Distributed control of mobile robots with CORBA

Eric Colon[a], Hichem Sahli[b], Yvan Baudoin[a]
[a]Royal Military Academy – [b]Vrije Universiteit Brussel
eric.colon@rma.ac.be

Abstract

As robotics systems are becoming more complex, distributed, and integrated, there is a need to promote the construction of new systems as composition of reusable building blocks. System modularity and interoperability are key factors that enable the development of reusable software. The work described in this paper consists in two parts: the design of a generic control framework using CORBA as its communication middleware and the development and integration of different robots, sensors and processing components.

1. Introduction

For years, searchers have focused on embedded intelligence providing ad hoc implementation. Solutions have been tied to existing platforms, limited by software and hardware constraints (processor, memory, OS, communication, ...) or implementation costs. Everything had to be on-board and was limited by hardware capabilities. Nowadays hardware is far more affordable and wireless communication has become fast and reliable. It has consequently become easier to communicate and to implement distributed applications and with the recent progresses of the Internet, the notion of service has become familiar to many of us. Generally speaking a robot is already by itself a complex system but in order to perform useful tasks it must be equipped with external sensors and actuators. These have in most cases their own control system resulting obviously in a distributed architecture. For non-specialists, developing software for a single robot, without speaking of multi-robots systems, can rapidly become a nightmare. What is needed in robotics is a software framework that enables the easy development of distributed applications by providing functions that hide and automate low level mechanisms and provide the developer with a high level environment and let him concentrate on intelligent aspects of the application.

It is evident that this initiative is not unique and that other researchers and laboratories have been faced with similar frustrations when developing their control software and have consequently developed their own frameworks based on similar requirements. One of the best known examples is certainly the module generator GenoM. It is a tool that helps building real-time software architectures and corresponds to the functional level of the architecture developed at the LAAS-CNRS (Alami, R & all, 1998). MCA2 is a software framework with real-time capabilities that is rapidly gaining in popularity (available at http://mca2.sourceforge.net). The DCA framework (Peterson, L.; Austin, D.; Christensen, H. 2001) has been developed to control a mobile manipulator. It relies on a process-algebra for specifying tasks. The main drawback in GenoM, DCA and MCA2 are their proprietary communication mechanism. Some frameworks solve the aforementioned limitations by building up on communication middleware. Miro (Enderle, S & all, 2001) is a distributed object-oriented framework developed in C++ for Linux that is based on CORBA technology. It offers synchronous and asynchronous communication and configuration capabilities through XML files. MARIE (Mobile and Autonomous Robotics Integration Environment) is a programming environment which aim is to develop an integration framework based on the mediator design pattern for distributed systems. MARIE uses ACE (Schmidt, D. C. & Huston, & S. D., 2003) as its communication library. All interactions between applications are done asynchronously. Orca (Brooks, A. & all, 2005) is an emerging open-source suite of tools for developing component-based robotic systems (available at http://orca-robotics.sourceforge.net). The Miro weak point is its behaviour engine and the lack of

remote management. MARIE's approach, as mentioned in (Côté, C. & all, 2004) suffers from many drawbacks, namely, overhead, complexity and system resource management. Our framework, which name is CoRoBA, tries to merge the strong points of the ones mentioned above while minimizing weak points. But as in all real projects, compromises have to be made and a perfect solution will never exist.

The work described in this paper consists in two parts: the design of a generic control framework using CORBA as its communication middleware and the development and integration of different robots, sensors and processing components.

Section 2 presents software requirements that have been identified by various review and that must be satisfied by the framework. In section 3 we discuss the selection and usability of middleware for distributed robot control system and detail Design Patterns that guided the software development. Section 4 covers Component design and implementation and gives some examples of existing components including a multi-robot 3D simulator. Section 5 describes a control application that has been developed with CoRoBA. Section 6 concludes with comments on further research directions.

2. Requirements

Developing reusable software is an incremental and iterative work as illustrated by the figure 1. It begins with the conceptual design. In this phase, requirements are captured, high level architecture is produced and the purpose and function of the components are described. At the second phase, the specification design is created. In this stage, the object model may be created, interfaces and sequence diagrams defined. In the final stage, the implementation design, the final details are laid down and physical systems and technologies are selected. At this stage we have sufficient details for starting the implementation. These phases may be repeated several times during the development cycle.

