they can be realized.

2.2.1 Information sharing metaphors

Modules need to exchange data as they execute their computations: there are two possible paradigms that can be considered. The first option is to allow data to be exchanged through a centralized entity, often called *blackboard*, where modules can publish and retrieve data: modules read their inputs from the blackboard and then publish outputs for other modules. The second paradigm is based on the concept of *data-ports*: a module reads from its input ports, processes the information and pushes the results to its output ports; a cascaded connection between input and output ports make them work as filters.

Notice that *blackboard* and *data-ports* are related each other. Both the two mechanisms can be implemented on a framework using the same underlying communication system. However, from the point of view of the developer, there is a clear distinction between their use: when data-ports are used it is necessary to explicitly connect input ports of one module with output ports of another module, while with a blackboard this connection is demanded to a system component that is in charge of the management of publishers and subscribers.

Using a blackboard-based paradigm, modules read inputs from the blackboard and publish their output after each iteration, using unique names to identify them. The blackboard-based paradigm is often adopted when large objects have to be shared (e.g., images) and module execution can be synchronized with respect to input updates (i.e., a module is run after its input is updated by other modules) and it is usually realized by shared memory. This approach has two benefits: 1) it provides a simple way to introspect and debug the execution of the system, as there is a centralized object where all relevant information is accessible; 2) a set of well-known names has an implicit meaning for the developer (and the debugger), who knows how to find the data of interest; 3) if modules publish also their internal state on the blackboard, together with input/output data, it is possible to freeze the system by saving the whole blackboard and resume the execution by reloading this snapshot later on.

The blackboard metaphor allows for state inspection, freeze and resume, but a pure blackboard-based system can exacerbate the design of a robotic system for which the data-ports metaphor is more suited. If the framework provides only a centralized blackboard paradigm for information sharing, the solution is typically to define well-known places in the blackboard where a producer and a consumer can exchange data. This creates an unnecessary coupling between them and it prevents the possibility of introducing an intermediate module that further processes the information before the last module uses it (e.g., the robot pose that is computed by a localization module may need a subsequent filter module before it can be used by a navigation module). Another concern of a pure blackboard style communication method is the difficulty in dealing with situations where not only the last updated value is needed, but the whole history of changes. A typical example are the sensor readings used by a SLAM algorithm: losing some of them (for example for scheduling issues) can prevent the algorithm to build a consistent map. The producer/consumer model of data sharing is well-known in computer science. It is usually solved by some (possibly bounded) data queue, i.e., containers with FIFO (first-in first-out) accessing paradigm: the producer pushes pieces of data into the queue and the consumer retrieve them. Finally, a data-sharing mechanism is also needed to allow multiple producers/consumers: for example, the sensor queue can be used not only by the SLAM algorithm, but also by another module that is schedule at a different rate.

The data-ports paradigm is best suited for asynchronous execution (modules wait for data coming from their input ports), when modules need the whole history of updates for a given input (e.g., ports are implemented as buffer queues), or when a socket-based communication is used to share data

2. http://www.cs.wustl.edu/~schmidt/ACE.html

among modules. Many architectures in robotics are composed by cascades of filters. Modules can be seen as filters of data flowing from their inputs through their outputs, they are connected in cascade. This is the typical example where the data-ports paradigm is preferable.

The potential disadvantages of a pure data-port solution derive from the lack of a common place (addressing space) where all data can be uniformly accessed. This makes it more difficult to inspect and log data and may lead to unnecessary duplication and exchange of data.

Summarizing, both data-sharing paradigms have application to specific situations, and thus a software framework for robotics should provide the means to accommodate both. From our point of view, the blackboard metaphor better supports distributed software development, facilitating access and sharing of information; consequently, it has a central role in OpenRDK. On the other hand, several frameworks, such as OpenRTM-aist, OROCOS, Orca, IPC/TDL, etc., mainly support the data-ports paradigm. Among the few existing frameworks that make use of a pure blackboard-based information sharing model. In particular, SPQR-RDK requires modules to publish their data in a Shared Information Register, in well-known places where other modules can read it. Cognitive Map [21] uses the Psyclone Whiteboard system [22], that uses a blackboard-type information sharing mechanism, where the blackboard can also host shared data streams. Player [19], [20] has a dedicated blackboard interface and a local-memory implementation that allows clients to read and write named pieces of data that reside in the Player server. A similar centralized approach is used also by The MOOS framework [23] (Mission Oriented Operating Suite) where modules are arranged in a star-like topology, and publish their information in a centralized blackboard called MOOSDB. It is worth noticing that recently also ROS adopted a centralized publish/subscribe blackboard mechanism implemented through messages on a proprietary communication protocol.

OpenRDK, specifically, provides both the blackboard and the data-ports paradigms on the same data, as will be better explained in Section 3. An object called "repository" is used as a blackboard, where modules publish and retrieve the relevant information. Each piece of information, being input, output, parameter or state information, is called "property". A framework feature, called "property links", allows for connecting outputs of modules to inputs of other modules, thus realizing also the data ports paradigm. In this way, OpenRDK can take advantage of both centralized blackboard-type communication and data-ports-based semantics on the same data. To our knowledge, it is the only framework that integrates the two paradigms at the same time on the same data, without requiring the module developer to take care or to know how these data will be actually delivered to other modules.

2.2.2 Information sharing implementation

From the implementation point of view, in practice, there are two main mechanisms that can be used: modules within the same process (i.e., if they are realized as call-back functions or threads) usually rely on *shared memory*, while modules distributed among different processes/hosts typically use some *inter-process communication* service.

Shared memory is the most efficient communication method, in particular when large data structures need to be shared (e.g., images, maps). In the case of threads, concurrency management primitives are necessary. Moreover, shared memory itself is not enough to provide specific semantics to access shared data: for example, if two modules act as a producer/consumer couple, the framework must provide a mechanism to implement some kind of data "queue" in the shared memory.

The choice of distributing modules over multiple processes and machines prevents information sharing to be implemented purely with shared memory. This makes it necessary to introduce an additional communication mechanism that works between different hosts. However, this feature should be provided by the framework in a transparent way with respect to the developer of functional modules. In fact, the interconnection layout must not be defined during the functional module development, since it introduces unneeded complexity, by coupling the *computation function* and the *computation layout*.

The shared memory can be effectively used also to implement both the blackboard model and the data-ports one. While implementing a blackboard model using a shared memory is straightforward, the data-ports paradigm requires some considerations. One solution is to use queue collection structures, possibly protected with a locking mechanism to avoid concurrent accesses to the queues themselves. However, this does not allow for multiple readers and can be inefficient due to the necessary lock of the whole collection of objects for each operation. Active processes can be used instead: they are in charge of dispatching objects directly to the consumer(s), when they are pushed by the producer(s). The main drawback of this method is the requirement that active objects need to be scheduled themselves, in addition to the functional modules.

In the case of multiple processes, there are many existing solutions both for processes on the same machine (e.g., DCOP, DBUS, IPC, COM+) and across a network (e.g., CORBA). Regarding this issue, the Object Management Group[3] (OMG) is currently working to the new version of the Data Distribution Service (DDS) specification, a standard that is becoming accepted by a great number of companies. DDSs are middleware systems aiming at distributing data in real-time among applications, using the publish/subscribe paradigm. The most notable DDS implementations are OpenSplice[4] and

3. http://www.omg.org
4. http://www.prismtech.com/section-item.asp?id=175&sid=18&sid2=10

RTIDDS[5], which adhere almost completely to the OMG DDS specification.

