INTEGRATION BETWEEN TWO ROBOTIC PROGRAMMING FRAMEWORKS

Luiz Chaimowicz*, Daniel Silva Chaltein de Almeida*

* *VeRLab – Laboratório de Visão e Robótica*
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, MG.

Emails: `chaimo@dcc.ufmg.br, chaltein@dcc.ufmg.br`

Abstract— With the widening availability of stable robotic platforms, there has been a great effort in developing good programming frameworks in robotics. Unfortunately, most of these efforts are done in parallel and the resulting frameworks are completely independent from each other. This paper presents a methodology for integrating two of these frameworks: ROCI and Player. The main objective is to create a mechanism that allows developers to benefit from the strengths of both platforms when developing software for different robots.

Keywords— Robotic programming frameworks, ROCI, Player.

1 Introduction

In recent years, there has been an increasing interest in the software side of robotics. With the widening availability of stable robotic platforms, developers can focus on software issues such as modularity, code reuse, algorithm efficiency and software architecture. The general goal is to be able to abstract from the hardware, writing programs that are not tied to specific architectures and can be easily reused with different robotic platforms.

With this general objective in mind, different programming frameworks have been proposed in the robotics community. For example: Player (Gerkey et al., 2001), Carmen (Montemerlo et al., 2003), ROCI (Chaimowicz et al., 2003), CLARAty (Nesnas et al., 2003), Miro (Utz et al., 2002) and, more recently, the Microsoft Robotics Studio (Microsoft, 2006). Despite some efforts for creating standards (RETF, 2003) and integrating some these platforms (Côté et al., 2004), it is still very difficult to use them in conjunction and benefit simultaneously from the strengths of each framework.

In this paper, we present a methodology for integrating two of these programming frameworks: ROCI and Player. ROCI (*Remote Object Control Interface*) has been developed by the GRASP Laboratory at the University of Pennsylvania – EUA with the collaboration of the Vision and Robotics Laboratory (VeRLab) at UFMG – Brazil. It is an objected oriented, strongly typed programming framework that facilitates the development of distributed applications for dynamic multi-robot teams. On the other hand, Player is one of the most used frameworks in the robotics community. It allows the control and simulation of several robotic platforms and incorporates various algorithms for robot navigation, localization, mapping, etc.

For us, this integration is particularly important for two reasons. Firstly, we want to be able to use some of the robotic platforms already targeted by Player, including Player's 2D and 3D simulators Stage and Gazebo, with ROCI code already developed and used in other robots. Secondly, we intend to further investigate the challenges of robotic software integration, starting a path that will eventually allow the development of more generic, reusable code in robotics.

This paper is organized as follows. In the next section, we describe ROCI, the software framework we have been developing. Section 3 gives an overview of the Player framework. Section 4 discusses the methodology used for integrating these frameworks while Section 5 shows some experiments preformed for validating this methodology. Finally, section 6 brings the conclusion and directions for future work.

2 ROCI

ROCI is a self-describing, objected oriented, strongly typed programming framework that facilitates the development of robust applications for dynamic multi-robot teams (Chaimowicz et al., 2003; Cowley et al., 2004; Cowley et al., 2006). In ROCI, each robot is considered a node which contains several processing and sensing modules and may export different types of services and data to other nodes.

Each node runs a kernel that can be considered a high level OS. This kernel mediates the interactions of the robots in a team, handling task allocation and execution, managing the network and maintaining an updated database of other nodes in the ROCI network. ROCI tasks are encapsulated in this kernel providing a very lightweight and efficient way of identifying relevant processes running on the nodes and giving

users remote control over them. Also, the kernel automatically load, in runtime, all the modules necessary for running a certain task. No extra step is necessary during compilation to link several objects, which is common in frameworks that deal with several source code libraries. A new task in ROCI can be "built on-the-fly" dynamically connecting independent modules.

