CoRoBA, an Open Framework for Multi-Sensor Robotic Systems Integration

Eric Colon
Unmanned Ground Vehicles Center
Royal Military Academy
Avenue de la Renaissance,30, B-1000 Brussels, Belgium
eric.colon@rma.ac.be

Hichem Sahli
Dept. Electronics & Information Processing
Vrije Universiteit Brussel
VUB-ETRO, Pleinlaan, 2, B-1050 Brussels, Belgium
hichem.sahli@vub.ac.be

Abstract - **This paper presents the recent developments of a distributed framework for integrating multi-sensor robotic systems and controlling robots. It is based on the communication middleware CORBA.**

Index Terms - **control framework, distributed computing, CORBA**

I. INTRODUCTION

Computer systems are the backbones of all robotic applications. Since many years, searchers have developed ad-hoc programs for every new system. It is consequently difficult to build on existing systems and to reuse existing applications. There is a crucial need for reusable libraries, control framework and components. Efforts in this direction have focused on autonomous systems while we are also targetting tele-operation. Some are based on proprietary communication libraries like in [1] and [2], others are based on CORBA (Common Object Request Broker Architecture) like in [3] and [4].

Before going further, we give two useful definitions. A **framework** is a reusable, "semi-complete" application. It provides generic modules which generally need to be customized and extended in function of the application. An **architecture** is an instance of the framework. It is composed by selected modules which are customized and completed by application specific modules. A good designed framework should allow to implement different control architecture.

It is evident that such a framework has to be based on robust communication libraries and to claim to be open it must subscribe as much as possible to existing standards. When considering communication libraries it appears that one communication middleware has been present for more than 10 years and has now reached its maturity, this middleware is CORBA. This standard is briefly presented in section II. In the section III we explain the framework and component design while the software implementation is summarized in section IV. The 3D simulator that has been developed and integrated in the framework is briefly described in section V. The section VI gives practical informations on the framework use. The section VII considers different examples of typical tele-robotic applications and shows how the framework can be used in each case. The last section discusses some issues when integrating existing systems.

II. COMMUNICATION MIDDLEWARE

CORBA is actualy a specification of the Object Management Group (OMG). Presently more than 30 implementations are available on the market. Some are freely available others are commercial products. Their common characteristic is that none of these versions implements all specifications. While the third version of CORBA has been published, most of the CORBA implementations conform partially to the version 2.3 or 2.6 of the specifications.

CORBA offers different communication solutions that give the developer a large freedom when implementing distributed applications. 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 *en masse* when data change. For these reasons the OMG introduced the Event Service and the Notification Service.

In event-based communication we do not speak anymore about client and server but about suppliers and consumers (see Fig. 1). The CORBA specifications define different methods for sending and receiving events: consumers and producers can push or pull the events. 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

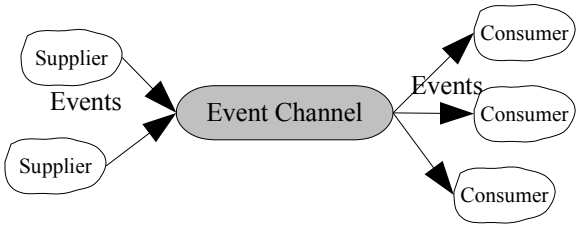

Fig. 1 Event based communication model

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 prioritization. The advantages of this communication method is counterbalanced by the complicated consumer registration (multiple interfaces, bidirectional object reference handshake, ...). Not all CORBA libraries implement the Notification Service.

III. FRAMEWORK AND COMPONENT DESIGN

The framework architecture defines how the different components are integrated into the framework and how they are interrelated. Framework services are divided into:

- Structural services: these offer the basic services that will be used by other components (Name server, Time server, configurator, supervisor, ...).
- Application services: the building blocks of an application (sensors, navigation, ...).

Because we cannot preclude of any use or special needs, the different communication methods 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. Because each service is managed individually, event communication does not offer any advantage. Moreover, by writing interfaces and methods we can create CORBA object references that are registered with the NameService and are used by other services to locate them.

