INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GRAS: a Research & Development framework for Grid services

Martin Quinson

N° 5789

Janvier 2006

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--5789--FR+ENG

INRIA LORRAINE

GRAS: a Research & Development framework for Grid services

Martin Quinson

Thème NUM — Systèmes numériques
Projet Algorille

Rapport de recherche n° 5789 — Janvier 2006 — 24 pages

Abstract: Grid platforms federate large numbers of resources across several organizations. While their promises are great, these platforms have proven challenging to use because of inherent heterogeneity and dynamic characteristics. Therefore, grid application development is possible only if robust distributed services infrastructures, *e.g.* for resource and data discovery, resource monitoring or application deployment, are available. These infrastructures, which are large-scaled distributed loosely-coupled applications, are very difficult to design, develop and tune.This paper presents the *Grid Reality And Simulation* (GRAS) framework that allows grid developers to first implement and experiment with such an infrastructure in simulation, benefiting from a controlled and fast environment. The infrastructure can then be deployed *in situ* without code modification.

We first detail the design goals and the implementation of GRAS, and contrast them to the state of the art. We then present a case study to highlight the fundamentals of GRAS and illustrate its ease-of-use. In addition, we quantify the complexity of a code example using either GRAS or several other communication solutions. We also conduct tests over LAN and WAN networks to assess the performance. We find that the code using GRAS is simpler and shorter than any other solution while achieving better performance than most of the other solutions.

Key-words: Grid computing, Grid services development, development tools.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

GRAS : un environnement de recherche et développement pous les services sur la grille

Résumé :

Les plates-formes de grilles fédèrent de nombreuses ressources mises à disposition par des organisations différentes. Bien que le potentiel de ces solutions soit important, ces plates-formes se sont révélées difficiles à utiliser à cause de leurs caractéristiques hautement hétérogènes et dynamiques. Le développement d'applications pour la grille est impossible sans de solides infrastructures pour la découverte des ressources et des données disponibles, pour le monitoring des ressources ou encore pour le déploiement des applications.

Ces infrastructures (qui sont des applications distribuées à large échelle et faiblement couplées) sont elles-mêmes très difficiles à concevoir, à développer et à optimiser. De plus, ces infrastructures posent la plupart du temps des difficultés algorithmiques complexes.

Cet article présente l'environnement GRAS (*Grid Reality And Simulation*). Il permet aux développeurs d'implémenter et d'expérimenter de telles infrastructures sur simulateur dans un premier temps afin de leur offrir un environnement controllé et rapide. Les infrastructures peuvent ensuite être déployées *in situ* sans modification de code.

Nous détaillons tout d'abord les objectifs et l'implémentation de GRAS avant de comparer ce projet à l'état de l'art du domaine. Nous présentons ensuite un exemple de programme utilisant cet environnement pour en illustrer la simplicité d'usage. Par ailleurs, nous quantifions la complexité d'un autre programme selon qu'il est implémenté avec GRAS ou d'autres solutions de communications. Enfin, nous présentons une campagne d'expériences visant à mesurer l'efficacité des applications ainsi obtenues. Nous montrons que GRAS simplifie l'écriture des services distribués tout en garantissant de meilleures performances que la plupart des autres solutions existantes.

Mots-clés : Calcul distribué á grande échelle, Développement de services distribués, Outils de développement.

1 Introduction

Grid computing consists in federating heterogeneous and distributed computing resources in order to aggregate their computational, communication and storage capacities. A platform resulting of the sharing of local resources between several organizations is called a grid [11]. Such platforms are very promising but are very challenging to use because of their intrinsic heterogeneity in terms of hardware capacities, software environment and even system administrator orientations. That is why any application developers has to rely on distributed services infrastructures such as the Globus MDS and GIS [12] for resource and data discovery, the Network Weather Service (NWS, [26]) for resource monitoring, NetSolve [2] or DIET [6] for application deployment. These infrastructures, which are large-scale distributed loosely-coupled applications, are in turn challenging to develop and to tune. Furthermore, the underlying distributed algorithms are generally extremely complex and difficult to study.

