Real-time Software for Mobile Robot Simulation and Experimentation in Cooperative Environments

Andreu Corominas Murtra[1], Josep M. Mirats Tur[1],
Oscar Sandoval[1], Alberto Sanfeliu[1,2] ⋆

[1] Institut de Robòtica i Informàtica Industrial, IRI (UPC-CSIC). C/Llorens i Artigas, 4-6, 2nd floor. Barcelona, Spain. www-iri.upc.es
[2] Universitat Politècnica de Catalunya, UPC. Barcelona, Spain.

Abstract. This paper presents the software being developed at IRI (Institut de Robòtica i Informàtica Industrial) for mobile robot autonomous navigation in the context of the european project URUS (Ubiquitous Robots in Urban Settings). In order that a deployed sensor network and robots operating in the environment cooperate in terms of information sharing, main requirements are real-time performance and the integration of information coming from remote machines not onboard the robot. Moreover, the project involves a group of eleven industrial and academic partners, therefore software integration issues are critical. The proposed software framework is based on the YARP middleware and has been tested in real and simulated experiments.

Key words: Mobile robot software, real-time, sensor networks

1 Introduction

Research in robotics is experiencing a steady incoming of new hardware components, platforms and devices, with the aim of overcoming perception and actuation limitations of current robotic systems. These hardware novelties need software to be operative, but developing such a software is a time consuming and error prone task. Therefore, good practices in software development are required in robotic laboratories in order to economize engineering time and share results and modules between research teams. Also, simulation of robot systems is a generalized task that saves a lot of power and human energies, but the danger of recoding algorithms for both simulation and experimentation arises. All these topics have been recently discussed within the robotic research community [1–4].

⋆ Research conducted at the Institut de Robòtica i Informàtica Industrial of the Universitat Politècnica de Catalunya and Consejo Superior de Investigaciones Científicas. Partially supported by Consolider Ingenio 2010, project CSD2007-00018, CICYT project DPI2007-61452, and IST-045062 of the European Community Union. [acorominas,jmirats,osandoval,asanfeliu]@iri.upc.edu

In the last years some interesting middlewares have been presented which can be downloaded as open source software [5–8]. These projects coincide on being real-time oriented and based on fully independent processes running in the same machine or in a network of computers, thus they require a fast and robust inter-process communication tool to operate.

Our context is that of the URUS european project [9] involving open research fields such as network robot systems and cooperative robotics. Different experiments, as transport of goods or evacuation of people, are envisaged in outdoor urban scenarios, involving a camera network recently deployed on the URUS environment and a team of heteregeneous robots running in it. In order to be successful with the software integration and experimentation a good software practice and a process communication approach are mandatory.

In this work we present a software framework based on the YARP middleware [10]. YARP was initially written for people working with humanoid robots hence involving a lot of hardware devices to be controlled. We do use YARP in our context to provide communication capabilities between different processes of the whole system, whether these processes are running onboard the same robot or not. Therefore, our framework is developed with the aim of being executed in a decentralized network of computers, being flexible to accept an heterogeneous set of devices and algorithms and being independent of whether the data sources are real or simulated platforms and devices, or files with stored off-line data.

This paper is organized as follows: section 2 overviews the whole software structure, section 3 presents the knowledge basis of our system while section 4 focuses on the involved processes and their designed hierarchy. In section 5 the graphical user interface is presented. Real-time experiments, both simulated and real, are presented in section 6, validating the operability of the presented software. Finally, section 7 summarizes the main conclusions of the work.

2 Framework Overview

The proposed software framework has the main goal of providing a mobile robot with autonomous navigation capability. Moreover, our mobile robot is thought to be running in a cooperative environment, that is, an area where other mobile robots are also operating and where a sensor network is deployed. The whole system should provide a set of services in an urban environment such as transportation of goods and people, cleaning or surveillance. Figure 1 shows the proposed navigation software framework in this context including hardware devices providing data (grey boxes), processes running concurrently (white boxes), and YARP connections (black arrows) building a network of processes that exchange data. This figure also indicates with an asterisc the processes that are running in a remote machine (not on-board the robot) and, thus, a wireless link is required to connect them to on-board processes. The running mode variable indicates if the robot is tele-operated (RM=0), executing a path (RM=1) or following a visual target (RM=2).

