ROCI: A Distributed Framework for Multi-Robot Perception and Control

Luiz Chaimowicz, Anthony Cowley, Vito Sabella, Camillo J. Taylor

GRASP Laboratory – University of Pennsylvania, Philadelphia, PA, USA, 19104

{chaimo, acowley, vsabella, cjtaylor}@grasp.cis.upenn.edu

Abstract

This paper presents ROCI, a framework for developing applications for multi-robot teams. In ROCI, each robot is considered a node which contains several modules and may export different types of services and capabilities to other nodes. Each node runs a kernel that mediates the interactions of the robots in a team. This kernel keeps an updated database of all nodes and the functionalities that they export. Multi-robot applications can be built and changed dynamically, connecting modules that may be running in different nodes over the network. As an example, we present an obstacle avoidance task implemented using our framework and also discuss the use of ROCI in a multi-robot scenario.

1 Introduction

As sensors, actuators, microprocessors and wireless networks become cheaper and more ubiquitous it has become increasingly attractive to consider employing teams of small robots to tackle various sensing and manipulation tasks. In order to exploit the full capabilities of these teams, we need to develop effective models and methods for programming distributed ensembles of sensors and actuators.

Applications for distributed dynamic robotic teams require a different programming model than the one employed for most traditional robotic applications. In the traditional model, the programmer is faced with the task of developing software for a single processor interacting with a prescribed set of sensors and actuators. He or she can typically assume that the configuration of the target system is completely specified before the first line of code is written. On the other hand, when developing code for multi-robot dynamic teams, we must account for the fact that the number and type of robots available at runtime cannot be predicted. We expect to operate in an environment where robots will be added and removed continuously and unpredictably. Further, we must expect an environment where the robots will have heterogeneous capabilities; for example, some may be equipped with camera systems, others with range sensors or specialized actuators, some agents may be stationary while others may offer specialized computational resources. This implies that the program must be able to identify and marshal all of the resources required to carry out the specified task automatically.

This paper presents ROCI (Remote Objects Control Interface), a self-describing, objected oriented, strongly typed programming framework that allows the development of robust applications for dynamic multi-robot teams. The building blocks of ROCI applications are self-contained, reusable modules. Basically, a module encapsulates a process which acts on data available on the module's inputs and presents its results as outputs. Thus, complex tasks can be built connecting inputs and outputs of specific modules. These connections are made through a pin architecture that provide a strongly typed, network transparent communication framework. A good analogy is to consider each of these modules as an integrated circuit (IC), that has inputs and outputs and does some processing. Complex circuits can be built wiring several ICs, and individual ICs can be reused in different circuits.

The core control element in the ROCI architecture is the ROCI kernel. There is a copy of the kernel running in every entity that is part of the ROCI network (robots, remote sensors, etc.). These entities are considered ROCI nodes and any information acquired or processed in a certain node can be exposed to others. The kernel is responsible for managing network and maintaining an updated database of all the nodes and services in the ROCI network. The kernel is also responsible for handling module and task allocation and injection. It allows applications to be specified and executed dynamically, by connecting available pins and transferring code libraries to the nodes.

ROCI incorporates some features that are already present in modern distributed software environments such as the Open Agent Architecture [7] and the Grid Computing [4]. Some frameworks for cooperative robotics have already included advances such as hier-

archical and reusable objects [1], distributed sensing and actuation capabilities [5], abstraction and modularity [8], and task decomposition [9]. Also, the use of modern programming languages [2] and graphical interfaces for task specification [6] are important advances. But, in spite of that, most of the programming architectures for distributed robots still rely on traditional programming models and are specific for certain types of robots and control architectures. Thus, we believe that ROCI will certainly be a valuable contribution to the multi-robot programming field.

This paper is organized as follows: the next section describes the ROCI framework, giving details about its structure and its main features. Section 3 shows the implementation of an obstacle avoidance task using ROCI and Section 4 describes the use of ROCI in a multi-robot scenario. Finally, in Section 5 we conclude the paper and discuss the next steps in this work.

