A Generic Software Platform for Controlling Collaborative Robotic System using XML-RPC

G. Glez. de Rivera, R. Ribalda, J. Colás, and J. Garrido (*IEEE Member*)
Escuela Politécnica Superior. Universidad Autónoma de Madrid
Email: guillermo.gdrivera@uam.es, ricardo.ribalda@uam.es, jose.colas@uam.es, javier.garrido@uam.es

***Abstract*—** This paper describes a software platform used for controlling any set of collaborative robots. The platform is specially designed for users without special skills on hardware design or communication topics. The platform provides a standard to simplify the addition of new hardware devices. The system runs over Linux operating system; it is accessible through different programming languages. Calls among architecture processes are performed using XML-RPC. Data transport is TCP-IP based; therefore the system is accessible from a conventional Internet link. Some experiments are performed in order to detect the programming languages that better fit in the architecture and the better web server for operating. It was found that php, in comparison with C language, reduces more than three times the speed of call processing, and more that seven times in comparison with c-sharp language (MONO implementation). Using CGI to access an Apache server is twice faster than using s standalone server.

I. INTRODUCTION

At the moment there is a great demand of robotic systems to solve complex tasks in fields as manufacturing, construction, transportation, medicine and others. Furthermore, in web-controlled systems, robots play the role of a physical mediator, enabling people to remotely acquire information, explore, manipulate, communicate, and interact physically with other people far away [1-6]

Currently, in robotics research groups it is not strange to find small very specialized research groups in certain subjects but without any background in others, like the robotic field. This implies multiple and very diverse disciplines, it is difficult to find research groups with the multi-disciplinary degree needed. The main idea of the present work is to design and implement a development platform that facilitates the work of researchers in fields like AI (artificial intelligence), Neuronal Networks, Navigation, and in general, applications where the use of robots is required. Additionally, it would be interesting to integrate this platform with the necessary tools to establish, in a simple way, the dialogue between different platforms, and offer support for techniques and algorithms directed towards collaborative agents and distributed systems. The development is completed with a software layer that allow to access from high level to all the resources of the robot, as well as it is possible an extension or modification.

II. OBJECTIVE

The objective of this work is to present a generic and flexible platform that allows abstracting of the hardware problem to the software specialist, especially in the subjects related to robotic cooperative systems. It also serves as an aid to the hardware designer, who can count on a finished and proven system. The hardware specialist can add peripherals to the system, just fulfilling a simple standard previously defined. The final result is formed by a hardware structure and some applications/libraries.

The hardware structure is composed of a movable platform equipped with sensors and network interfaces to allow collaboration between different platforms.

The software part is composed of the drivers necessary to handle the sensors and actuators and the protocols necessary to intercommunicate robot-robot and robot-application. For the end user of the platform, the system is a "black box" accessible through simple remote calls.

III. GDRBOT PLATFORM

In this section we describe the GdRBot platform, specially its software component. The GdRBot platform is a free and standard implementation of a robotic system, very flexible and easy to use.

The platform is formed by two types of elements: *Clients* and *Servers*, see figure 1, running in different types of hardware platforms. A client makes requests and monitors the servers through XML-RPC calls. If a client is unable to use the physical medium of the server's network, it sends the requests through a bridge. The server is installed in the robots, and clients can be installed in the robot or in any processor based system. Clients installed inside the robots are used to do survival tasks, as avoiding crashes, and to do cooperative tasks, as playing a soccer match. In a few words, the server handles the hardware and the client does the logic work.

In practice, each robot is built around an element of relatively powerful control, with numerous network interfaces and a non-defined number of transducers /actuators, to support the maximum number of possible tasks.

The platform must govern all these elements in a simple and flexible way.

The reasoning on which elements compose the server follows a top/down strategy and it concerns the robot exclusively.

