A Grid for autonomous robot control

Fabrice Sabatier
Supélec*
sabatier_fab@metz.supelec.fr

Amelia De Vivo
Universitá di Salerno†
amedev@unisa.it

Stéphane Vialle
Supélec⋆
Stephane.Vialle@supelec.fr

Hervé Frezza-Buet
Supélec⋆
Herve.Frezza-Buet@supelec.fr

Abstract

A Grid of computing resources can be an interesting solution for remote distributed robot control. It can allow to find available machines when the usual ones are too loaded or not powerful enough, to run redundant computations on different sites for fault tolerance, to support remote robot control when partner laboratories have to work with the robotic system. We designed a Grid architecture across Internet, composed of machines from two research campus, in France and Italy, connected by a VPN, under the DIET GridRPC environment. This Grid supports distributed and remote redundant control of an autonomous robot.

Our work proves that a Grid solution is feasible for our applications. This paper introduces our Grid architecture, with low and high-level Grid services adapted to robot control, and describes a specific GridRPC library we designed for improving the job of robotic application developers.

1 Motivations and Objectives

In the past few years, many researches have been using the Internet as a medium for remote robot control and studies exist about the effects of factors like uncertain time delay, data loss and security problems [6, 5]. Anyway, when we talk about autonomous robots, the problem is more complex. An autonomous robot does not simply executes commands. It has to be able to decide about a lot of questions, and generally needs more processing power than available on its onboard processor. With this kind of robots it can make sense to distribute computation among different machines, including some in remote partner laboratories. Here are some typical situations needing distributed and remote computing:

- Some robotic applications are computationally very heavy and need a powerful and expensive machine. Since this is not a common case, it is not convenient to devote a parallel machine or a cluster to the robotic system. Just when needed, we look for a powerful machine on our LAN, or in remote partner laboratories.
- Some robotic applications are composed of embarrassingly parallel modules working on the same input data. The whole processing can slow down the robot, so we can split it in different tasks and distribute them on different machines.
- A solution based on a single machine is not fault tolerant. For being sure the robot mission will not fail, we can launch multiple copies of the application on different machines. When a failure happens on a fast one, the robot mission will slow down but will continue, controlled by a slower one.
- When using non devoted resources some variable slow down can appear depending on the load status. Dynamically switching between different machines can be a solution to avoid overloading.
- We can share our robotic system with our partners, but we cannot give them all our computational resources. Then, they must run some of their robotic applications on their own machines, controlling our robots from their laboratories.

Various kinds of systems can be built on classical remote client-server mechanisms across encrypted tunnels (like *ssh* links) on the Internet. But currently Grids of computing resources are emerging and could be an interesting solution for our purpose. Grids aim to become comfortable distributed environments, hiding heterogeneousness and complexity of distributed systems [2, 4] and our collaboration with remote partners seems well suitable to Grid philosophy. Anyway our applications have three issues that can give trouble in a Grid environment. First, they must interact frequently with

*Supélec, 2 rue Edouard Belin, 57070 Metz, France
†Universitá di Salerno, Dipartimento di Informatica e Applicazioni "R. Capocelli", Via S. Allende, 84081 Baronissi (SA), Italy

