OpenRDK: a modular framework for robotics software development

Daniele Calisi, Andrea Censi, Luca Iocchi, Daniele Nardi[*][†]

2008-03-04

Abstract

In this paper we conduct an analysis of existing frameworks for robot software development and we present OpenRDK, a modular framework focused on rapid development of distributed robotic systems. It has been designed following users' advice and has been in use within our group for several years. By now OpenRDK has been successfully applied in diverse applications with heterogeneous robots and as we believe it is fruitfully usable by others we are releasing it as open source.

1 Introduction

More than sixty years after the creation of the first programmable computers, programming is still an art [9] rather than a science. The advancements of hardware capabilities, and the networking revolution of the last decade, exacerbate the problem because new applications of unprecedented complexity are possible and desirable.

The discipline of software engineering aims at providing paradigms, methods, and tools for designing and realizing complex software systems. The problem with such paradigms, methods, and tools is that they tend to come and go in a few years, a time scale that is incomparable with other engineering fields. Object oriented languages (C++, Java), UML, semantic Web, distributed services (Web 2.0), agile languages (Python, Ruby, etc.) are just a few examples of a very fast evolution of methodologies and techniques in less than twenty years.

The situation is even worse when we look at software for robotic applications. In this paper, we intend "robotic applications" as complex systems based on mobile robotics technology and we consider robotic software to be in the middle ground of two fields: it is as complex as other standard ICT software, and it has the same stringent real-world requirements as industrial automation. On the one hand, developing a full control stack for a team of heterogeneous robots is as hard as any other large distributed software system; however, standard tools of ICT software might not be applicable because of the real-world issues

[*]D. Calisi, L. Iocchi, D. Nardi are with the Dipartimento di Informatica e Sistemistica "A. Ruberti", "Sapienza" Università di Roma, via Ariosto 25, I-00185 Rome, Italy.

[†]A. Censi is with the Control & Dynamical Systems department, California Institute of Technology, 1200 E. California Blvd., 91125, Pasadena, CA.

like network unreliability or real-time constraints. On the other hand, formal methods used in automation for developing distributed control systems, like the IEC 61499 standard [14, 15], are not really applicable to robotics, because they were designed only for relatively simple systems.

The scientific community is conducting an intense effort to find a common structure in robotic applications, both from a conceptual and from an implementation point of view, in order to achieve a wide deployment of standard design techniques, architecture styles, and reusable components. This resulted to be very difficult because of the wide range of domains where robots can be exploited, the variety of forms and functions that a robot can have, and, moreover, because of the diversity of people involved in robotics [2]. It is then natural that people working with robots felt the need to develop their own solutions.

Although the primary development of mobile robots will eventually be the industry, nowadays the coolest-acting robots (e.g., those seen in DARPA Grand Challenge, or the Mars rovers) still come from academia or other research environments. Consequently, much of the related work on robotic frameworks makes sense only for the needs of a relatively small academic research group.

Also our work on OpenRDK is focused on the needs of a research group, whose aim is to create innovative algorithms for complex mobile robotic systems, without spending more time than necessary on the software infrastructure. The research group we are considering here is formed by many people (e.g., 10 undergraduate students) working for a short time (e.g., 6 months) on a complex project (e.g., multi-robot heterogeneous exploration in a disaster scenario), usually with no or only a few experience on robotic applications and on software engineering. Hence our main goals are modularity and code re-usability, but we need to take into account also other real-world related problems, like efficiency, noisy perception, unreliable networks, etc.

The contributions of this paper are twofold. On the one hand, we conducted a deep analysis of the issues that a robotic software framework should address, and how existing frameworks take their choices. On the other hand, we present our own framework, designed to meet the goals described above through its unique combination of design choices.

The software can be downloaded from `http://OpenRDK.sourceforge.net` and can be used under the terms of the GPL (GNU General Public License).

2 Analysis of software robotic frameworks

In this section we describe existing robotic software frameworks and their main characteristics, goals, features and drawbacks, grouping them according to their design choices.