Inspired by classical control applications, application services have been divided in three categories: sensors, processors and actuators. 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 (see also section V).

A service is composed of a main thread in which runs the Object Request Broker (ORB) and a service thread that runs its own loop (Fig. 2). The service thread can be

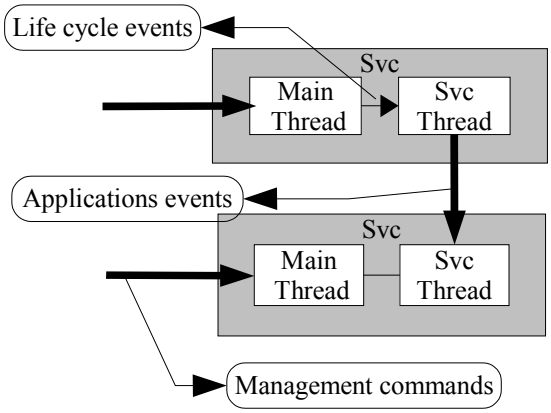

Fig. 2 Service structure

managed remotely. It admits the following commands: start, pause, wakeup, stop. The 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). The normal synchronous 2-way can also be used with methods declared in the idl interfaces (see section IV). The Table I lists what happens for the different service categories in the different modes.

TABLE I
WORKING MODES FOR COROBA SERVICES

	Sensor	***Processor***	***Actuator***
synchronous	Push event each time new sensor values are available	Process data and push events each time a new event is received	Sends data to external system each time a new event is received
Periodic	Reads the sensor values and push events at periodic intervals	Process data and push event at periodic intervals	Sends data periodically to external system
External	Reads sensor values and push events when externally triggered	Process data and push events when externally triggered	Sends data to external system when externally triggered
Synchronous (2 way)	Reads and returns sensor values	Process data received as parameters and return values	Sends data received as parameters to external system.

The service can also be remotely destroyed. Its life cycle is resumed in the Fig. 3.

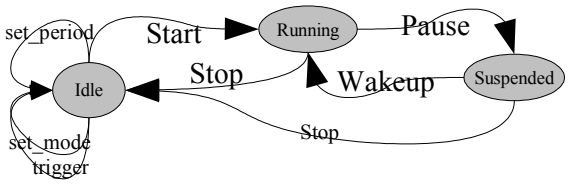

Fig. 3 Service Life Cycle

IV. SOFTWARE IMPLEMENTATION

A. Corba Libraries

For the software implementation of the framework we consider two programming language, i.e. C++ and Java. We have chosen the TAO implementation among others as the CORBA library for C++ developments for the following reasons: TAO [ ] has been ported to many operating systems including almost all UNICES, Win32, VMS, QNX, ... and TAO is based on the very popular ACE library which is a (efficient) platform-independent communication library. In [5] we showed that TAO has all the characteristics to satisfy the software requirements of the control framework. Concerning Java, as long as we use the 2-way communication scheme, the SUN® implementation coming with the Java Development Kit is sufficient but for the AMI or or the Notification Service we have used the popular CORBA implementation JacORB.

In [6] we presented in more details the communication options of CORBA and we analyzed which one is more suited to the different needs in a control framework. A demonstration application has also be presented, it allows a CORBA 2-way remote access to a classical serial port. In order to be invoked remotely, the methods have to be defined in CORBA interfaces written with the CORBA Interface Definition Language (idl). The idl compiler generates classes (C++ classes in the case of TAO, Java classes with JacORB) that allow the remote invocation of methods and the transfer of parameters over the network in a transparent manner. The client that calls a method is not aware that the object that implements it is not local to the process.