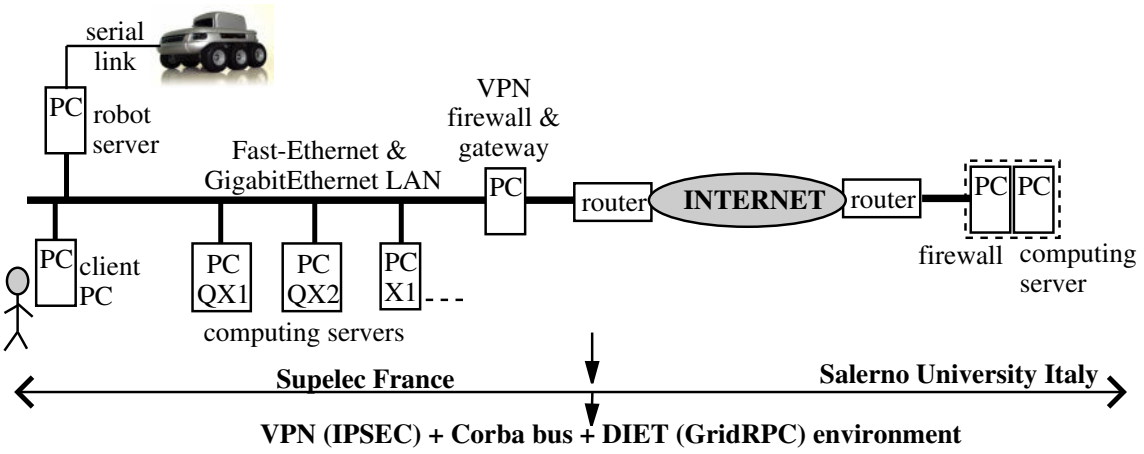

Figure 1: Overview of the Grid testbed.

robot sensors and motors, not just running in memory with limited IO. Second, they have to run when robot is needed, not depending on the load status of the Grid. Third, they have to take synchronization into account because robots are compound of several organs moving and acting in parallel.

In this paper we investigate interest and difficulties when running robotic applications on a Grid of computing resources. Our first objective is to evaluate if it is really possible to use a Grid to improve distributed and remote control of autonomous robots. The second is to develop a Grid architecture with adapted low-level robot control modules, Grid middleware and high-level Grid services for our purposes. Finally, our last objective is to design an easy-to-use API for making robotic researchers overcome the difficulties of Grid programming.

2 Overview of the Grid testbed

Our current testbed is composed of a robot, a set of mono- and multiprocessor PCs in Supelec (Metz, France), a monoprocessor PC in Salerno University (Italy), a VPN joining all these machines and the DIET Grid environment. See figure 1.

Robot features

We used a navigating *Koala* robot by the K-Team society. It has onboard controllers for driving its different organs, and is controlled by an external devoted PC, connected through a serial link. This PC runs the *Koala Server* and all robotic applications are clients of it. As discussed further, this server has been modified to be adapted to Grid computing.

Robotic application under test

This application is about robots navigating in indoor dynamic environments, where they cannot use only predetermined maps. Some unexpected obstacles can appear. We assume that artificial landmarks are installed at known coordinates. When switched on, the robot makes a panoramic scan (moving its camera turret), detects landmarks, computes an optimized triangulation, and self-localizes [8]. Then it can compute a theoretical trajectory to circulate. Self-localizations are computed at intermediate positions for compensating errors during long trajectories and when unexpected obstacles are detected. Image transmission from the *Koala Server* to its clients has been optimized using JPEG compression.

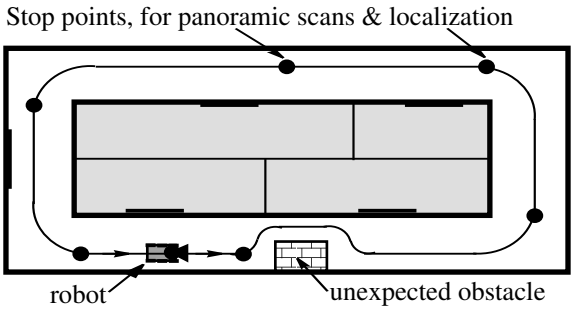

Figure 2: Autonomous robot following long trajectories.

VPN configuration

An IPSEC-based VPN [3] has been installed to support the Corba bus required by the DIET Grid environment. IPSEC implements both AH (Authentication Header)