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!). In software, this leads to dividing an application in smaller "modules", mutually decoupled software units with precise interfaces. In the following we will use the term "module" to refer to this kind of entity, regardless of the use of other words in other frameworks (e.g., component, service, client, driver, etc.).

2.1 Concurrency model

Dealing with a modular architecture, the first choice that needs to be taken is about how to arbitrate their (concurrent) execution. We classified three possibilities, as follows.

- Modules are *processes* distributed over one or more machines. In this case, developers have the highest freedom. The major drawback is the need of some communication infrastructure that allows for inter-process communication.
- Modules are *threads* inside a single process. With multi-threading, sharing information is easily accomplished using shared memories, but this requires that the framework provides some mechanism for data access concurrency and thread synchronization.
- Modules are built as *call-back functions* and there is a single process (i.e., a scheduler) that repeatedly calls them in response to some event or periodically. The call-back functions are preferable for frameworks whose focus is on low-level device interaction: in particular, call-backs allows for a higher scheduling control, that can be used to enforce real-time constraints. On the other hand, writing this kind of call-back functions is difficult, because they need to return quickly to the scheduler and thus distribute the computation over more than one call.

The distributed process model is a very common choice: most existing frameworks use this architecture. Nevertheless, these choices are not mutually exclusive, and some frameworks make use of a hybrid architecture.

For example, OROCOS[1] [3] 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[2] (a RT-Middleware [1] implementation) and SPQR-RDK [8], our previous software robotic framework, use a very similar architecture. CLARAty[3] [11] relies on ACE (Adaptive Communication Environment[4]) for modules spawning at run-time: its modules can be either processes or threads.

In Player[5] [6] 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 other 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.

[1] http://www.orocos.org
[2] http://www.is.aist.go.jp/rt/OpenRTM-aist
[3] http://claraty.jpl.nasa.gov/man/overview/index.php
[4] http://www.cs.wustl.edu/ schmidt/ACE.html
[5] http://playerstage.sf.net

2.2 Information sharing model

Another problem that arises in a modular framework is how to share information among modules. There are two metaphors that can be used to model the exchange of information among modules:

- modules have input and output "ports" from which they can receive or send messages to other modules that are connected to these ports;
- there is a central object where modules "publish" their data and where they can read other modules data, using some sort of addressing scheme.

In practice, there are two main mechanisms that may be used: modules within the same process may use shared memory, while modules distributed among different processes/hosts have to use some inter-process communication service.

If modules are implemented as threads or as call-back functions, shared memory is the most efficient communication method. In the case of threads, some concurrency management primitives (locks, etc.) are necessary. We remark that the shared memory mechanism cannot allow for all semantics we wish to have. An example is if two modules act as a producer/consumer couple: in this case the framework must provide a mechanism to implement some kind of data "queue" in shared memory.

OROCOS uses "lock-free data ports" to exchange information among modules in real-time. It is possible for a module to invoke other modules' methods and access to their parameters, send events, etc. SPQR-RDK makes modules publish their data in a Shared Information Register, in well-known places where other modules can read it.

In the case of multiple processes, there exist many solutions both for processes on the same machine (e.g., DCOP, DBUS, IPC, COM+) and across a network (e.g., CORBA).

Currently, the Object Management Group[6] (OMG) is working to the new version of the Distributed Data Service (DDS) specification, a standard that is becoming accepted by a great number of companies involved in the development of this kind of software. DDS are middleware systems aimed at distributing data in real-time among applications, using the publish/subscribe paradigm. The most notable DDS implementations are OpenSplice[7] and RTIDDS[8], which adhere almost completely to the OMG DDS specification. These software, anyway, have been never used in robotic application, and are indeed focused in other kinds of domains; their focus is generality and multi-platform support, rather than ease of use and efficiency.