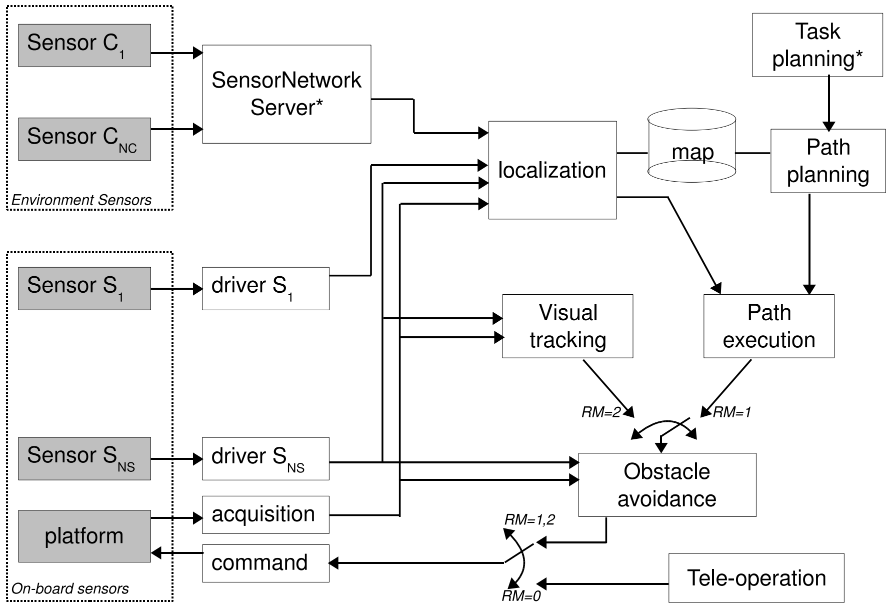


Fig. 1. Network of processes building up the proposed software framework for autonomous navigation in cooperative environments. Grey boxes are devices while white boxes are processes. Processes running in a remote machine are marked with *.

Our design divides the proposed software in three parts: knowledge-basis, processes and graphical user interface. The *knowledge basis* is a set of classes implementing the data base and methods to deal with it, representing all that the mobile robot 'knows'. In our context, this refers to the environment model (map) and the methods to efficiently operate with it. *Process classes* implement the core of this software design. These classes are organized in a three-level hierarchy in order to exploit C++ modularity and inheritance. This set of classes implements the basic process loop (layer 1), the process network (layer 2) and the specific device drivers and algorithms (layer 3). Finally, the *graphical user interface* (GUI) provides a mean to display real-time data while allowing the user to drive the robot in a tele-operation running mode.

3 Knowledge Basis

The set of classes implementing the a priori knowledge of the robot builds up the knowledge basis. In our map-based navigation case, this a priori knowledge is given by a map of the environment and different geometric methods to operate with it. This map could also be implemented off-board, as a data server providing answers to requests about distances, angles or line-of-sights. However, since some navigation algorithms perform thousands or even millions of operations per second related to the map, the "server approach" is unfeasible for real-time applications. This fact forces us to load the knowledge basis to the local memory of the computers where processes requiring it are running.

In the context of our network robot system the map is described in a data file of about $40KB$, using a compatible format with Geographical Information Systems (GIS). The model represents the environment as a list of obstacles, each one described with metric and semantic information. When a robot initializes, it requests the map to a map server. The map server replies sending the map file and the robot loads it to a program variable, thus the knowledge basis is now in the local memory and processes requiring it have faster access to their methods.

For other applications this knowledge basis could be a dictionary, as a set of objects to be identified, or a set of faces to be recognized (parameterized or not). Obviously, if there were no real-time constraints in our application, we could implement this knowledge basis as a data base server running in a machine not onboard the robot.

4 Processes

The hierarchy of the classes implementing processes is organized in three layers. The first layer defines a basic process class. A second layer implements the interfaces, defining data packets and connections between processes, thus the process network is completely stablished. This second layer is based on the communication capabilities of the YARP middleware. Finally, the third layer of the hierarchy implements specific algorithms and drivers to control devices.

Figure 2 shows the C++ class hierarchy for the involved processes in our framework. In the next subsections we detail each layer of this hierarchy.