Most of the currently deployed infrastructures rely on lightweight communication libraries. For instance, the NWS uses a specific portability and communication library perfectly fitted to its use; the first NetSolve versions used raw sockets and a basic application protocol; many projects of the former AppLeS group (like the *Effective Network View* [21] or *A Parameter Sweep Tool* [8]) rely on the *AppLeS Multi-Protocol Interprocess Communication* (AMPIC[1]). These facilities are application-specific, and hence likely to be difficult to reuse in a more general context. The key problem here is that none of the classical standards of parallel computing is suited to the development of distributed service infrastructures.

Another difficulty posed by grid platforms is their dynamic characteristics that prevent reliable reproduction experiments and hinder faithful algorithm comparisons. As a result, developers typically spend inordinate amount of time and energy to establish stable development and evaluation environments. A solution to alleviate these problems is to use simulation. However, the resulting implementations are typically confined to proof-of-concept prototypes. Such implementations would need a complete rewrite before being useable *in situ*.

[1] URL: http://grail.sdsc.edu/projects/ampic/

This paper introduces the *Grid Reality And Simulation* (GRAS) framework, which aims at easing the development of distributed event-oriented applications. It constitutes at the same time a convenient development framework and an efficient grid runtime environment, allowing the *same unmodified code* to run both on top of a simulator and on real distributed platforms (using two specific implementations of its API). This solution combines the better of both worlds: developers benefit from the ease-of-use and control of the simulator during most stages of development cycle while seemlessly producing efficient real-life-enabled code. While GRAS does not pretend to address all issues pertaining to grid computing, we believe that its combined simulation/*in situ* approach is the key to the rapid and easy development of effective grid infrastructures.

The remainder of this article is organized as follows: §2 details the goals and the approach of GRAS. §3 compares our work to the state of the art in the field. §4 presents a case-study to illustrate the use of GRAS. §5 provides some experimental results, both in term of communication performance and code complexity. §6 concludes the paper.

2 The GRAS project

2.1 Targeted Applications

GRAS is designed to build well-tested distributed infrastructures offering a specific service to grid applications and middlewares. For instance, one could use GRAS to build grid computational servers comparable to NETSOLVE [2] and DIET [6], or platform monitoring sensors like the NWS ones [26]. Such infrastructures are constituted of several entities dispatched on the various hosts of the platform and collaborating with each other using some specific application-level protocol. The primarily targeted application class is thus the class of loosely coupled collections of communicating processes using an application-level protocol.

Such applications are more easily described using an *event-driven* model than with a *SPMD* model [23]. The GRAS framework relies on the *active message* paradigm and a high level message passing interface. After an initialization phase where message types are declared and callback functions attached

to the arrival of given message types, the GRAS typical main loop polls for incoming messages and dispatches them to the corresponding callbacks. It is still possible to explicitly wait for a given message (for instance an answer to a previous message). Any message arriving in the meantime is queued for further use.

We now detail our design goals and their implications on the framework implementation.

2.2 Development Framework for Distributed Applications

It is well known that the development of distributed applications is much more difficult than for centralized and sequential applications. Concurrent algorithms introduce specific difficulties like race conditions or deadlocks. Moreover, usual development techniques do not apply because the application is split in several entities interchanging messages. This process multiplication makes it very difficult to conduct step-by-step execution in a debugger to understand why the program does not behave as expected.

This situation is even more complicated in a typical grid setting for two reasons. First, the processes are usually distributed over several sites introducing different hardware and administrative orientations. Thus, the programing environment is likely to be different on each site, and getting each process compiled on each site taking the library location and compiler settings into account can become difficult. The second main issue comes from the scale of typical grid platforms, which can range from a dozen of hosts to several hundreds (or even more). Naturally, increasing the number of hosts dramatically increases the probability that at least one host is out of order at a given time. The resulting technical difficulties often distract the developers from the algorithmic challenges posed by grid applications, making the development even more challenging.

Under such conditions, a classical answer is to develop and tune the applications on one single host before deploying it at large on real platforms. This approach may induce a limitation on the number of processes co-located on the same host due to application specific constraints. For instance, the NWS sensors test the available bandwidth using active measurements. It is thus difficult to place more than one such sensor per machine.