The previous specification of the OMG, the well-known CORBA, is used by MIRO[9] [12], which is process based. OROCOS uses CORBA to provide also inter-process communication through a couple of ad-hoc modules, and OpenRTM-aist uses CORBA also for communication among threads inside the same process. Other frameworks rely on other middlewares, such as ACE (Adaptive Communication Environment[10], that is used by, e.g., CLARAty) or ICE

[6]http://www.omg.org
[7]http://www.prismtech.com/section-item.asp?id=175&sid=18&sid2=10
[8]http://www.rti.com/products/data_distribution/RTIDDS.html
[9]http://smart.informatik.uni-ulm.de/MIRO/index.html
[10]http://www.cs.wustl.edu/ schmidt/ACE.html

(Internet Communications Engine[11], that is used by, e.g., Orca[12] [10]).

The advantage of using a third party middleware for communication is that they are broadly experimented and stable, but there are also some drawbacks: often their goals, like multi-platform/multi-language support and application independence, are not of primary importance in robotics. Writing a proprietary middleware can be a difficult and a long work, but allows to make it fit the specific needs of the robotic application, without unnecessary added complexity.

Among frameworks that provide their own inter-process communication infrastructure, we mention Microsoft Robotics Studio, that 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 connection between Player server and the clients is done over a TCP link, using a proprietary protocol, that can be tuned to the application need (e.g., frequency of updates, etc.). The MOOS[13] (Mission Oriented Operating Suite) communication model has a star-like topology, in which the modules publish their information in a centralized blackboard called MOOSDB and connect to it as clients, sending information over TCP links using a proprietary protocol.

IPC (Inter Process Communication)[14] and TDL (Task Description Language)[15] [13] are two software packages developed at CMU that, extending C++ semantics, provide for a technique to send C++ objects through a network between servers and clients (IPC) and for the definition of call-back procedures that are called when some event (e.g., some packet arrival) occurs. Although it is general enough to be used in any kind of application that is based on asynchronous events, its main uses have been robotics.

MOAST[16] (Mobility Open Architecture Simulation and Tools) makes use of the RCS/NML library[17] (Real-time Control Systems/Neutral Message Language).

In MARIE[18] [7], 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 inter-connection and the use of data coming from the various modules, that, in their turn, can be developed using any kind of library or framework available.

OpenRDK provides its own communication infrastructure in which a blackboard object, called repository, is involved in data sharing among modules (threads) running inside the same process. Through a mechanism called "property sharing", modules are allowed to refer to data residing on a different process in the same way as it was on the same one. Property sharing is accomplished using both TCP and UDP, and can be tuned, in the configuration phase, by the means of many options.

[11]http://zeroc.com/ice.html
[12]http://orca-robotics.sourceforge.net/orca/index.html
[13]http://www.robots.ox.ac.uk/%7Epnewman/TheMOOS/
[14]http://www.cs.cmu.edu/ ipc/
[15]http://www.cs.cmu.edu/ tdl/
[16]http://moast.sf.net
[17]http://www.isd.mel.nist.gov/projects/rcslib
[18]http://marie.sf.net

2.3 Tools

Tools are a very important feature for a software framework, because they can speed up development and help in finding bugs and analyze their algorithms. Among the most widely used tools in a software robotic framework, we find:

- a simulator;
- utilities to log sensor data or module activity;
- remote inspection tools to observe/modify the module state;
- a configuration utility.

When developing complex algorithms that requires a long time of experimentation, a realistic simulator can be handy. 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 simulator, for example MOAST is strictly connected with the USARSim[19] simulator, while OpenRDK provides modules that can be used to connect both to USARSim and, through Player, to Stage and Gazebo.

Almost all frameworks provide some facility to log data on a file to be analyzed or used off-line. Some frameworks contains viewers that allow to visually analyze the saved logs. OpenRDK provides two configurable modules that can be used to log and replay data.

OpenRTM-aist provides RTCLink, a graphical tool to connect components and save a configuration for later use. MARIE can make use of RobotFlow[20], a 3rd party tool that allows for visual module integration and debugging.

Remote inspection can be a crucial utility, since it allows for on-line examination of modules behavior and often it comprise also the possibility of modifying parameters without the need to restart the application. OpenRDK, like many other frameworks, comprises also a specific visual tool for remote inspection and visualization.