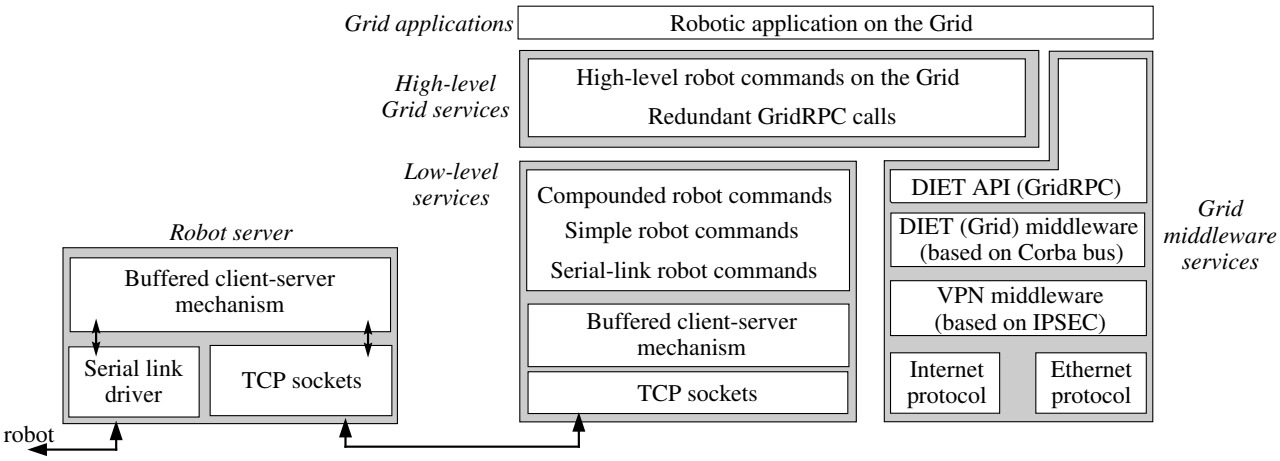

Figure 3: Grid software layers of the Grid for robot control.

and ESP (Encapsulating Security Payload) and IPSEC packets can cross intermediate Internet router as they were simple IP packets. IPSEC only requires 500/udp port to be opened and protocols 50 (ESP) and 51 (AH) to be authorized on the destination gateway, so that also the most restrictive security policies are compatible with the connection among Grid components.

Grid environment

The Grid environment we use is DIET [1] (Distributed Interactive Engineering Toolbox). It can be considered a Grid Problem Solving Environment, based on a Corba bus and a Client/Agent/Server scheme. In this scheme a Client is an application that submits a problem to the Grid through an agent hierarchy. Agents have a list of the Servers running in the Grid and choose the best one for the client request. Generally an Agent makes its choices according to the output of some performance forecasting software, that monitors the Grid resources and gives information about workload, bandwidth, etc. Finally, DIET supports both the synchronous and asynchronous Grid-RPC calls [7].

Experimented Grid services

Two pilot modules of the robotic application were redesigned for the Grid environment: localization (based on panoramic-scan) and navigation. They are very frequently used modules, allowing to build many basic robotic applications. So, they are the first high-level Grid services we have implemented.

3 Grid software architecture

Figure 3 shows the software layers of our Grid architecture. At the top layer, the *Grid application* is almost like a classical application. It can give the robot high-level commands, requiring the Grid services ”Localization”, ”Navigation” and ”Lightness”. These services can be concurrent, but the user has to explicitly coordinate robot devices.

High-level Grid services are implemented through GridRPC calls, requesting low-level services. A synchronous call is for a single robot function, an asynchronous call is for a robot function running in parallel to others, several concurrent asynchronous calls are for redundant computations. When a high-level service runs a redundant computation, it waits only for the first GridRPC call to finish, ignoring or cancelling the others. All these Grid programming details are hidden to the user that can focus on robotic problems.

The low-level robot services are organized like a three-layers command stack, on top of a *buffered client-server mechanism*. This mechanism allows concurrent accesses to the *Koala server* through *TCP sockets* and to the *Koala robot* through a *serial link driver* running on the Koala server machine (see figure 3). The *Koala Server* is a multithreaded and buffered server. It supports concurrent requests to different services, to simultaneously control different robot devices. It also accepts concurrent requests to the same service, allowing redundant computations for fault tolerance. In any case the robot accomplishes each needed action just once, buffering all previous commands and results. Low-level robot services are called from high-level Grid services distributed across the Grid. A high-level Grid service can be called by another one or by application pro-

