Evolution of the robotic control frameworks at INRIA Rhône-Alpes

S. Arias J. Lahera-Perez A. Nègre N. Turro

May 11, 2011

Abstract

Intense efforts have been carried out in the last decades to define and implement frameworks to ease the development of robotic applications. This led each research group to propose their own solution, well suited for their needs, however no common framework has been adopted. But today we have the feeling that a peculiar framework has some of the qualities required to meet with general acceptance as far robotics research is concerned : the open source robotics platform ROS developed by Willow Garage. At INRIA Rhône-Alpes, we are such a research group that developed its own framework, HUGR. In this paper, we present the requirements that ruled its design and how we now envision migrating to ROS.

1 Context

During the last decade, the way robotics has been addressed at INRIA Rhône-Alpes has changed a lot. We went from custom made robots (biped robot [1], an autonomous vehicle [2]) to more ordinary, off-the-shelf, platforms : BlueBotics wheelchair [3], Parrot AR.Drone [4], Aldebaran Robotics Nao [5] and even a standard Lexus car [6]. The evolution of the hardware went together with software architecture changes : hard real-time monolithic applications built on top of slow CPUs running VXWORKS have been replaced by more decoupled designs. We now separate the low layer, often provided by manufacturers, that runs real-time control loops such as PIDs, and higher level software requiring less precise scheduling. Most of our scientific contributions lie in these high level algorithms, so the definition of our robotic framework must take into account several key requirements such as :

- The ease of use by researchers and students, combined with a fast learning curve;
- The ability to run the same software on several robotic platforms, using the same cheap and powerful computing hardware as desktop PCs;
- Enforce software modularity, and re-use of software from others (either older contributors from the research team or open source projects);
- The availability of tools such as simulators, data logging and data replay capability that can replace time consuming real experiments.

According to these requirements, in section 2 we present the middleware and tools we designed. Then in 3 we explain why we consider the adoption of a third part robotic middleware (ROS [7]). At the end, we give some benchmarks concerning performance and we expose feedback on how we migrate our applications and tools.

2 The HUGR middleware and the simulator

To meet the above mentioned requirements, we decided to define a toolkit, CYCABTK [8] - at first aimed to ease development on mobile robot Cycab - including a 3D simulator and a piece of software called a middleware that implements a blackboard-based [9], publish-subscribe paradigm.

2.1 Middleware

The first function of a middleware is to provide an abstraction layer between the application layer and the low layer. That means that application does not communicate directly with the drivers but with the middleware and does not matter about the hardware management. In the other side, the drivers only have to manage the hardware and are protected from application crashes by the middleware layer.

The middleware, called HUGR, we developed within the CYCABTK toolkit, offers an answer to those different points, is open source and – as much as possible – easy to use by the robotics community.

This middleware uses the concept of a blackboard-based publish-subscribe architecture : the core of the middleware is a shared memory where different applications can write or read a set of variables, as depicted in Fig. 1. HUGR is developed in C++ and depends on POSIX system libraries, so it works on standard Linux system. It is worth noting that our middleware does not stand on hard real-time constraints : the real-time characteristics added to the Linux especially since kernel 2.6.x version have been proved sufficient for our applications.