4.1 Layer 1: Basic Process Class

This single class defines a generic process as an independent thread. The protected variables of this class are listed below:

```
int status; //=0 when process runs ok. Otherwise is an error code
int partnerId; //id of the partner responsible of that process
int machineId;  //id of the machine where this process is running
char label[20];  //short label identifying the process
int sleepPeriod; //sleep period [us] to regularize loop period
ofstream logFile; //file to print log messages
ofstream dataFile; //file to print data
timeval timeStamp; //time stamp of the process output data
int processThreadID; //thread id
pthread_t processThread; //thread variable
```

This variable set has been found as the minimum common set that all our processes need to be operative. The variable status indicates if a process is running with no trouble (status=0) or if some error or unexpected situation is

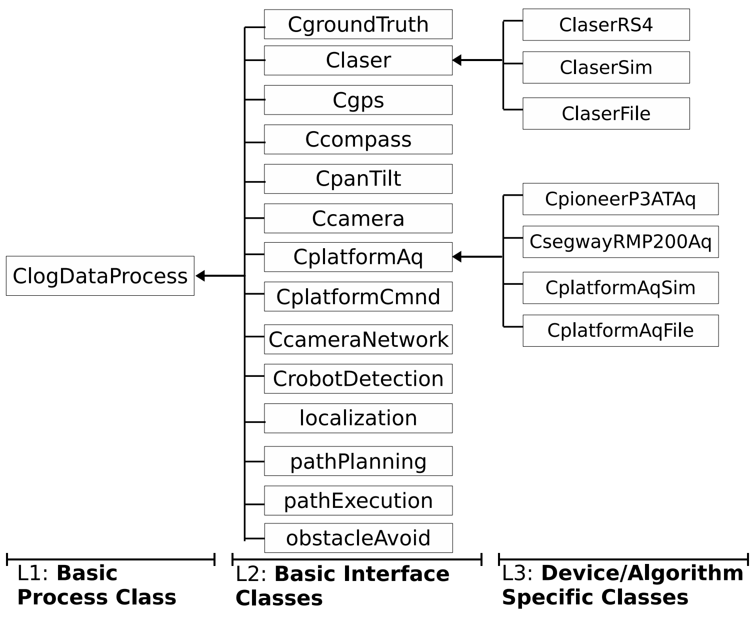


Fig. 2. Hierarchy of classes involved in the process implementation. Claser and CplatformAq (acquisition) basic interfaces are unfolded to their related specific classes for illustrative purposes. Boxes are classes and arrows imply inheritance relation.

encountered (status=errorCode). The variable partnerId identifies which partner is responsible of the process among a group of partners working in the project. The variable machineId carries the identification of the machine on which the process is running. The label string is used to shortly define the process as, for instance, 'gps', 'frontLaser' or 'obstacleAvoidance'. The sleepPeriod integer, defined in microseconds, is the pause that the process will execute to regularize its output to a given output frequency, specially for those cases where process stuff is very low and data output is not required to be fast. Two files are also members of a process, one to keep log messages during execution and the other to save process data. Both, log messages and process data are always printed with a time stamp value provided by the variable timeStamp (TS). Last two variables are needed to run the process as a separate thread.

For this basic process class, we have the following public member functions:

```
ClogDataProcess(int ptid, int rid, char *labelStr); //constructor
virtual ~ClogDataProcess(); //destructor
int writeLogFile(char *msg);//prints message with TS to logFile
int writeDataFile(char *msg);//prints message with TS to dataFile
virtual void printAlive(); //prints alive message to std output
virtual void process()=0; //Main method processing the data
virtual void printData()=0; //prints data content to dataFile
virtual void sendData()=0; //sends data content (publish data)
void startRun(); //Throws the thread calling the run() method
void endRun(); //Cancels the run() this process
static void *run(void *thisPnt); //Main loop
```

The proposed set of public member functions is also designed to satisfy the minimum common needs for all the processes. The constructor initializes the status to -1, sets the variables partnerId, machineId and labelString, and opens the logFile and the dataFile. Destructor will close these files. We have also the member functions writeLogFile() and writeDataFile(), that print a given message in the log/data file with the current time stamp. The virtual member function printAlive() prints a basic alive message to the standard output. If desired, it can be overridden to print a more specific alive message. After that, we find three pure virtual member functions that are just named in this class but not implemented. The process() member function will contain all the process work and it will be implemented in the last layer of the hierarchy, that of the device/algorithm specific classes. The other two functions will be implemented in the second layer of the hierarchy: the printData() member function printing the whole data packet that the process outputs to the dataFile, and the sendData() member function sending a data packet through a communication channel (publishes data). Finally, there are three member functions implementing the starting of the thread, its main loop and its end or cancel condition. The run() member function is the main loop of the thread and it is detailed in the following code:

```
while (1)
{
       thisProcess->process();//main job of this process
       thisProcess->printData();//prints data to data file
       thisProcess->sendData();//writes data to output ports
       sleep(thisProcess->sleepValue);//adjusts output frequency
}
```