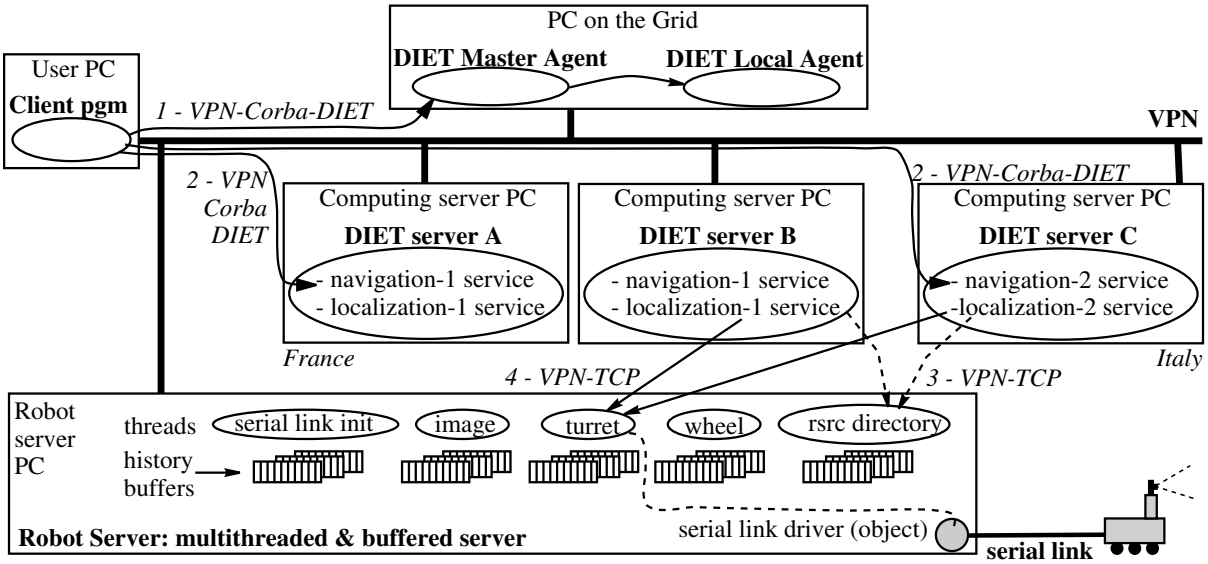

Figure 4: Deployment of the Grid architecture.

cesses, through the Grid middleware.

The Grid middleware consists of the DIET Grid environment [1] on an IPSEC-based VPN [3]. Practically a user program (client application) makes GridRPC calls to high-level Grid services that, in turn, make other GridRPC calls (see figure 4). They contact the DIET agents, running somewhere on the Grid, to know the addresses of the most suitable computing servers. Then the user program contacts directly these servers to get Grid services. This communication happens on the VPN using the DIET protocol on a Corba bus.

Then, each computing server establishes a direct communication with the *Koala Server*. This communication happens on the VPN again, but bypasses the Corba-based DIET protocol and goes directly on TCP sockets. This way the robot server can send large camera images to the requesting Grid servers avoiding Corba slow down. After processing, Grid servers use the Corba bus just for returning small results (such as a computed localization) to the client machine.

4 Grid oriented robot server

Specific needs for robot control

From a computer science point of view, a robot is a set of independent devices (wheel, camera turret, camera, infra red sensors, ...) that usually work in parallel and need synchronization. So, each robot organ is considered as a resource that can be accessed through a service.

The robot server groups all elementary services related to the different robot organs. It is composed of several sub-servers, one per robot organ (see figure 4), attached to different ports. Each sub-server is a multithreaded one, in order to be able to serve several clients in parallel, and can lock its resource when needed. For example, several clients (actual Grid computing servers) can connect to the camera in parallel to make different image acquisitions and image processing during the robot motion, but only one client can command a motor at a time.

Adding a resource directory service

Service port numbers depend on the configuration of the robot server PC. High-level Grid services on Grid computing servers must connect to different ports of the robot PC server for requiring various services.

In order to improve the independence between the robot server and the Grid computing servers (the actual clients of the robot server), we included a *resource directory service* in the robot server (see figure 4). Clients have to know only the port number of this directory service, that sends them the directory of all available resources and associated service ports. Moreover, updating the robot server (i.e. updating a low-level Grid service) does not need for changes in high level Grid services.

