A Grid Architecture for Comfortable Robot Control

Stéphane Vialle[1], Amelia De Vivo[2], and Fabrice Sabatier[1]

[1] Supelec, 2 rue Edouard Belin, 57070 Metz, France
Stephane.Vialle@supelec.fr, sabatier_fab@metz.supelec.fr
[2] Universitá degli Studi della Basilicata, C.da Macchia Romana, 85100 Potenza, Italy
devivo@unibas.it

Abstract. This paper describes a research project about robot control across a computing Grid, first step toward a Grid solution for generic process control. A computational Grid can significantly improve remote robot control. It can choose at any time the most suitable machine for each task, transparently run redundant computations for critical operations, adding fault tolerance, allowing robotic system sharing among remote partners.
We built a Grid spanning France and Italy and successfully controlled a navigating robot and a robotic arm. Our Grid is based on the GridRPC paradigm, the DIET environment and an IPSEC-based VPN. We turned some modules of robotic applications into Grid services. Finally we developed a high-level API, specializing the GridRPC paradigm for our purposes, and a semantics for quickly adding new Grid services.

1 Motivations and Project Overview

This paper introduces a Grid architecture for comfortable and fault tolerant robot control. It is part of a larger project aiming to develop a grid solution for generic process control.

Motivations. In several real situations robots must be remotely controlled. This is because we install robots where an application requires them. It can be a not computer-suitable environment, for example, for temperature constraints. Sometimes the building where the robots are is far away from that one where the computing centre is. In such a case probably the computer maintenance team is in the computing centre and it could be uncomfortable and expensive to have some computers near the robots.

Simple robots, like robotic arms, just need to receive commands and sometimes to send feedback. A simple remote application can manage the situation, but a devoted machine makes sense only if the robot is continuously used. An integrated environment, like a Grid environment, can run the robotic application on the first available computer when needed.

Complex robots, like navigating robots, are equipped with sensors and devices acquiring data about the surrounding environment. They send these data

to a remote server running complex computations for deciding robot behavior. Navigating robot applications sometimes, but not always, need a powerful machine. A computational Grid could find a suitable machine on the fly, avoiding to devote an expensive computer to the robotic system.

Finally, a computational Grid can be useful and comfortable in several other situations. It can automatically switch to an unloaded computer when the current one gets overloaded, guaranteeing some QOS to time-constrained robotic applications. Some applications are embarrassingly parallel and a Grid can offer a different server for each task. A single application execution is not fault tolerant for critical missions. A Grid can automatically run the same application on different machines, so that if one fails, another one can keep robot control. In a robotic research environment, a Grid allows remote partners to easy share a robotic system.

Project Roadmap. In 2002 we started a four-step project about remote robot control across a Grid. A careful evaluation about delay, security policies, Internet uncertainty and so on [4, 6] was mandatory and the first step was about it. We worked with a *self-localization* application for a single navigating robot [7], using "ssh links" and a simple client-server mechanism. We focussed on ad-hoc overlap techniques for amortizing the Internet communication, and at the end we achieved a reasonable slow down [8]. In the second step we built and experimented a Grid based on a secure VPN and DIET [1]. We added a *navigation* module, we turned both modules into Grid services, and we designed and implemented an easy-to-use and easy-to-expand robotic API [2]. In our Grid configuration the robot, a client and some computing servers were in a single site in France, while another server was in Italy. The third step is a working in progress. Using our API we quickly and easily added a *lightness detection* Grid service for environment checking. Then we extended the Grid with other two sites in France, each one hosting just a computing server. Finally we added a second robot (a robotic arm) and a related control module. In the fourth step we will investigate the way to adapt our software architecture to the Globus middleware. The current solution is very suitable for our applications, but Globus is an example of more generic and standard middleware.

2 Robotic Applications and Grid Testbed

Hardware Resources. The physical resources on our Grid are two robots, some PCs at Metz Supelec campus, two PCs in two different sites in Metz and a PC at Salerno University, see figure 1. Our robots are an autonomous navigating *Koala*, with several onboard devices, and a robotic arm. Both are connected to external servers through serial links. Each server is a devoted PC controlling basic robot behaviors. All robotic applications are clients of these servers.