IPC (Inter Process Communication) [24] and TDL (Task Description Language) [25], [26] are two software packages developed at CMU that, by extending C++ semantics, provide for a technique to send C++ objects through a network between servers and clients and for the definition of call-back procedures that are called when some event (e.g., some packet arrival) occurs. Although IPC is general enough to be used in any kind of application that is based on asynchronous events, its main uses have been in robotic applications.

Among frameworks that provide their own inter-process communication infrastructure, Microsoft Robotics Studio uses its own Decentralized Software Services (DSS) in order to achieve inter-service communication. DSS works over TCP links using plain HTTP protocol or an XML-based protocol called DSSP (DSS Protocol). Also the connections between the Player server and the clients, and between the MOOS modules and the MOOSDB blackboard, are implemented over a TCP link, using proprietary protocols, that, in Player, can be tuned to the application needs (e.g., frequency of updates, etc.). Also ROS uses its own message-based communication infrastructure between modules (both for data exchange and for service requests and responses). In MARIE [27], [28], a framework that encourages code reuse from other existing robotic projects, the main concept involved in modules arbitration and information sharing is the *mediator design pattern*, i.e., a central entity that is responsible of the interconnection and the use of data coming from the various modules, which, in turn, can be developed using any kind of library or framework available.

Regarding the use of third-party middlewares for communication, CORBA is used by MIRO [29], which is process based. OROCOS also uses CORBA to provide also inter-process communication through a couple of ad-hoc modules, and OpenRTM-aist provides CORBA transport facilities along with other possibilities (e.g., ICE, DDS, etc.). Other frameworks rely on other middlewares, such as ACE (Adaptive Communication Environment, that is used by, e.g., CLARAty) or ICE (Internet Communications Engine[6], that is used by, e.g., Orca. MOAST (Mobility Open Architecture Simulation and Tools) [30] makes use of the RCS/NML library[7] (Real-time Control Systems/Neutral Message Language).

These middlewares, anyway, aim at generality and multi-platform support, rather than ease of use and efficiency. Therefore, their use in robotic applications is not common and almost all the existing frameworks using a communication infrastructure provide its own middleware, that fits the specific needs of the robotic application. Unfortunately, all these middlewares cannot communicate to each other, thus preventing the use of modules developed in one framework within another framework, without writing suitable wrappers.

Summarizing, both shared memory and IPC are suitable techniques for information sharing: the first is more adequate for implementing a blackboard metaphor, while the second is naturally used for implementing data-ports. Among other advantages/disadvantages of these choices, we highlight that data-ports implemented with IPC mechanisms require additional computation and memory with respect to a shared memory implementation of a blackboard. When these data refer to large objects that are often updated, IPC-based data-ports either consume time and memory to send the whole large objects at every update or require them to be kept updated by the modules themselves (sending only updates) with ad-hoc procedures.

OpenRDK adopts a memory sharing approach, which is an obvious choice for a blackboard metaphor, while it provides two solutions for inter-process communication. The first one is based on a proprietary protocol over TCP or UDP, and can be tuned, in the configuration phase, by means of many options, as explained later in Section 3.2.4. As an alternative, OpenRDK allows also the use of 3rd-party middlewares (e.g., a DDS [31] or ROS communication protocol) in a transparent way for the developer.

2.3 Tools and interoperability

Tools are essential for a software framework, because they can speed up development and help in finding bugs and analyze behaviour and performance of implemented methods. Given the challenges in robotic software development, tools are even more critical. A first set of tools includes: (i) remote inspection tools to observe/modify the module state at run-time; (ii) on-line and off-line configuration utilities; (iii) logging and debugging tools for sensor data and sensor processing. Such tools are typically embedded in the software framework and their implementation can benefit from some of the above discussed design options. A second set of tool is concerned with complex components that are often available as add-ons. One typical example is given by robotic simulators. In this case, their accessibility can depend on the features for interoperability provided by the framework. In this section, we highlight some interesting approaches for making both types of tools available for software development.

Remote inspection can be a crucial utility, since it allows for on-line inspection and analysis of module behavior and often it includes also the possibility of modifying parameters on the fly, without the need to restart the application. For example, ROS includes a suite of tools that allows the developer to inspect the messages exchanged between nodes (modules), as well as logging and replaying facilities, while OpenRDK provides a specific utility for on-the-fly configuration and remote inspection of the modules behavior.

Regarding the configuration of single components and the whole system, OpenRTM-aist provides RTCLink, a graphical

5. http://www.rti.com/products/data_distribution/RTIDDS.html
6. http://zeroc.com/ice.html
7. http://www.isd.mel.nist.gov/projects/rcslib

tool to connect components and save a configuration for later use. MARIE can make use of RobotFlow[8], a 3rd party tool that allows for visual module integration and debugging. ROS and OpenRDK allow for specifying the components layout through XML files.

Almost all frameworks provide some facility to log data on a file to be analyzed or used off-line. Some frameworks include viewers that allow to visually analyze the saved logs. OpenRDK provides two configurable modules that can be used for logging and replaying data, as well as a specific visual tool for remote inspection, visualization and on-the-fly configuration.

When developing complex applications that require a long experimental validation, a realistic simulator can be handy. Since there are already many free and commercial simulators, each of which with its specific goals, the majority of software frameworks for robotics do not include their own simulator, but provide modules to connect to the majority of them (e.g., USARSim, Stage, Gazebo, Webots, etc.). This is easier when the framework supports interoperability (as discussed below). However, some frameworks provide their own simulator, for example Microsoft Robotics Studio, and Player that comprises in its package both a 2D (Stage) and a 3D (Gazebo) simulator. Other frameworks provide modules to interface to these or other simulators, for example MOAST is strictly connected with the USARSim simulator[9] [32].

The ability to interoperate with many other different components is not specific to the use of robotic simulators, but it spans over a large variety of components, being them small computational units, devices, or whole systems that are written by different teams using different frameworks. Full interoperability requires at least two elements: a standardization of protocols (how do systems communicate?), and a standardization of interfaces (what is the abstraction of a sensor? what is the interface of a localization module?). While protocol standardization is the most obvious issue, defining the interfaces is a problem that requires more careful thought. Common interfaces between components enforce modularity and code reuse. A great effort is being put on designing standard interfaces for typical components of robot systems. This includes standard data structures for sensor classes and standard interfaces for common robotic services. At present, this effort is ongoing and it is not clear if the goal will be reached through a formal standardization of data structures, services and algorithms for robotics [4], [5], [21] or a de-facto standard (e.g, ROS).

Regarding interoperability, the most notable example of existing frameworks is MARIE, whose main aim is code reuse from other existing projects: through the use of the mediator design pattern, the MARIE framework allows for an easy interconnection and use of data coming from the various modules, that, in their turn can be developed using any kind of library or framework available.