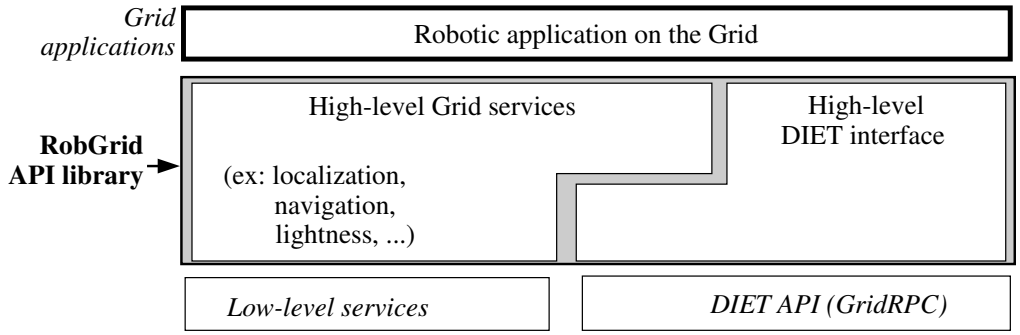

Figure 5: Details of the high-level Grid architecture.

Using history buffers

When redundant computations are run, the robot has to perform each action only once but has to return the same subsequent sensor values to all redundant clients.

To satisfy these constraints we introduced history buffers in the robot server: one buffer per resource and per algorithm controlling the resource. When a Grid computing server connects to the robot server and asks for a service, it sends an execution identification number and a set of serial link commands to be transmitted to the robot by the robot server (more precisely by the subserver attached to the resource). Redundant Grid computing servers will run the same algorithm and will send the same execution identification number. When receiving a new low level request, the robot server transmits it to the robot and stores the result in the buffer associated to this execution. When receiving again the same request from another Grid computing server which is in late, it just returns to this server the result stored in the buffer.

Buffers have a limited size, and slow Grid computing 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. Moreover, if the network load changes during the execution, the Grid computing server sending the new next request may change too. In this case the robot server goes ahead, driven by the new fastest Grid computing server. This mechanism leads to improve fault tolerance and load balancing.

5 High-level Grid interface

5.1 Grid service mechanism

Figure 5 shows how the higher layer of our Grid architecture encapsulates complex low-level details.

As a programming interface, we designed a high-level library. It is based on GridRPC functions, and it makes easy to call high-level Grid services. For example, a simple API function can concurrently request several instances of a high-level Grid service and wait for the first that finishes. The programmer does not need to deal with complex synchronization mechanisms.

An example of application program is illustrated on figure 6: the robot gets images and measures lightness during its navigation toward a point, where then it self-localizes.

Our API syntax is C++-like. First, we create a session handle to manage communication between the application and the requested Grid services. Then, we create a client object for each Grid service we need. The constructor parameter allows to specify the number of concurrent instances for the required service. In this example we use two instances of critical services (navigation and localization). Each client object totally encapsulates the DIET mechanism to communicate with its corresponding Grid service. The next step is the DIET session start, that starts communication between the application and the Grid services.

Each client object requests a connection between the required Grid service and the *Koala server*. This establishes a TCP communication with the robot server, initializes the related robot devices and resets the associated buffers. Different Grid services can be simultaneously connected to the *Koala server*.

Each service can be called through its connected client in a synchronous (`light->Call()`) or asynchronous way (`nav->AsyncCall()`). All redundant

```
// Create a session handle to control the
// communications between the application
// and the Grid services.
Session *session = new Session();

// High level Grid services allocation
// (pointing out the number of redundant
// calls to run). Initialization of the
// DIET communication mechanisms.
LocClient *loc = new LocClient(2)
NavClient *nav = new NavClient(2);
LightClient *light = new LightClient(1);

// Start the communication between the
// client application and the services
// specified previously.
session->Start();

// TCP conection between the GRID services
// and the robot server, initializion of
// robot server resources and buffers.
loc->Connect();
nav->Connect();
lihght->Connect();
```

```
// Move the robot and measure the
// lightness in parallel
nav->AsyncCall(x,y,theta);
while (!nav->Probe()) {
  light->Call();
  ...
}
...

// Localization based on panoramic scan
// and landmark detection
loc->Call();
Res = loc->GetResult();
...

// Disconnect Grid services from robot
// server, reset some robot server buffers
delete loc;
delete nav;
delete light;

// Close the communication session between
// the application and the Grid services
delete session;
```

Figure 6: Source code of a Grid application example

Grid services are called concurrently. An API function allows to check if an asynchronous operation has finished (`nav->Probe()`). Redundant calls are considered finished as soon as one of the Grid service instances finishes. The winner Grid service is identified, while the others are cancelled or ignored. The client of a redundant service gets results from the winner Grid service (`loc->GetResult()`).