[19]http://usarsim.sf.net
[20]http://robotflow.sf.net

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
OpenRTM-aist	threads	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
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

Figure 1: Summary of known software robotic framework and their architecture choices

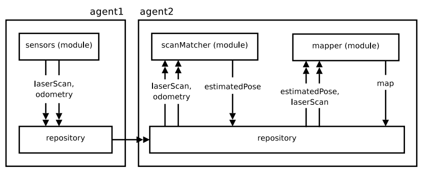


Figure 2: An example of two OpenRDK agents, running on two different machines

3 OpenRDK architecture

OpenRDK is a framework written entirely in C++ running on Unix-like operating systems (Linux, OS X). In this section we describe its architecture and the rationale for our design choices.

3.1 Key concepts

- In our framework the main entity is a software process called **agent**. A **module** is a single thread inside the agent process; modules can be loaded and started dynamically once the agent process is running.
  In Figure 2 we see an example. Two agent are executed on two different machines and three modules run inside them: hwInterface retrieves data from a laser range finder and the odometry by a robotic base; given these two piece of information, scanMatcher uses a scan-matching algorithm in order to estimate the robot positions over time; mapper uses the estimated robot positions, together with the laser scans, to build a map of the environment.

- An agent **configuration** is the list of which modules are instantiated, together with the value of their parameters and their interconnection layout. It is initially specified in a configuration file.

- Modules communicate using a blackboard-type object, called **repository** (see Figure 2), in which they publish some of their internal variables (parameters, inputs and outputs), called **properties**. A module defines its properties during initialization, after that, it can access its own and other modules' data, within the same agent or on other ones, through a global URL-like addressing scheme. Access to remote properties is transparent from a module perspective; on the other hand, it reduces to shared memory (OpenRDK provides easy built-ins for concurrency management) in the case of local properties.

- Special queue objects also reside in the repository and they share the same global URL-like addressing scheme of other properties.
  In Figure 2, the hwInterface module pushes laser scan and odometry objects into queues, that are remotely accessed by the scanMatcher module, which, in turn, pushes the estimated poses in another queue, for the mapper to access to them. Finally, the mapper updates a property which contains a map.

3.2 Concurrency model

We thought long and hard about how to implement concurrency in the OpenRDK. We would not leave behind the efficiency of call-back functions, but we saw, through years, that our typical user found it difficult to implement that kind of functions. The multi-threading solution is a good compromise, although this required an infrastructure for concurrent data access and synchronization. Another reason to implement modules in this way is that we often deal with big data structures like maps or images, and the use of a shared memory for these objects is the best solution. The bias on multi-threaded applications, with respect to multi-process applications, results also in a more compact structure of the overall system: it is easier to configure and run a system comprising many agents, each of which contains its own set of modules parametrized for its specific task. Finally, this allows OpenRDK to contain also very small modules, for example for data filtering purposes, without losing too much in efficiency.

At run-time, each module (thread) is waiting on a semaphore until some event occurs. Typical events are fixed interval timeouts, new data on a queue or the change of a property value; modules can wait on more than one event.

3.3 Repository, properties and URLs

As we said above, the repository is the place where all modules publish the data they want to share with others. Published properties are inputs, outputs and parameters. A path-planner module, for example, could publish the current and target robot poses as inputs, and the resulting path as output.

Each property is assigned a globally-unique URL with the following syntax:

```
rdk://<agent-name>/<property-path>
```

in which `<agent-name>` is a unique name of the agent, while `<property-path>` usually contains the name of the module followed by the name of the property. Some examples follows:

```
rdk://agent1/hwInterface/odometry
rdk://agent2/scanMatcher/odometry
rdk://agent2/scanMatcher/estimatedPose
```

Globally unique URLs allow every module to transparently access a property on any other agent (on any other host). For details on remote property sharing, see Subsection 3.8.