B. Service interfaces and class hierarchy

The base service declares generic management methods in the *Service* interface. All interfaces are put in a module named RMA. The file RMA_Service.idl contains the following code:

```
module RMA {
  interface Service {
    // typedef and enum are omitted here
    // Exception definitions and raises are omitted here

    SvcMode get_mode();
    void set_mode (in SvcMode mode)

    Msec get_period();
    void set_period (in Msec period)

    void trigger() raises(BadMode,NotRunning);

    Msec get_duration () raises(NotRunning);

    void start() raises(CannotStart, AlreadyRunning);
    void stop() raises(NotRunning);
    void pause() raises(NotRunning);
    void wakeup() raises(NotRunning);

    SvcInfo get_info();

    oneway void destroy();
  }
};
```

The CORBA idl compiler generates header (*.h) and code files (*.cpp) for the client and the server side as well as an empty implementation class of the interface methods defined in the idl file. The programmer has to fill in the empty bodies with the code to perform the desired actions.

We have defined three interfaces that inherits from the *Service* interface: *Sensors, Processors and Actuators*. The idl declaration file for the *Sensor* interface is listed below.

```
#include “RMA_Service.idl"
module RMA {
  interface Sensor : Service {
    exception SensorException {string msg_error;};
    any  get_sensor_value() raises (SensorException);
  };
};
```

Those interfaces are considered as abstract and they may not be instantiated. The real services are derived from those three interfaces. Some examples are Motion_Command_Sensors, Motion_Command_Processor and Vehicle_Actuator. The idl declaration file for the *Motion_Command_Sensor* interface is listed below.

```
#include "RMA_Sensor.idl"
module RMA {

  typedef long MotionCommand;
  typedef sequence<MotionCommand> McmdSeq;

  interface Motion_Command_Sensor : Sensor {
    typedef short NumberOfAxis;
    MCmdSeq get_motion_command();
   NumberOfAxis get_number_of_axis();
  };
};
```

What we have presented until now is sufficient to write a classical client server application and to invoke the methods in 2-way calls. For using events we need other CORBA objects that are defined in the NotificationService specifications. All those objects are defined in idl files and the implementation classes are available in the TAO libraries. Objects of those classes have to be instantiated in the different services. The Sensor uses a Supplier, the Processor has both a Consumer and a Supplier and the actuator instantiates a Consumer.

We also need to add code for creating and managing the service thread. This operation is realized by inheriting from an ACE class, i.e. ACE_Task_Base. This class defines several virtual functions that have to be implemented in the service code:

```
virtual int svc (void);
virtual int close(void);
virtual int suspend(void);
virtual int resume(void);
```

Each service is implemented as a class that inherits from automatically generated CORBA classes and from implementation classes. The implementation class name in our example is Motion_Command_Sensor_i. The structure of the classes is illustrated in the Fig. 4.

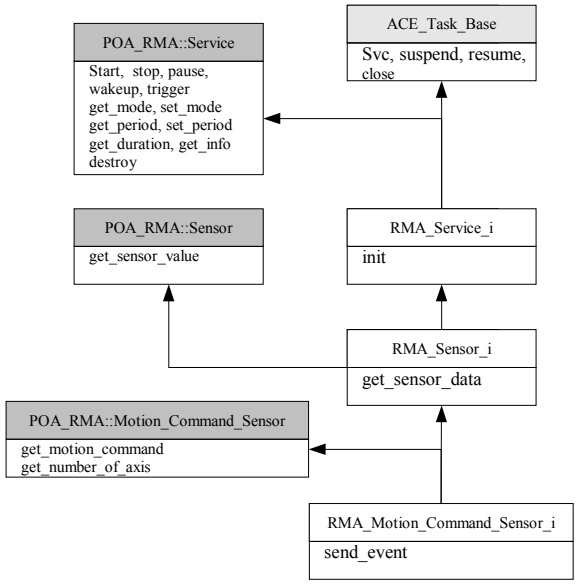

Fig. 4 Class inheritance of the Motion_Command_Sensor_i

V. SIMULATOR

MoRoS3D is a 3D robotic simulator which is written in Java. It is based on Java3D for all 3D aspects. The simulator has ben designed to allow distributed control of many identical or different models of mobile robots. Currently only flat terrain models are allowed but in the next future 3D terrain following will also be implemented. Tri-dimensional elements have been divided in different categories: robots, obstacles and terrain. Elements geometry can be read from files or directly created using Java code. The simplified data structure of the virtual world is represented in the Fig. 5.