Finally, client objects and DIET session are deleted: each Grid service is contacted to flush the corresponding event buffers on the *Koala server* and disconnection takes place.

5.2 High-level Grid service programming

End users write robot applications calling high-level Grid services, but sometimes they need to write new high-level Grid services. For example, to implement new image processing, to call new low-level robot services after a robotic system upgrade, or to compose other Grid services. Our programming interface offers a generic skeleton, so that it is easy to develop a new high-level service.

Each high-level Grid service is a set of sub-services:

- Connection to the related *Koala server* service. This is based on a low-level routine, identified by a unique number, that sends command to a robot device and gets related output. Such a service can be shared with other redundant high-level service instances (same low-level routine identification number and same target resource).
- Disconnection from the *Koala server*
- Buffer reset. Depending on the low-level service, this operation can be done before running a robot control operation, or after the last redundant call has finished.
- Execution of the related robotic operation, like *navigation(x,y,theta)*. It consists of a set of basic robot control commands (like *move_straightforward(x,y)*). Each basic command is sent to the *Koala server* across TCP sockets as a set of very low-level robot commands (serial link commands), to be relayed by the serial link driver and finally executed on the robot.

Sub-services for connection/disconnection to/from the robot server are automatically called from constructors and destructors of high-level client objects. Other sub-services have to be called explicitly from the user application through high-level client objects.

5.3 Development modularity

To improve our application we decided to make a lightness measure during long robot trajectories.

For this purpose we implemented a new module as a new high-level Grid service. It moves the robot camera to a pointed out position, catches an image and measures the average lightness. This high-level service requires the *Koala server* for camera motor control and image acquisition.

All development and tests were done in less than two days. The new service was immediately fully compatible with the rest of the Grid environment.

6 Performance measurement

Most of our Grid performance evaluations are based on the execution time of the localization module. It is composed of a panoramic scan of the camera by its motorized turret, images acquisition with related JPEG compression and transmission, landmark detection and an optimized triangulation. The localization module uses multithreading to overlap image transmission and image processing.

6.1 Comparison to classical execution

When the Supelec LAN is unloaded, running the whole localization module on the client PC without the Grid environment leads to an execution time close to 8.5*s*. Under the same conditions, but using the Grid environment without redundant computations leads to approximately the same execution time. So, the Grid has a negligible overhead on robot localization when running on the LAN of the robot.

6.2 Grid performances across Internet

In order to have benchmarks across Internet during 24h, with respect to the mechanical organs of the robot that must not run too long, we run three localizations each 15 minutes. The client application was calling just one localization service on the Grid (no redundant calls). Figure 7 shows the execution time, function of the date of the execution. Running the service at Supelec, on the same LAN than the robot, leads to 8.5*s* for a complete localization when Supelec LAN is unloaded and to 10.5 when it is loaded. Execution time was not function of the selected server because most of time is spent in camera move and image get.

On the PC of Salerno University the execution time increased to 15.5*s* from 8pm to 9am (during the night), due to Internet communication overhead, but appeared very stable. So, when our robot has to evolve out of the classic working period, our Grid architecture across Internet allows to easily make remote control with a slow down less than 2, and with standard environment (no real time OS, no devoted communication links, ...). During the day the execution time changes a lot and is much greater: slow down can reach 10, but the remote control still succeed.