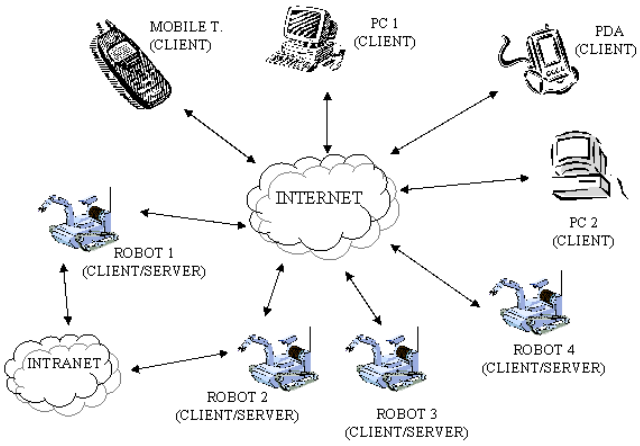

Fig. 1. System architecture. It is formed by two types of elements: Clients and Servers connected trought any type of network.

A. *Architecture*

The design should provide a system that allows to carry out 3 types of actions:

- *Survival*: The robot must be an autonomous entity with the possibility of making decisions by itself. For example: "to avoid crashes".
- *Remote Control*: The user should be able to control a robot in a remote way for carrying out actions, (for example: moves), or to show us its environment sensations where it is moving (for example: "show us the camera").
- *Interoperability*: The robots must be able to interact with each other to carry out common tasks, such as design a floor plan building.

To provide these actions we have two options:

- One server for each action type in every robot.
- One unique server to carry out the tasks in every robot.

If the first solution is chosen (several servers) we will have to face problems like the arbitration in the use of the robot's elements hardware (for example: "when we want to move a wheel towards remotely and the survival system opts for the opposite."). The solution for this problem goes by the generation of a POSIX like semaphore which arbitrates the use of the hardware. This solution complicates the development of the control of the hardware. Likewise, we will generate three different elements, with different interfaces, multiplying for three the necessary efforts for the generation of the same ones. However, as a positive aspect, we will have an specialized and very efficient software that supports interruption driver tasks.

If the one-server solution is adopted, we will not have to face competition problems for the hardware, neither we will duplicate efforts in the development of the same ones. However, the software will be less efficient and will not support interruptions. Nevertheless, these mechanisms can be substituted by pooling.

Since simplicity is one key in our platform, we have opted for a system built only with one server.

B. *Remote process*

Once we have already decided on a architecture based on a unique server for each control unit, we must decide the way of communicating with this server. We need a simple system, (if the development gets complicated the platform will not make sense), that supports in a simple way, great quantity of languages (we can not force the development team to use a particular language), it must be standard (so it is open to new languages) and that it can works in several operating systems in a shadow way.

The most mature alternatives that exist at the present are: Corba, DCOM, SOAP, RMI and XML-RPC (http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto-corba.html).

- Corba: It is a popular language to write distributed software guided to objects. It is very supported and possesses an enviable IDL, but it is very complex. It requires very sophisticated clients and it is difficult to implement.
- DCOM: It is the answer from Microsoft to Corba. It is easier of using; however, it only works in MS-Windows.
- SOAP: It is based on XML+HTTP, but it specification is not very good, and it has unnecessary elements.
- RMI: It is the system of distribution of Java language. It is very potent and easy to use, however it can only be used from Java. It is not a good option for our system that seeks just the opposite, to be as most supported as possible.
- XML-RPC: It is based on the HTTP protocol to transport and XML to code. It is, therefore, a highly standardized and easy system. Likewise, it possesses implementations for almost any well-known language. In the case of using a language that does not have implementation of XML-RPC, it is very simple to develop it, the standard does not have more than 10 pages. To summarize: XML-RPC means simple and standard. As compensation it is necessary to highlight the overload caused by HTTP and XML. At the present, there are more than 100 official implementations of XML-RPC that give support to more than 40 programming languages (http://xmlrpc.scripting.com/directory/1568/implementations).