Another way to support interoperability is to rely on standard middlewares (e.g., CORBA, ICE, etc.). This allows for the communication with any middleware-aware framework, by requiring an agreement on the interfaces to be used to exchange information and on the semantics of the exchanged data. However, without a reference standard to implement, a practical approach is just to provide the “glue” that allows the use of several other libraries, tools or middlewares commonly used in practice in developing robotic applications. This is the approach chosen to make simulators available. The approaches that rely on well-specified structures for internal communication (that can be obtained with both the blackboard and the data-ports model), make interoperability with external components easier to achieve. In other cases, specific software components acting as bridges must be used (again both in the blackboard and the in data-ports model).

3 OpenRDK

In this section we address an implementation of the choice ”thread + blackboard”, as the basis for a framework for robotic applications. Our aim is to discuss the options available and the trade-offs between them, and propose the OpenRDK solution.

We structure the discussion following the main issues presented in Section 2: concurrency model, information sharing, tools and interoperability.

3.1 Concurrency Model

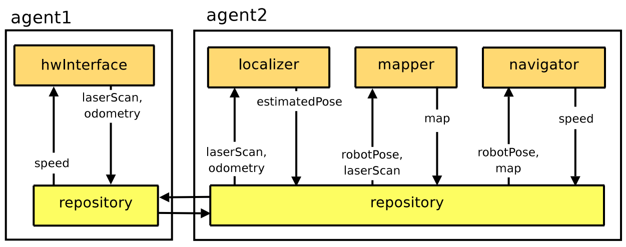
In OpenRDK, each **module** is a thread, executed inside a process called **agent**. OpenRDK uses POSIX Threads. Modules are pieces of code which are compiled as shared libraries, independently of OpenRDK’s core. At run-time, modules are instantiated by dynamically loading the shared library based on the configuration, specified as an XML file, that contains the modules parameters. A typical deployment of an OpenRDK application comprises several agents distributed on different machines, and several threads inside each agent.

In order to avoid the limitations due to a pure thread-based framework (see Section 3.2.4), OpenRDK provides also a mechanism that allows for a transparent inter-process communication. This means that, although our framework uses the thread-based paradigm, modules can reside on different processes and communicate as if they were in the same process. Figure 2 shows an OpenRDK applications formed by two agents (possibly running on two different machines) and a set of mobules communicating each other through the repository.

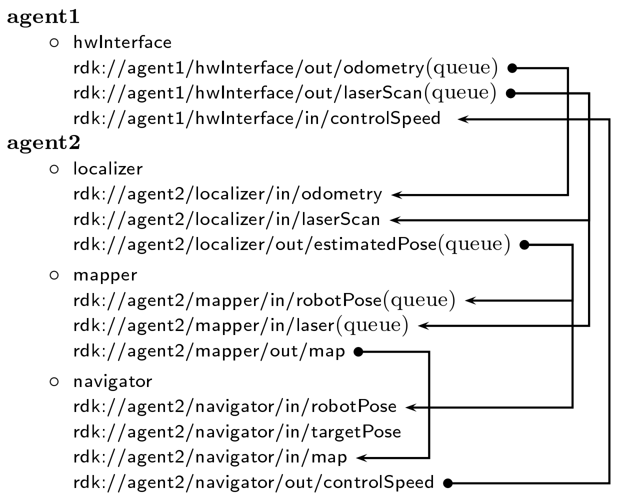
Finally, two features enrich module management in the OpenRDK framework. The first is a simple event-based scheduler that reduces the computation overhead: at run-time, each module (thread) is typically waiting for some event(s) to happen: new data to consume in a queue, the change of a

8. http://robotflow.sf.net
9. http://usarsim.sf.net

property value, a timeout, etc. The second is that modules are treated as "plug-ins" and can be loaded and instantiated at run-time.



(a) block diagram



```
agent1
  ○ hwInterface
    rdk://agent1/hwInterface/out/odometry(queue)
    rdk://agent1/hwInterface/out/laserScan(queue)
    rdk://agent1/hwInterface/in/controlSpeed
agent2
  ○ localizer
    rdk://agent2/localizer/in/odometry
    rdk://agent2/localizer/in/laserScan
    rdk://agent2/localizer/out/estimatedPose(queue)
  ○ mapper
    rdk://agent2/mapper/in/robotPose(queue)
    rdk://agent2/mapper/in/laser(queue)
    rdk://agent2/mapper/out/map
  ○ navigator
    rdk://agent2/navigator/in/robotPose
    rdk://agent2/navigator/in/targetPose
    rdk://agent2/navigator/in/map
    rdk://agent2/navigator/out/controlSpeed
```

(b) properties and links

Fig. 2. An example of two OpenRDK agents, running on two different machines: (a) shows a block diagram with named data flow; (b) shows the properties of each module and the links between them.

3.2 Information sharing model

The discussion of the information sharing model first addresses the general features of the blackboard model, then it addresses three additional issues arising in the implementation of the blackboard model: the use of data ports to model the typical cascade of filters that is needed for sensor data processing; the use of queues to enrich the blackboard model with capabilities to store the system evolution (as opposed to the last state); the efficient implementation of the information sharing model on different components of the heterogeneous distributed system.

3.2.1 Blackboard data-sharing metaphor: repository, properties and URLs

As already mentioned, the blackboard metaphor supports modularity by providing access to the state of computational structures. OpenRDK uses the metaphor of a centralized blackboard, called **repository**, where modules read and write data. Each piece of data is called a **property**. Properties can be inputs, outputs, state variables, or parameters for the module.

Each property has a URL of type rdk://agent/module/property, where the agent name is different from the host name as there can be more agents on the same host. These URLs incorporate both an addressing scheme and the possibility to access properties residing on different agents (processes), referring them using the full URL that includes the name of a remote agent. The repository itself takes care of establishing connections and negotiating with the remote host (see Section 3.2.4 for details). Inter-process communication can be implemented with different protocols, thus allowing interoperability with other frameworks and middlewares.

3.2.2 Data-ports data-sharing metaphor: property links

We extended our blackboard-based repository in order to handle also the data-ports information exchange paradigm, without loosing the benefits of a blackboard, by means of *property links*. These are analogous to Unix filesystem symbolic links and introduce a level of indirection that naturally drives the development towards independent and interchangeable modules. With links, a module's input property is linked to another module's output property, so that the two modules do not need to be aware of each other. Links are specified in a configuration file; since the data flow is not hard-coded, modules can be re-used for different applications. Links can point to remote properties as well, and this allows to distribute the computation in a way which is completely transparent to the module developer (the module always reads its own input properties without taking care which other properties they are linked to).

The property-link mechanism allows for transparently using both blackboard and data-ports paradigms for all data that is shared among modules. Other frameworks make use of both mechanisms, but either requiring additional modules, different code, or the two mechanisms are used for different kinds of information (e.g., ROS uses a blackboard metaphor for parameters and data-ports for inputs and outputs).

3.2.3 Data queues

OpenRDK implements two models for sharing data between modules: publisher/reader and producer/consumer. Regular properties realize the former, and special "queue" properties implement the latter. Both objects share the same URL-based addressing scheme and they can also be used across multiple processes.

Queues are smart FIFO containers that:

- support multiple readers; thread-safeness is ensured without object duplication;
- own the objects that are pushed into them and take care of garbage collection, by destroying the objects when no reader is interested in them anymore;

- allow for subscribing modules to listen to particular objects entering in the queue, and to be awaken on such event.
- are 'passive' objects: no additional thread is required.

