Grid Programming for Distributed Remote Robot Control

Fabrice Sabatier
SUPELEC
2 rue Edouard Belin
57070 Metz, France
sabatier_fab@metz.supelec.fr

Amelia De Vivo
Universitá di Salerno
Via S. Allende
84081 Baronissi (SA), Italy
amedev@unisa.it

Stéphane Vialle
SUPELEC
2 rue Edouard Belin
57070 Metz, France
Stephane.Vialle@supelec.fr

Abstract

A computational Grid can be an interesting solution for distributed remote robot control. It can provide computational resources when the usual ones are too loaded or not powerful enough. It can support fault tolerance allowing redundant computations. It can make possible the robotic system sharing with remote partners.

We designed a Grid architecture across Internet, including resources from two laboratories, one in Italy and one in France. It is based on the DIET GridRPC environment and supports distributed remote redundant control of an autonomous robot.

This paper focuses on high-level Grid services and on a specific GridRPC library we designed for improving the robotic application development.

1. Introduction

Remote control of autonomous robots is more complex than just sending commands to a robot. 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 robot it can make sense to distribute computation among local and remote machines. In the following we illustrate typical situations needing distributed remote computing:

- Some, but not the most, robotic applications are very computation demanding and need powerful expensive computers. It is not convenient to devote a parallel machine or a cluster to the robotic system. Just when needed, we can look for a suitable machine on our LAN or in a remote laboratory.
- Some robotic applications consist 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.
- Using a single machine is not fault tolerant. For being sure the robot mission will not fail, we can run multiple copies of the application on different machines. If a failure happens on the fastest one, the robot slows down but its mission can continue.
- We do not use devoted resources, so performance depends on the load status. Dynamically switching between different machines can be a solution for avoiding degradation.
- We can share our robotic system with remote partners, but we cannot give them our computational resources. Then, they must run their robotic applications on their own machines, controlling our robots from their laboratories.

Although studies exist about the effects of factors like uncertain time delay, data loss and security problems [6, 5] about robot control through the Internet, our applications have further issues that can give trouble. They must interact frequently with robot sensors and motors and must deal with synchronization because each robot has several devices working in parallel.

Anyway our previous work showed that we can use remote distributed control with a reasonable delay. Computational Grids could be an interesting solution for our purpose because they can hide heterogeneity and complexity of distributed systems [2, 4]. Collaboration with remote partners is well suitable to Grid philosophy and our first tests with Grid and robots achieved encouraging results.

Moreover complex robotic applications can be built using basic modules. Some of them are frequently used and it can be very comfortable to have a repository of such optimized building blocks. We re-designed some basic modules of a robotic application, turning them into Grid services to be used by application developers. Finally we designed

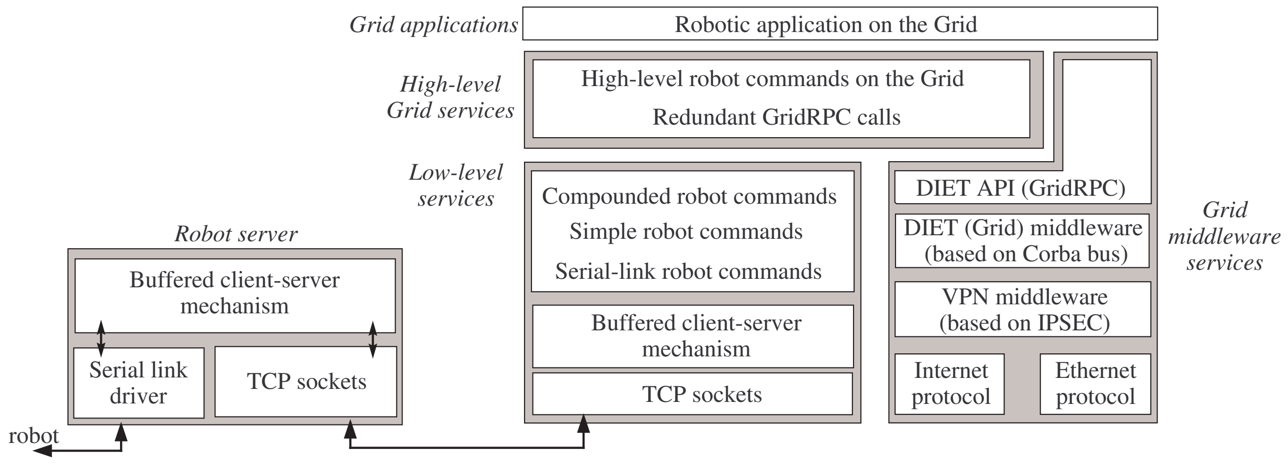
Grid applications
Robotic application on the Grid
High-level Grid services
High-level robot commands on the Grid
Redundant GridRPC calls
Low-level services
Compounded robot commands
Simple robot commands
Serial-link robot commands
Buffered client-server mechanism
TCP sockets
DIET API (GridRPC)
DIET (Grid) middleware (based on Corba bus)
VPN middleware (based on IPSEC)
Internet protocol
Ethernet protocol
Grid middleware services
Robot server
Buffered client-server mechanism
Serial link driver
TCP sockets
robot