The control functionality needed by such kernel is made possible by self-contained, reusable modules. Each module encapsulates a process which acts on data available on its inputs and presents its results on well defined outputs. Thus, complex tasks can be built by connecting inputs and outputs of specific modules. 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. It is defined in an XML file which specifies the modules that are needed to achieve the goal, any necessary module-specific parameters, and the connections between these modules. At runtime, these connections are made through a pin architecture that provides a strongly typed, network transparent communication framework. A good analogy is to view each of these modules as an integrated circuit (IC), that has inputs and outputs and does some processing. Complex circuits can be built by wiring several ICs together, and individual ICs can be reused in different circuits.

The main interface between a human operator and the robot team is the ROCI Browser. The browser displays the multi-robot network hierarchically: the operator can browse nodes on the network, tasks running on each node, the modules that make up each task, and pins within those modules. The browser's main job is to give a user command and control over the network as well the ability to retrieve and visualize information from any one of the distributed nodes. Specifically, using the browser, the user can start, stop and monitor the execution of tasks in the robots remotely, change task parameters, send relevant

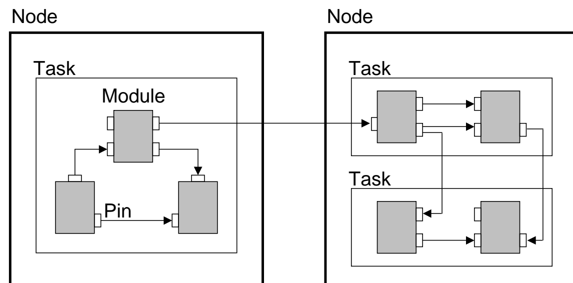

Figure 1: ROCI Architecture: tasks are composed of modules and run inside nodes. Communication through pins can be seamlessly done between modules within the same task, modules in different tasks, or in different nodes.

control information for the robots or even to tap into and display the outputs of pins for which display routines exist. Also, elaborated missions can be constructed within the browser using *scripts*. Mission scripts can be generated online or offline, and specify a sequence of actions that should be performed by a team member.

ROCI has been actively used in various projects, ranging from vision-based control (Rao et al., 2004) to multi-robot missions in outdoor environments (Chaimowicz et al., 2005). As mentioned, once the modules have been implemented, it is very simple to reuse them in different tasks: the user simply has to specify the desired modules and their pin connections in the task declaration.

To illustrate this, we present two task diagrams that use the same controller module in different scenarios (Figure 2). Basically, we have a high level controller module that drives a robot through a series of waypoints. This module receives two inputs: the robot pose (x, y, θ) and a list of waypoints (X_i, Y_i) and outputs velocities (v, ω) to the robot's low level controller. This waypoint controller is completely self-contained and can be used in combination with different modules that export and import the correct pins.

Figure 2(a) shows an implementation where the waypoint controller receives input from a GPS and a waypoint planner and outputs velocities to a clodbuster robot. This configuration was used to navigate teams of robots in urban environments as part of the DARPA - MARS2020 project (Chaimowicz et al., 2005). The waypoint controller (the exact same module) has also been used for some indoor demos in the GRASP Laboratory in the configuration shown in Figure 2(b). In this case, instead of GPS, an overhead camera is used for localization: the robot is tagged with a colored blob and a color blob detector is used to compute its position. The waypoint list is given by a user interface running in the ROCI Browser and the robot being controlled is a Segway. It is important to mention that some of the modules are running in different machines but, since the pin architecture is network transparent, the programmer can abstract from this.

3 Player Framework

The Player framework is a collection of tools that enables research in robotics. It is composed by the Player Device Server, a 2D simulator called Stage, and a 3D simulator called Gazebo. Its main goal is to simplify software development and reuse in robotics, specially for multi-robot systems (Gerkey et al., 2003). Player is one of the most used programming frameworks in the robotics community and has drivers to a wide range of robots. This is why it was primarily chosen in this work.