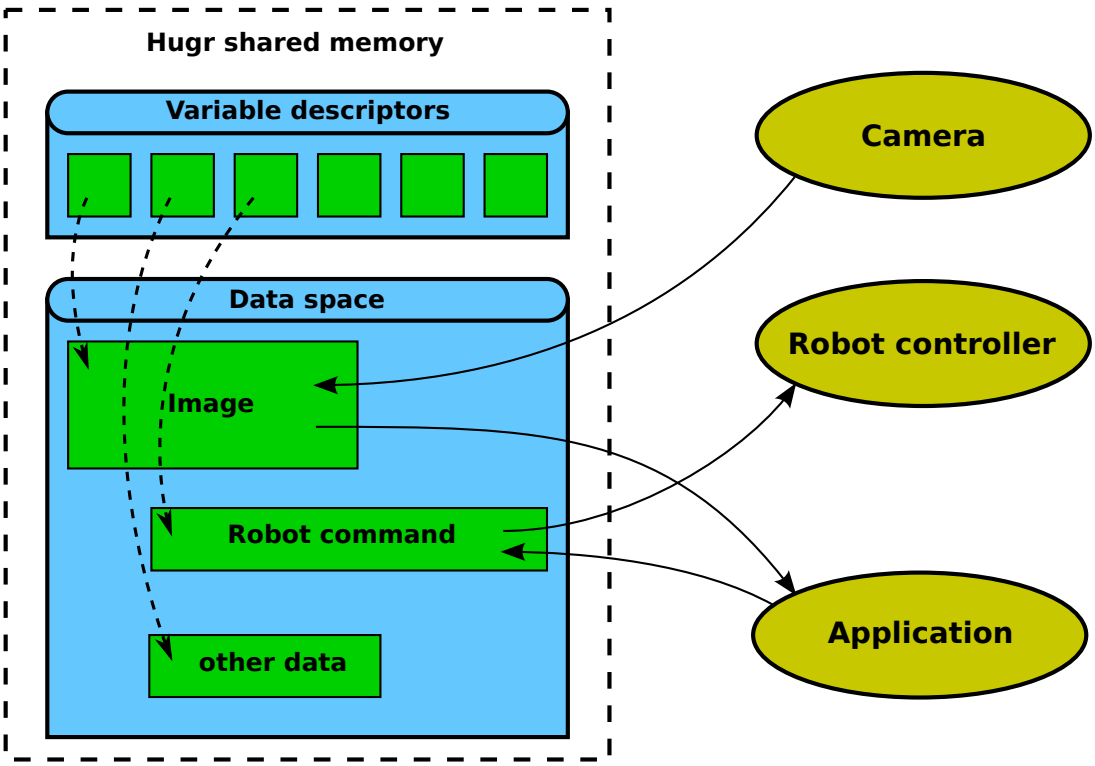

Figure 1: HUGR architecture: the middleware offers a shared memory where sensor drivers, robot controllers, viewers and other applications can write and read a set of variables.

The main features of HUGR are listed below :

Shared memory. All the data are stored in a POSIX shared memory in order to minimize the transfer time between applications.

Serialization. To easily read and write a variable into the shared memory, HUGR uses the BOOST serialization library [10].

Synchronization. The synchronization between software components of an application can be implemented by blocking reads of variables. Then writing those variables causes the emission of a signal that can "wake" the listener module. This mechanism is used when some processing must be performed each time a new sensor data is available.

Time management. Each variable is automatically timestamped whenever its value is modified.

Data recording/replaying. The `hugrlog` and `hugrreplay` tools are intended to record and to replay back a set of variables by optionally scaling the time.

Networking. A TCP server enables to share variables within the network. This tool duplicates a variable in the HUGR shared memory on computer A into the HUGR shared memory of computer B.

Visualization tools. A Web server (`hugrweb`) provides a convenient way to display all the HUGR variables and their contents (may need appropriate plug-ins according to the structure of the variable). For real-time visualization, dedicated viewers are available for some sensors like camera images and LIDARs.

2.2 Simulator

In addition of the middleware, our robotics team needed a simulator in order to develop algorithms faster and to validate experimental applications before testing on the real platforms.

When we decided to implement our own simulation tool, only a low number of simulators were dedicated to robotics simulation and were not adapted to our needs : real-time execution, specifics various sensors and robots simulation (LIDAR, camera, Cycab robot), 3D rendering for camera simulation, etc. The simulator we developed was based on the open source 3D rendering engine MGENGINE [11]. One of the main interest of this simulator was its integration with the HUGR middleware which make the simulator transparent to the applications. The data produced by simulation is accessed the same way that real data. The same application could thus be used with the simulator and with real platform without even having to recompile.

The figure 2 represents a screenshot of a view from the simulator, where simulated omnidirectional and fish-eye camera, the simulated LIDAR and the simulated mobile robot are rendered within a car park environment.

2.3 Shortcomings

Although HUGR was successfully used during several years by our team on the Cycab platform, we were aware of some of its limitations :

- Its audience was very small and purely local to INRIA Rhône-Alpes, thus we could not share algorithm implementation or experimental data. Neither could we use third part modules without re-writing them.
- We lack the manpower to maintain, let alone enhance this middleware, originally written by enthusiastic PhD students.

As a result we consider the adoption of a third part robotic middleware.

3 Current evolution

The French robotic ecosystem offers several robotic frameworks : GENOM/PRS [12], AROCAM [13], RTMAPS [14], etc., but none of them fulfilled our requirements which were :

- Free, and open source;
- No loss of functionality with respect to HUGR;
- An already sufficiently large adoptance from the robotic community.

3.1 Migration to ROS

Last year, we figured out the ROS middleware developed at Willow Garage [7] might be a good replacement for HUGR. Indeed, ROS matches most of HUGR features :

- Like HUGR, ROS behaves like a blackboard : it is in charge of the data routing between modules producing data (filling topics) and consumer modules that are subscribed to those topics.