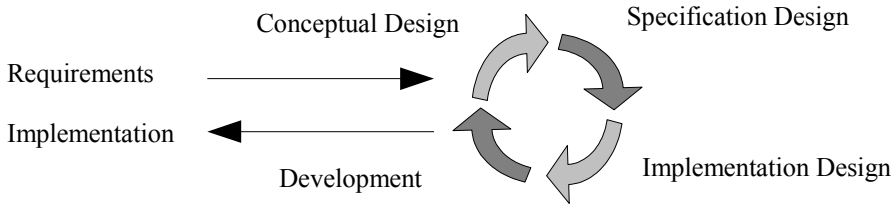

Figure 1. Software development phases

But this is not the end of the story. As the needs change, the code must also be adapted. The life cycle of object-oriented software has typically several phases, namely prototyping, expansionary and consolidating phases (figure 2).

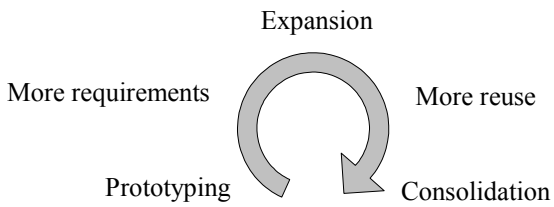

Figure 2. Software life cycle

At this stage the code has to be re-organized or re-factored. This means that from the beginning, software has to be designed with change in mind. In order to minimize these modifications or to simplify them, we need at least to focus on two points: requirements and design patterns. As we will

see in the next section, using design patterns in early stages of software development prevents later refactoring; that is because design patterns model what in the code needs to be changed. What is presented in this text is the first iteration in the development process.

Requirements have been derived from three different categories: In the first place we gathered requirements that apply to every networked applications, that is distribution, communication, computing and performance issues, etc.

Secondly, from a decomposition of applications based on increasing complexity we derived robotics control patterns. Identified Control Patterns classified from the simplest to the most complex one are:

- Direct control
- Sensor data processing
- Monitoring
- Direct tele-control
- Supervised and autonomous control
- Multi-robot control
- Multi-user control

The last category that generated requirements is development and deployment support.

We obtained finally a list of about 40 requirements. Developing a framework that meets all the requirements is certainly an utopia. In the architecture and specification design we have had to make choices and compromises. Most of the identified requirements have been implemented while others have been theoretically addressed.

3. Design and Implementation of the framework architecture

Framework and control architectures

A framework provides a kind of skeleton. The framework architecture is composed by families of related patterns and components, it defines how the different components are integrated into the framework and how they are interrelated. It defines also how components communicate with each other. It is generally admitted that a framework reverses the control paradigm; components written by the programmer are called back by the framework in function of events. The Framework dictates the architecture of applications developed with it.

A control architecture defines the design of a set of components in which perception, reasoning, and action occur. It also specifies the specific functionality and interface of each component, and the interconnection topology between components. The control architecture specifies which components are used and how they collaborate in a concrete application. The framework architecture must be flexible enough to allow different control architectures to be build using the same components (Figure 3).

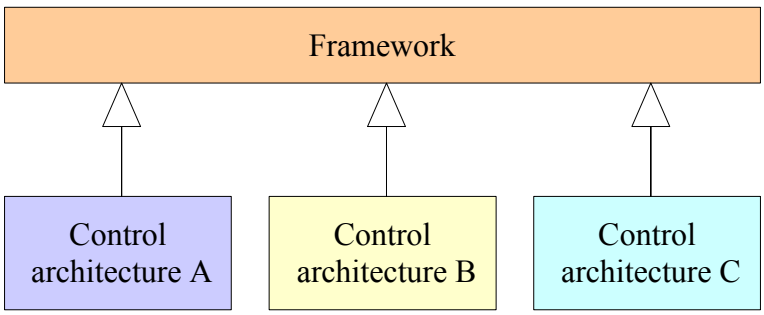

Figure 3. Different Control Architectures developed with the framework

Communication middleware

The first decision when developing distributed application is the choice of the communication middleware.