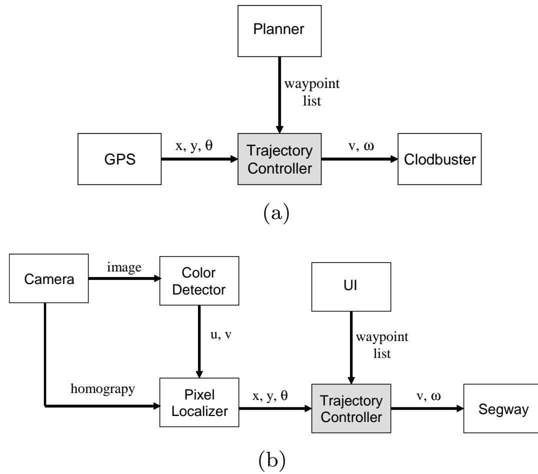

Figure 2: Diagrams of two tasks using the same waypoint controller module: (a) outdoors with a GPS and a planner; (b) indoors with an overhead camera and an user interface.

3.1 Player

Player is a device server that provides a powerful, flexible interface to a variety of sensors and actuators. Each robot or sensor is linked to a device through a specific driver and is accessed by a client program using the interface provided by that driver. In this way, Player acts as a *hardware abstraction layer*. There are standard interfaces which are supported by many drivers, one for each type of robot or sensor. For example, the *position2d* interface is supported by several drivers, such as the *p2os* for Pionner robots and the *rflex* for RWI robots. Among other things, this interface can control linear and angular velocities and feedback odometry information. A program that uses it is able to control, without any change or recompilation, different kinds of robots. The only modification would be in a configuration file, used by the player server, that contains the declaration of each device and its respective driver.

A client-server architecture allows programs to access and control the physical devices. Client programs connect to the player server that, as explained above, connects physically to every device in the system. Typically, clients have methods for reading and writing data and configuring the devices using the interfaces explained above. Consequently, clients have access to the hardware in a simple and transparent way. All communication is done using a Transmission Control Protocol (TCP) socket. Since communication follows that standard, player clients can be written in any language that supports sockets.

As mentioned before, there is a configuration file that informs the player server of each device, its respective driver, and which port it is connected to. Note that there can be more than one server and a client may connect to multiple servers. For every server, each interface is provided with an index such that there can be multiple interfaces of the same kind in the server. Figure 3 shows an example of a configuration file. It declares drivers for a robot (*p2os*) and a GPS sensor. The GPS declaration could be interpreted as follows: there is a device on the /dev/ttyS18 port controlled by the *garminnmea* driver that is accessed through the gps interface with index 0.

```
driver (
   name "p2os"
   provides ["odometry:::position2d:0"]
   port "/dev/ttyS0"
)
driver (
   name "garminnmea"
   provides ["gps:0"]
   baud "19200"
   port "/dev/ttyS18"
)
```

Figure 3: Example of a Player configuration file.

3.2 Stage

Stage is a 2D simulator that provides virtual Player robots which interact with simulated rather than physical devices. Various sensor models are provided, including sonar, scanning laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry. In Stage, each simulated entity acts like its real counterpart. One of its main objectives is to provide data that represents as close as possible the data given by real robots and sensors. Even the data transmission ratio of specific devices is simulated. In addition, player clients can not tell the difference from real hardware to Stage's device simulations, so tests and real experiments can be done using the same programs. Another important feature is that Stage is scalable to large robot populations. Consequently, it is widely used in multi-robot simulations. Stage can provide a simulation very close to the real experience, saving a lot of time and effort when developing and testing the programs.

3.3 Gazebo

Gazebo is a 3D simulator for Player applications. Like Stage, it is capable of simulating robots, sensors and objects, but does so in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics). Therefore, Gazebo can simulate environments with much more details. Because it deals with a 3D environment, Gazebo usually is used with small numbers of robots, due to the heavy image and physics processing.

4 Platform Integration

ROCI and Player are equivalent in the sense that their main objective is to allow the development of modular and reusable code for robotics. But, as described in sections 2 and 3, they have several features that complement each other. As mentioned, our primary interest in this integration is to take advantage of these complimentary features in the development of new applications. Specifically, in the near future, we want to target some Player enabled platforms (such as the Pioner AT3 robot and the stage simulator) with our outdoor navigation modules developed in ROCI.