2 ROCI Architecture

2.1 Introduction

ROCI is a dynamic, self-describing, object-oriented, strongly typed programming framework for distributed sensors and actuators. It provides programmers with a network transparent framework of strongly typed modules - assemblies of metadata, byte code, and machine code that can consume, process and produce information. ROCI modules are injectable (they can be automatically downloaded and started on a remote machine), reusable, browseable, support automatic configuration via XML, and provide strongly typed pin based communications. These features, coupled with a dynamic database of available nodes and network services, allows a programmer to write code that utilizes networks of robots as resources instead of independent machines.

ROCI is developed in C# using the Microsoft .NET platform. Modules are not limited to this language however, and several of our own modules are written in mixtures of C# and C++. The system makes use of XML to provide basic configuration options, and object reflection to enforce type safety and autodiscovery.

2.2 Modules and Tasks

The building blocks of a ROCI application are ROCI modules. A module is a computational block that encapsulates a process, taking an input, performing some operation on it, and making the result of that operation available as an output. There is no specification in the code relating to where input should come from or where output should go; the only specification is the type of data this computational block deals with. To create a new module, the application developer has to decide on the types of data to be input and output and then inherit from the ROCI module parent class, implementing a few virtual functions related to the allocation and de-allocation of resources required by the new module. Since the modules are designed with no knowledge of their runtime environment, they can be wired into the ROCI network with a great deal of flexibility.

ROCI modules are further organized into tasks. A ROCI task is a way of describing an instance of a collection of ROCI modules to be run on a single node and how they interact at runtime. Tasks represent a family of modules that work together to accomplish some end goal – a chain of building blocks that transforms input data through intermediate forms and into a useful output. A task can be defined in an XML file which outlines the modules that are needed to achieve the goal, and the connectivity between these modules. Tasks can also be defined and changed dynamically, by starting new modules and connecting them with the outputs and inputs of other modules. As will be explained in the next section, the connection between modules is made using pins. Pins can connect modules within the same task on the same computer, between tasks on the same computer, or between two tasks on different computers. Figure 1 shows an example of this architecture.

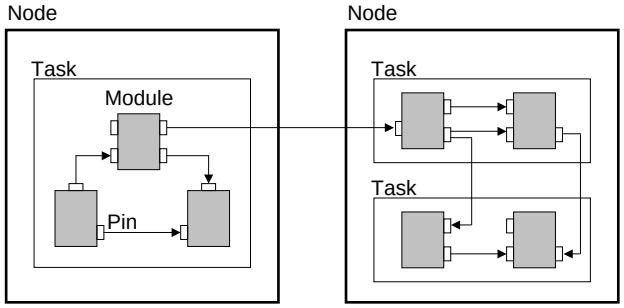

Figure 1: Roci architecture: tasks are composed of modules and run inside nodes. Communication through pins can be seamlessly done between modules within the same task, modules in different tasks, or in different nodes.

2.3 Pin Based Communication

The wiring that connects ROCI modules is the pin communications architecture. Pin communications in ROCI are designed to be network transparent yet high performance. Basically, a pin provides the developer with an abstract communications endpoint. These endpoints can either represent a data producer or a data consumer. Pins in the system are nothing more than strongly typed fields of a module class, and thus

connecting a producer pin to a consumer pin is as simple as setting a reference to the producer in the consumer's field. Modules' pins are automatically exposed and discovered by the ROCI kernel through reflection – an important feature when programming dynamic networks.

Whenever a consumer pin registers itself to a producer pin, the ROCI subsystem determines whether the modules are within the same task domain. If they are, the consumer pin is assigned a reference to the producer pin. If not, ROCI creates a Remote Procedure Call channel and assigns a proxy reference to the consumer.