Grid Environment. We chose the DIET [1] (Distributed Interactive Engineering Toolbox) Grid environment. It supports synchronous and asynchronous

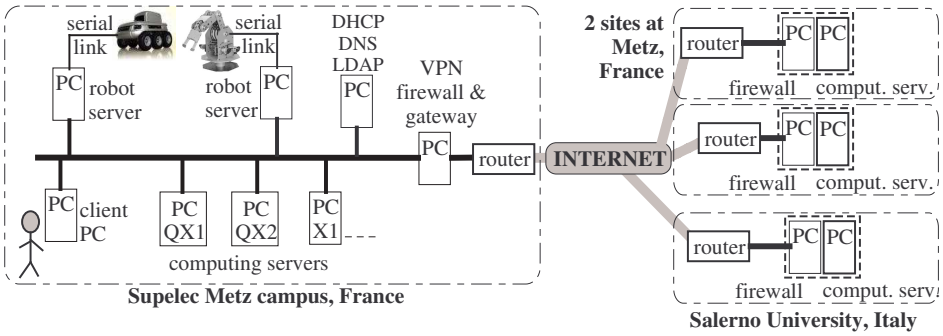

Fig. 1. Grid Testbed for robot control: 2 robots and four sites.

Grid-RPC calls [5], and can be considered a Grid Problem Solving Environment, based on a Client/Agent/Server scheme. A Client is an application that submits a problem to the Grid through an Agent hierarchy. This avoids the single Agent bottleneck. Agents have a Grid Servers list and choose the most suitable for a Client request, according to some performance forecasting software.

DIET communication is Corba-based, but this was a problem for our institutes security policies. In order not to relax our respective security levels, we created an IPSEC-based VPN [3]. This just requires the 500/UDP port opened and the ESP and AH protocols authorized on the destination gateway.

Robotic Testbed Applications. The testbed application for the *robotic arm* is a simple action loop, while the testbed application for the *Koala* robot consists of three complex modules: self-localization, navigation and lightness detection. Our robot navigates in dynamic environments, where no complete pre-determined map can be used. Artificial landmarks are installed at known coordinates. When switched on, the robot makes a panoramic scan with its camera, detects landmarks and self-localizes [7]. Based on its position, it can compute a theoretical trajectory to go somewhere. For error compensation, new self-localizations happen at intermediate positions. During navigation the robot checks the environment lightning and, eventually, signals problems. For this purpose it moves its camera and catches images. The *Koala Server* always sends its clients JPEG-compressed images.

3 Software Architecture and Grid Deployment

Figure 2 shows our Grid architecture. At the toplevel, a *Grid application* is almost like a classical application. It calls *RobGrid API* functions (see section 6) to achieve Grid services. Our Grid services implement robotic application modules, so that they appear as high-level robot commands, such as "Localization"

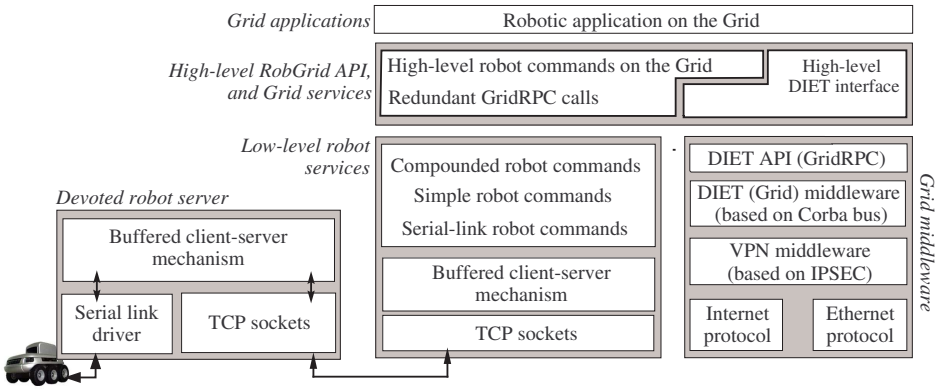

Fig. 2. Software architecture overview

or "Navigation". Users can call them concurrently, but in this case they are responsible for robot devices coordination and synchronization.