Another challenge is then to reproduce the platform heterogeneity. To assess for instance how the application behaves when a node is placed on a node slower than the other ones, it is possible to use classical UNIX process priority mechanism with `nice`(1). This only works for processing capacity heterogeneity, and simulating communication heterogeneity implies to use emulation solutions such as MicroGrid [27] and ModelNet [24].

Both projects aim at emulating a grid or an internet-like environment while enabling repeatable results. They allow to run applications on a virtual platform by trapping every relevant library call (socket-related calls, `gethostname`, *etc.*) and mediating them. In MicroGrid for instance, the computing resources are simulated using a local scheduler, which allocates CPU time slices to each process according to a predetermined simulation rate. The network is simulated through the DaSSF simulator [16] which mediates all communications. In ModelNet, communications are not simulated but emulated. They are sent to a router core constituted of specific nodes running a modified version of FreeBSD to emulate the behavior of a configured target network: it offers the same rates, delays and losses as the target network.

We expect GRAS users to run their applications quite frequently during the debugging cycle, and therefore need a very fast evaluation scheme. Since they precisely simulate the whole architecture, the emulation solutions presented above, or complete simulators such as SimOS [14] (for complete machine simulation, down to instruction level) or NS [4] (for complete network simulation, down to packet movement) do clearly not fit our needs.

Instead, we decided to base our work on the SIMGRID [7] environment. This toolkit provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments. It is built upon an efficient trace-based discrete event simulation kernel. Thanks to the rather simple models used, SIMGRID allows to simulate up to some thousand processes on a single workstation and is several orders of magnitudes faster than complete simulators or emulators.

Another interesting feature of SIMGRID in our context is that all the *processes* constituting the application are changed to threads of a single process during the simulation, and that they are run one after the other in an exclusive manner. This can be seen as a problem limiting the simulation performance,

but this is an advantage in our context since it greatly eases the application debugging by allowing the step-by-step execution approach.

To allow the same unmodified code to run both on top of a real platform and in a simulation mode, GRAS needs to virtualize the operating system and its system calls. For instance, `time` calls should return the current *simulated* time rather than the current real time on the machine running the simulation, which is meaningless within the simulation.

In this view, GRAS defines explicit wrappers to system calls such as `time` and `sleep`. These wrappers also act as portability layer. Indeed, the system calls' prototypes and semantics may change from one operating system to another, forcing the user code to take these variations into account to remain portable. The GRAS system call wrappers thus provide a consistent and portable interface masking the specificity of each OS to the programmer.

Another difficulty is to report the computation sequence durations into the simulated reality. For instance, when the user code executes a computation lasting W Mflop, the corresponding simulated process has to be blocked for W/ρ virtual seconds if the virtual host on which it runs delivers a constant throughput of ρ Mflop/s. GRAS thus provides a way for the user to specify whether the duration of a given sequence of code should be reported in the simulated world or not. W is either provided by the user or automatically benchmarked.

This mechanism can be used to completely virtualize some code sections in the simulation. For instance in an application achieving a distributed matrix product, the simulator purpose is only to validate the interactions of the distributed processes while the actual product result does not matter. In such cases, some specifically marked code sequences will be skipped and the simulated process will be blocked for the corresponding amount of simulated time, allowing for an even better simulation acceleration factor.

2.3 Efficient Grid Runtime Environment

Performance concerns. As our goal is to allow the development of real programs, not only simple algorithmic prototypes, the associated runtime environment has to be suitably optimized. The communication layer deserves a lot of attention because of the distributed settings of the targeted applications.

The *Native Data Representation* (NDR) constitutes an efficient data representation first demonstrated by PBIO [10] and used in GRAS. This methodology (sometimes described as "The receiver makes it right") consists in sending the data structures as they are represented in memory on the sender side. Then, if the receiver architecture matches the sender one, the data can be placed in memory without any analysis, completely avoiding the encoding costs. If the architectures do not match, this is the receiver responsibility to convert the remote data representation to the local one. The performance of GRAS is discussed in §5.

Portability. Any decent grid runtime environment has to address the operating system and hardware heterogeneity issue. In this view, the system call virtualization mechanism discussed above is used as a portability layer over the different operating systems, ensuring that any user code built on top of GRAS remains portable across architectures.