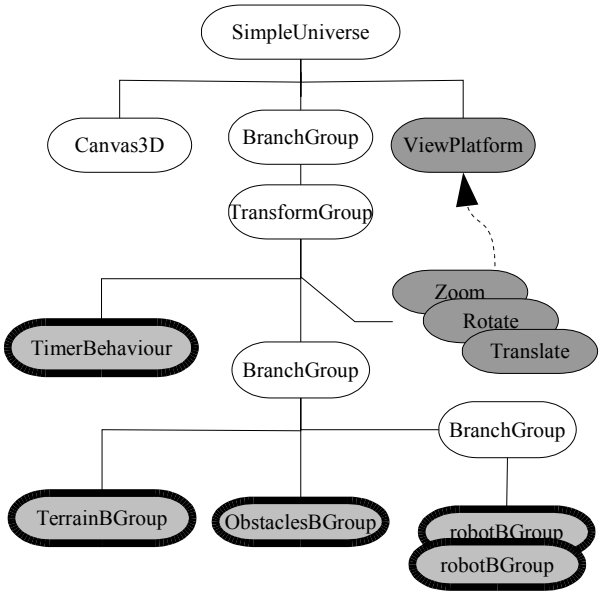

Fig. 5 Virtual world structure

Collision detection between the robots and between robots and obstacles works efficiently. It does not use the collision behaviors delivered with the SUN Java3D libraries because this one is implemented as an asynchronous detection system. Therefore collision detections are sometimes reported severall frames after the robot has entered into the obstacles! We consequently implemented a collision detection system based on the Java3D picking mechanism. At this moment only one type of robot has been implemented, a Nomad200 as we still use such a model in our laboratory.

Sensors are off course necessary to simulate any useful robot application. Therefore a first version of a distance sensor has also been defined and implemented. We can see the graphical representation of the measuring process in the Fig. 6.

A TimerBehavior generates timer events which are propagated to other collision and sensor behaviors.

The class SensorBGroup, which extends BranchGroup, defines the geometry of the sensors and also contains a LaserBehaviour that performs the operations simulating the measurement process. This behaviour is triggered by a timer event generated by the global TimerBehaviour of the simulator. The measured distances are visually depicted by variable length red rays. The distance are stored in a local array that is also accessible from CORBA methods (see below).

The simulator has naturally CORBA capabilities. The ORB is run in a separate thread that allows dispatching the CORBA calls to the right objects.
The idl definition of CORBA interfaces are listed below. The *Nomad* interface is derived from a generic *Robot* interface. The *Sensor* interfaces is also included in this file.

```
module MoRoS3DCorba
{
 interface Robot
  {
  void move_rel (in float dist, in float angle);
   void move_abs (in float x, in float y, in float angle);
  };

interface Nomad: Robot
 {
```

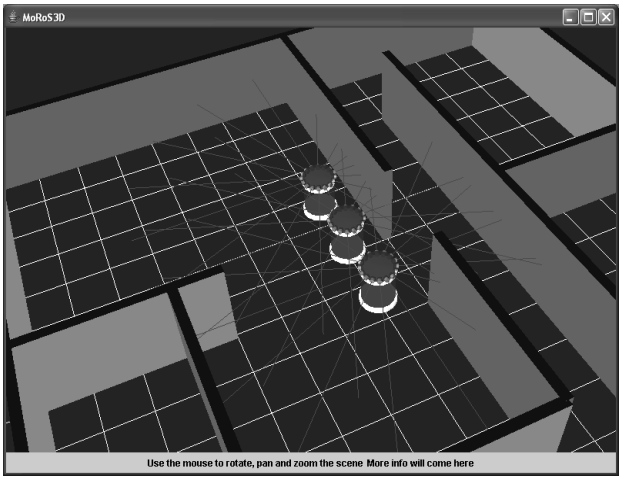

Fig. 6 The simulator with 3 robots

```
    long pr(in long t_pr, in long s_pr, in long r_pr);
    long vm(in long t_vm, in long s_vm, in long r_vm);
    long dp(in long x, in long y);
    long da(in long th, in long tu);
    void get_rc(out long x, out long y, out long steer, out long turret);
    void get_rv(out long vel_trans, out long vel_steer, out long vel_turret);
  };

  typedef long Sensor_data[16];
  interface Sensor {
        Sensor_data get_data();
  };
};
```