As the producer generates data it assigns it's pin an updated reference to the latest data. This assignment causes the pin to fire messages to all of the registered consumers in the network, alerting them to the availability of fresh data. A typical usage of this system is for a module to make a blocking call to one of its input pins which returns once the input pin has gotten new data from the output pin it is registered to. Alternatively, a module can ask its input pin to copy over whatever data is available immediately, whether it is new or not. Pin data is time stamped, allowing the consumers to determine how current their data is. Once all the consumers have completed processing their data, the managed environment in which ROCI runs automatically marks the data for garbage collection, freeing the programmer from any memory management issues which may arise with complex producer/consumer interconnections.

One can set a pin's input in two distinct ways that are each useful in different situations. On a lower level, pins can be connected dynamically during program execution. This can be accomplished by querying nodes on the ROCI network for available pins – usually with some type constraint – and may involve dynamically creating local pins to bind to the discovered remote pins. However, the simpler way of binding pins together is via the XML descriptions that define ROCI tasks.

Strongly typed pins enforce that only pins of the same type are connected to each other. The exchange of strongly typed objects instead of raw data eliminates potential software bugs increasing the robustness of the system. Robustness is also a consequence of the self-describing nature of pins. Since we can find out the exact type of a pin instance, we can dynamically guarantee that it will only be connected to a compatible pin.

2.4 ROCI Kernel

The kernel is the core control element of ROCI. The kernel manages the Remote Procedure Call (RPC) system, the real-time network database, module and task allocation and injection, and a Web Services like interface for remote monitoring and control. There is a copy of the kernel running in every entity that is part of the ROCI network (robots, remote sensors, etc.).

The RPC system provides interfaces for module management, injection and communication, as well as providing a web-based interface to the current status of the network. The real time database contains information on all of the modules, tasks and communications channels within the network. ROCI's database can be used to locate an appropriate sensor or actuator to solve a problem, find software modules that are needed in local computation, and identify computer utilization and congestion across the network. The database provides modules with a bird's eye view of the environment, allowing them to locate and utilize each and every hardware and software resource on the network.

It is important to mention that ROCI's kernel exposes its interfaces through Simple Object Access Protocol (SOAP), a cross-platform RPC standard. This standard allows non-ROCI programs and utilities to easily interoperate with and utilize the resources of the ROCI network.

2.5 ROCI Browser

The job of presenting this network of functionality to the user falls upon the ROCI Browser. The browser's job is to give a human user command and control over the network as well as the situational awareness necessary to make informed decisions about network operations. The ROCI network is presented hierarchically: the human operator can browse nodes on the network, tasks running on each node, the modules that make up each task, and even certain pins within those modules. The browser can be used to monitor the status of running tasks or even to tap into and display the outputs of pins for which display routines exist.

Using the browser, the user can also decide to start or stop a task running on any node on the network. When the user requests a task be started on a given node, the kernel running locally on that node first ascertains whether or not the proper versions of component modules are available locally (strong versioning is a useful feature of the .NET Framework). If they are not, it queries the network for a node that does have the modules in question and downloads them automatically. Once the byte code of the modules that comprise the task all exists locally, the task is loaded.

One of the important features of the ROCI architecture is that modules and pins are self-describing

entities. Thus, when the user browses through the tasks, he or she can immediately have a complete description of the modules and pins in use. Given this information, the browser can automatically start appropriate modules locally to tap into the remote data for visualization or processing purposes. This can be very useful for debugging purposes during development and for situational awareness during deployed execution.

3 Simple Obstacle Avoider in ROCI

As mentioned, an application in ROCI may be composed of multiple tasks. Tasks can be specified connecting several building blocks (modules), each one offering a specific service. The connections determine the data flow from one block's outputs to another block's inputs. In fact, these connections can be made seamlessly between modules in different tasks, even if they are running on different nodes.

To demonstrate this, we have developed and successfully executed a simple obstacle avoidance task using one of our ClodBuster robots equipped with an omnidirectional camera and IEEE 802.11b wireless network [1]. In this task, the robot's heading is computed based on a range map constructed from an omnidirectional edge image of the environment (Figure 2). The edge image and the heading direction can be displayed simultaneously by another task running on a remote computer.