The proposed solution for integrating ROCI and Player relies on a client-server structure. Initially, we want to use data generated by ROCI modules to control robots through a player interface. So, ROCI will act as a server, exporting data to player clients. For this, we are using TCP Sockets which allow a transparent multi-platform networked communication. Moreover, this client-server structure is very general and may allow other clients developed in other frameworks to get data from ROCI as well.

As described in section 2, objects called pins are responsible for the communication between modules in ROCI. All data produced and consumed by the modules is encapsulated in these objects. Thus, on the ROCI side, we created a specific module called *TCPServer* for exporting pins. This module receives a pin, converts this pin from its internal strongly typed format to a raw text message and sends it over the network. This module may work with any type of pin by calling an abstract method that converts it to raw text. This method is declared in the pin base class and redefined in each of the pin classes that inherits from it. This ROCI module is also responsible for establishing network connections: besides the pin conversion it has a thread that keeps listening to connection requests from clients.

On the Player side, we implemented a simpler solution. We placed the code for receiving ROCI data directly on the the player client program. Basically, the program connects to the ROCI server module, gets the data and uses it to control player devices through the player server, using the default interfaces (for example, the interface *position2d* for moving the robot). Figure 4 shows a diagram of the integration: a specific module in ROCI is responsible for sending data over a TCP/IP network. This data is received directly by the player program that interfaces with the devices through the player server.

An alternative solution on the Player side would be to define ROCI as a device inside the player server architecture providing some interfaces to access it. As mentioned, Player already uses standard sockets and TCP/IP connection to

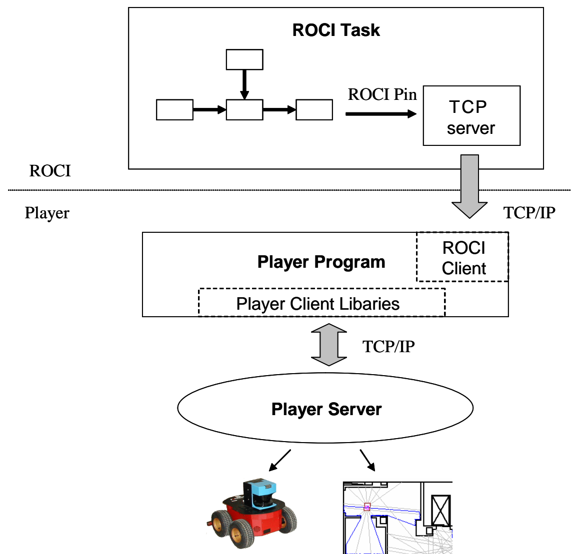

Figure 4: Integration between ROCI and Player.

communicate client programs and devices, so this would be a natural solution. But for now, since our main objective is to create player clients to control robots based on ROCI data, we opted for making things simpler and connect the player client directly to ROCI instead of going through the burden of writing new devices and interfaces. This will eventually be necessary for developing more complex applications, in which bidirectional communication is necessary.

5 Experiments

In order to test the integration mechanism described in the previous section, we performed a couple of experiments in which a ROCI task was used to guide simulated and real robots interfaced through Player.

In these experiments, a single robot was teleoperated from a remote computer using a joystick. The ROCI task was composed by three modules: one module to get the information from the joystick, another to translate the joystick data to linear (v) and angular (ω) velocities and a third one, the *TCPServer* module explained in the previous section, to send these velocities as raw byte streams over the network to the player program. As mentioned in Section 2, a task in ROCI can be started and monitored through the ROCI Browser. Figure 5 shows the browser during the execution of this task. Specifically, the tabs on the bottom shows the modules that make up the task and the popup window displays the inputs and outputs of the TCPServer module.

On the Player side, we implemented a client program that receives these streams, converts them back to velocities and sends them to the robot through a specific interface. As discussed in Section 3, one of the greatest benefits of the

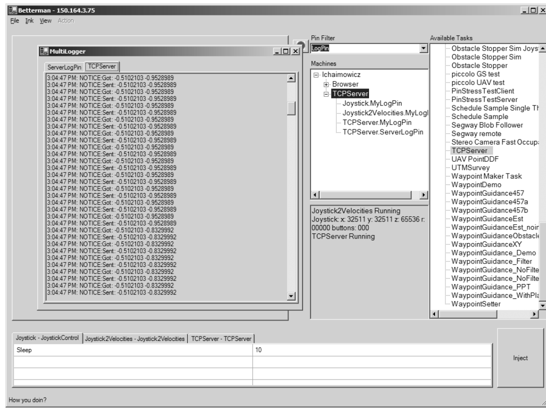

Figure 5: ROCI Browser during the execution of the task.