Along writing new services, some existing code or applications will need to be integrated in the framework. We can distinguish different types of integration: database, library and application. Integration is generally made by providing a facade, aka an object wrapper [Design pattern: wrapper facade]. The wrapper is in this case a distributed object and its implementation manages the interaction with the existing system. When developing the facade, one has to capture the business model of the existing system and develop interfaces that reflects this model. For wrapping libraries, we need to pay attention to serialize data access if existing libraries are not thread-safe. The difficulty for wrapping applications depends on the available mechanisms available for communicating with it: API, network messages (traditional client/server), file-based communication (batch program).

If we consider the case of a Nomad200, we have an API which is available for 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 JacORB idl compiler applied to this file generates the stubs and skeletons necessary to write client and server applications. A Java class NomadImpl extends one of the generated class (NomadPOA) but in order to be able to receive events, it also implements the interface org.omg.CosNotifyComm.StructuredPushConsumerOperations. This interface defines one method that is called by the NotificationService mechanism to send events to the consumer.

VI. UTILISATION

The following steps have to be followed in order to build a useful applications from the different services.

- Start the NameService.

The NameService is a kind of phone book that stores the name of the services and their CORBA Object Reference, that is, how to contact the service (it contains e.g. The server name or ip address, the port and a private key).

- Start the NotificationService

This service is used to create and managed Event Channels It registers with the NameService.

- Start the different services.

We take the Motion_Command_Sensor service as example to explain the starting sequence whose startinginfo are showed in Fig 7.

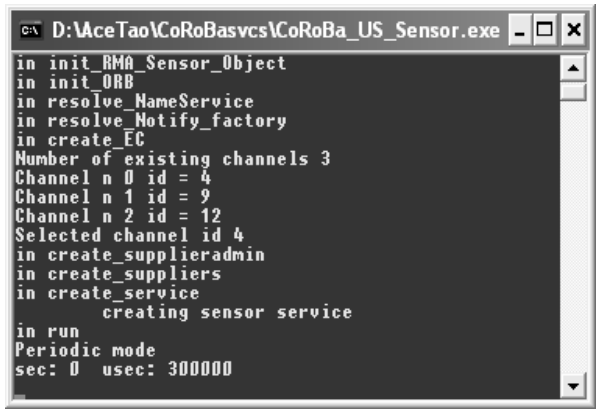

Fig. 7 Start sequence info for a Sensor service

The *main()* function creates an instance of the utility RMA_Sensor_Object and performs the following actions:

1. Initializes the RMA_Sensor_Object
2. Initializes the ORB
3. Tries to resolve a running NameService
4. Tries to resolve the NotificationService from the Naming service
5. Creates EventChannels
6. Creates the Supplier Admin
7. Creates a Supplier
8. Creates the Service

After all objects have been created, the *run()* method of the RMA_Sensor_Object is called; this method calls the *run()* method of the ORB, starting the CORBA operations.

Each service is a console application that takes parameters on the command line to select the input and/or input channels as well as the name that is registered with the NameService (each instance of a service class has a CORBA Reference and can be registered with a NameService).

start CoRoBa_Sensor.exe -s RMA_Sensor1 -e 0

The Fig. 8 shows examples of names that are registered with the NameService.

A console management application (CoRoBa Remote Control or crc) has been developed to remotely manage the life cycle of the services. It gets the reference to the CORBA objects from the NameService.

crc -n RMA_Sensor1 -c *command*

The available commands are start, stop, pause, wakeup, info. We can also change the working mode:

crc -n RMA_Sensor1 -m *mode* <-p *period*>

The last two lines of the Fig. 7 illustrates this command. The mode has been changed to periodic with a period of 300 ms.