3.4 Object marshalling and unmarshalling

Marshalling is the process of transforming run-time objects in a form suitable for storage or transmission. In OpenRDK, each class implements a read/write methodThrough the same interface functions, it is possible to write both in XML (typically for storage purposes, for example to save configuration files that can be easily read by a human) and in a more efficient binary format (for transmitting over the network).

We chose not to provide any automatic mechanism to generate this kind of functions (for example, using an IDL-like language), because platform independence is not among our goals and, moreover, because in this way we have a great flexibility that allows for the implementation of ad-hoc serializations (e.g., for lossy/lossless compression of maps and images).

3.5 Configuration and object persistence

A configuration is a list of modules to be loaded and executed, their interconnections and the values of their parameters. Agent configurations are saved as XML files and contain a serialized representation of each module and its properties. Since parameters are properties, they can easily be saved using the mechanism seen in the previous subsection. Moreover, since we save all module properties in the configuration file, this technique can also be used to save the state and load it afterwards. For example, we can have the robot building a map of an environment, save it once in the configuration file, and then use it for all subsequent runs.

3.6 Property links

The device of "property links", analogous to Unix symbolic links, introduces a level of indirection in the repository that allows to make the modules as decoupled as possible.

The main problem that links solve is the fact that two modules needing to share an information must agree on some well-known place (in our case, a URL) where this information is to be accessed, and this creates an unnecessary coupling between them. For example, consider in Fig. 3(a) where the module `scanMatcher` writes on the property `estimatedPose` and module `mapper` reads from there. If we were to introduce another module `kalmanFilter` as a filter, we would have to change `scanMatcher` or `mapper`.

The solution with links is that each module reads from and writes to its own properties, and then one module's input is "linked" to another module's output. For example, the impasse of Fig. 3(a) is solved in Fig. 3(b) by using links: each module writes on its own properties, and links are used to determine the interconnections.

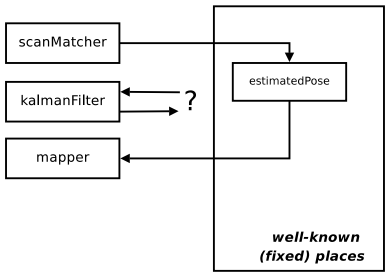
Links can point to remote properties as well, and this allows to distribute the computation in a way which is completely transparent to the module developer.

Links are specified in a configuration file; since the data flow is not hard-coded, modules can be easily re-used for different applications. In practice, this encourages the developers to create many small re-usable modules instead of big monolithic ones.

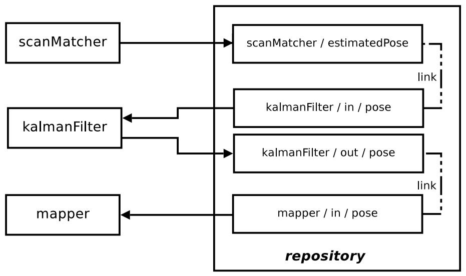
3.7 Queues as object dispatchers

There are two typical use cases for sharing data between modules: either the subscriber module needs only the current value, or it needs the full history of changes. For example, a simple path-planner module usually needs only the current position of the robot and the current map, regardless of their previous value or content. On the other hand, a localization or a mapping module may require the whole history of odometric measurement and range-finder scans.

This naturally leads to the concept of a queue, i.e., a container where the objects are managed in a FIFO way. The OpenRDK provides a special queue object, that has the following characteristics:



(a) Well-known places



(b) OpenRDK URLs and links

Figure 3: The difference between a "well-known places" and OpenRDK "URLs and links" architecture

- it manages thread-safeness when multiple readers access the same objects, but avoids object duplication;
- it takes care of garbage collection, keeping a reference counter and destroying the objects when no reader is interested in them;
- it allows subscribers to listen to particular objects entering in the queue, i.e., a module can be sent in sleep mode until a particular kind of object enter in a queue;
- it is a 'passive' object: no additional thread is required to manage a queue.