Please note that in this basic process class neither the process connections nor the data packets are still defined, since each process uses a different number of inputs and outputs and works with different kind of data. The second layer of the hierarchy will define and manage these issues.

4.2 Layer 2: Basic Interface classes

Classes in the second layer implement communication between processes, that is, they define the network connecting processes and data packets passing through that network. This layer is motivated by the fact that several implementations of a given algorithm or sensor driver use the same inputs and outputs and manage the same data packets. The idea within this layer is to define, for each interface class, which are the required inputs, the provided outputs, and the format of the data packets going through these inputs and outputs. It is only in this layer where YARP, the chosen middleware, is used to support the communication network. Such a layer is critical since we are working in a project involving several industrial and academic partners, and is in this layer where integration guidelines must be carefully respected [11]. Only if we faithfully follow these guidelines we will enjoy the integration work as an assembling of "little black boxes".

As an illustrative example we show the localization basic class, implementing the communication layer for all specific localization algorithms. Figure 3 shows this class as a black box accepting inputs from several real-time observations and outputing a data packet containing the estimations of the robot pose, velocities and related uncertainties. Hence, the localization basic class is in charge of putting a localization specific module in the right place within the network of processes.

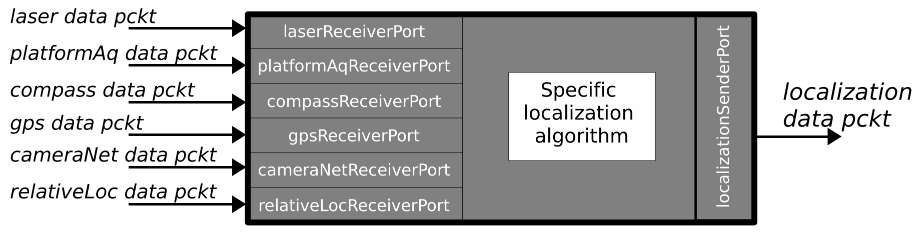


Fig. 3. Concept of the localization basic class. Inputs, outputs and data packets are defined at this basic interface layer.

To fully implement an interface, we need to define the format of the data packets provided by each interface. With this aim, we have designed a set of structs named xPacket for each content format travelling throughout the process network. Moreover, we have a set of classes inheriting yarp ports, specialized to send or receive a given data packet. In figure 3, the localization basic class has, for instance, a laserReceiverPort, an object in charge of receiving real-time observations from a laser driver process, always storing the last one. In the output side, the localization process, publishes a localization data packet through a localizationSenderPort, with the format specified in figure 4.

In terms of integration, and following the illustrative localization case, a given process P requiring real-time localization data only has to incorporate a "localizationReceiverPort" object and connect it to the output port of the localization process. Doing this, the process P has available in its local memory the last estimation of the robot position, published by the localization process.

However, these interface classes do not implement the process() member function presented at section 4.1, thus they 'do nothing', but the robot has to sense and move. The next section details the third layer of the presented software framework, where specific algorithms and drivers are implemented.

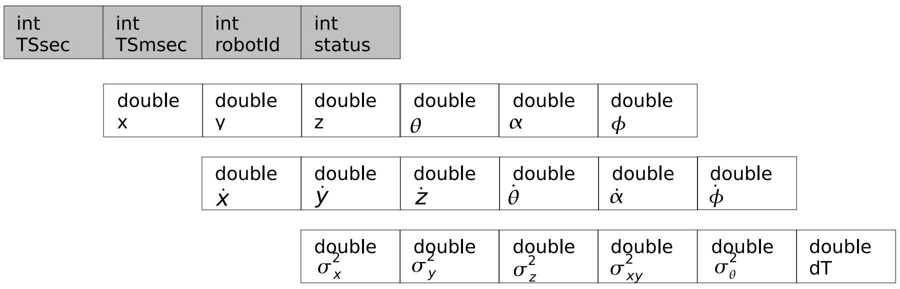


Fig. 4. Output data packet for the localization. All specific localization algorithms publish the same data packet. Grey fields form the common header of all data packets.