Another big advantage of XML-RPC is that it supports Reflection, that is, the services can be auto-described.

To finish, we would like to highlight the last great advantage of XML-RPC: our robots are open to the Web world. Today, the protocol that web systems use to intercommunicate is usually XML-RPC. To carry out webs applications integrated with robots has never been easier.

C. *Transport*

Once we have decided on our server and the communications nature, we must decide on the way we will use to transport requests. The answer seems very simple: TCP/IP.

TCP/IP supports multitude of physical supports and you can even encapsulate easily in any protocol as ATM.

Working with great quantity of physical supports is a great utility. In robotic there are some scenarios where the use of a particular physical support is totally discarded, and others where the work conditions are so specific that it is only possible to use a particular one (robots for use in space).
We have chose a set of physical supports among the following ones: Ethernet, 802.11, Serial port, Parallel port , GSM modem , USB, Bluetooth and IrDa, although the use of another one, due to the use of TCP/IP, will not imply design changes

In some cases, we will need to adapt the network to work with lower delays. We will solve this problem using external tools to the platform

D. *Operating system*

Once we have selected the high level transport protocol, the procedure for remote calls and the design of the server, the following step is the operating system selection.

This selection is one of the most important of the system, because it will significantly affect the efficiency, adaptability and flexibility of our platform.

Since the communication will be made through XML-RPC+TCP/IP, the interoperability is guaranteed, but we will need to analyse other different elements to these ones, such as yield, tweaking, connectivity and compatibility with different hardware architectures (http://www.kernel.org).

The option more indicated in this case is LINUX. Linux is easily to modify for the final user, because it has a lot of documentation and sources to the user's disposition, likewise it supports a great quantity of architectures (32-bit, Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64, AXIS CRIS, and Renesas M32R architectures). Thus, we will be able to substitute (interchange) the robot hardware without problems.

Besides, Linux also works very well with not very powerful hardware.

As compensation, we need to highlight that some manufacturers hide information about their devices, what disables their use in Linux. However, we can affirm that with Linux, the platform will be open up to more hardware than with others that we can choose.

Another reason could be that Linux is focused to network and security. The nucleus of Linux has support for infinity of physical mediums for IP, as well as routing, bridging, firewalling, etc.

To carry the software to other operating systems would not suppose a great work, although it would reduce their functionality, when supporting less architectures hardware and less physical networks.

E. *Programming language*

The last important decision to make is the programming language to use to implement the robot server.

It is necessary to keep in mind that XML-RPC protocol uses HTTP like transport protocol, so we will need also a Web server. Since we have a powerful control unit and Linux has a wonderful Web server called Apache, we will make use of it, although if we do not have enough computational power or we decide to reduce that cost, it would be possible to choose a lighter Web server or implement it by ourselves.

Once solved the problem of the HTTP protocol, we will need to choose an extensively tested, stable and powerful language that allows the communication with the hardware in a simple way. It can seem that the use of an interpreted language is the solution, because it would allow to absorb of the operating system, but later we will see that this solution, in the same way that separates us from the operating system, and it takes us away from the hardware, it also lowers the yield. Since the nature of our server is exactly the opposite (access to the hardware), the language must be compiled.

Now, for what we have argued above, it can seem that the best option is the use of a language that is very related with the hardware, just as the assembler. However we will quickly realize that we must carry out complex tasks at low level (XML-RPC), and that we would be restricting our system to a specific platform hardware.

For all this, it seems to be that the inflection point among a high-level language like Java and a very low level language like assembler, it is the C language. It allows us to make very high level tasks, due to the libraries while it does not move us away from the hardware, to which we can continue access in a simple way.

F. *Final specifications*

Finally, the system software specifications are the following:

TABLE 1
PLATFORM SPECIFICATIONS

Architecture	Monoserver
RPC	XML-RPC
Transport	TCP/IP
Operating System	Linux
Language	C

IV. PLATFORM DESIGN

Once analysed the development requirements, now the platform design will be detailed studied.