Finally, objects in the queues have a timestamp for data synchronization purposes.

Some examples of using these structures are presented in Figure 3.

```
RDK2::ROdometry* odom = new RDK2::ROdometry(/* ... */);
session−>queuePush("odometry", odom);
```

(a) Pushing data into a queue (module hwInterface)

```
session−>subscribeQueue("odometry");
// "odometry" is linked to "rdk://agent1/hwInterface/odometry"
// in the configuration file
```

(b) Subscribing to a queue (module localizer)

```
while (session−>wait(), !exiting) {
    vector<const RDK2::ROdometry*> v =
        session−>queueFreezeAs<ROdometry>(ODOMETRY_URL);

    for (size_t i = 0; i < v.size(); i++) {
        const ROdometry* odom = v[i];
        ... // process odometry data in the queue
    }
}
```

(c) Reading from a queue (module localizer)

```
void Map::read(Reader* r)
 throw (ReadingException)
{
 r−>startReading("map");
 width = r−>read_i32("width");
 height = r−>read_i32("height");
 data = unrle(r−>read_string());
 r−>doneReading();
}
```

```
void Map::write(Writer* w)
  const throw (WritingException)
{
 w−>startWriting("map");
 w−>write_i32(width,"width");
 w−>write_i32(height,"height");
 w−>writeString(rle(data),"data");
 w−>doneWriting();
}
```

(d) Object serialization

Fig. 3. Some code examples of common operations in the OpenRDK framework.

3.2.4 Data sharing implementation: inter-thread, inter-process and inter-machines

In OpenRDK, data sharing happens through a combination of shared memory and network communication; this is transparent from the module's point of view. Modules interact only through the repository interface; if they reside in the same process, then data sharing takes place using shared memory; otherwise, network communication is used, in different forms, as explained below.

OpenRDK provides facilities for a transparent management of concurrent memory access. An implicit per-property locking mechanism is used for simple data-types: in this case, the single module is not responsible for locking/unlocking and deadlocks are prevented. However, when data to be exchanged are large (e.g., maps or images), explicit locking/unlocking primitives are responsibility of the developer.

Inter-agent (i.e., inter-process) communication is accomplished using the *property sharing* mechanism: if a module wants to read or write a remote property (e.g., because a property link points to a property that resides on another agent), a copy of that property (cache) is created in the first agent and is kept synchronized transparently by the repository itself, using a publish/subscribe mechanism.

There are some parameters for tuning the synchronization behavior:

- The subscriber can request a property update every time it changes on the remote repository (`ON_CHANGE`) and optionally set a minimum interval between two subsequent updates. As an alternative, it may request the update to be sent at fixed intervals (`PERIODIC` in OpenRDK terms).
- The subscriber can request to use one of two transport protocols: UDP or TCP.
- OpenRDK implements a data reconstruction layer for some specific objects which can be split in multiple packets and reconstructed at the destination repository.
- Some objects (e.g., images) can be transmitted using `LOSSLESS` (default) or `LOSSY` compression.

The solution we implemented allows for the maximum efficiency: if the data is to be shared between modules on the same host, then shared memory is used; otherwise, the network communication is transparent to the interface used by the module user. Moreover, different clients can request the same property by specifying different options that affect the property updates.

3.3 Tools and interoperability

In this section, we discuss a number of features of a framework for robotic software development that are specifically concerned with the ability to fast prototyping complex applications. This category of tools is particularly relevant to our goals, since it cannot be acquired as an add-on component of a framework. In particular, we argue that OpenRDK choices in the design and implementation of the models for information sharing are key to support the development of tools for fast prototyping.

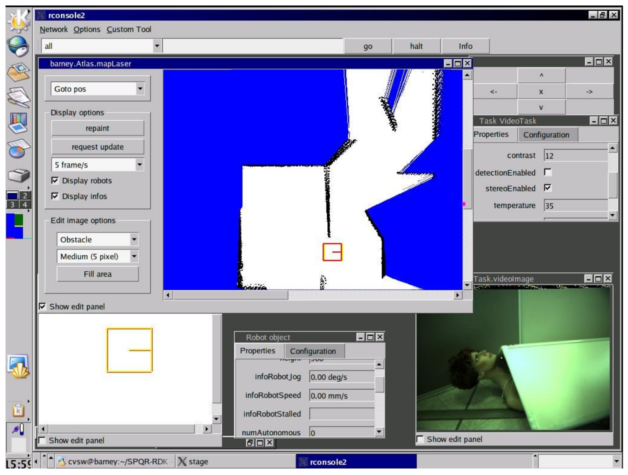
In this section we first look at the problem of remote inspection and debugging, then modularity and configuration, logging and debugging, and, finally, interoperability.

3.3.1 Remote inspection and debugging

Debugging requires to follow the progress of a module behavior in order to detect failures. The inspection tools become easily available, by exploiting the introspection capabilities of the blackboard architecture.

The set of information that are necessary for a proper debugging can easily grow and some data must be visualized using graphical viewers (e.g., images, maps, trajectories), and in an integrated way (e.g., concurrent view of the current map, the current laser scan and the current trajectory). Moreover, the possibility to tune module parameters on-the-fly drastically improves performance of the modules in different applications. The use of a graphical user interface that can interact with the system by visualizing relevant information and change parameters is thus a valuable tool.

OpenRDK provides its own graphical tool for remote inspection and control: **RConsole**. We use it both for the main control interface of the robot and for debugging while we develop the software. RConsole exploits the property sharing mechanism: it is simply an OpenRDK agent that includes a module that displays a GUI. Graphical widgets visualize the internal module state and allow the user to change their parameters while running. Advanced viewers allow to interact with images and maps, moving robot poses, seeing visual debugging information provided by modules, etc. (see Figure 4).



5 frame/s
☑ Display robots
☑ Display infos
Obstacle
Medium (5 pixel)
☑ Show edit panel
contrast 12
detectionEnabled ☐
stereoEnabled ☑
temperature 35
infoRobotJog 0.00 deg/s
infoRobotSpeed 0.00 mm/s
infoRobotStalled
numAutonomous 0
☐ Show edit panel

Fig. 4. RConsole: the graphical tool for remote inspection, on-the-fly configuration and parameter tuning.

3.3.2 Modularity and Configuration

Modularity and code reusability are the most important issues addressed by the analyzed frameworks. In particular, one of the goals of modularity is to reduce the efforts that are needed to adapt a module to different applications and to exchange similar modules within the same application to test their effectiveness. Unfortunately, the implementation of this kind of modularity is not straightforward. The main difficulty consists in defining which *interface* modules should adhere in order to be interchangeable. One could use classes and interfaces provided by the programming language; however, this makes the interface too rigid.

The way modularity and module interchangeability is achieved in OpenRDK is through the use of the repository and property links. Each module publishes inputs and outputs in the repository so that the flow of information is described by property links. OpenRDK enforces a type checking on properties, but it does not perform semantic checks. Thus, the *interface* between modules reduces to conventions regarding the data types accepted by the modules in their input properties. The advantage of this choice is that existing modules are very loosely connected. The disadvantage is that, given that interfaces are not formally defined, the system cannot check for correctness of the interconnection.