We implemented our Grid services using the DIET GridRPC interface, according to the *RobGrid API* semantic rules (see section 5). Depending on the service, it can make a synchronous call to a low-level service (to control just one robot device), an asynchronous call to a low-level service (to simultaneously control more robot devices), or several concurrent asynchronous calls to low-level services, for example to run redundant computations on different Grid servers. When a Grid service runs redundant computations, it waits just for the first one to finish, ignoring or cancelling the others. All these details are hidden to the programmer that can focus on robotic problems.

Each low-level robot service is implemented as a three layers stack. A *buffered client-server mechanism* allows different and/or redundant tasks to concurrently access a robot server through *TCP sockets*. This supports the fault tolerance strategy of our grid (see section 4). Finally a *serial link driver* runs on each devoted robot server.

The Grid middleware consists of the DIET API, a Corba bus and a secure IPSEC-based VPN. We configured IPSEC so that our grid has a main node at Supelec, including a LAN segment and a gateway-and-firewall PC. Each of the other three sites have just a stand alone computer, with a lighter configuration. It does not include a gateway, so that the stand alone PCs can only communicate with the main node. The client machine can access all Grid services only if it is on the main node. Of course this solution is not suitable for a larger Grid and we are currently working for removing this limitation.

IPSEC requires just the 500/UDP port for security keys exchanging, and the AH and ESP protocols for secure communication. This made possible to deploy an IPSEC-based VPN without changing our local security policies.

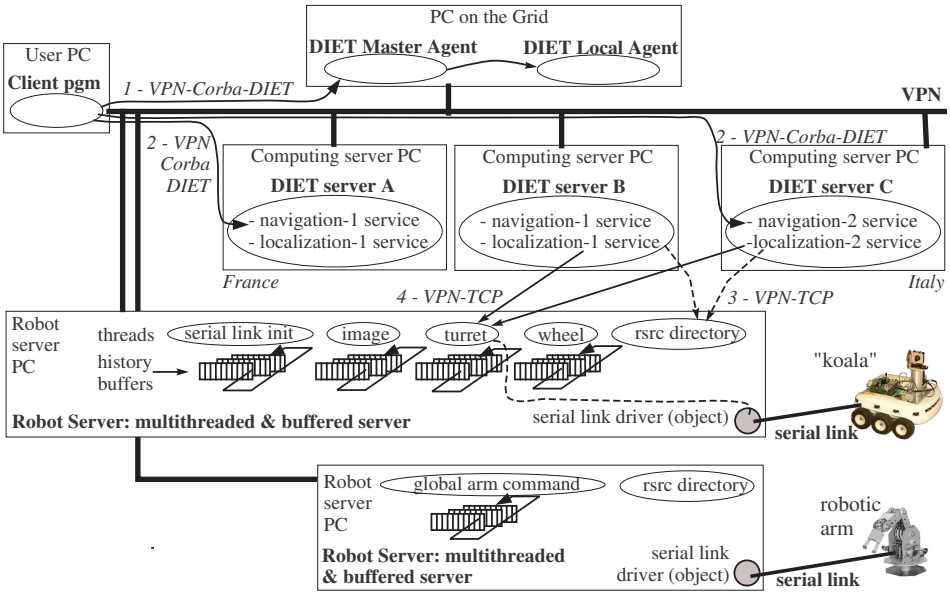

Fig. 3. An operation sequence example

In figure 3 there is an example of (*Koala*) robot control across our Grid. The client application, on a client PC, requires some Grid services through the *RobGrid API*. The underlying DIET functions contact the DIET agents running somewhere on the Grid to know the addresses of the most suitable computing servers. Then the user program contacts them directly through the DIET protocol on the Corba bus, and each computing server establishes a direct communication with the *Koala Server*, using TCP sockets instead of the DIET protocol. This way camera images to be sent to the Grid servers can avoid Corba encoding. After processing, only small results (such as a computed localizations) has to be sent to the client machine across the Corba bus.