Player framework is the possibility of using the same program to control both simulated and real robots. Taking advantage of this feature, we performed two sets of experiments. Firstly, we use the ROCI task to guide a simulated robot in Stage. Figure 6 displays a snapshot of the stage simulator with the trajectory performed by the robot in a virtual environment.

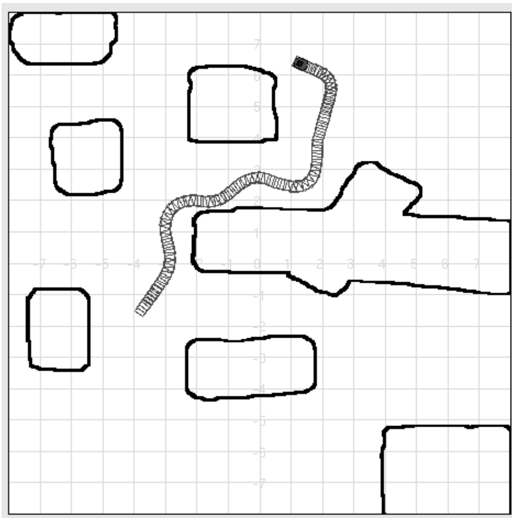

Figure 6: Robot trajectory in the Stage simulator.

The second set of experiments were performed using a Pioneer 3 All Terrain Robot (P3AT), shown in Figure 7. It is a four wheel, differential drive robot equipped with GPS, encoders, gyroscope, laser range scanner among other sensors. An on board laptop is responsible for running the player code. During the experiments, we logged the linear (v) and angular (ω) velocities commanded by the joystick and the velocities executed by the robot. The results, presented in the graph of Figure 8, show that the robot programmed in Player is able to follow the remote commands received from the ROCI task. The corresponding robot trajectory navigating in an indoor environment is shown in Figure 9.

Figure 7: Pionner 3 All Terrain (P3AT) used in the experiments.

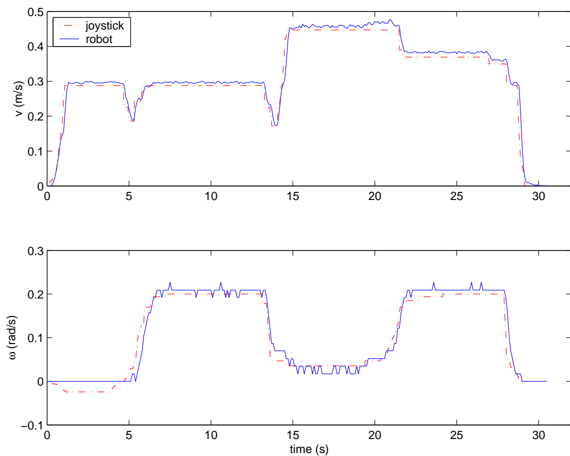

Figure 8: Linear and angular velocities commanded by the joystick (dashed) and executed by the robot (solid).

These experiments successfully demonstrated the integration between ROCI and Player. More sophisticated experiments, including sensor based navigation in outdoor environments, are being planned to fully take advantage of the benefits of both frameworks.

6 Conclusion

In this paper, we presented a methodology for integrating two robotic programming frameworks: ROCI and Player. The main advantage of this integration is to allow the use of features and code from both frameworks, leveraging the notion of code reuse. We used a client-server approach, in which ROCI acted as a server sending data to player programs responsible for interfacing the robots. Experiments with simulated and real robots were successfully performed demonstrating the feasibility of this integration.