Thanks to a plain C ANSI implementation using tools like `autoconf` for compile-time configuration, the GRAS framework is itself highly portable. It is known to run at least on the following platforms: Linux (x86, AMD64, IA64, ALPHA, SPARC, HPPA and PPC); Mac OS X; Solaris (SPARC and x86); Irix and Aix.

The only platforms on which GRAS is known not to work are the Windows operating system family and the ARM processor family. Both issues are currently worked on.

Ease of use. As discussed in §4, the provided API aims at remaining simple and easy to use. In addition, GRAS provides the grounding features needed by any advanced distributed application such as logging facilities (in the spirit of the log4j project[2]), error handling and exception raising primitives, advanced data containers (dynamically sized arrays, hash tables, *etc.*), as well as basic configuration support.

[2] URL: http://jakarta.apache.org/log4j

3 Related Work

The duality of our design goals (from ease of development to efficient execution) naturally connects the GRAS framework to different literature domains, which we shall present now.

3.1 Development Framework for Distributed Applications

Solutions such as **Vampir** [15] and **TotalView**[3] allow the programmer to explore the communication patterns of MPI applications to identify the performance bottlenecks and other problems. These solutions provide a graphical representation of captured data to ease their understanding. The main drawback stands in the volume of data provided. Since they do not suppose any semantic on the sent bytes, they have to show all of them. In contrast, GRAS can use its high-level messaging infrastructure to provide a synthetic and potentially preferable view of the run. Moreover, these tools are designed to work with MPI, which is neither suitable for grid computing nor for the development of loosely coupled collections of communicating processes.

The main advantage of GRAS compared to these solutions stands in the use of a simulator, allowing to reproduce the experiments under controlled conditions. Due to the changing nature of grid platforms, two subsequent execution of the same code will necessary face different conditions, possibly triggering different behaviors of the application. It is thus not possible to thoroughly and reliably test a grid application without using a simulator.

Dimemas [3] is the performance predictor associated with the VAMPIR tool. This tool aims at the so-called "*what if analysis*": given the application trace as recorded by VAMPIR, it tries to interpolate what the application execution would be on another (possibly larger) platform. This may give some hints about the scalability of the code, but does not help to develop and debug the application.

Macedon [19] aims at providing a unified framework to compare large-scale overlay algorithms and thus help developing large-scale overlay services. It allows to specify distributed algorithms in a concise domain-specific language,

[3]URL: http://www.etnus.com/

which enables fair comparisons of the merits of individual algorithms rather than artifacts of particular implementations [19]. This specification can then be used to generate a code that runs unmodified in live Internet setting or on top of ModelNet [24]. Therefore, even though algorithms encoded within MACEDON lead to working implementations, they remain prototypes which efficiency is arguable. Lastly, computation times are not taken into account by the simulator, limiting this approach to communication bound applications.

GRAS can be considered as an evolution of the Direct Execution Simulators such as **LAPSE** [9] or **MPI-SIM** [18]. These systems allow to evaluate and tune (in a simulated environment) parallel programs using MPI. The sequential code sections are timed by direct execution and their effect is then reported into the simulator. GRAS follows the same approach and extends it to heterogeneous platforms and to applications presenting irregular communication patterns.

3.2 Efficient Grid Runtime Environments

We now present some classical communication libraries and solutions distributed application. Note that they only compare to a subpart of GRAS since none of these solutions allow the users to develop and debug their applications easily using a simulator.

The classical message passing communication libraries like **PVM** [22] or **MPI** [17] were designed for high performance computing on clusters. They are thus particularly adapted to applications presenting a regular communication and synchronized execution patterns. On the opposite, GRAS aims at loosely coupled applications with potentially highly irregular communication and execution patterns. Moreover, GRAS aims at offering efficient communication of structured data while PVM or heterogeneous implementations of MPI use XDR for data encoding, impacting badly the performance. The MPICH implementation of MPI (against which GRAS is compared in §5) is homogeneous on Linux.