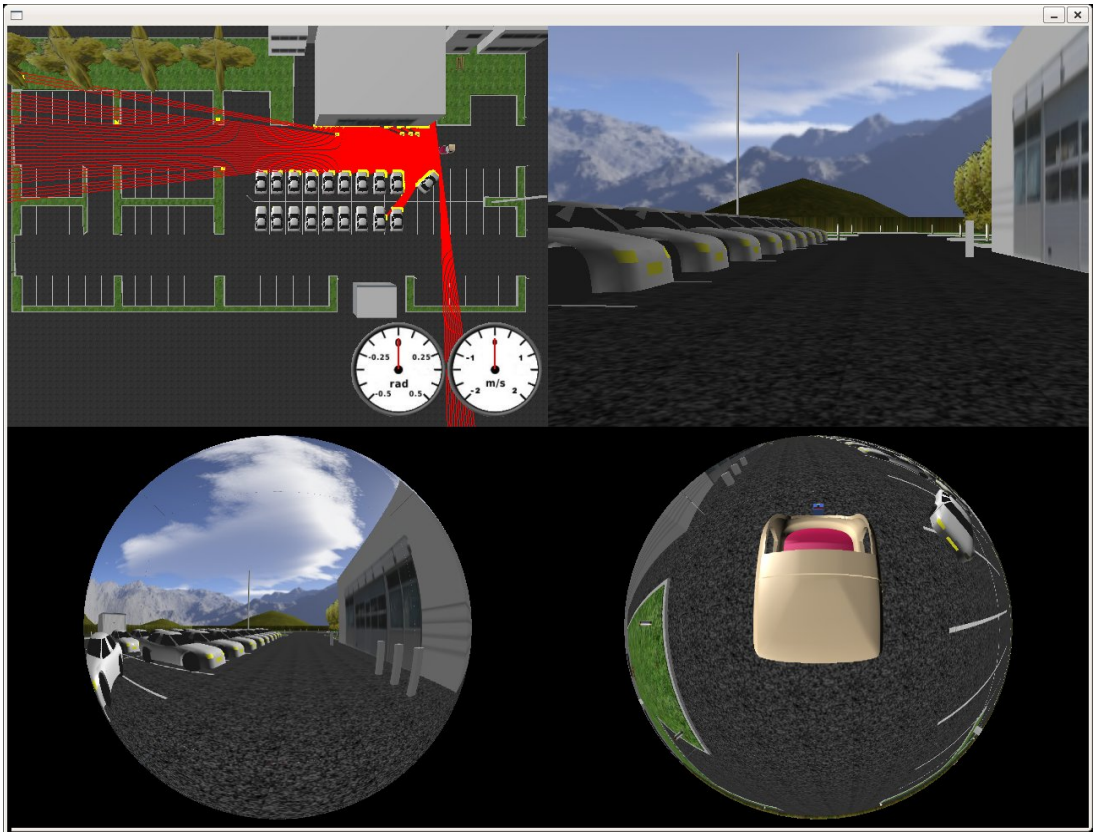

Figure 2: Screenshot from the MGENGINE based simulator provided with the CYCABTK toolkit. Simulated mobile robot, environment, laser impacts, omnidirectional and fish-eye camera are presented on this image.

- Sensors are abstracted through the use of pre-defined data types (`Images`, `LaserScans`, `JoinState`, `Odometry`, etc.), thus high level modules can be implemented without knowledge of the type of sensor providing the data.
- Each sensor data is timestamped by the middleware.
- ROS provides tools for real-time recording of data, and replay of this data.
- Many drivers for hardware we use are already available.

Moreover, ROS provides some more benefits :

- Multi-langage support : especially C++ and PYTHON (increasingly used for application prototyping).
- Sexier visualization tools (see packages `rosbag` and `rviz`).
- A wide spectrum of predefined robotics data structures : `OccupancyMaps`. `Paths`, `PointClouds`, `GridCells`. Thus, high level algorithms can be implemented in a portable way.
- ROS is on the verge of becoming a de facto standard and is already taught in several universities [15].
- PhD Student works can be disseminated more easily with other teams using ROS, they can share their software in order to be used by other groups and built on top of each other work.