A. *The client*

The client is a program or service that carries out requests or monitors a server. It is composed of three parts:

- *Application*: It is contributed by the final user and is developed in the language selected by him. It will be able to run on any operating system.
- *Robotics library*: It is contributed by our platform for some languages. It simply abstracts the user from the XML-RPC protocol.
- *XML-RPC library:* It is contributed by a third part. It implements the XML-RPC protocol. If necessary it is based in open standards, so it is possible to develop from zero without being too complex.

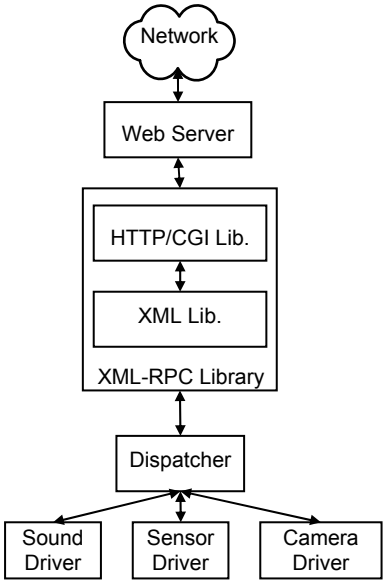

Fig. 2. Robot server block diagram.

B. *The Robot server*

The server is installed in the robot. Basically, it is composed of the following parts (figure 2):

Network Interfaces: They allow the robot to communicate in different networks, and different physical media.

Control Unit: It carries out the logical actions of the server. It receives through the network interfaces XML-RPC requests which must be dealt independently. It can communicate with the hardware through different elements, such as: Serial ports, PCI, Parallel port, USB, etc. From a global view, it is composed of:

- *WEB Server*: Its function is to negotiate and process HTTP requests using CGI, it transmits these requests to the Sensor Server.
- *Sensor Server:* It is executed through CGI and its function is to analyse the Web Server requests. Then, it processes them and returns a Web page where the answer is coded. The sensor server is composed of the following elements:
  - *XML-RCP library*: It parses the requests carried out by the Web server and evaluates their parameters. Likewise, it executes the corresponding function for this request. It is composed internally of two libraries: HTTP and XML.
  - *Dispatcher*: It arbitrates the access to the different devices to avoid conflicts. Likewise, it distributes the request to the different drivers.
  - *Nucleus:* It allows to access the hardware (network devices like the different sensors/actuators).
  - *Drivers:* They carry out the low level access to the different devices.

V. Communication Stack

Once detailed the architecture high level design, we will describe how a request is manipulated by the different elements in the platform (figure 3):

An application decides to transmit a request, for that it uses the robotic library or the XML-RPC library. If the request is incorrect an error is returned.

The request is transformed into a XML document that will be transmitted using the HTTP protocol.

The communication is divided into TCP/IP frames that will travel through one of the different networks in function of the message destination. In the case of lost frames, TCP retransmits the frame to obtain semantic transparency.

If it is necessary, a bridge will retransmit the request to a different physical network without modifying its content.

Once this frame has arrived to its destination, the packages will be decoded to form a HTTP request which will be transmitted to the WEB server.

The Web server will identify the message and it will send the message to the robotic server using CGI. If the HTTP protocol is not completed an error page will be generated.

The robotic server obtains the XML document which will be parsed by the XML-RPC library. If a nonexistent method is requested, or the protocol is not completed, a XML document it is returned with an error message.

The drivers will pass the request to the different hardware elements.

The answer obtained from the drivers will be transformed into a XML document using the XML-RPC library.

The XML document is transmitted to the Web server that retransmits it to the client making all the previous steps in inverse order.

VI. An Example of Using the Application

As one example of the platform capacity for collaborative work, we will try to solve the problem of carrying out a work coordinated among a group of robots, supervised by a client PC [7]. For this case, in the platform we will install two different clients located at the robots:

Survival Client: It constantly pools over error conditions inside the robot, such as crashes.