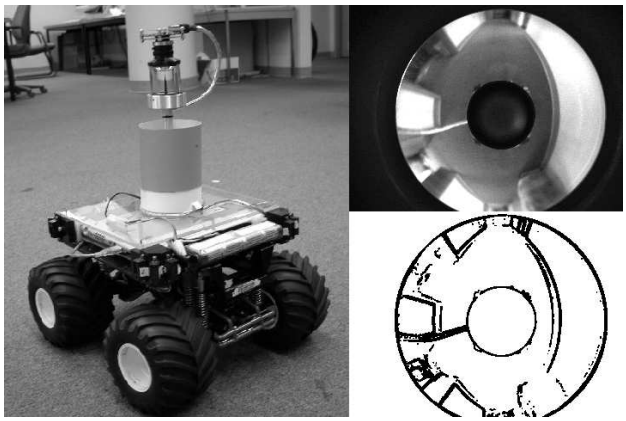

Figure 2: A Clodbuster robot and an image captured by the omnicam, before and after the edge extraction.

A diagram of the present application implemented in ROCI can be seen in Figure 3. It is composed of three tasks, two running in the robot and one running in a remote computer. The main task is the Obstacle Avoider that is comprised of 5 modules: the OmniCam captures an image and exports it to other modules through a video pin. The Edge Detector processes this image and makes it available for the Range Mapper that computes a desired bearing for the robot. The Range Mapper also receives calibration parameters from the OmniCam and exports a video pin containing the input image with the heading direction highlighted. The bearing is exported and used by the Robot Controller module which generates inputs to the Grasp Board that is the interface to the servo motors. Running in the same node, there is also a task that reduces the resolution and subsamples the video stream exported by the Range Mapper so it can be better transmitted over the network. Finally, there is a Video Preview running on another computer that allows a remote operator to observe the video broadcast by the robot.

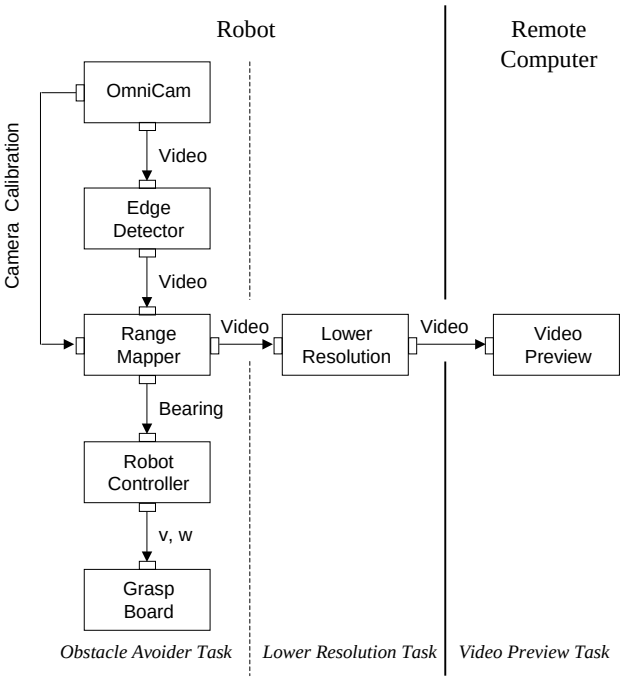

Figure 3: Diagram of the tasks and modules for the obstacle avoider implemented in ROCI.

We need to reinforce that each one of these modules is self-contained and can run independently of the others. Consequently, different applications can be specified and performed reusing some of these modules and adding new ones. For example, the Omnicam module could be used in a tracking task connected to some feature extractor module or the Edge Detector could receive input from a regular camera in a different task specification. The idea is to have a library of modules that can be executed by robots according to their capabilities. Also, during execution, connec-

tions between modules can be dynamically made or changed allowing modules to receive different inputs from different sources. For example, in the application described above, a remote operator could repin the input of the Lower Resolution module to any one of the modules that output a video pin. The operator has this information, since modules and pins are self-describing and the network database is continuously updated. Consequently, the operator could observe images directly from the Omnicam module or from any other module that is exporting a video pin at the moment.