Future work is directed towards two different fronts. Firstly, we want to improve the proposed methodology to allow bidirectional communica-

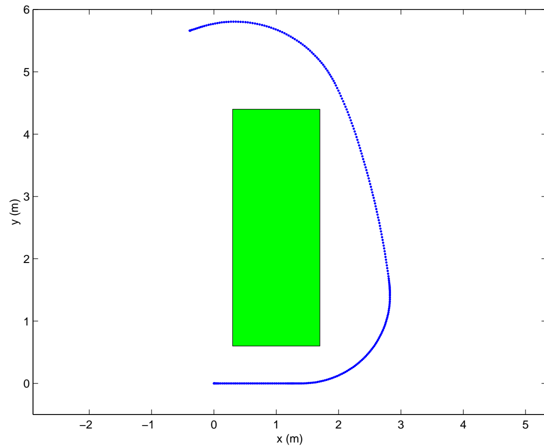

Figure 9: Trajectory performed by a P3AT robot.

tion. This would allow, for example, that data gathered from player sensors be feedback to ROCI modules, or that a robot controlled by ROCI have inputs generated by a player program. The second front is to use the ROCI-Player integration for developing other robotic applications. For example, we have a multi-robot scenario in which we want to use the outdoor navigation tasks developed in ROCI (Chaimowicz et al., 2005) with a group of Pionner robots controlled by Player. In this case, the reuse of large portions of code will simplify the programmer's job and allow them to focus on the implementation of application specific code.

Acknowledgments

The authors would like to thank Marco Fonseca Marino for preliminary work on the platform integration. This work was partially supported by CNPq and FAPEMIG.

References

Chaimowicz, L., Cowley, A., Gomez-Ibanez, D., Grocholsky, B., Hsieh, M. A., Hsu, H., Keller, J. F., Kumar, V., Swaminathan, R. and Taylor, C. J. (2005). Deploying air-ground multi-robot teams in urban environments, *in* L. E. Parker, A. Schultz and F. Schneider (eds), *Multi-Robot Systems: From Swarms to Intelligent Automata*, Vol. III, Kluwer, pp. 223–234.

Chaimowicz, L., Cowley, A., Sabella, V. and Taylor, C. J. (2003). ROCI: A distributed framework for multi-robot perception and control, *Proceedings of the 2003 IEEE/RJS International Conference on Intelligent Robots and Systems*, pp. 266–271.

Cowley, A., Chaimowicz, L. and Taylor, C. J. (2006). Design minimalism in robotics programming, *International Journal of Advanced Robotic Systems* **3**(1): 31–36.

Cowley, A., Hsu, H. and Taylor, C. J. (2004). Distributed sensor databases for multi-robot teams, *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pp. 691–696.

Côté, C., Létorneau, D., Michaud, F., Valin, J.-M., Brosseau, Y., Raïevsky, C., Lemay, M. and Tran, V. (2004). Code reusability tools for programming mobile robots, *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems*.

Gerkey, B. P., Vaughan, R. T. and Howard, A. (2003). The player/stage project: Tools for multi-robot and distributed sensor systems, *Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003)*, pp. 317–323.

Gerkey, B. P., Vaughan, R. T., Stoy, K., Howard, A., Sukhatme, G. S. and Mataric, M. J. (2001). Most valuable player: A robot device server for distributed control, *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1226–1231.

Microsoft (2006). Microsoft Robotics Studio. http://msdn.microsoft.com/robotics/.

Montemerlo, M., Roy, N. and Thrun, S. (2003). Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit, *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems*, pp. 2436–2441.

Nesnas, A., Wright, I., Bajracharya, M., Simmons, R. and Estlin, T. (2003). CLARAty and challenges of developing interoperable robotic software, *Proceedings of the IEEE/RJS International Conference on Intelligent Robots and Systems*, pp. 2428–2435.

Rao, R., Taylor, C. J. and Kumar, V. (2004). Experiments in robot control using uncalibrated overhead imagery, *Proccedings of the International Symposium on Experimental Robotics (ISER)*.

RETF (2003). Robotics engineering task force. http://www.robo-etf.org.

Utz, H., Sablatnog, S., Enderle, S. and Kraetzschmar, G. (2002). MIRO - middleware for mobile robot applications, *IEEE Transactions on Robotics and Automation* **18**(4).