4.3 Layer 3: Specific device/algorithm classes

This last layer of the hierarchy implements the specific processes of drivers controlling hardware and algorithms for navigation tasks, that is, it implements the member function process() that remainded a pure virtual function in the first and second layers of the hierarchy. It is precisely in this layer where robotic researchers have to program their own algorithms to solve the different navigation tasks. The only restriction when programming a specific device or algorithm class is to agree with the related interface, a fact that appears naturally in object oriented languages as C++, when class inheritance is used.

For each basic interface related to a device family, we have a class implementing a simulation of that device family, a class reading off-line data for that device family, and a class for each physical device that we have in our laboratory. For instance, the basic class being in charge of the acquisition of the platform data (CplatformAq), has four inherited classes implementing the above mentioned cases: CplatformAqSim, CplatformAqOffLine, CplatformAqSegwayRMP200, CplatformAqP3AT (see figure 2).

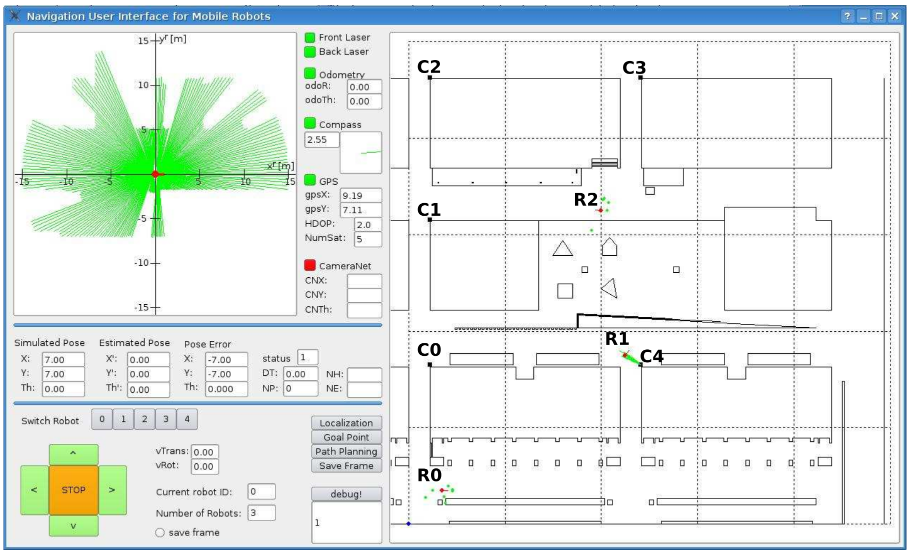
The key point of the proposed software architecture is that all these four specific classes inherit the basic CplatformAq class, thus from the point of view of communications, these four classes have the same interface and manage the same data packets, and, therefore, for a process requiring platform data it is completely transparent which kind of platform (simulated, off-line or real) is currently providing the real-time data. To keep the real-time in off-line executions, the sleepPeriod of the process reading a data file is adapted at each iteration according to the time stamp increment between the two last data rows.

This approach facilitates also the integration work. For instance, a team requiring the localization data for its task allocation research do not worry on which specific algorithm is performing the localization. This team only needs to incorporate a localizationReceiverPort to its module and to connect this port to the output port of the localization process.

5 Graphical User Interface

The developed graphical user interface allows monitoring the navigation experiments. Figure 5 shows a snapshot of this GUI for a simulated case.

On the right side of the screen snapshot in figure 5 we can see a map representing the $10000m^2$ campus outdoor area where the robots are expected to operate. In this map, we can see three robots (R0..R2) as red dots and five fixed cameras (C0..C4) as black squares. We can also see simulated GPS data for robots R0 and R2 positions as green spots on the map layout (R1 was out of gps coverage), and how the camera network process is detecting robot R1 with camera C4 and is sending range-bearing observations, each one depicted as a green segment.