Although the OpenRDK queues implementation is rather complex, it is kept "under the hood" and they actually make module writing much easier. See for example Figure 4, in which we show the communication between two modules, `hwInterface` and `scanMatcher`: the first module takes the odometry reading from the sensor device and pushes in a queue called "odometry", on another agent, the module `scanMatcher` subscribes to that queue during initialization and then is able to retrieve the values in a very simple way.

3.8 Inter-agent information sharing

As we described above, information sharing among modules that are executed inside the same agent (process) is accomplished using the repository. Inter-agent (i.e., inter-process) communication is accomplished by two methods: through *property sharing* and *message sending*.

```
// Module hwInterface, on agent1
RDK2::ROdometry* odom = new RDK2::ROdometry(/* ... */);
session->queuePush("odometry", odom);
```

```
// module scanMatcher, on agent2: during initialization
session->subscribeQueue("odometry");
// "odometry" is linked to "rdk://agent1/hwInterface/odometry"

// module scanMatcher, on agent2: during execution
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
    }
}
```

Figure 4: Example of using a queue object in OpenRDK (see also Figure 2)

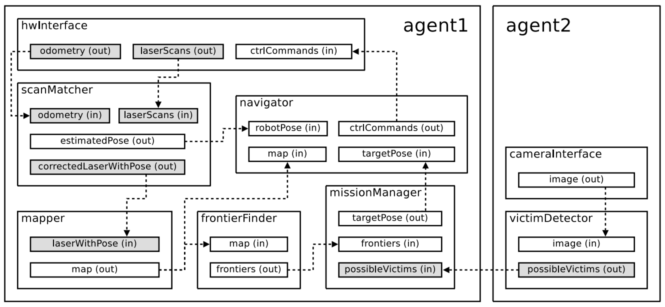
In the first case, one agent refers to properties of another agent by specifying the name of the remote agent in the URL of the property (this is usually done when specifying a property link in the configuration). The repository is in charge of requesting the remote property value and publish it in a local copy (proxy) for the requesting module to read.

Property sharing can be tuned by using a set of parameters, that can be specified in the configuration files and are explained in the following.

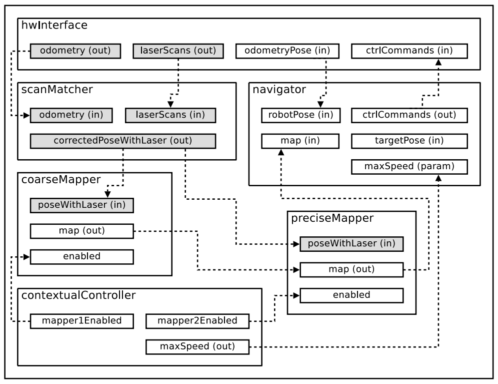
- The subscriber can request a property update every time it changes on the published repository (`ON_CHANGE`) and optionally set a minimum interval between two subsequent updates. As an alternative, it may request that the update have to be sent at fixed intervals (`PERIODIC` in OpenRDK terms).
- The subscriber can request to use one of two transport protocols: UDP or TCP.

OpenRDK also partially implements a data reconstruction layer, i.e., some object can be split in multiple packets and reconstructed in the destination repository. Moreover, for some objects can be requested a `LOSSLESS` (default) or `LOSSY` compression. For example, if an image have to be sent to an image processing module, chances are that it need the image as it was on the source side. On the other hand, if the property is requested solely for visualization purpose, a lossy compression is more effective.

In addition to the property sharing mechanism, we also implemented a standard message-sending (mailbox) method, for when this feature is a more straightforward mapping with the semantic of the application. When a module wants to send a message to another module, being it on the same agent or on a remote one, it writes the address of the recipient on the message object and push it into a special queue called “outbox”. Receiving a message requires to subscribe to the “inbox” queue and to be able to discern interesting messages.