4 Low-level Robot Services

Devoted Robot Servers. A robot is a set of independent devices (wheel, camera, infra red sensors, articulation, grip...), that can be considered as a set of independent resources accessible through related elementary or low-level services.

A devoted robot server collects all elementary services for its connected robot. Each service is attached to a different port (see the bottom of figure 3) and is multithreaded, so that it can serve several clients. If necessary, it can also lock the related resource. For example, several clients (actually Grid servers)

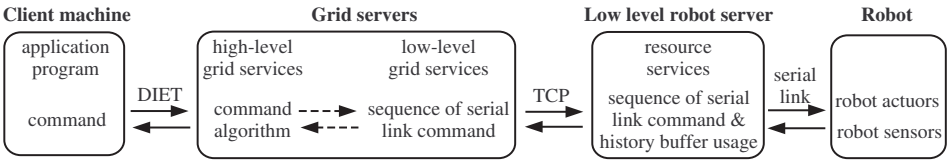

Fig. 4. Translation steps from an application command to robot device commands

can simultaneously connect to the camera for taking images while the robot navigates, but only one client at a time can drive the camera motor.

Resource Directory Service. In order to make the Grid servers independent from the robot server, we added a *resource directory service* (see the bottom of figure 3). Clients have just to know its port number for achieving the directory of all available resources and related ports. This way low-level robot services upgrades have no impact on Grid services.

History Buffers. Redundant computations require the same robot data for all redundant clients, so we associate a number of *history buffers* to each robot device. On figure 3 each robot resource has three buffers. Each Grid service has a default number of history buffers, but applications can modify it.

Before to reset a history buffer, we have to consider that several redundant Grid services can be using it. Each Grid service needs a *reset strategy* depending on the related robot device and on the Grid service algorithm. All reset strategies are implemented in the *RobGrid* API (see section 6).

Slow Grid servers asking for too old execution are rejected and finally cancelled. However, slow servers get results faster because they have not to wait for robot actions. So, few servers are actually rejected and cancelled. If the network load changes during the execution, the Grid server that sends the new next command may change too. In this case the robot server goes ahead, driven by the new fastest Grid server.

5 Grid Services

Grid services implement robotic application modules and, of course, we can add other of them in future. Our *RobGrid* API introduces a semantics for Grid service programming. This makes Grid service development easy for robotic researchers.

For adding a new Grid service it is not required to deal with low-level details and to manage the communication with the robot server. Grid services have just to implement four sub-services and to define a *reset strategy* for history buffers:

– **Connection**. This is done by a generic Grid service that contacts the resource directory service, asks for the required TCP port and connects the application to a low-level service.

– **Disconnection**. This is done by a generic Grid service that closes the application connection to a TCP port.
– **History Buffer Reset**. This is a simple reset request for one of the history buffers associated to the related device. An index identifies the required buffer, according to the buffer reset strategy of the Grid service.
– **Robotic Command Execution**. This is something like *navigation(x,y,theta)*. Grid services execute each high-level robot command calling the staked up low-level robot commands. Each stack layer decomposes a higher-level command in more simple commands. For example the navigation command above can include *move_straightforward(x,y)*. At the bottom of the stack there are basic commands (serial link commands) to be sent to the devoted robot server across TCP sockets. Here the serial link driver gets the robot to execute the lower-level commands. Figure 4 illustrates the translation steps from an application command to robot device commands.

We experimented our semantic rules adding a new Grid service. It implements the *lightness measurement* module for the navigating robot application. Then we extended our testbed with a robotic arm and a related application module. The same low-level robotic library drives both the robotic arm and the navigating robot. In order to integrate the arm application module we added a new Grid service and related low-level services. In both cases we encountered no major difficulties.