🟩 Front Laser
🟩 Back Laser
🟩 Odometry
odoR: 0.00
odoTh: 0.00
🟩 Compass
2.55
🟩 GPS
gpsX: 9.19
gpsY: 7.11
HDOP: 2.0
NumSat: 5
🟥 CameraNet
CNX:
CNY:
CNTh:

Simulated Pose		Estimated Pose		Pose Error					
X:	7.00	X':	0.00	X:	-7.00	status	1		
Y:	7.00	Y':	0.00	Y:	-7.00	DT:	0.00	NH:	
Th:	0.00	Th':	0.00	Th:	0.000	NP:	0	NE:	

Switch Robot 0 1 2 3 4
vTrans: 0.00
vRot: 0.00
Current robot ID: 0
Number of Robots: 3
○ save frame
1

Fig. 5. GUI snapshot.

On the left of the snapshot (figure 5), we can see the simulated onboard sensor data for the selected robot (R0 in this case). Leds near each sensor label indicate whether the status of the sensor driver is ok (green) or if some problem occurs on providing data (red). In the shown case, the cameraNetwork led is in red since there are no detections for R0 (the selected robot), since it is out of the camera network coverage. On the bottom left there are also the control buttons to move the robots and to change the selection of the current robot.

6 Real-time experiments. Position Tracking example

6.1 Simulated experiments

We first show a simulated experiment on position tracking. The localization filter is a process fusing data coming from six simulated device processes: platform acquisition (odometry), front laser, back laser, compass, gps and camera network observations. Moreover, during this execution we have a process moving the platform and updating the simulated ground truth and the GUI. All these processes run in real-time, providing and receiving data through the YARP network. The localization filter process do not worry about where are the computers providing data arriving to its data ports (see figure 3) and whether these data is simulated or real. This localization process is completely ready to be exported to a real experiment with no change on the code. Figure 6 shows the map frame after the execution of this simulated experiment.

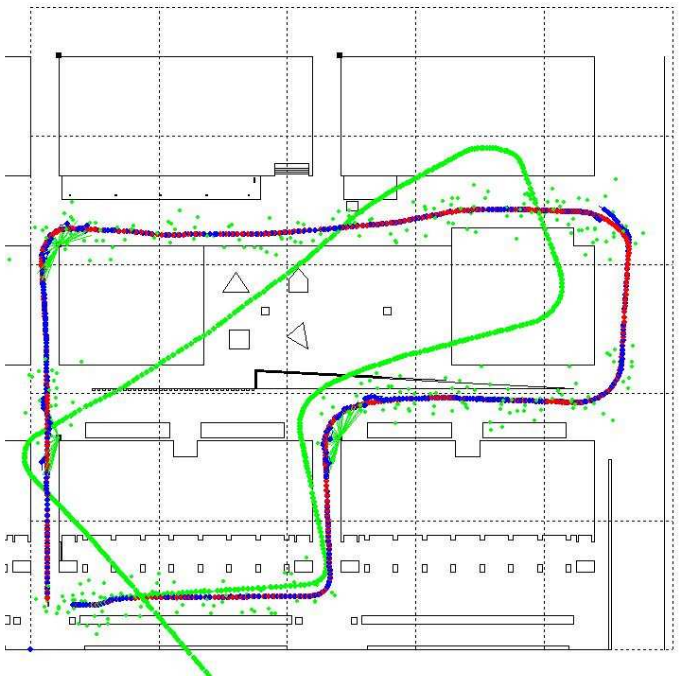


Fig. 6. Simulation of a position tracking experiment. Red poses are for ground truth. Blue poses are the output of the filter. Green poses form the odometry path. Little green dots are GPS data. Green segments are camera detections.

6.2 Real-world experiments

This experiment is a position tracking experiment of the segway platform RMP-200 of figure 7 (left). This position tracking is processed at real-time, since the filter output rate was about $3Hz$ and the maximum robot speed was about $0.5m/s$. Since the overall camera network infrastructure and robot detection algorithms are not yet fully operative, the localization filter only fuses onboard sensor data, coming from a front laser, a back laser and the odometry of the platform. However, the robot position is sent throughout the ouput port(see figure 3) and a remote computer connected to this port can see the position of the robot. In this experiment we have validated that the proposed software is operative in real conditions, but also we have ascertained that integration of the provided localization service can be easily done if a receiver process incorporates a localizationReceiverPort object and connects it to the output port of the localization process. Figure 7 (right) shows the map frame after the execution of this real experiment.