Another concept that is used to improve the modularity and the code reuse is "configurability". At *system level*, a configuration is the arrangement of the system as defined by the interconnections of its parts [33]. At a *module level*, a configuration is the specification of its parameters and possible operation modes. The key for improving code reuse is to decouple the configuration from the functional code, in such a way that the same module (i.e., the same code) can be used and maintained separately from its instantiation in an application (i.e., its parametrization and its interconnections with other modules in that singular application).

Following the common solution adopted in other frameworks, OpenRDK saves the configuration (i.e., the module arrangements and their own parametrization) in a file. An OpenRDK configuration file for an agent is an XML file containing the list of modules to be loaded, a description of their interconnections (i.e., property links) and the (initial) values of their properties (e.g., parameters, static inputs, etc.). In this way, often modules need no update in their code in order to be used in another system, another application or even by another group.

The same mechanism has the double function of configuring the modules and offering a way to freeze and restart later the execution of the system (if the module state is properly specified by the set of properties it saves in the repository), or to use static inputs (e.g., a static map of a known environment) that have been saved in the configuration file instead of being produced by other modules.

3.3.3 Logging and debugging sensor data

The use of log files to save the history of important runs is a common habit, especially in those community that deal with problems that do not require an active behavior of the robotic component. For example, the SLAM community can work with the sensor logs from the Rawseeds project[10] or from the Radish project[11]. These (multi)sensors datasets are used to reconstruct a representation of the environment the robot is run into and to compare the results of different methods or different parameter values. Replaying a sensor log is much faster even than using simulation and the fact that it provides always the same sensor data history can be useful to study

10. http://www.rawseeds.org
11. http://radish.sourceforge.net

some critical situations. These log files encode the whole history of sensor readings, including the timestamps, so that it is possible to replay a specific run as it were. OpenRDK provides a configurable module that, reading from a sensor queue, is able to write a log file containing the sensor data. This file can be processed off-line using third-party tools or used in conjunction with another OpenRDK module that provides the "playback" feature. This latter module can replace the hardware interface module or the simulator module: it is thus possible to replay a specific log so that other functional modules behave as in the actual saved run.

3.3.4 Interoperability with simulators

The robot simulators supported directly by OpenRDK are Stage/Gazebo (belonging to the Player/Stage project and using the common Player [19] interface), USARSim [32], the standard robot simulator for the RoboCup Rescue competition, and Webots, used to simulate NAO soccer robots. The modules implementing the support of these simulators typically expose properties that are used both for controlling the platform (e.g., controlling speed), and for sensor reading, (e.g., reading odometry pose).

It is worth noting that driver modules representing the same robot model (and also the same real robot and its simulated counterpart) can be used interchangeably in a transparent way, because the corresponding OpenRDK modules have the same *interface*. Every module that exposes odometry and speed properties, can be considered a robot and connected to a navigation module, regardless of the actual implementation.

3.3.5 Interoperability with other frameworks

The key issue to achieve run-time (i.e., inter-module) interoperability is that the communication layer should be abstracted from the functional modules. The OpenRDK framework provides an easy way to deal with different interfaces by establishing the proper connection between the chosen protocol (or middleware) and the repository. This is done by a single protocol/middleware manager module, that exposes the relevant information using the chosen protocol. Moreover, based on the specification of property links in the configuration file, that include a protocol section, virtually no other modification is needed to the functional modules.

For example, consider the problem of developing an OpenRDK application whose user interface has to be browser-based. The solution is to develop a module that implements a simple web server, maps HTTP URLs to OpenRDK URLs (the mapping is immediate), and serves GET and POST requests by retrieving and changing the values of properties accordingly. Therefore, a complete user interface could be implemented in a few lines of HTML/Javascript. Moreover, these "bridge" modules can allow for additional customization of the link synchronization, that are specific for the protocol they manage. In particular, for example, the HTTP handler module allows to specify an XPath query in order to isolate the desired value from the rest of the response.

The design choices that make this process easy to develop are the presence of the repository (i.e., a blackboard), and the policy of forcing the modules to register all the relevant information (inputs, outputs, parameters, state variables, etc.) as properties in the repository. In this way, a generic module is able to read the whole repository and to expose it to the outside using the required protocol. Note also that while the communication between OpenRDK agents (Section 3.2.4) is of type "push" (publish/subscribe), the HTTP server realizes a communication of type "pull" (more precisely, a REST (REpresentational State Transfer) model [34]).

Thus one can easily implement on top of OpenRDK any data-driven interface that will emerge as standard in future, as an ordinary module (i.e., without or with limited changes to the framework core). In this way, OpenRDK has been enriched with a module that allows its repository to interact with data coming from a Data Distribution Service for Real-Time Systems (DDS)[12] [31]. In the same way, OpenRDK applications can interoperate with other applications using a different middleware (e.g., ROS components).

4 FLEXIBLE DEVELOPMENT USING OPENRDK

One important feature for a robot development framework is the ability to deal with different robotic platforms and to implement different robotic architectures.

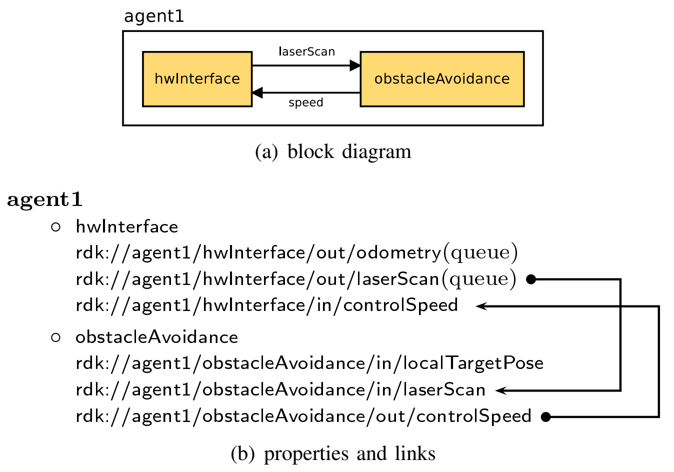
OpenRDK has been successfully used in a wide range of robotic applications, especially in the context of the RoboCup[13] competitions: RoboCupRescue Real Robots, RoboCupRescue Virtual Robots, RoboCup@Home, and also RoboCup Standard Platform League, with the humanoid robot Nao[14] from Aldebaran Robotics, where OpenRDK currently runs on the Nao's internal computation unit. In addition, the OpenRDK framework currently runs also on Gumstix and Beagleboard mini-board systems.

Robotic applications, that are characterized by the relationships among "Sense", "Plan", and "Act", have been deeply studied in the past and many deliberative, reactive and hybrid architectures have been proposed (e.g., [35]). OpenRDK allows to easily implement different kinds of robotic architectures.

This section shows how OpenRDK allows to build diverse applications using different robotic paradigms. While each paradigm puts some requirements on the underlying software framework, the flexibility of OpenRDK allows to compose the functionalities of modules according to any of these paradigms.