Collaboration Client: It carries out the collaborative tasks among the robots.

A client installed in the PC that will take charge of monitoring the robots.

The client inside the robot, can be developed in any programming language, but for the same operating system used by the robot (in this case Linux).

The client inside the PC can be developed in any language and in any operating system. If operating system is not compatible with the physical medium of robots network, a bridge will be used.

VII. Experimental Results

Because of the platform architecture can be adapted to different languages and different web servers, a set of experiments are made to evaluate the best configuration.

4.1.- Choosing a language: In this first experiment, we do a number of calls to the robotic server from different clients implemented in the robot but in different programming languages.

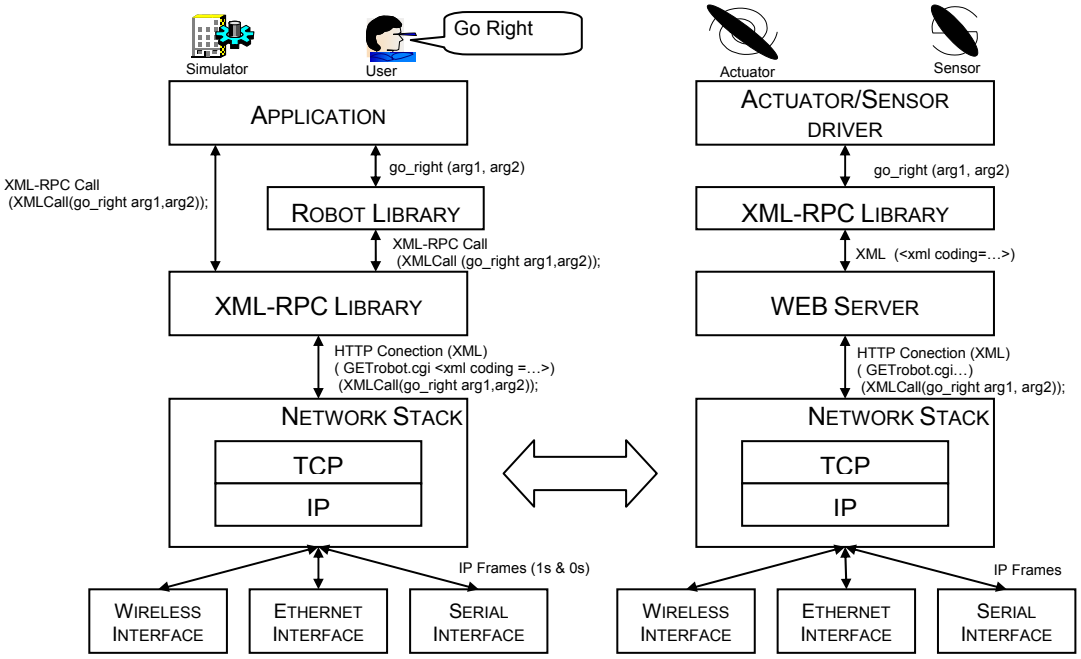

Fig. 3. Protocol Stack example. It is showed how an order (Go Right) is processed from the user or simulator to the involved sensor or actuator.

All calls are made to a server running as an Apache's CGI, and because calls are from one robot to itself, the network delays are inappreciable. In this case delays that affected to this experiment are delays for creating TCP/IP frames (managed by the kernel, so they are constant in every language), delays dued to inserting a XML petition on HTTP protocol (they depend on how they are implemented), delays dued to XML parsing (they also depend on how they are implemented, but they are harder process than managing HTTP protocol) and finally delays dued to calls processing.