In distributed applications, programs need to invoke operations in other processes, often running in different computers. To achieve this, the following programming models are available:

- The Remote Procedure Call (RPC) which allows clients to call procedures in server programs running in separate processes and generally in different computers from the client.
- The Remote Method Invocation (RMI) that allows objects in different processes to communicate with each other.
- The distributed Event-based programming model that allows objects to be notified when events they have registered interest in have been emitted.

As most of current distributed systems is written in object oriented languages, we have only considered the two last models.

After having compared different Middlewares, we selected CORBA for our framework implementation. CORBA provides RMI and Event-based communication. CORBA is actually a specification of the Object Management Group (OMG). Presently more than 30 implementations are available on the market. Some are free software others are commercial products. The TAO implementation has been selected among others as the CORBA library for C++ developments for the following reasons. For developers of distributed and embedded applications who have stringent performance demands, TAO is a freely available, open-source, and standards-compliant real-time implementation of CORBA that provides efficient, predictable, and scalable quality of service (QoS) end-to-end. Unlike conventional implementations of CORBA, which are inefficient, unpredictable, non-scalable, and often non-portable, TAO applies the best software practices and patterns to automate the delivery of high-performance and real-time QoS to distributed applications.TAO has been ported to many operating systems including almost all UNICES, Win32, VMS, QNX, etc. TAO is based on the very popular ACE library which is a (efficient) platform-independent communication library.

Framework and Design Patterns

A Framework captures the design decisions that are common to its application domain. Applications based on frameworks can be built faster and have similar structures. They are consequently easier to maintain. The drawback is that applications are particularly sensitive to changes in the framework interfaces. Design Patterns actually help reducing these changes.

Design Patterns brings a theoretical foundation to software engineering. A pattern describes a problem which occurs over and over again and then describes the core of the solution to that problem, in such a way that you can reuse it again and again. Sets of interrelated patterns tailored specifically to work well together are called frameworks. In a framework-based development effort, the majority of the application is provided by the instantiated framework. Frameworks provide four primary usage strategies: instantiation, generalization, parametrization and extension.

Design Patterns involved in the Framework architecture design and how they are implemented have been presented in (Colon E. & Sahli H.,2005). We provide here a summary of those patterns; they fall into the following categories: Architecture Patterns and Distribution Patterns.

- Component-based Architecture Pattern organises system into replaceable units with opaque interfaces. It increases the robustness of the architecture in the presence of maintenance and is highly reusable in a variety of circumstances.
- The Channel Architecture Pattern is useful when data within a stream is sequentially transformed in a series of steps. It improves throughput capacity with the replication of units allowing efficient processing of multiple data in different stages of processing. It also improves reliability and safety through the simple addition of redundant processing.

- Remote Method Call Pattern extends the programming model used to invoke services locally and does the same think when the client and server do not reside in the same address space. The Client does not communicate directly with the Server but via a Client Stub that contacts the Server Stub, which invokes the specified method on the Server. The low-level network operations are hidden to the Client and Server.
- The Broker Pattern may be though of as a symmetric version of the Proxy Pattern. It provides a Proxy Pattern in situations where the location of the clients and servers are not known at design time. The Broker Pattern is a very effective means for hiding remoteness of clients and servers that greatly simplifies the creation of systems with symmetric distribution architectures.
- The Data Bus Pattern further abstracts the Observer Pattern by providing a common (logical) bus to which multiple servers post their information and where multiple clients come to get various events and data posted to the bus. The Data Bus Pattern is basically a Proxy Pattern with a centralized store into which various data objects may be plugged along with metadata that describes its contents.

CORBA communication models

CORBA offers different communication solutions that implement the Design Patterns described in the previous subsection. It give the developer a large freedom when implementing distributed applications.