Here we develop an example corresponding to (a simplification of) an application in the realm of Rescue Robotics, in a

12. http://portals.omg.org/dds/
13. http://www.robocup.org
14. http://www.aldebaran-robotics.com/eng/Nao.php



(a) block diagram

agent1
- hwInterface
  rdk://agent1/hwInterface/out/odometry(queue)
  rdk://agent1/hwInterface/out/laserScan(queue)
  rdk://agent1/hwInterface/in/controlSpeed
- obstacleAvoidance
  rdk://agent1/obstacleAvoidance/in/localTargetPose
  rdk://agent1/obstacleAvoidance/in/laserScan
  rdk://agent1/obstacleAvoidance/out/controlSpeed

(b) properties and links

Fig. 5. An example of reactive system implemented with OpenRDK: (a) shows a simplified block diagram where the Sense and Act primitives are highlighted (repository has been omitted for simplicity); (b) shows the properties in the repository and the links between them.

bottom-up fashion. The goal of Rescue Robotics is to develop robots to assist human rescuers during emergency operations. The main capabilities needed by such a robot are:

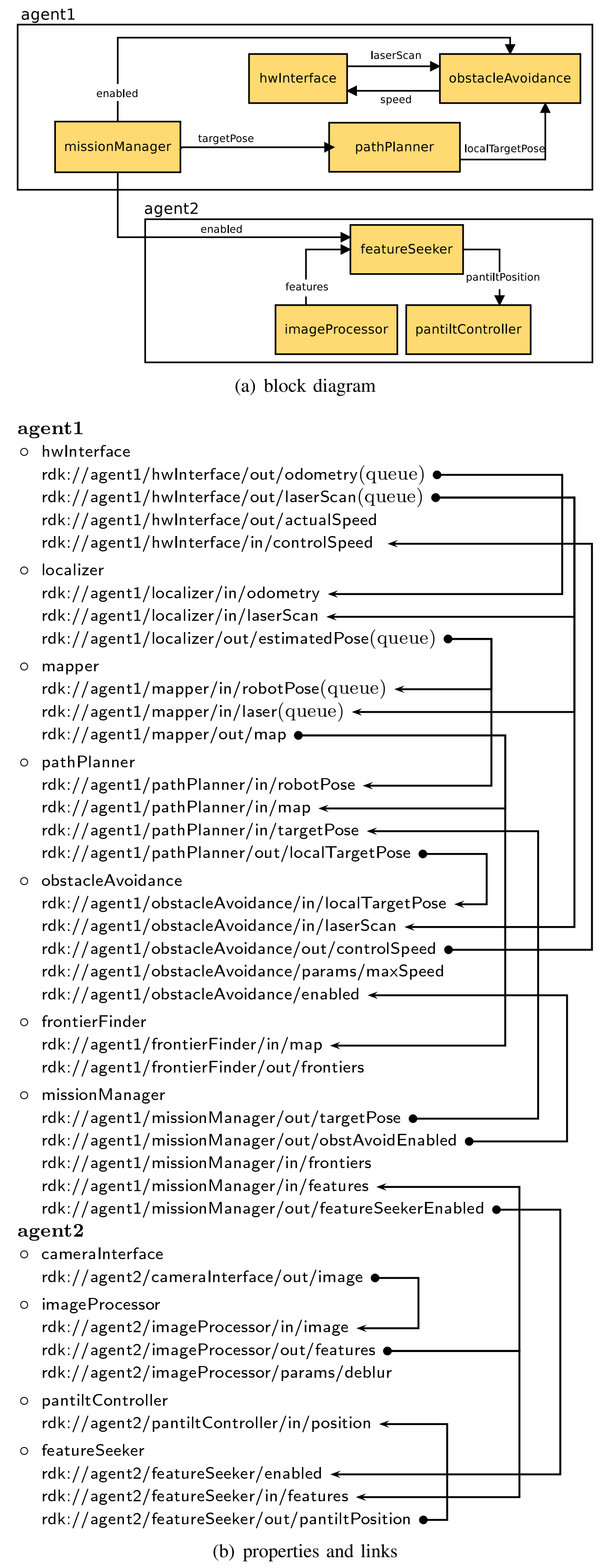
- building a map of an unknown environment;
- being able to move autonomously in a cluttered scenario;
- reporting to the human rescuers the interesting features found during the exploration (for example, possible victims or threats).

The reference robot is a four-wheeled skid-steering mobile base. A laser range finder is mounted on the base for mapping and obstacle avoidance, while a pan-tilt camera is used to identify trapped human victims or possible treats in the environment. Details about the algorithms being used in this system can be found in [36].

4.1 Step 1: reactive system

We start our project by implementing a simple obstacle avoidance behavior, that makes use of the laser range finder to steer the robot through the environment to a target position. The target is given in robot local coordinates, thus no localization in a global frame is required. This reactive behavior can be developed and tested independently, for example using RConsole to feed the local target pose. The system can be realized by connecting the modules as indicated in Figure 5: the hwInterface module is responsible for sensor data gathering and actuator controls allowing for hardware abstraction, the obstacleAvoidance module implements an obstacle avoidance method.

This simple application realizes a pure reactive robotic system, that will be extended in the next sections.



(a) block diagram

agent1
- hwInterface
  rdk://agent1/hwInterface/out/odometry(queue)
  rdk://agent1/hwInterface/out/laserScan(queue)
  rdk://agent1/hwInterface/out/actualSpeed
  rdk://agent1/hwInterface/in/controlSpeed
- localizer
  rdk://agent1/localizer/in/odometry
  rdk://agent1/localizer/in/laserScan
  rdk://agent1/localizer/out/estimatedPose(queue)
- mapper
  rdk://agent1/mapper/in/robotPose(queue)
  rdk://agent1/mapper/in/laser(queue)
  rdk://agent1/mapper/out/map
- pathPlanner
  rdk://agent1/pathPlanner/in/robotPose
  rdk://agent1/pathPlanner/in/map
  rdk://agent1/pathPlanner/in/targetPose
  rdk://agent1/pathPlanner/out/localTargetPose
- obstacleAvoidance
  rdk://agent1/obstacleAvoidance/in/localTargetPose
  rdk://agent1/obstacleAvoidance/in/laserScan
  rdk://agent1/obstacleAvoidance/out/controlSpeed
  rdk://agent1/obstacleAvoidance/params/maxSpeed
  rdk://agent1/obstacleAvoidance/enabled
- frontierFinder
  rdk://agent1/frontierFinder/in/map
  rdk://agent1/frontierFinder/out/frontiers
- missionManager
  rdk://agent1/missionManager/out/targetPose
  rdk://agent1/missionManager/out/obstAvoidEnabled
  rdk://agent1/missionManager/in/frontiers
  rdk://agent1/missionManager/in/features
  rdk://agent1/missionManager/out/featureSeekerEnabled

agent2
- cameraInterface
  rdk://agent2/cameraInterface/out/image
- imageProcessor
  rdk://agent2/imageProcessor/in/image
  rdk://agent2/imageProcessor/out/features
  rdk://agent2/imageProcessor/params/deblur
- pantiltController
  rdk://agent2/pantiltController/in/position
- featureSeeker
  rdk://agent2/featureSeeker/enabled
  rdk://agent2/featureSeeker/in/features
  rdk://agent2/featureSeeker/out/pantiltPosition

(b) properties and links

Fig. 6. An example of a hybrid deliberative-reactive system implemented with OpenRDK.