These different representative languages have been chosen:

1. C using XML-RPC-c library available at http://XML-RPC-c.sourceforge.net/: it is a general purpose compiled language.
2. PHP using The Inutio XML-RPC Library available at http://scripts.incutio.com/XML-RPC/: Interpreted language designed for creating dynamic web pages.
3. Java using The Apache's Library available at http://ws.apache.org/XML-RPC/: Object based language interpreted with precompilation to bytecode.
4. c# (Mono platform available at http://www.mono-project.com/ using the XML-RPC.NET library available at http://www.xml-rpc.net/: New object based language, interpreted with precompilation to bytecode.

Figure 4 shows the results of such experiment. As it can be observed, the most efficient language is PHP, this is because the library used for XML-RPC makes XML parsing very fast. Also, the HTTP transport is made by a PHP's core library, which is also very fast. After PHP, the next fastest language is C. The reason for C been slower than PHP (despite being a compiled language) is that the library used for XML-RPC uses internally very slow libraries for HTTP transport (http://www.w3c.org/Library/) and XML parsing (http://www.jclark.com/XML/expat.html). The worse behaviours are produced by c-sharp and Java clients. Java shows some dispersion in its graphic; this is because, its virtual machine is unpredictable. Mono is the slowest language, and this is because the implementation used (MONO) is still too immature.

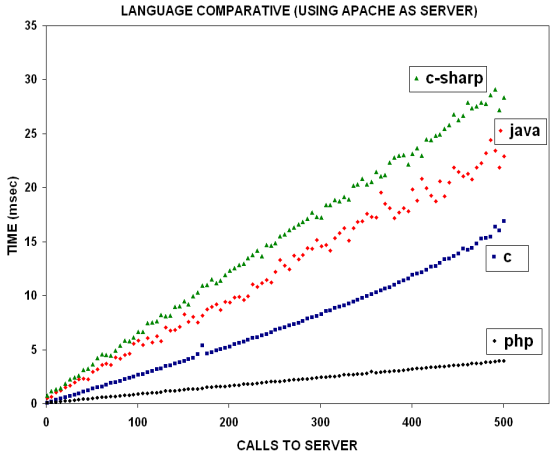

Fig. 4. Language comparative. It is represented time taken by calls realized by different languages.

4.2.- Choosing Web server Type: In this experiment, two different clients (PHP and C) are tested over two different servers. One server is an Apache's CGI , and the other is an standalone server with the robotic server as part of it. The standalone server is based on Abyss web server available at http://abyss.sourceforge.net/.

The graphic of figure 5 shows that C works much better with the Apache's CGI than the standalone server. In other hand, PHP work better on the standalone server, but in a lower order.

In all cases, every language shows a linear behaviour.

If the server is standalone, it is always running, and a fault in a driver can make the whole robotic system crash.

4.3.- Multiple Calls: When its needed to make polling to the server, its possible to choose between two different methods.

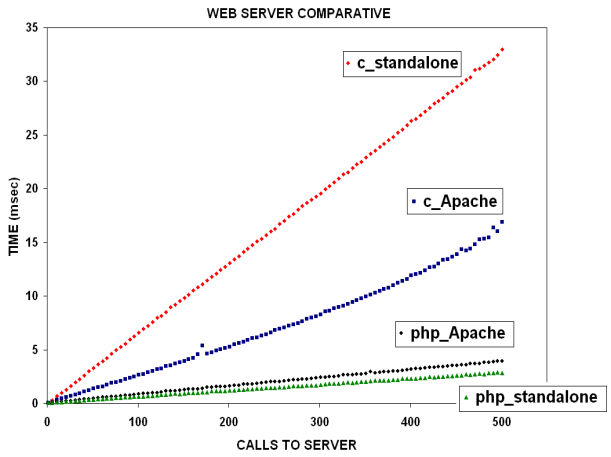

Fig. 5. Web server comparative. It is represented time taken by calls realized by c and php languages to two different servers Apache-CGI and standalone.

The first option is doing these calls synchronously (before doing next call, we wait for the response to came back). The second option is doing the calls asynchronously (all the calls are made at once, without waiting for the responses).