As will be discussed in the next section, these capabilities are especially important in multi-robot applications in which the number of robots, communication and sensor constraints may change dynamically during execution.

4 Multi-Robot Scenario

Let us consider a multi-robot task in which n robots must perform a visual reconnaissance of a certain area. Each robot i is equipped with a GPS and an omnidirectional camera and its objective is to send its position $\boldsymbol{x_i}$ and an image captured from that position to a base station. In ROCI, this task could be specified by three modules, as shown in Figure 4. The Camera captures images and exports them as a video pin. It also outputs the camera parameters which are not being used by any module at this moment. There is another module to get the robot's GPS coordinates and a third one that simply processes the video and the robot's position and sends it to the base station. Initially, each robot can act independently from its teammates (it is a loosely coupled task), but the ROCI kernel running in each robot continuously updates its database, keeping track of the other network nodes.

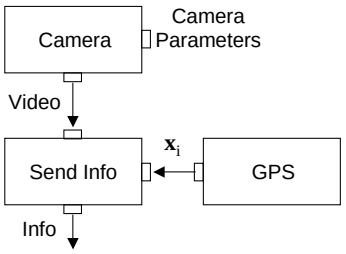

Figure 4: Diagram of a reconnaissance task running in each robot.

Now, suppose that one of the robots (for example, robot k) loses its GPS information. In order to continue performing its task, it should find a way of replacing that information source. Since the other robots in the team may still be able to compute their positions, robot k can rely on a cooperative localization scheme to localize itself [3]. The cooperative localization works as follows: each robot j in the neighborhood of k computes the position of the other robots in its field of view based on its GPS and camera image and exports this information. Then, based on the position estimates received from its neighbors, robot k will be able to localize itself.

Using the ROCI framework, it is easy for robot k to start the cooperative localization dynamically. First of all, it has an updated list of the other robots in the network that can perform localization since the ROCI kernel maintains this information. So, robot k will be able to inject a new localizer module in some of its teammates and dynamically pin it to the modules that are already running to get the information that it needs (Figure 5a). The localizer will receive information from the camera (image and calibration parameters) and the GPS and export the estimated position of all the robots that are visible ($\boldsymbol{x_{j1}}, \boldsymbol{x_{j2}}, \ldots, \boldsymbol{x_{jk}}$). Robot k will also start running a local module called Position Estimation to get the position estimates from its neighbors and compute its position, automatically repinning the input of the Send Info module to the output of this new module, as shown in Figure 5b. Thus, this dynamic reconfiguration allows the robot that lost its GPS to continue executing its task.

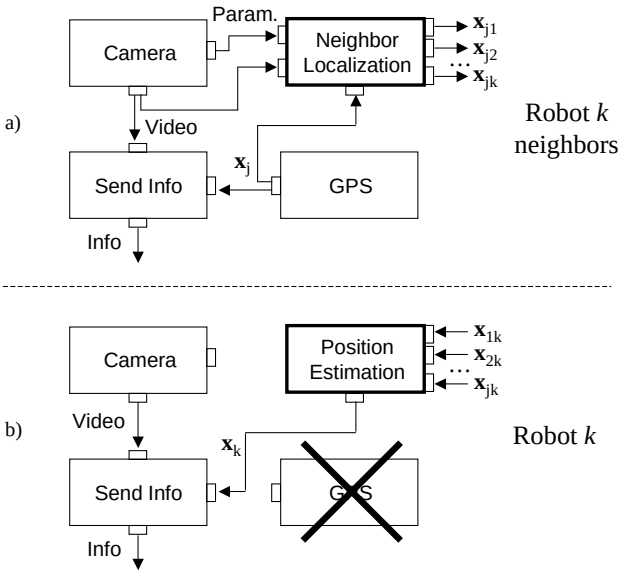

Figure 5: Scenario if robot k loses its GPS: a) robot k injects a new localizer module on its neighbors and b) robot k starts a local module and dynamically repins its other modules to use the new information.