6 The RobGrid API

To make the robotic application development easy, our *RobGrid* API offers classes and objects for Grid service interfacing. They hide DIET communication details and automatically initialize some DIET data structures. The application has to explicitly call three sub-services of each Grid service: *connection*, *disconnection* and *command execution*. *RobGrid* provides very friendly functions for this purpose. The *buffer reset* sub-service, instead, is called in a transparent way, according to the predetermined *reset strategy* of the Grid service which is implemented in a *RobGrid* object.

When creating a local interface object, a user just specifies a redundancy factor. The local interface object chooses the requested Grid servers, waits for the first one to finish, cancels the others, and returns the results.

Moreover, it is possible to request a Grid service through a local interface object in an asynchronous way. The API provides functions for testing and waiting for a service completion. See [2] for details about *RobGrid* API usage.

7 Experimental Results

Experimentation on a Local Sub-Grid. In real conditions, researchers working on new control algorithms are near the robotic system, for avoiding unexpected problems. A comfortable solution is to use a laptop, a wifi connection and

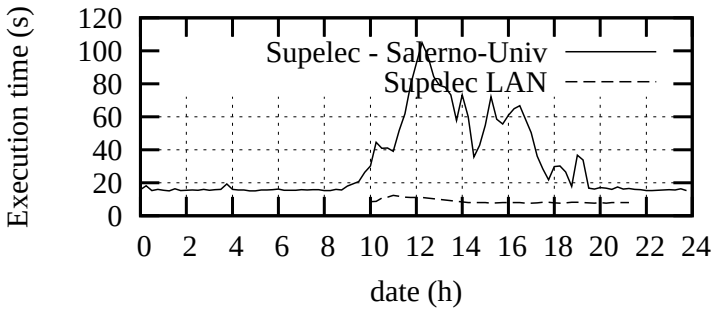

Fig. 5. Limited slow down across the Internet

a computing server for large computations. The wifi connection allows to control the robot and to walk around the robotic system when needed, and a computing server avoids to overload the laptop. But generally a computing server is a shared resource and can be loaded. A local grid including several servers can choose an unloaded machine on the fly. It can also run redundant computations on different servers and get the first available result.

Table 1 shows experimental performance on a local sub-grid for the self-localization module. We run redundant computations for critical tasks on several servers and wait for the first to finish. Execution time decreased until 9.5-8.5s against 13.5s measured with a single shared server. Last columns of table 1 show there is no remarkable overhead using DIET on this benchmark.

Performance Across the Internet. We measured the self-localization performance when the localization service runs in Italy. During 24 hours its average execution time elapsed from 15.5s during the night up to 100s during the day (see figure 5). In both cases remote localization succeeded and during the night the slow down was limited to a 2 factor compared to a local computation. So, running robot control services in Italy can be an interesting solution during the night.

Fault Tolerance. We experimented a complete long application for the navigating robot, with several localization and navigation steps, on the whole Grid.

Table 1. Execution time on a local sub-grid for the self-localization service. *Devoted resources* and *overloaded laptop* are not real situations

Laptop-wifi		Laptop-wifi+local sub-Grid		Laptop-wifi+devoted rsrcs	
Basic pgm (Std load on the laptop)	Optimized pgm (Overloading the laptop)	One Grid server	N redundant Grid servers	One server across an unloaded Grid	One server across a ssh link
14.34s	11.23s	**13.5-10.5s**	**9.5-8.5s**	8.65s	8.65s

Table 2. Execution time of the localization Grid service on different nodes

Grid machines	Computing PC server at Supelec (France)	Desktop PC at Metz (France)	Computing PC server at Salerno University (Italy)
day run	9.5s-8.5s	**21.5s-23.7s**	≈ 100s (large variations)
night run	9.5s-8.5s	21.5s-23.7s	≈ **15.5s**

Robot camera was simultaneously controlled by two Grid servers, one in France and the other in Italy. We killed the local server and the robot camera continued to execute its panoramic scans, slower, controlled by the remote server from Italy. Then we run again the server in France, and the client application speeded-up. So we achieved a fault tolerant behavior when a server went down, avoiding the robot mission failure.