3.1.1 Learning curve

The first contact with ROS is a bit awkward, since you have to make yourself familiar with a full set of non standard commands to manipulate ROS software components : e.g `rosmake` for compilation, `rosrun` for execution, `rosdep`, `roscore`. Although those commands are custom wrappers to standard tools like `cmake`, it seems very tedious to by-pass them. However, a full set of tutorials available on the ROS Web site [16] introduces each command and concept step by step.

As a result, after one week of work, it has been possible, for example, to :

- Implement a ROS driver for one of our stereo camera;
- Write a ROS interface to the embedded low level controller of BlueBotics wheelchair, and then reuse all the navigation stacks provided by ROS to build an impressive demo;
- Reuse a third part ROS driver interface to control the Parrot AR.Drone. This driver has been provided by the Mobile Robotics Lab of the Southern Illinois University to the ROS community short time after Parrot drone has been put on sale.

Rewriting high level modules is painless for us since the ROS and the HUGR share the same concepts (blackboard architecture, separate Unix processes for each module and the core daemon). It mostly consisted in :

- Replace the HUGR middleware API calls with the equivalent ROS functions;
- Change the data structures to fit standard predefined ROS messages;
- Make some small adjustments on the communication with the middleware, since HUGR uses blocking read and polling mechanisms whereas ROS uses callbacks.

As a consequence, the overall feeling is that useful applications can be easily build on top of ROS, but most of its internal mechanisms remain arcane, and hidden.

3.1.2 Performance

Given the complexity of the ROS framework, we were worried by its performances and its appropriateness to our experimental setup. In order to dispel this uncertainty as soon as possible, we carried out the following comparison between HUGR and ROS : we studied the latency induced by the middleware between a video camera driver and client module. The camera driver receives images at 30 frames per seconds and stores them in a ROS topic or a HUGR variable, adding a timestamp information. At the same time, we run a client which either registers a callback in ROS or performs a blocking read on a variable, waiting for a new image. When receiving a new image, the client computes the difference between the current system time and the timestamp of the image. This difference is the latency induced by the middleware. We conducted this experiment using two different scenarios. In first case, the Linux system was not stressed, and in the second case, we wrote all the images to the disk in real-time (about 10 gigabytes of data for a fifteen minutes experiment), which heavily disrupts the Linux scheduling.

The results of this experiments are displayed in tables 1 and 2. In the low-load configuration, the latency induced by the middleware is very low most of the time ($< 1ms$), despite the size of the data, either with ROS or with HUGR. Obviously, due to the non real-time characteristics of the Linux version we use, some jitter is present and some higher latency occurs, but within acceptable range. In the high-load configuration, the mean value of the latency remains under $1ms$, but it is spikier, with an higher standard deviation. In some cases, delays as large as $10ms$ have been observed, but on a very small set of measurements. This might be a problem for some very specific algorithms, but usually, a few late measurements over a fifteen minutes run does not hinder much our experiments. Moreover, we usually do not log data at the same time as we run high level processing. Nevertheless, it is worth noting that the latency induced using HUGR has a lower jitter, indicating that mechanisms used in ROS might be less efficient (but perhaps more powerful).