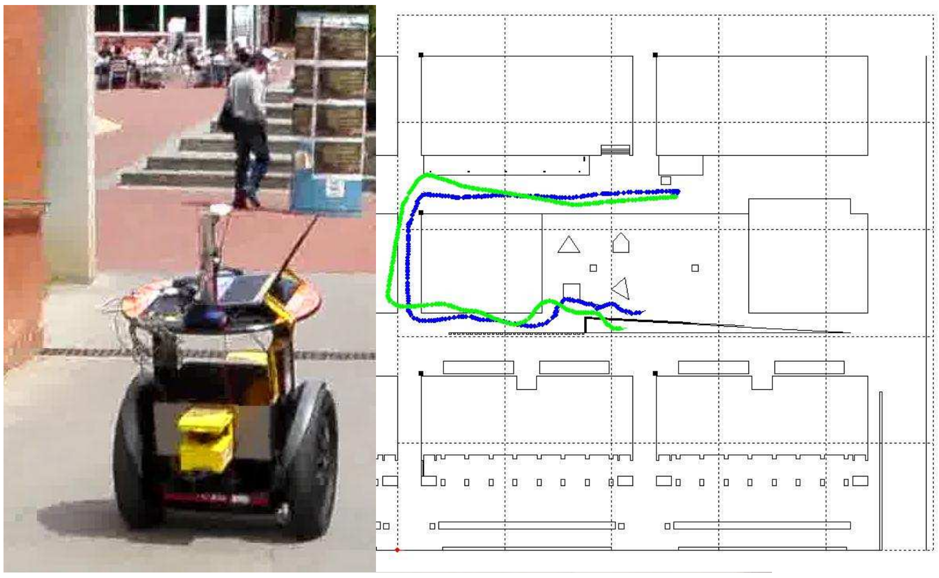


Fig. 7. Left: The segway robot with two lasers and one computer onboard. Right:Real position tracking experiment. Blue poses are the output of the filter. Green poses form the odometry path.

7 Conclusions

This paper presents a software architecture to solve navigation tasks for autonomous mobile robots operating in cooperative environments. We mean by a cooperative environment an area where a sensor network is deployed and a team of robots operates in it. This network robot system is the context of the URUS european project where eleven industrial an academic partners are developing joint research. Both engineering and social contexts of this project force to develop software following three main aims: real-time constraints for mobile robot navigation techniques, easiness on integration software modules and decentralized computing approach.

Real-time constraints in navigation techniques is a mandatory issue if we want that the robots operate autonomously in such environment. Easiness on integration is due to the fact that the proposed experiments demonstrating the validity of the whole project involve several partners and several robotic fields such as computer vision, data fusion or human-robot interaction. Finally, a network robot system approach implies that a set of computers are physically (wired or wireless) and logically connected to share any kind of data that each process requires and provides.

The proposed approach, based on the YARP middleware, satisfies these three aims and has been already tested in simulation and in a preliminary real outdoor experiment, showing its potentialities, specially in terms of integration.

References

1. H. Bruyninckx, "Robotics Software: The Future Should Be Open," *IEEE Robotics and Automation Magazine*, vol. 15, pp. 9–11, March 2008.
2. D. Brugali, C. Schlegel, T. Stumpfegger, and S. Tansley, "Third International Workshop on Software Development and Integration in Robotics, SDIR 2008," (Pasadena, USA. May, 2008.).
3. P. Fitzpatrick, G. Metta, and L. Natale, "Towards Long-Lived Robot Genes," *Journal of Robotics and Autonomous Systems*, vol. 56, pp. 29–45, January 2008.
4. A. Makarenko, A. Brooks, and T. Kaupp, "On the Benefits of Making Robotic Software Frameworks Thin," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (San Diego, California. October, 2007.).
5. `http://orca-robotics.sourceforge.net/`.
6. `http://www.orocos.org`.
7. `http://playerstage.sourceforge.net/`.
8. `http://eris.liralab.it/yarp/`.
9. A. Sanfeliu and J. Andrade-Cetto, "Ubiquitous networking robotics in urban settings," in *Proceedings of the IEEE/RSJ IROS Workshop on Network Robot Systems*, (Beijing, China. October, 2006.).
10. G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal on Advanced Robotics Systems*, vol. 1, no. 3, pp. 43–48, 2006.
11. M. Barbosa and M. Ransan, "URUS Communication Protocol," tech. rep., September 2007.