Even if the localization slow down was significant during the day, it was limited to some parts of the application and had a reasonable global impact. The whole application slow down was limited to a 2.5 factor during the day ($252s$ instead of $102s$).

First Scalability Experiment. To test the scalability of our Grid architecture, we added a new module to the navigating robot application (lightness measurement service), a new robot (a robotic arm) and 2 new nodes in the grid (2 PCs in two different areas in Metz). From a configuration point of view, we had a little trouble to extend the VPN and the Corba bus to PCs with dynamic IP addresses. Grid gateway and firewall configuration has to be updated each time an IP address changes.

About performance, we registered no slow down when our servers controlled both the robots simultaneously. Robot localization on servers from the new nodes took between 21.5s and 23.7s during the day. These nodes have low-speed connections, that is download at 512 KB/s and upload at 128 KB/s. They appeared interesting solution for redundant computations during the day, while the Italian server is a better solution during the night (see table 2). A larger Grid with several nodes improves the capability to find computing resources at any time with limited slowdown.

8 Conclusion and Perspectives

During the first steps of this project we built and experimented a secure Grid for robot control. We developed a friendly API for application programmers, a Grid semantics for high-level service developers, and low-level services supporting concurrent and redundant requests to robot devices. We obtained a comfortable and efficient environment for robotic experiments, and, finally, we showed that this Grid is easy to extend.

Currently we are working for adding new robots on different sites, and for allowing client applications to run from any Grid node. Next step is a porting of our software architecture on a more standard grid middleware, like Globus.

But our final goal remains a more generic Grid for *process control*, allowing researchers and engineers to easily share physical processes and related computing resources.

Acknowledgements

This research is partially supported by Region Lorraine and ACI-GRID ARGE research project.

Authors want to thank Hervé Frezza-Buet for low-level robot library development, Alexandru Iosup for optimized versions of the navigating robot application, and Yannick Boyé for preliminary implementation on the DIET environment.

References

1. F. Lombard J-M. Nicod M. Quinson E. Caron, F. Desprez and F. Suter. A scalable approach to network enabled servers. *8th International EuroPar Conference, volume 2400 of Lecture Notes in Computer Science*, August 2002.
2. A. De Vivo F. Sabatier and S. Vialle. Grid programming for distributed remote robot control. *13th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004). Workshop on Emerging Technologies for Next Generation GRID*, June 2004. Modena, Italy.
3. I. Foster and C. Kesselman. *N. Doraswamy and D. Harkins. Ipsec: The New Security Standard for the Inter- net, Intranets, and Virtual Private Networks*. Prentice-Hall, 1999.
4. L. Frangu and C. Chiculita. A web based remote control laboratory. *6th World Multiconference on Systemics, Cybernetics and Informatics*, July 2002. Orlando, Florida.
5. S. Matsuoka J. Dongarra C. Lee K. Seymour, H. Nakada and H. Casanova. Overview of gridrpc: A remote procedure call API for grid computing. *Grid Computing - GRID 2002, Third International Workshop Baltimore, Vol. 2536 of LNCS*, November 2002. Manish Parashar, editor, MD, USA.
6. S.H. Shen R.C. Luo, K.L. Su and K.H. Tsai. Networked intelligent robots through the internet: Issues and opportunities. *Proceedings of IEEE Special Issue on Networked Intelligent Robots Through the Internet*, 91(3), March 2003.
7. A. Siadat and S. Vialle. Robot localization, using p-similar landmarks, optimized triangulation and parallel programming. *2nd IEEE International Symposium on Signal Processing and Information Technology*, December 2002. Marrakesh, Morocco.
8. A. De Vivo and S. Vialle. Robot control from remote computers through different communication networks. *Internal Report*, January 2003.