In these experiments (figure 6), some calls have been done synchronously and asynchronously, from a C client running in the robot and in a PC on the Internet to a robot server running as an Apache's CGI.

If the client is located in the robot, the best method is the synchronous one, because the asynchronous method produces an overhead, and the network latency is zero.

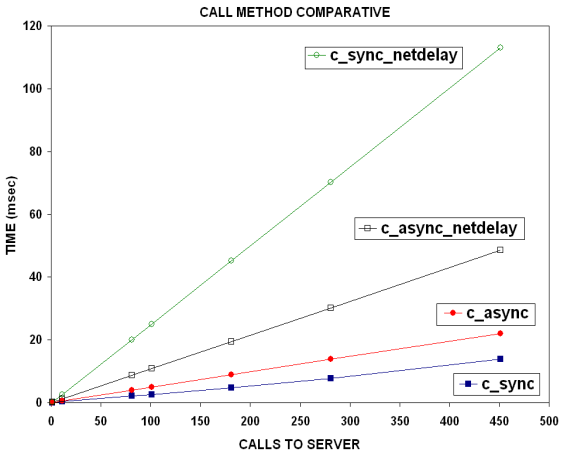

Fig. 6. Call method comparative. It is represented time taken by calls from a client inside the server or from an external client. Also it is represented the time taken comparative if calls are performed in asyncronal or syncronal way.

On the other hand, if the client is located in a PC in the Internet, the best method is the asynchronous, because the network latency is not negligible. In the asynchronous method, this latency just affect once, but in the synchronous it affects every call.

VIII. CONCLUSIONS AND FUTURE WORK

To reduce the time for robotic developments, it has been designed a platform that allows a non technical user to program and use a set of robots with great flexibility and simplicity.

The use of this platform guarantees the user will be able to create great variety of new applications, at the same time it allows the reusability of old platform elements. This platform is not dependent of any element, and so it is able to guarantee their durability in the future.

The platform described in this document completes all these characteristics. As future work we have created a directory system inside the platform, that will show which servers are working at each moment and which are the functionalities that they offer in order to facilitate cooperative tasks.

Because XML/RPC is not specially efficient for video streams transmission, two solutions are being studied: communication through RTP or HTTP without loading of XML/RPC. In last case it has been carried out some satisfactory tests.

REFERENCES

[1] D. Wang; X. Ma and X. Dai, “Web-based robotic control system with flexible framework”. Proc. ICRA '04. 2004 IEEE International Conference on Robotics and Automation, Volume: 4, pp:3351-3356.

[2] S. Dissanaike; P. Wijkman and M. Wijkman, “Utilizing XML-RPC or SOAP on an embedded system”, Proc. 24th International Conference on Distributed Computing Systems Workshops, 2004, pp438–440.

[3] X. Wang; M. Moallem and R.V. Patel.” An Internet-based distributed multiple-telerobot system”. IEEE Transactions on Systems, Man and Cybernetics, Part A, Vol:33, Issue: 5, Sept. 2003, pp:627 – 634.

[4] R. Marin, P.J. Sanz, P. Nebot and R. Esteller, “Multirobot Internet-Based Architecture for Telemanipulation: Experimental Validation” IEEE International Conference Systems, Man and Cybernetics, 2003, Vol: 4, 5-8 Oct. pp:3565-3570.

[5] K. Taylor and B. Dalton, “Internet robots: a new robotics niche”, IEEE Robotics & Automation Magazine, Vol: 7 , Issue: 1 , March 2000, pp:27-34.

[6] H. Hiraishi, H. Ohwada and F. Mizoguchi. “Web-based Communication and Control for Multiagent Robots”. Proc.IEEE/RSJ Int. Conference on Intelligence Robots and Systems. Victoria B.C. , (Canada), 1998, pp 120-125.

[7] G. Glez. de Rivera, K. Koroutchev, R. Ribalda and J. Garrido, “Occlusion Avoiding using Group Evolution Learning”. In preparation