	Hugr	ROS
Mean	0.35 ms	0.8 ms
Max	2 ms	2 ms
Std Dev	0.03 ms	0.6 ms

Table 1: Latency without any external load

	Hugr	ROS
Mean	0.5 ms	0.8 ms
Max	8 ms	10 ms
Std Dev	0.19 ms	0.9 ms

Table 2: Latency on a stressed environment

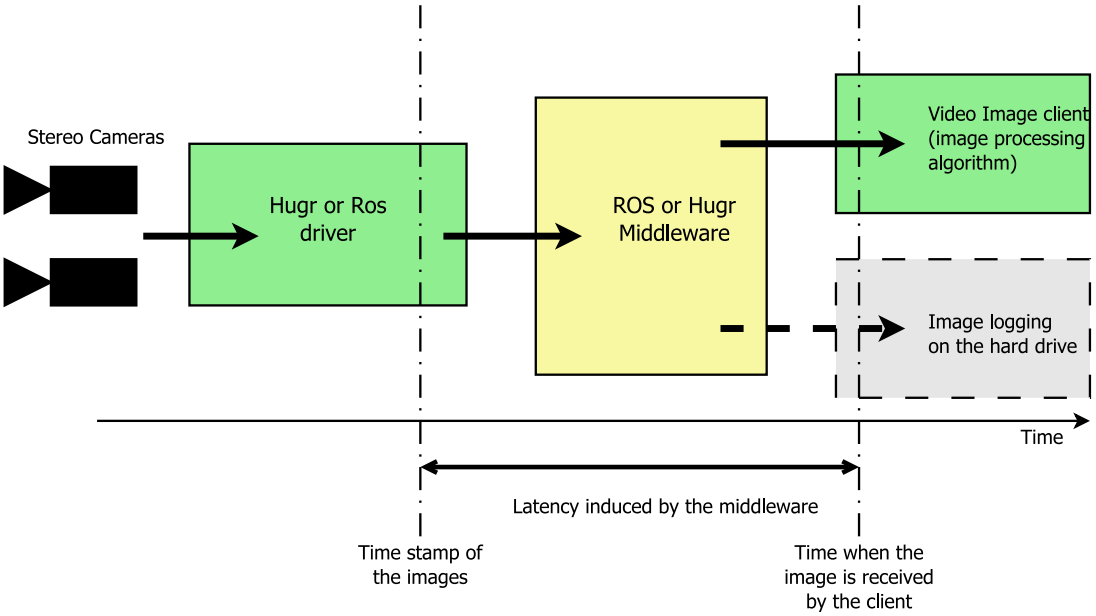

Figure 3: Setup for evaluation of the latency induced by the middleware in an experiment

3.2 Simulator

Initially, the CycabTK simulator was designed to work with the Hugr middleware. This dependency prevents other robotics team to use this simulator if they use any other middleware. According to this observation, and to make possible the use of the simulator in research project as "ANR PROTEUS" [17], we decided to integrate more than the Hugr support. Several solutions could be devised to support other middleware :

1. Replace all Hugr calls by the chosen middleware equivalent functions : this solution would be simple to realize but was not retained as it is not compatible with the support of several middlewares.
2. Create a connector between Hugr and other middleware : this is difficult to achieve since the data stored on the Hugr shared memory do not contain meta information concerning its type. A new layer should be added to convert the stored data into other middleware compatible data type and that can hurt the performances.
3. Remove any Hugr dependency from the simulation core and create a plug-in mechanism to interact with a given middleware. This solution requires to modify the simulation core but is more versatile and is better suited for maintainability and performance.

Given the considerations below, we chose the third solution. The plug-ins mechanism we chose implied to create a middleware specific module to all simulated components. These modules can then be dynamically attached to a simulated object. Technically, a callback function will be called before and after the simulation component in order to read data from the middleware (like a robot command) and/or to send data to the middleware (for example the simulated sensor data). The figure 4 shows the simulator architecture and the connection to middlewares (ROS and Hugr for example). This way a simulated component can provide connectors to several middlewares.

Using this architecture, we have already implemented ROS connectors for several sensors (cameras, LIDAR, GPS) and for a car-like robot. With minimal effort, we can now use ROS application like the visualization tool or the navigation stack, and we make the simulator much easier to use by academics partners.

4 Conclusion

In this article, we presented how the design and development of robotic applications evolved at INRIA Rhône-Alpes. We developed our own framework (mainly a middleware, Hugr, and a simulator); we used