Figure 1. Grid software layers.

an easy-to-use API for avoiding robotic researchers to deal with raw Grid programming.

2. Grid overview

The physical resources deployed on our Grid are a robot, some PCs in France (Supelec) and a PC in Italy (Salerno University). Our robot, an autonomous navigating *Koala*, has several onboard controllers driving its different devices. It is connected to an external devoted server through a serial link. This is a simple PC controlling basic robot behaviours. A robotic application is always a client of this server. An IPSEC-based VPN links all machines and supports the DIET Grid environment. On top of DIET we built our Grid architecture and services and an API library for robotic application developers.

2.1. Testbed application

The testbed application consists of three very frequently used basic modules: self-localization, navigation and lightness detection. Our robot navigates in dynamic environments, where no complete pre-determined map can be used. Anyway artificial landmarks are installed at known coordinates. When switched on, the robot makes a panoramic scan with its camera, detects landmarks and self-localizes [8]. 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 for average lightness computation. The *Koala Server* always sends its clients JPEG-compressed images. The three pilot modules were redesigned and turned into Grid services for robotic application development.

2.2. DIET environment

For our research we chose the DIET [1] (Distributed Interactive Engineering Toolbox) Grid environment. It supports both synchronous and asynchronous Grid-RPC calls [7], 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 list of the Grid Servers and choose the most suitable for the Client request, according to some performance forecasting software. Generally this kind of software monitors Grid resources and gives information about workload, bandwidth, etc.

DIET communication between Grid components is Corba-based, but our institute security policies do not allow this kind of communication. In order not to relax the security level of our respective organizations, we created an IPSEC-based VPN [3] linking our machines. This just requires the 500/udp port opened on the destination gateway.