CORBA relies on IDL (Interface Definition Language) compilers to generate stubs for clients and servers. Besides the 2-way method call, we can also make use of the Asynchronous Messaging Invocation (AMI) or of the event-based communication. The 2-way method is the most familiar to the programmer because it applies to remote calls the same principles as to a local method call. The method call blocks until the response is received from the remote object. It corresponds to a classical client-server scheme.

The AMI allows sending processing requests to a remote object without blocking the calling process. This later receives the response when this is available. A callback or a polling mechanism have to be used to get the response data. The AMI mechanism requires to modify the client but not the server which is unaware of this change.

There are many situations where the standard CORBA (a)synchronous request/response model is too restrictive. For instance, clients have to poll the server repeatedly to retrieve the latest information. Likewise, there is no way for the server to efficiently notify groups of interested clients when data change. For these reasons the OMG introduced the Event Service and the Notification Service.

The CORBA specifications define different methods for sending and receiving events: consumers and producers can push or pull the events (Figure 4). Implementations of the Events Service act as “mediators” that support decoupled communication between objects. Events are typically represented as messages that contain optional data fields. A primary goal of the Notification Service is to enhance the Event Service by introducing the concepts of filtering and configurability according to various quality of service requirements. Clients of the Notification Service can subscribe to specific events of interest by associating filter objects with the proxies through which the clients communicate with event channels. Furthermore, the Notification Service enables each channel, each connection, and each message to be configured to support the desired quality of service with respect to delivery guarantee, event aging characteristics, and event priority. The advantages of this communication method is counterbalanced by the complicated consumer registration (multiple interfaces, bidirectional object reference handshake, ...). Not all CORBA implementations implement the Notification Service.

Because we cannot preclude of any use or special needs, the different communication methods

described above are available in our control framework. In our design, we distinguish the control data flow from the management data flow. What concerns the control data flow, we have opted for an event based communication scheme while the management communication is based on the classical 2-way.

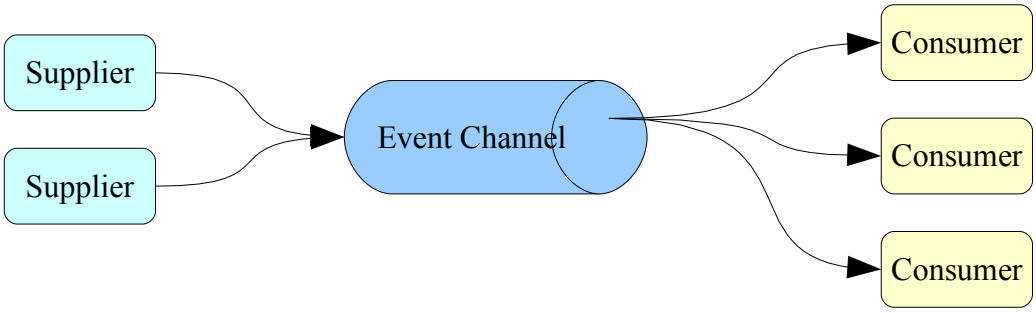

Figure 4. Event Channel Architecture

4. Design and Implementation of components

Components implementation

The component architecture actually implements the Hierarchical Control Pattern and the Message Queuing Pattern:

- The Hierarchical Control Pattern uses two types of interfaces: control interfaces that monitor and control how the behaviours are achieved and functional interfaces, which provide the services controlled by the other set of interfaces. The use of separate control and functional interfaces provides a simple and scalable approach when the system must be highly configurable.

- The Message Queuing Pattern provides a simple means for threads to synchronise and communicate information among one another using asynchronous communications implemented via queued messages. his pattern has many advantages.

CoRoBA Services have the generic architecture depicted in the figure 5.