4.2 Step 2: hybrid reactive-deliberative

A complex system is usually made up of more than one behavior and, possibly, by some deliberative component; in order to show how to add complex behaviours to the reactive system, we consider:

- a pathPlanner module, that, given a target pose in the global reference frame, computes a path towards that pose and outputs also a local goal for the obstacleAvoidance module;
- a featureSeeker module, that implements a scanning behavior that moves the camera in promising directions, in order to search for interesting features in the environment [37];
- a pantiltController module, that controls the pan-tilt unit;
- a imageProcessor module, that implements image processing routines needed to detect relevant features in the environment.

The first module is a path-planner that, given a target pose in the global reference frame, computes a path towards that pose. This pathPlanner module outputs continuously a local target pose for the robot, to be fed into the obstacleAvoidance module described in the previous subsection.

A level of deliberation is added to manage the overall mission behavior. Our way to implement this paradigm is by developing a planner module that is responsible for generating plans and decide possible re-planning steps. The planner module is also responsible for plan execution and monitoring by activating/deactivating the behaviors according to the current actions to be executed. The implementation of this module makes use of the Petri Net Plans (PNP) formalism, presented in [38], tailored for complex multi-robot plans. In particular, the PNP formalism allows for non-instantaneous actions, possible interrupts and failures. The resulting missionManager module is added to the system: exploiting data coming from the sensors, the map and the image processor, it decides the actions to be taken, following a plan written in the PNP formalism. The missionManager module acts upon the featureSeeker mode (different behavior can be adopted, for example when the system is near an interesting feature and should perform a more focused scan), the target pose of the pathPlanner, and the activation/deactivation of the obstacleAvoidance behavior (using a special property called enabled).

This version of the application is shown in Figure 6. The hwInterface and obstacleAvoidance remain the same as before. The vision subsystem (including the pan-tilt controller) runs on another agent: image processing can be computationally intensive, for this reason we decided to split the system into two processes, running on two different machines.

In Figure 6, some properties output relevant information about the robot base (hwInterface/out/actualSpeed), as well as being configurable parameters of the obstacleAvoidance module (obstacleAvoidance/params/maxSpeed) and the imageProcessor module (imageProcessor/params/deblur). The reason why we made these properties explicit in the figure (among many other parameters and outputs) will become clear in next section.

4.3 Step 3: implementing a context-based architecture

In [39], the context-based architecture has been introduced. A context-based architecture is actually a meta-architecture, in the sense that every existing architecture can be transformed in a context-based architecture, by including some additional components. A context-based architecture decouples the reasoning about the context, where the robot is working from the components that implement the mission functionalities (e.g., navigation, mapping, image processing, etc.). This architecture divides the components of a system in two sets: the first set is composed by *functional modules*, that are not aware of any context-related feature; in the second set, some modules are responsible to extract relevant information from functional modules, reason about these information and modify the behavior of the functional modules according to the context.

Using the OpenRDK framework, the context-aware subsystem can be added straightforwardly, without the need to change anything in the functional modules. Since there is no differentiation between inputs, outputs and parameters from the repository point of view, the parameters of the functional modules can be linked to the outputs of the context-aware subsystem, in order to change the behavior of the system.

This version of the application is shown in Figure 7. Another module contextSubSystem that adjusts the parameters of the modules obstacleAvoidance and imageProcessor is added. The rationale is that the current actual speed of the robot should cause some adjustment in the parameters of the victim detection (images are expected to be more blurred at high speeds), while relevant information from the image processor (e.g., the roughness of the terrain) should be used to regulate the obstacle avoidance movement.

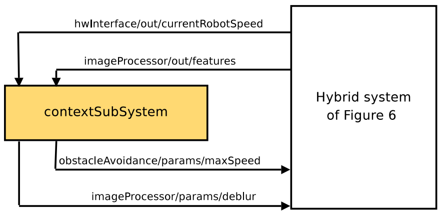

hwInterface/out/currentRobotSpeed
imageProcessor/out/features
contextSubSystem
Hybrid system of Figure 6
obstacleAvoidance/params/maxSpeed
imageProcessor/params/deblur

Fig. 7. An example of context-based system, built on top of the hybrid one of Figure 6: the context-aware module adapt other module parameters according to the current detected context.

A final remark about the complete system described in this section is that while some of the modules are designed to be application-specific (e.g., the imageProcessor detects human victims and exposes some data about the roughness of the floor), other modules can be reused in other domains (e.g., the obstacleAvoidance, localizer, mapper, etc.), *without any change in the code*. The very same modules can be loaded as plug-ins in another system, designed for a completely different application.

5 Conclusions

Robotic software development can become more effective and economical through the use of unified design techniques and the development of reusable and interoperable components. As a contribution to the ongoing discussion, in this paper we have addressed the major choices in the design of robotic software, trying to highlight merits and drawbacks of each alternative. More specifically, we have motivated the design of OpenRDK, a framework for robotic software components, developed in our laboratory. Our design choices in terms of concurrency model, data exchange and configuration management reflect the need for fast development of complex robotics applications. OpenRDK's most specific features are the multi-threaded multi-processes structure and the blackboard-type inter-module communication and data sharing. These allow to seamlessly distribute the computation among several hosts in a transparent way and encourage the users to develop many small decoupled modules with well-defined capabilities. In addition, we have built several tools in OpenRDK that exploit all the advantages of the underlying blackboard model, and we take special care of facilitate interoperability with other existing software resources, such as simulators, low-level robot operating systems, middleware and robotics libraries.

As it turns out, the solutions chosen by OpenRDK to achieve rapid prototyping, good performance and limited dependencies and interactions between different developers in the team, have led to an overall framework that nicely supports concurrent engineering and can be exploited to design the software for a wide class of robotic systems.

References

[1] IEC TC65/WG6, *IEC 61499-1: Function Blocks – Part 1: Architecture*, International Electrotechnical Commission, Geneva, Switzerland, 2005. 1

[2] ——, *IEC 61131-3: Programmable Controllers – Part 3: Programming Languages*, International Electrotechnical Commission, Geneva, Switzerland, 2003. 1

[3] D. Brugali, *Software Engineering for Experimental Robotics (Springer Tracts in Advanced Robotics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. 1

[4] H. Bruyninckx, "Robotics software: The future should be open," *IEEERobotics and Automation Magazine*, vol. 15, no. 1, pp. 9–11, Mar. 2008. [Online]. Available: http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=4476319 1, 2.3

[5] D. Brugali and E. Prassler, "Software engineering for robotics," *IEEE Robotics and Automation Magazine*, vol. 16, no. 1, pp. 9–15, Mar. 2009. [Online]. Available: http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=4799437 1, 2.3

[6] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2009. 1, 2.1

[7] "ROS website," http://www.ros.org. 1, 2.1

[8] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, "OpenRDK: a modular framework for robotic software development," in *Proc. of Int. Conf. on Intelligent Robots and Systems (IROS)*, Sep. 2008, pp. 1872–1877. 1

[9] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics," in *Int. Conf. on Intelligent Robots and Systems (IROS'06), Workshop on Robotic Standardization*, Dec. 2006. 2.1

[10] "Orca-robotics website," http://orca-robotics.sourceforge.net. 2.1

[11] P. A. Baer, R. Reichle, and K. Geihs, "The spica development framework – model-driven software development for autonomous mobile robots," in *Intelligent Autonomous Systems 10 – IAS-10*, 2008, pp. 211–220. 2.1