It is important to note that some aspects of the

cooperative task were not detailed in the description above. For example, we did not define exactly the concept of neighborhood in terms of sensing and communication in this paper. Also, we did not give any details about the cooperative localization or about the controllers and coordination techniques that should be used in this task. These points should be specified in the implementation of the task, but our main objective here is to present the ROCI framework, showing how it allows multi-robot teams to adapt to dynamic changes that typically occur during the execution of cooperative tasks.

5 Conclusion

In this paper we presented ROCI, a programming framework for distributed ensembles of sensors and actuators. Applications in ROCI are composed of tasks and can be built dynamically by connecting several modules, which gives a great flexibility to the programmer. Each module encapsulates a process that consumes data from its input and produces data on its output. Modules are completely self-contained and can be reused in different tasks and applications. A pin architecture is used to connect modules. These connections can be made seamlessly between modules in different tasks, even if they are running on different nodes, creating a network transparent programming environment. The ROCI framework is strongly typed, allowing the development of more robust yet high performance applications. Two of the ROCI's main features are the ROCI kernel and the self-describing nature of modules and pins. Together, they allow the creation of an updated view of network nodes, services, and data, providing situational awareness for users and applications. This is a key requirement for programming dynamic distributed multi-robot teams.

Our present and future work is direct towards implementing several multi-robot applications using the ROCI framework. Under the DARPA's MARS project, we are developing a new team of robots (both aerial and terrestrial) that will be fully programmed and controlled using ROCI. Several multi-robot capabilities are being developed for this team, such as outdoors navigation, cooperative localization, stereo obstacle avoidance and communication sensitive behaviors. Also, our multi-robot team will have to interact with other robots programmed in different frameworks, more specifically Player [5] and MissionLab [6]. We are working on a common interface between ROCI and these systems that will use SOAP and XML to exchange data between different platforms. This project will provide an excellent test bed for the ROCI framework, and we expect to have some multi-robot applications running very soon.

Acknowledgment

This work was in part supported by: DARPA MARS NBCH1020012 and NSF ITR (ANTIDOTE) CCR02-05336.

References

[1] R. Alur, A. Das, J. Esposito, R. Fierro, G. Grudic, Y. Hur, V. Kumar, I. Lee, J. Ostrowski, G. Pappas, B. Southall, J. Spletzer, and C. Taylor. A framework and architecture for multirobot coordination. In D. Rus and S. Singh, editors, *Experimental Robotics VII, LNCIS 271*. Springer Verlag, 2001.

[2] T. Balch. *Behavioral Diversity in Learning Robot Teams*. PhD thesis, College of Computing - Georgia Institute of Technology, 1998.

[3] A. Das, J. Spletzer, V. Kumar, and C. J. Taylor. Ad hoc networks for localization and control. In *Proceedings of the IEEE Conference on Decision and Control*, 2002.

[4] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[5] B. Gerkey, R. Vaughan, K. Stoy, A. Howard, G. Sukhatme, and M. Mataric. Most valuable player: A robot device server for distributed control. In *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems*, pages 1226–1231, 2001.

[6] D. MacKenzie, R. Arkin, and J. Cameron. Multiagent mission specification and execution. *Autonomous Robots*, 4(1):29–52, 1997.

[7] D. Martin, A. Cheyer, and D. Moran. The open agent architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128, 1999.

[8] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of 1999 IEEE International Conference on Robotics and Automation*, pages 1144–1151, 1999.

[9] R. Simmons and D. Apfelbaum. A task description language for robot control. In *Proceedings of the 1998 IEEE/RJS International Conference on Intelligent Robotics and Systems*, pages 1931–1937, 1998.