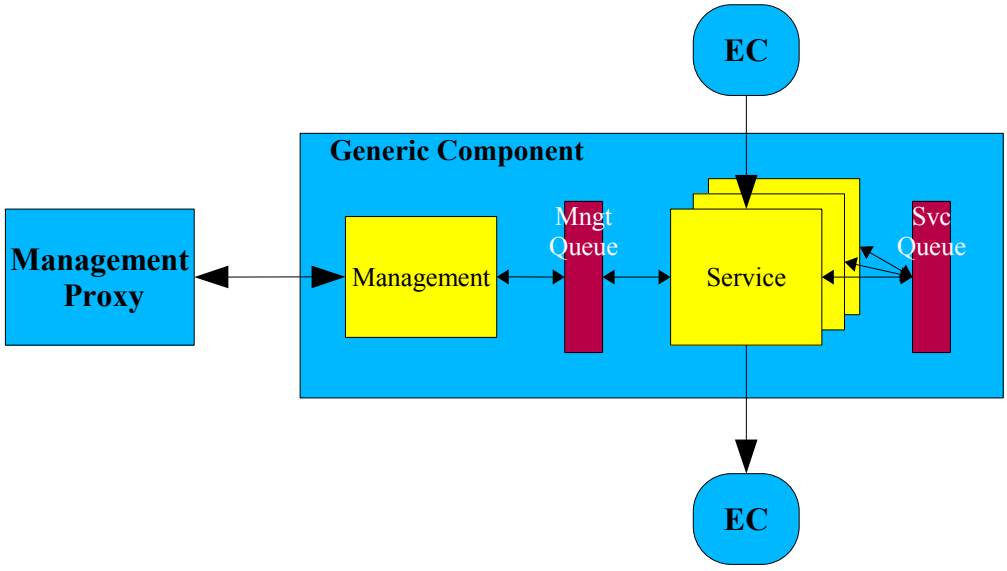

Figure 5. Generic Component Architecture

A service is composed of a main thread in which runs the Object Request Broker (ORB) and service threads that runs their own loop. The service thread can be managed remotely by invoking operations defined in the Service interface. It admits the following commands: start, pause, wakeup and stop. A service has 3 working modes that can be remotely selected: synchronous (process data when available), periodic (whose period can be changed) and external (synchronized on an external trigger). Threads in a service communicate by synchronized message exchange.

Component classic trilogy

Inspired by classical control applications, components have been divided in three categories: sensors, processors and actuators (Figure 6). Sensors have connections with the physical world and they output data to one Event Channel. Processors get their inputs from one Event Channel, they transform data and send the result to another Event Channel. Actuators have output connections with the physical world and received data from one Event Channel. Services can run on any machine in a network and are remotely managed by an administration application. Services register with a Naming Service that acts as a yellow phone book allowing to easily relocate services.

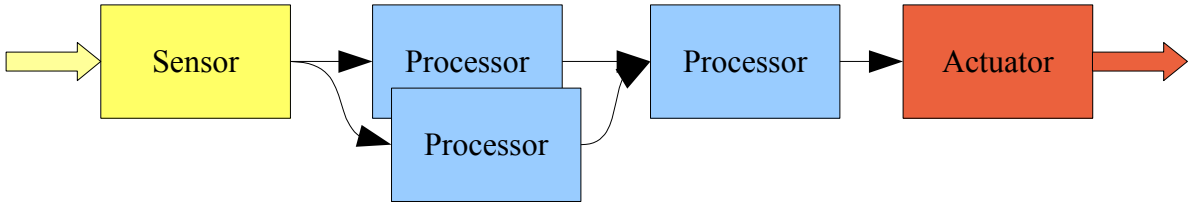

Figure 6. Typical components combination

Simulation

In order to easily test the capabilities of the framework , a 3D simulator for mobile robots has been developed. It is written in Java and Java3D. It can import 3D objects from VRML files. The characteristics of the simulator are:

- 3D environment with flat ground
- Multi robot simulation
- Obstacles collision detection (included collision between moving robots)
- Laser Distance sensors
- Planned and real followed trajectory visualisation
- Goal visualisation
- Fixed, tracking and on-board virtual cameras.

Two different views of the simulator are provided in the figure 7.

Presently 3 robots are available in the simulator: the Robudem (rectangular robot with 4 wheels) the Nomad and the Melexis(cylindrical robots with 3 synchronous wheels).

All simulated elements of the simulator have CORBA interfaces and can consequently communicate with CoRoBA components.