The **AppLeS Multi-Protocol Interprocess Communication** library (AMPiC) constitutes a simple solution to exchange messages between loosely

coupled applications and attach callbacks to their arrival in processes. It can convey fixed C structures (without dynamic arrays or other sort of pointers) over the socket, using secure connexion or on top of the MPI and Globus communication libraries. This only lacked the Native Data Representation to become an appealing grounding layer for the GRAS runtime environment.

The **Portable Binary Input/Output** library (*PBIO* – [10]) constitutes a very efficient data encoding solution. It allows to send traditional C structures in the native representation of the sender along with meta-data allowing the receiver to convert them to its own native representation if it differs from the sender one. Moreover, PBIO optimizes further the conversion by generating the needed routines in assembler at run time.

CORBA[25] is the standard of *Remote Method Invocation* (RMI), which allows to let several objects to interact. It is typically used in multi-tier enterprise applications and for applications integration. One of its advantages in this setting is to provide access to services regardless of their actual location. In our setting, this may become a disadvantage since the process location is very important to us. In other words, the applications targeted by CORBA are so different from our targets that the facilities provided do not really match our needs. For sake of implementations interoperability, the CORBA wire protocol is also less efficient than ours.

Nowadays, **XML** constitutes the *de facto* standard for interoperability, used in technologies such as **XML-RPC**[1] and **SOAP**[5]. One of its main advantages is to be a human-readable protocol, allowing for easy debugging and parsing. However, XML does not clarify the messages *semantic* by itself. There is currently a tremendous research effort in the web ontology field, but the conclusions are still unclear to us. Another huge disadvantage of XML from our point of view is the systematic data conversion to and from a textual representation which greatly impacts the performance of such solutions. This is why we decided to rule XML out in favor of an efficient binary representation. The benefits are discussed in §5.

4 Example: a Simple Matrix Multiplication RPC Tool

This section provides an example illustrating the effectiveness of GRAS. We present a simple system allowing to compute matrix multiplication remotely. It is composed of a client and a server: the client creates the matrices and send them to the server for computation. The server performs the multiplication and returns the result to the client. The whole example is about 100 lines long (accounting for memory management, error handling and the actual computation, omitted here because of space limitation). The presented example code is fully functional and can be run on top of GRAS as is. We now briefly comment the source code.

Data and messages description Concerning the data exchanged over the network, our example only uses matrices. Declaring this structure to GRAS is as easy as including the actual C `struct` statement in a `GRAS_DEFINE_TYPE` macro call (see Figure 1(a)). The structure definition is then saved to a string variable for further parsing and use at run time. This statement is also left unchanged for the compiler and there is no need of manual duplication.

There is no limitation on the datatype conveyed as message payload. It can be a structure, as in this example, or any other valid C type such as `enum`, `union` or pointers. Thanks to classical garbage collecting mechanisms, GRAS can deal with cycles of pointer references. They can be detected automatically on the sender side and reconstituted on the receiver side.

When the datatype contains pointers, as in this example, the user needs to provide additional semantic information. This is done with the `GRAS_ANNOTE` macro on lines 5-6. Here, it specifies that *ctn* is an array (and not a simple reference), which size is the result of the operation *rows* * *cols* (with rows and cols being the other fields of the structure).

Once the data types are declared, we need to describe the messages which may be exchanged over the network. Enclosing the data in messages allows to attach some semantic to them. For instance, the client sends to the server a `"mm request"` message containing the matrices to multiply instead of the

data directly. This helps the server to differentiate between the several request types.

Our tool declares two new messages conveying respectively a matrix multiplication request and the result of such a request. The first message accepts an array of two matrices as a payload while the second conveys an unique matrix. These declarations figure in 1(b).

On line 3, the macro `gras_datadesc_by_symbol` retrieves the structure statement definition saved by `GRAS_DEFINE_TYPE` and parses it automatically. The result is stored in the `matrix_d` variable. Lines 6-7 uses it to build the description of a 2 matrices long array. Line 5 declares the `"mm request"` message type using this description. Line 8 declares the `"mm answer"` message: it can convey one matrix.

Client side The function presented in Figure 2 acts as the main() of our client. On line 5, the GRAS infrastructure is initiated using the command line arguments. A new socket is opened on line 6. It will be used to contact the server. All known messages are then registered on line 7. The line 8 stands for the matrix creation. Being less relevant, this code is omitted. On line 9, a request conveying the prepared matrices is sent to the server, and the answer is awaited on line 10 up to 600 seconds.