(a) a single rescue robotic system application with two agents



(b) a context-based online configuration architecture

Figure 5: Two examples of configuration: gray properties are queues

3.9 Tools contained in the OpenRDK

RConsole RConsole is a graphical tool for remote inspection and management of modules. We use it as both the main control interface of the robot and for debugging while we develop the software. RConsole was very easy to implement thanks to the property sharing mechanism: it is just an agent that happens to have some module that displays a GUI. Through the reflection used in the repository, graphical widgets visualize the internal module state and allow the user to change their parameters while running. Advanced viewers allows to interact with images and maps, moving robot poses, seeing visual debug information provided by modules, etc.

Modules for logging and replaying OpenRDK provides a configurable module that, reading from a sensors 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 module that provides the "playback" feature.

Connection with simulators As we explained in the Section 2, OpenRDK provides modules that allow to connect to both USARSim and, through Player, to Stage and Gazebo. The modules expose the same interface of the real ones, thus resulting in a transparent behavior for the modules that connects to them.

4 Some applications of OpenRDK

The OpenRDK framework has been successfully used in a wide range of robotic applications.

4.1 Single rescue robotic system

Our group is involved in rescue robotics, whose goal is to develop robots to assist human rescuers during emergency operations. The main capabilities needed by such a robot are:

- to build a map on an unknown environment;
- to be able to move autonomously in a cluttered scenario;
- to report to the human rescuers the interesting features found during the exploration (for example, possible human victims that are entombed or trapped, or possible treats).

The system has been developed as an OpenRDK agent. The real robot was equipped by two personal computers and two agents run on each of them: in this way, we were able to divide the computation weight among two machines. In particular, the first was responsible for the robot mapping and navigation subsystems, as well as the mission manager module; the second machine contained the modules for vision processing. In this application, one example of property sharing is that the vision module published a queue of "possible human sightings" that was read remotely by the mission manager module on the other PC. See Figure 5(a) for details.

By simply substituting the real sensor and robot modules by modules that connected to a simulator, we have been able to test exactly the same software system in both real and simulated scenarios. The simulated rescue scenario allowed us to conduct experiments with a large number of robots in a large environments.

4.2 Assistive robots

The RoboCare Project[21] [5] aims at building a system for assistance of the elderly and the impaired person. Such non-invasive technology should be easily integrated in the environment and able to interact with the person and to monitor his behavior. Such technology is a distributed and heterogeneous system. For example, some of the main components is a multi-camera system that can follow the human in the environment and track his position, a wheeled robot that can move in the environment and interact with the human through a human-robot interface, and a PDA that the assisted person can use to interact.

In this project, two OpenRDK agents are involved and interconnected to a pre-existent system. One of them is responsible of managing the mobile robot. It includes modules for localization in a known environment as well as path-planning and dynamic obstacle avoidance. Another OpenRDK agent is running connected with the camera tracking system and is responsible for sending the image data to the PDA and to send the tracked human position to the robot agent.

[21]http://robocare.istc.cnr.it

4.3 Context-based online configuration

In a recent work of ours [4], we studied the possibilities of a system that is able to control the behavior of other modules by using "contextual" information. Using the OpenRDK framework, we were able to test this idea in a straightforward way. The only thing we needed was to implement the contextual controller and connect its outputs to the *parameters* of the other modules. In this way, the contextual controller was able, for example, to reduce the maximum speed of the path planner, when the situation required slower movement, or to switch between two mapper modules. In Figure 5(b) we can see a simplified diagram of this system.

5 Conclusions and on-going work

In this paper, after a deep analysis of the many existing robotic frameworks and how they address the most important issues, we presented our own OpenRDK framework. Our design choices reflect the need for fast development of complex robotics applications in a research environment.

With respect to the other full-featured frameworks, OpenRDK's most unique 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.

The most immediate future work is to extend the Quality of Service (QoS) settings provided by the OpenRDK, regarding network property sharing, to include some others defined by the DDS specification. For example, at the moment OpenRDK communication happens either over TCP or UDP; in the case of noisy wireless networks neither behave well, as UDP messages are simply lost, and TCP keeps resending old data (thus aggravating the network overload). It would be useful to have a mechanism similar to DDS's "latency budget" which keeps resending data only for a fixed period of time. Other DDS-inspired ideas worth exploring are rules for detecting whether a peer is not reachable anymore and acting consequently.