[12] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of Int. Conf. of Robotics and Automation (ICRA'01)*. IEEE, 2001, pp. 2523–2528. 2.1

[13] "The Orocos project website," http://www.orocos.org. 2.1

[14] "OpenRTM-aist official website," http://www.is.aist.go.jp/rt/OpenRTM-aist. 2.1

[15] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "RT-middleware: distributed component middleware for RT (robot technology)," in *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, Aug. 2005, pp. 3933–3938. 2.1

[16] A. Farinelli, G. Grisetti, and L. Iocchi, "SPQR-RDK: a modular framework for programming mobile robots," in *Proc. of Int. RoboCup Symposium 2004*, D. Nardi *et al.*, Eds. Heidelberg: Springer Verlag, 2005, pp. 653–660. 2.1

[17] I. Nesnas, "CLARAty: A collaborative software for advancing robotic technologies," in *Proc. of NASA Science and Technology Conference*, Jun. 2007. 2.1

[18] "CLARAty website," http://claraty.jpl.nasa.gov. 2.1

[19] T. Collet, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proc. of the Australasian Conf. on Robotics and Automation (ACRA 2005)*, Dec. 2005. 2.1, 2.2.1, 3.3.4

[20] "The Player project website," http://playerstage.sourceforge.net. 2.1, 2.2.1

[21] V. Ng-Thow-Hing, K. Thorisson, R. Sarvadevabhatla, J. Wormer, and T. List, "Cognitive map architecture," *IEEE Robotics & Automation Magazine*, vol. 16, no. 1, pp. 55–66, Mar. 2009. 2.2.1, 2.3

[22] K. Thórisson, T. List, C. Pennock, and J. DiPirro, "Whiteboards: Scheduling blackboards for semantic routing of messages & streams," in *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence*, I. K. R. Thórisson, H. Vilhjalmsson, and S. Marsella, Eds., Jul. 2005, pp. 8–15. 2.2.1

[23] "The MOOS website," http://www.robots.ox.ac.uk/~pnewman/TheMOOS/. 2.2.1

[24] "Interprocess communication (IPC) website," http://www.cs.cmu.edu/~ipc/. 2.2.2

[25] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Proc. of Int. Conf. on Intelligent Robots and Systems (IROS)*, vol. 3, Victoria, BC, Canada, Oct. 1998, pp. 1931–1937. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=724883&isnumber=15658 2.2.2

[26] "TDL - Task Description Language website," http://www.cs.cmu.edu/~tdl/. 2.2.2

[27] C. Cotè, Y. Brosseau, D. Letourneau, C. Raïevsky, and F. Michaud, "Robotic software integration using MARIE," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 55–60, March 2006. 2.2.2

[28] "MARIE website," http://marie.sourceforge.net. 2.2.2

[29] S. Sablatnog, S. Enderle, and G. Kraetzschmar, "Miro - middleware for mobile robot applications," *IEEE Transaction on Robotics and Automation*, vol. 18, pp. 493–497, Aug. 2002. 2.2.2

[30] "MOAST website," http://moast.sourceforge.net. 2.2.2

[31] D. Calisi, F. Fedi, A. Leo, and D. Nardi, "Software development for networked robot systems," in *Proc. of the 7th IFAC Symposium on Intelligent Autonomous Vehicles (IAV)*, 2010. 2.2.2, 3.3.5

[32] S. Balakirsky, C. Scrapper, S. Carpin, and M. Lewis, "USARSim: providing a framework for multi-robot performance evaluation," in *Proceedings of the International Workshop on Performance Metrics for Intellingent Systems (PerMIS)*, Gaithersburg, MD, USA, 2006. 2.3, 3.3.4

[33] I. of Electrical & Electronics Engineers, *IEEE Standard Computer Dictionary - A Compilation of IEEE Standard Computer Glossaries*. IEEE, Jan. 1991. 3.3.2

[34] R. Fielding and R. Taylor, "Principled design of the modern web architecture," *ACM Transaction on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002. 3.3.5

[35] R. Murphy, *Introduction to AI Robotics*. Cambridge, MA, USA: MIT Press, 2000. 4

[36] D. Calisi, A. Farinelli, L. Iocchi, and D. Nardi, "Autonomous navigation and exploration in a rescue environment," in *Proceedings of IEEE International Workshop on Safety, Security and Rescue Robotics (SSRR)*, Kobe, Japan, June 2005, pp. 54–59. 4

[37] V. Navalpakkam and L. Itti, "An integrated model of top-down and bottom-up attention for optimal object detection," in *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, New York, NY, USA, Jun. 2006, pp. 2049–2056. [Online]. Available: http://ilab.usc.edu/publications/doc/Navalpakkam_Itti06cvpr.pdf 4.2

[38] V. A. Ziparo, L. Iocchi, D. Nardi, P. F. Palamara, and H. Costelha, "Petri net plans: a formal model for representation and execution of multi-robot plans," in *AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 79–86. 4.2

[39] D. Calisi, L. Iocchi, D. Nardi, C. Scalzo, and V. A. Ziparo, "Context-based design of robotic systems," *Robotics and Autonomous Systems (RAS) - Special Issue on Semantic Knowledge in Robotics*, vol. 56, no. 11, pp. 992–1003, Nov. 2008. 4.3

Daniele Calisi Daniele Calisi is with the SIED Laboratory in Rome from 2004, in 2010, he got the Ph.D at Dipartimento di Informatica e Sistemistica (DIS) of "Sapienza" University in Rome, Italy. He worked for many projects regarding robotics. He has been a visiting scholar in the Tadokoro Laboratory of Tohoku University, Sendai, Japan, and in the Neuroinformatics Group of University of Osnabrueck, Germany. His main research topics are robot motion planning, obstacle avoidance, machine learning and software frameworks for robotics. http://www.dis.uniroma1.it/~calisi/

Andrea Censi Andrea Censi received a Laurea Specialistica (M.Eng.) degree in control engineering from "Sapienza" University in Rome, Italy in 2007. Currently he is a Ph.D. student in Control & Dynamical Systems at the California Institute of Technology. His research interests include estimation and filtering for robotics and other artificial and biological systems. http://www.cds.caltech.edu/~ender/

Luca Iocchi Luca Iocchi is Assistant Professor at Facolta' di Ingegneria (Faculty of Engineering), "Sapienza" Universityin Rome, Italy, Dipartimento di Informatica e Sistemistica (DIS), since 2002. He has been Post-Doc and visiting researcher at SRI International, Menlo Park USA. He has been working mainly in the field of cognitive robotics, realizing teams of mobile robots with cognitive capabilities for applications in dynamic environments (including soccer robots, rescue robots, domestic robots). His main research interests include: reasoning about actions and planning under incomplete knowledge, cognitive robotics, multi-robot coordination, mobile robot localization navigation and mapping, robot learning, vision-based people and object tracking. http://www.dis.uniroma1.it/~iocchi/

Daniele Nardi Daniele Nardi is Full Professor at "Sapienza" University of Rome, member of Dipartimento di Informatica e Sistemistica, where he was employed since 1986 (as Full Professor since 2000). His current research interests are mainly in the field of artificial intelligence in the area of knowledge representation and reasoning, and cognitive robotics, multi agent and multi-robot systems. http://www.dis.uniroma1.it/~nardi/