Server side We now discuss the code of our server, presented in Figure 3. The function `server()` (lines 13-24) acts as the main() function for the server. After the needed initializations, it registers the function `request_cb()` as a callback to the incoming `"mm request"` messages on line 20 and waits for an incoming message on line 21. The callback code presented on lines 1-12 is also relatively straightforward.

The request payload is casted and stored in a variable on line 3. Lines 7-9 allocate the needed memory to store the result. Line 10 stands for the actual matrix multiplication, which code is omitted for sake of clarity. The result is sent back to the `expediter` (passed to the callback as argument) on line 11.

Deployment Once the code of all system components is written, GRAS has to be instructed about how to glue them together. Figure 4 constitutes a simple deployment file. It specifies that the program `server` has to be launched with

```
1  GRAS_DEFINE_TYPE(s_matrix,
2  struct s_matrix {
3    int rows;
4    int cols;
5    double *ctn GRAS_ANNOTE(size,
6                            rows*cols);
7  };);
8  typedef struct s_matrix matrix_t;
```

(a) Data type definition.

```
1  void register_messages(void) {
2    gras_datadesc_type_t matrix_d
3        =  gras_datadesc_by_symbol(s_matrix);
4    gras_msgtype_declare("mm request",
5      gras_datadesc_array_fixed("matrix_t[2]",
6                                 matrix_type,2));
7    gras_msgtype_declare("mm answer", matrix_d);
8  }
```

(b) Message type definitions.

Figure 1: Data types and message types definitions.

```
1   int client(int argc,char *argv[]) {
2     gras_socket_t from, toserver=NULL;
3     matrix_t request[2], answer;
4
5     gras_init(&argc, argv);
6     toserver=gras_socket_client(argv[1], atoi(argv[2])); /* host port */
7     register_messages();
8     /* Prepare the request for the server (omitted) */
9     gras_msg_send(toserver, gras_msgtype_by_name("mm request"), &request);
10    gras_msg_wait(600,gras_msgtype_by_name("mm answer"),&from,&answer);
11  }
```

Figure 2: Client side.

```
1   int request_cb(gras_socket_t expediter, void *payload_data) {
2     /* 1. Get the payload into the data variable */
3     matrix_t *data=(matrix_t*)payload_data;
4     matrix_t result;
5
6     /* 2. Make some room to return the result */
7     result.rows = data[0].rows;
8     result.cols = data[1].cols;
9     result.ctn = xbt_malloc0(sizeof(double) * result.rows * result.cols);
10    /* 3. Do the computation (code omitted), and send the result back */
11    gras_msg_send(expediter, gras_msgtype_by_name("answer"), &result);
12  }
13  int server (int argc,char *argv[]) {
14    gras_socket_t sock;
15
16    gras_init(&argc,argv);
17    sock = gras_socket_server(atoi(argv[1])); /* first arg: port */
18    register_messages();
19
20    gras_cb_register(gras_msgtype_by_name("mm request"),&server_cb_request_handler);
21    gras_msg_handle(600.0);
22  }
```

Figure 3: Server side.

```
1  <?xml version='1.0'?>
2  <!DOCTYPE platform_description SYSTEM "surfxml.dtd">
3  <platform_description>
4    <process host="Tremblay" function="server">
5       <argument value="6000"/>       <!-- port number -->
6    </process>
7    <process host="Fafard" function="client">
8       <argument value="Tremblay"/>   <!-- server host -->
9       <argument value="6000"/>       <!-- port number -->
10   </process>
11 </platform_description>
```

Figure 4: Deployment description file.

the string "6000" as unique argument on a machine named `Tremblay`, and that the program `client` has to be launched with the provided arguments on the machine `Fafard`.

This is used to do the actual deployment on top of the simulator, but unfortunately, GRAS cannot deploy the code automatically on real platforms yet, requiring the user to deploy its code manually using tools like `ssh`. Nevertheless, it uses this file to get the name of the components to build, and write the corresponding `main()` function calling the right user function after initialization.