Several CoRoBA components communicating with the simulator have been developed:

- Sensors: Joystick control, Distance sensors, Robot pose, Goal Sensor, Trajectory sensor
- Processors: Simple collision avoidance algorithm, Fuzzy controller for goal seeking and trajectory following
- Actuators: Nomad , Robudem and Melexis.

The simulator can also be used to visualize the motion of real robots or robots simulated in another

simulation (distributed simulation). In the future it could be extended to build virtual worlds based on the detection of obstacles by real sensors.

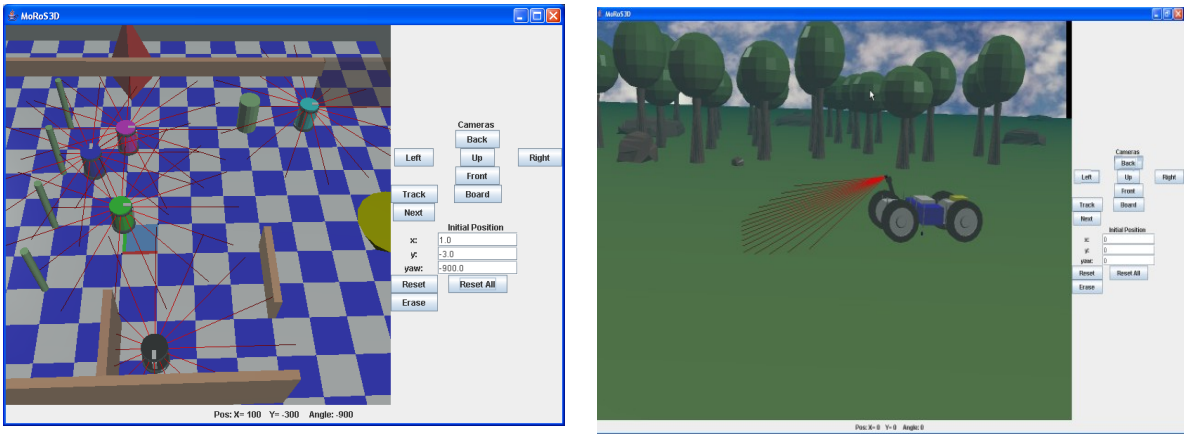

Figure 7. Multi-robot indoor and single robot simulation

System integration

While new services are designed from scratch, existing applications and systems will need to be integrated in the framework. Depending on the case this integration can be a trivial or a tricky job. We can distinguish different types of integration: database, library and application. Integration is made by providing a façade, aka an object wrapper following the Wrapper Façade pattern. The wrapper is in this case a distributed object and its implementation manages the interaction with the existing system. When developing the façade, one has to capture the business model of the existing system and develop interfaces that reflects this model. When wrapping libraries, we need to pay attention to serialize data access if existing libraries are not thread-safe.

If we consider the case of a Nomad200, we have an API which is available for the Linux OS. So we can write an actuator that translates CORBA method invocations to motion command functions of the API and a Sensor that reads the different sensor' values and propagates them as events. The methods listed in the *Nomad* interface keep the same signature as original functions of the Nomad200 API.

The integration of the second robot (Robudem) is simpler because we do not have to wrap any existing API. On the Robudem main computer runs a server that transfers data via shared memory between the Linux side and the RT-Linux side of the control application. The data structure that contains the robot kinematic information but also the motion commands are exchanged via sockets. As only one client is allowed to connect to the robot, the CoRoBA wrapper component breaks with the design standard presented above that separates sensors from actuators. It is actually a processor that receives motion commands via events, propagates the commands via sockets to the Robudem server, receives the kinematics data as the reply from the socket server and finally forwards the kinematic information via the output event channel. The control of the robot is implemented by other CoRoBA components.

5. Fuzzy Control Application

This application controls a simulated four-wheeled robot in an outdoor environment. The aim is to travel along way-points that are provided by the Goal Provider (Figure 8). The way-points are read from a file when the components are started. The Goal Scheduler compares the actual position of the robot with the coordinates of the way point. When this is reached, an event is sent to the Goal Provider in order to jump to the next way point. The Position Sensor propagates the position events that are used by the Goal Scheduler and the Goal Controller. The Goal Controller used a fuzzy logic control engine to navigate to the way point.