Fig. 8 Tree of the Services registered with the NameService

How can we combine the different services together ? On the command line we can pass the channel number we want the process to communicate through (-e channel number -i input_channel -o output_channel). If the channel does not exist, the service will create it (and all other channels below it). Each channel receives a unique id number from the NotificationService. We can see an example in the Fig. 7. In this case, three channels exist and the first one is selected by the service. It means that the Sensor will send the events on the first channel (id=4).

It can be a joystick that sends the raw X-Y data. We can use an actuator that is connected to the physical system (or to the simulator) that listens directly on this channel (Fig. 9 top) or we can use a processor that will first transform the raw data and then send the result on an other channel (Fig. 9 bottom). The event domain and type are indicated in the figure.

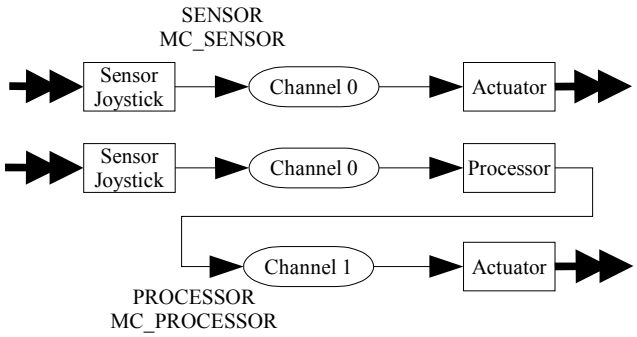

Fig. 9 Combination of Sensor, Actuator and processor

VII TELE-ROBOTIC CONTROL PATTERN

In order to identify reusable components, we introduced the robotic control patterns concept (a **pattern** is a proved design solution to a general problem). The robotic control patterns are classified from the simplest one to the most complex one. This classification has helped us to identify basic components for each pattern and to make common components as generic as possible. Considering complex application patterns at design time allows integrating their requirements during the design of generic services.

We show that most of the cases are trivially satisfied thanks to the framework flexibility.

- Direct control:sensor (joystick, console, GUI) and actuator
- Monitoring: We can connect to any channels and intercepts all events in order to display or save it.
- Sensor data processing: it is a combination of sensors and processors.
- Direct tele-control: A sensor must be send information about the robot's environment. A GUI has to connect to the channel and display the data.
- Assisted tele-control: The user commands must be fused with sensors' information in processors.
- Autonomous navigation: Sensors + processors + actuators
- Multi-robot coordination:Each robot corresponds to one object reference that can be resolved in a NameService. Each robot can be controlled by a separate process and a processor can coordinates the robot activities.
- Multi-user cooperation:each user can from its computer control one or more robot.

VIII CONCLUSIONS

We present in this paper the most recent developments of a framework that should allow to develop more flexible and more robust distributed robotic applications. Off course the chosen design has still to be refined in function of the 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 the terrain following has certainly to be implemented.

REFERENCES

[1] KU. Scholl, J. Albiez, B. Gassmann, “MCA – An expandable Modular Controller Architecture”, Third Real-Time Linux Workshop , 2001, Milan, Italy,

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand, “An architectue for autonomy”,International Journal of Robotics Research, Special issue on “Integrated Architectues for Robot Control and Programming”, 1998.

[3] http://marie.sourceforge.net/

[4] S. Engerle & al, “MIRO:Middleware for autonomous mobile robots”, Telematics Applications in Automoation and Robotics, TA2001 preprints ,p 149-154, July 2001, Weingarten, Germany.

[5] Eric Colon, Hichem Sahli, “Software modularity for mobile robotic applications”, CLAWAR2003, p 417 – 424, September 2003.

[6] Eric Colon, “Evaluation of CORBA communication models for the development of a robot control framework “,HUDEM04, Brussels,June 2004.