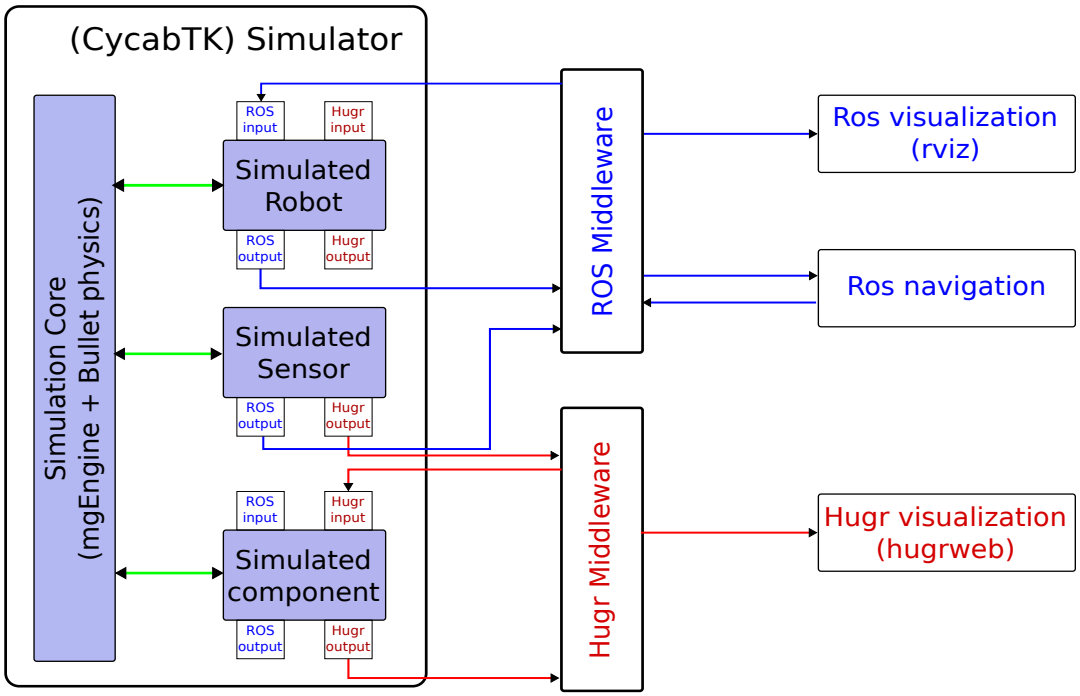

Figure 4: Architecture of the simulator and connection to middlewares (here ROS and HUGR).

it successfully but only locally at INRIA Rhône-Alpes.

We also presented why and how migrating to ROS seems to be the next logical step. Henceforward ROS will be preferred over HUGR middleware, and within our simulator. For this reason we also intend to extract the simulator from the CYCABTK toolkit package and remove all its HUGR dependencies. This way, we let the users implement the plug-in interface mechanisms they need for the middleware they favor.

We hope that the adoption of ROS will enable more collaborations among the robotics community, since it will ease the exchange of experimental sensor data sets, and even algorithms. We also hope that the ROS public modules repository will act both as a showcase for our developments, and as a place to find a wide range of up-to-date device interfaces.

References

[1] Gérard Baille, Philippe Garnier, Hervé Mathieu, and Roger Pissard Gibollet. Le cycab de l'INRIA Rhône-Alpes. Research Report RT-0229, INRIA, 1999. Projet SERVICE ROBOTIQUE.

[2] Gérard Baille, Pascal Di Giacomo, Hervé Mathieu, and Roger Pissard Gibollet. L'armoire de commande du robot bipède bip2000. Research Report RT-0243, INRIA, 2000.

[3] BlueBotics SA Autonomous Modular Vehicule related Web Site. `http://www.bluebotics.com/automation/AMV-1/`.

[4] Parrot AR.Drone Web Site. `http://ardrone.parrot.com/`.

[5] Aldebaran Robotics Web Site. `http://www.aldebaran-robotics.com/`.

[6] Mathias Perrollaz, Mao Yong, Amaury Nègre, Christopher Tay, Igor E. Paromtchik, and Christian Laugier. The ArosDyn Project: Robust Analysis of Dynamic Scenes. In *11th International Conference on Control, Automation, Robotics and Vision*, Singapore, December, 07-10 2010.

[7] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[8] CycabTK toolkit Web Site. `http://cycabtk.gforge.inria.fr`.

[9] Steven A. Shafer, Anthony Stentz, and Charles E. Thorpe. An architecture for sensor fusion in a mobile robot. In *IEEE International Conference on Robotics and Automation*, pages 2002–2011, San Francisco, CA, USA, April, 7-10 1986.

[10] Boost Serialization Library. `http://www.boost.org/doc/libs/release/libs/serialization/doc/index.html`.

[11] Massive G Engine Web Site. `http://mgengine.sourceforge.net/`.

[12] Sara Fleury, Matthieu Herrb, and Raja Chatila. GenoM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture. In *International Conference on Intelligent Robots and Systems*, pages 842–848, Grenoble, France, 1997.

[13] Cédric Tessier, Christophe Cariou, Christophe Debain, Roland Chapuis, Frédéric Chausse, and Christophe Rousset. A Real-Time, Multi-Sensor Architecture for fusion of delayed observations: Application to Vehicle Localisation. In *9th International IEEE Conference on Intelligent Transportation Systems, ITSC 2006*, pages 1316–1321, Toronto, Canada, September 2006.

[14] Fawzi Nashashibi, Bruno Steux, Pierre Coulombeau, and Claude Laurgeau. RTMAPS a framework for prototyping automotive multi-sensor applications. In *IEEE Intelligent Vehicles Symposium 2000*, Dearborn, MI, USA, October, 3-5 2000.

[15] ROS Courses List. `http://www.ros.org/wiki/Courses/`.

[16] ROS Web Site. `http://www.ros.org/wiki/`.

[17] ANR PROTEUS Web Site. `http://www.anr-proteus.fr/`.