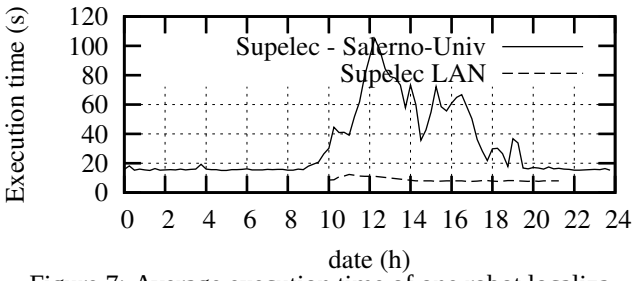

Figure 7: Average execution time of one robot localization, controlled across the Grid from Salerno University and from Supelec LAN, function of the execution date.

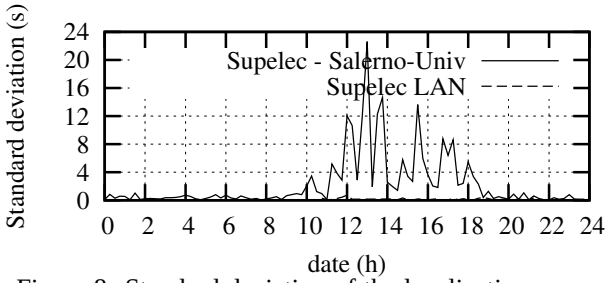

Figure 8: Standard deviation of the localization execution times introduced on previous figure.

6.3 Performance of local redundant computation

When the Supelec LAN was loaded, running redundant computations just on the Supelec part of the Grid showed an execution time again around 8.5*s*. In fact we observed that the fastest service to compute the localization changed from one localization call to the next, and sometimes from one localization step to another.

Using simultaneously several servers allows to run each step at the speed of the fastest service (depending on the load on the servers and on the network). So, redundant computing appeared well supported by our Grid, and useful to limit slow down for our applications that exhibit some kind of time constraints.

7 Fault tolerance achievement

In order to test our Grid programming environment, we developed a long application using redundant calls to localization and navigation services. Robot moves from one position to another, making a panoramic scan and a

self-localization at each point. This first test exhibited fault tolerance across Internet.

The robot camera was simultaneously controlled by two DIET servers (one in France and the other in Italy). Because of Internet communication overhead, the local DIET server (in France) was always the first to control the robot, while the other one just got the previous results. We killed the local DIET server and the robot camera continued to execute its panoramic scans, slower, controlled by the remote DIET server in Italy. When we re-run the DIET server on the local part of the Grid, the client application successfully called it again, and the robot camera control automatically speeded-up.

We measured the whole application slow down in the fault tolerance test during the day. Localization was simultaneously run at Metz and Salerno. Then we stopped and further restarted it at Metz. Even if the localization module slow down was significant, it has a reasonable impact on the whole application. This is because the application bottleneck is the navigation module. It takes longer than localization because of robot mechanical constraint. Finally, the whole application slow down was limited to a 2.5 factor during the day (252s instead of 102s).

8 Conclusion and perspectives

We designed and built a small but operational Grid architecture for autonomous robot control across the Internet, leading to an easy use of remote servers for extra-CPU needs and redundant computing. This Grid architecture is a beginning solution to the uncomfortable situations described in the introduction section.

Moreover, the Grid API that we have designed and implemented allows to decrease the development times of new Grid services and new applications relative to robot control. But we aim to develop a more generic Grid environment for robot control, with many high-level Grid services, easy to interface with different robot servers.

Now, we are planning to increase the Grid size, adding new services and new sites, for checking if our system scales. Our VPN and Grid should extend soon to other Italian, French and Rumanian laboratories. With a larger Grid, covering more sites in Europe, we hope to obtain fault tolerance with limited slow down at any date.

Acknowledgements

Authors want to thank Region Lorraine and ACI-GRID that have supported a part of this research.

References

[1] E. Caron, F. Desprez, F. Lombard, J-M. Nicod, M. Quinson, 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] I. Foster and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann publisher, 1998.

[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] I. Foster, C. Kesselman, and S. Tuecke J. Nick. Grid services for distributed system integration. *Computer, 35(6)*, 2002.

[5] L. Frangu and C. Chiculita. A web based remote control laboratory. *6th World Multiconference on Systemics, Cybernetics and Informatics*, July 2002. Orlando, Florida.

[6] R.C. Luo, K.L. Su, S.H. Shen, 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] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, 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.

[8] 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.