2.3. Grid architecture

Figure 1 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 syn-

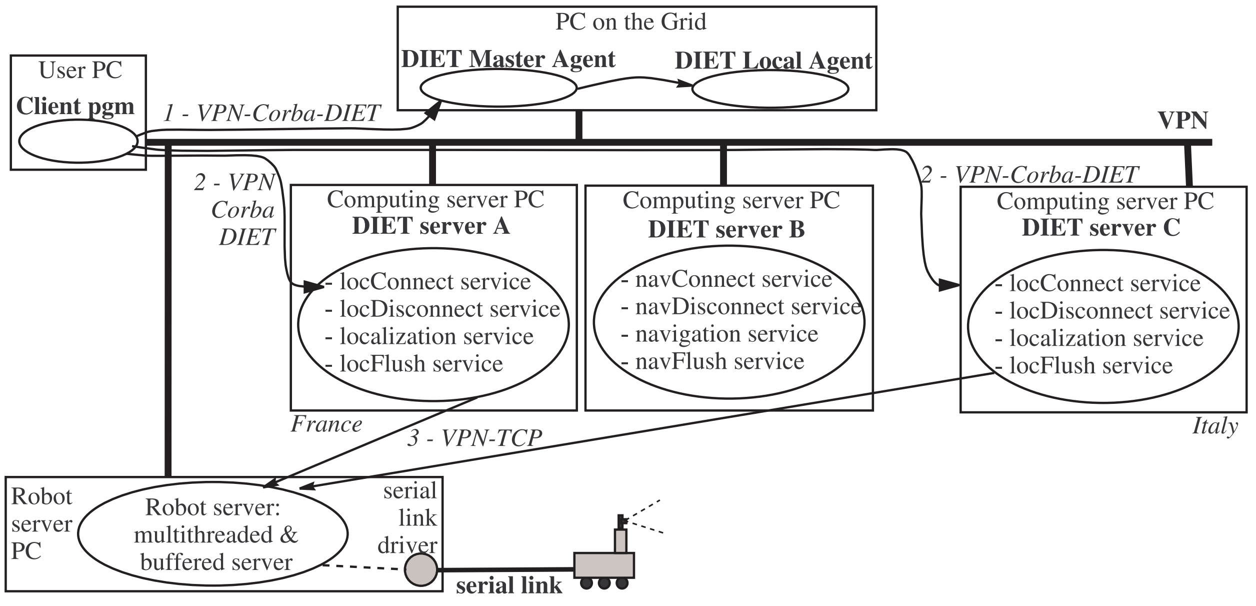

Figure 2. Deployment of the Grid architecture.

chronous 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 1). 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 processes, 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 2). 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.

3. High-level Grid interface

3.1. Grid service mechanism

Figure 3 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 fig-

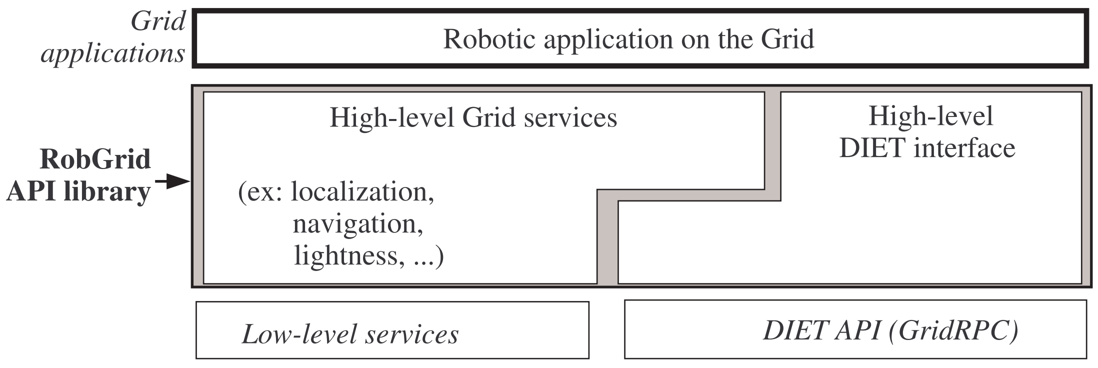
Grid applications
Robotic application on the Grid
RobGrid API library
High-level Grid services
(ex: localization, navigation, lightness, ...)
High-level DIET interface
Low-level services
DIET API (GridRPC)

Figure 3. Details of the high-level Grid architecture.

ure 4: 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 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.

3.2. 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.

```
// 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 4. Source code of a Grid application example

4. Development test

4.1. 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.

4.2. 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.

5. Performance measurement

We run several tests on our Grid, to check if it could efficiently support robot control:

- Localization time on the local Grid (same LAN as the robot) was the same as on a devoted no Grid PC (approximately $8.5s$). Grid software overhead has no impact.
- When the local LAN was loaded, local redundant computation showed again an execution time around $8.5s$. Simultaneous usage of several servers allows to run each step at the speed of the fastest one (depending on servers and network load).
- Measured on 24 hours, when the localization service run in Italy, its average execution time elapsed from $15.5s$ during the night up to $100s$ during the day. In

both cases remote localization succeeded and during the night the slow down was limited to a 2 factor.

- 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 $102s$).

These tests show that robot control across our France-Italy Grid can lead to successful fault tolerance thanks to redundant computations. Moreover the measured slow down is quite limited and not too impacting on our application performance.

6. Future perspectives

Our Grid API is currently a work in progress. We aim to develop a more generic Grid environment for robot control, with many high-level Grid services, easy to interface with different robots and robot servers.

Moreover we are going to extend our Grid deployment to other French, Italian and Rumanian laboratories. Our goal is to test if our Grid architecture scales when using a larger Grid spread.

7. 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. T. 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. Luo, K. Su, S. Shen, and K. 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.