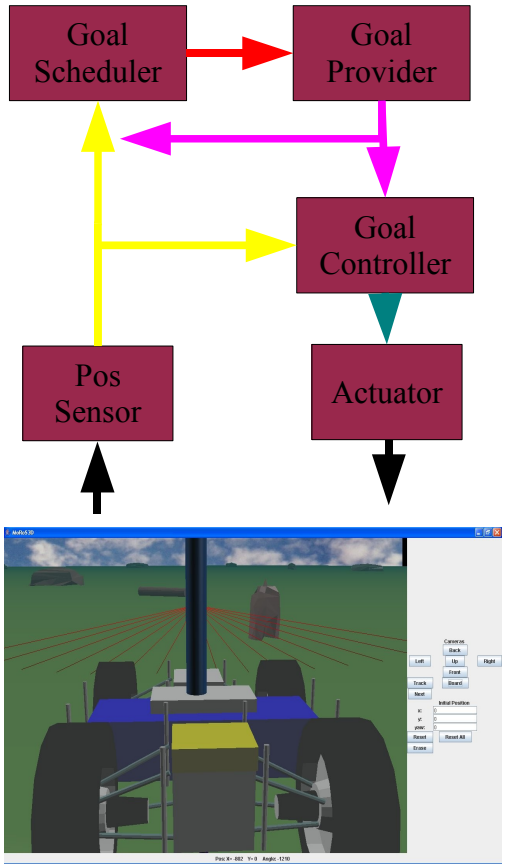

Figure 8. Distributed Fuzzy Control Architecture

6. Conclusion and further work

We present in this paper the most recent developments of a framework that allows to develop more flexible and more robust distributed robotic applications. This is the first iteration of the development cycle and the chosen design will need to be refined in function of results that will be obtained with different real applications. What is certainly missing for the moment are graphical tools for managing and monitoring the processes and for configuring the services. The simulator has also to be improved to become more flexible and terrain following has certainly to be implemented.
In order to further validate the framework, we need more demonstration applications; some could be simply build up by combining existing components in different ways while others will require the development of new components. We are also working on a general distributed Behaviour engine for multi-robots applications. In the next future, we will integrate other robots (Melexis developed at the VUB, a Cartesian scanner at the RMA) and real sensors (Laser, GPS, US, MD...).

7. References

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M. &. Ingrand, F. (1998), An Architecture for Autonomy, *International Journal of Robotics Research*, special issue on "Integrated Architectures for Robot Control and Programming", 1998

Brooks, A.; Kaupp, T.; Makarenko, A.; Orebäck, A. & Williams, S. (2005). Towards Component-Based_Robotics. Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005). In Press.

Colon, E. & Sahli, H. (2003), Software Modularity for Mobile Robotic Applications, Proceedings of Clawar2003, Sep 2003, Catania, Italy

Colon E. & Sahli H. (2005), CoRoBA, a multi mobile robot control and simulation framework, submitted to International Journal of Advanced Robotic Systems.

Côté, C.; Létourneau, D.; Michaud, F.; Valin, J.-M.; Brosseau, Y.; Raievsky, C.; Lemay, M.; Tran, V. (2004). Code Reusability Tools for Programming Mobile Robots, Proceeding IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2004.

Enderle, S.; Utz, H.; Sablatnön, S.; Simon, S.; Kraetzschmar, G.; Palm, G. (2001). MIRO: Middleware for autonomous mobile robots, in Proceedings of Telematics Application, pp 149-154, July 2001, Weingarten, Germany

Peterson, L.; Austin, D.; Christensen, H. (2001) DCA: A Distributed Control Architecture for Robotics, Proceedings IROS 2001 - IEEE International Conf. on Intelligent Robots and Systems, 29 Oct. – 3 Nov., 2001, Hawaï, USA,

Schmidt, D. C. & Huston, & S. D. (2003). *C++ Network Programming: Systematic Reuse with ACE and Frameworks*, ISBN 0-201-79525-6, Addison-Wesley Longman