So far we talked about QoS only for communication, but for robotics we can also talk about QoS for the computations, as there are many algorithms whose output has a quality that can be tuned: for example particle filters, in which one can have more precise estimates by using more particles, or RRT path-planning, in which one can find shorter paths by expanding more nodes. Integrating this "computation QoS" in the framework, by providing some means for the modules to declare a QoS seems particularly interesting because it addresses a need which is very particular to robotics.

In the last three years, OpenRDK has been used by many people in our group, for the most part undergraduate students, in diverse research projects, like rescue robotics and assistive robotics. During the RoboCup competition, the OpenRDK has been used contemporaneously in three different leagues (RoboCup@Home, RoboCupRescue Real Robots and RoboCupRescue Virtual Robots), and its characteristics allowed us to share basic modules among the leagues (for example the obstacle avoidance or the localization modules) while focusing on league-specific developments. For RoboCup 2008 we will use this framework also for the development of the team of Nao humanoid soccer robots.

In conclusion, we are not worried by the apparent proliferation of robotic frameworks. We believe that in software engineering as in nature innovation happens through evolution and natural selection in a bazaar-like atmosphere of creative chaos. In this context, the equivalent of 'intelligent design' would be design-by-committee: as the maxim goes, *a camel is a horse designed by committee*; many technologies praised at their birth as sturdy stallions turn out to be, in hindsight, lame camels. Committees are useful though to crystallize technologies when they are well understood; but this is not still the case for robotics software. The fact that there are currently lots of frameworks projects available is not necessarily a bad thing, it just means that the field is alive and well and that there are many directions being explored.

References

[1] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. RT-middleware: distributed component middleware for RT (robot technology). In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, pages 3933–3938, August 2005.

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

[3] Herman Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings of Int. Conf. of Robotics and Automation (ICRA'01)*, pages 2523–2528. IEEE, 2001.

[4] Daniele Calisi, Alessandro Farinelli, Giorgio Grisetti, Luca Iocchi, Daniele Nardi, S. Pellegrini, D. Tipaldi, and Vittorio A. Ziparo. Contextualization in mobile robots. ICRA'07 Workshop on Semantic Information in Robotics, 2007.

[5] A. Cesta, G. Cortellessa, M.V. Giuliani, L. Iocchi, R.G. Leone, D. Nardi, F. Pecora, R. Rasconi, M. Scopelliti, and L. Tiberio. The robocare assistive home robot. *Autonomous Robots*, 2007. to appear.

[6] T.H.J. Collet, B.A. MacDonald, and B.P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA 2005)*, December 2005.

[7] Carle Cotè, Yannick Brosseau, Dominic Letourneau, Clément Raïevsky, and Francois Michaud. Robotic software integration using marie. *International Journal of Advanced Robotic Systems*, 3(1):55–60, March 2006.

[8] A. Farinelli, G. Grisetti, and L. Iocchi. SPQR-RDK: a modular framework for programming mobile robots. In D. Nardi et al., editors, *Proc. of Int. RoboCup Symposium 2004*, pages 653–660, Heidelberg, 2005. Springer Verlag. ISBN: 3-540-25046-8.

[9] Donald E. Knuth. *The Art of Computer Programming.* Four volumes. Addison-Wesley, 1973–.

[10] 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*, December 2006.

[11] I.A. Nesnas. Claraty: A collaborative software for advancing robotic technologies. In *Proc. of NASA Science and Technology Conference*, June 2007.

[12] S. Sablatnog, S. Enderle, and G. Kraetzschmar. Miro - middleware for mobile robot applications. *IEEE Transaction on Robotics and Automation*, 18:493–497, August 2002.

[13] Reid Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings Conference on Intelligent Robotics and Systems*, October 1998.

[14] IEC TC65/WG6. *Programmable Controllers – Part 3: Programming Languages.* International Electrotechnical Commission, Geneva, Switzerland, 1993.

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