5 Experimental Evaluation

This section aims at evaluating more quantitatively the GRAS framework. In that view, we implemented a simple communication example using several communication libraries. The code simplicity was then measured using classical metrics and the communication performance was compared in different settings.

The chosen message is involved in the Pastry [20] application protocol. It is returned by all contacted nodes when a new node joins the system. Figure 5 presents the C definition of this data type. This structure mainly contains one array of integers and one array of sub-structures, which themselves contain an array of integer. All arrays sizes are fixed at compilation time. The whole structure is 84 bytes long.

In this experiment, we compare GRAS to the following solutions: the MPICH implementation (version 1.2.5.3) of the MPI standard; the OmniORB

```
1 typedef struct {
2   int which_row;
3   int row[COLS][MAX_ROUTESET];
4 } row_t;
```

```
1 typedef struct {
2   int id, row_count;
3   double time_sent;
4   row_t *rows;
5   int leaves[MAX_LEAFSET];
6 } welcome_msg_t;
```

Figure 5: C definition of the exchanged message.

implementation (version 4.0.5) of the CORBA standard; PBIO, which is presented in §3.2 and a hand-rolled solution using the expat XML parser. To our knowledge each of these implementations are amongst the best solutions in their categories.

User code complexity. In this section, we compare the complexity of the code that the user has to write to exchange this message. This comparison, presented in Table 1, uses two classic code complexity metrics: The McCabe Cyclomatic Complexity shown on the first line is the amount of functions plus the occurrence count of `for`, `if`, `while`, `switch`, `&&`, `||`, and `?` statements and constructs. This metric assesses the code complexity and its maintenance difficulty [13]. The second line reports the number of lines of code (not counting blank lines and lines that contain nothing but comments).

The OmniORB column presents two series of numbers because CORBA prevents the developer from using the arrays from the standard library and forces them to use its own representation of data collection. Thus, the modification when distributing an existing sequential code is more consequent with CORBA than with other solutions. To recap this, the first number is the code for the actual exchange over the network while the number in parenthesis also accounts for the mandatory conversions from and to the CORBA data collection types.

	GRAS	MPI	PBIO	OmniORB	XML
McCabe Cyclomatic Complexity	8	10	10	12 (20)	35
Number of lines of code	48	65	84	92 (195)	150

Table 1: Complexity and size of the different implementations.

MPI is quite simple, the main difficulty being that it requires manual marshalling and unmarshalling of data. PBIO exempts the user of these error-prone tasks, but requires the declaration of data type description meta-data. OmniORB requires the user to override several methods of classes automatically generated from an IDL file containing the data type description. The CORBA initialization is also a bit longer that of other solutions. Interestingly enough, the XML solution is by far the most complicated one. It may be an artefact of our use of the expat parser, but since it is usually considered as the fastest XML parser, we decided to use this solution anyway.

GRAS automatically marshals the data according to the type description, which is also automatically retrieved from the C structure declaration (*cf.* §4). This allows GRAS to be the less demanding solution from the developer perspective, according to both metrics.

Communication performance. To quantify communication performance, we conduct some experiments involving computers of different architectures (PPC, SPARC and X86), and at different scales. Figure 6 presents the timings measured when the data is exchanged between processes placed onto the same host. Figure 7 presents the timings measured on a LAN. The sending architecture is indicated on the row while the receiving architecture is shown by the column (for instance, the most down left graphic was obtained by exchanging data from a PPC machine to a X86 one). Figure 8 presents the timings measured in an intercontinental setting: data is exchanged from the previously used hosts located in California, to an X86 host placed in France. The X86 machines are 2GHz XEONs, the SPARC are UltraSparc II and the PPC are PowerMac G4. The SPARC machines are notably slower than the other ones while X86 and PPC machines are comparable. All hosts run Linux. The LAN is connected by a 100Mb ethernet network, and both sites are connected to a T1 link.

Each experiment were run at least 100 times, for a total of more than 130 000 runs. Moreover the different settings were interleaved to be fair and equally distribute the external condition changes over all the tested settings.

The first result of these experiments is the relative portability of communication libraries. This version of PBIO does not work on the PPC architecture while MPICH fails to exchange data between little-endian Linux architectures

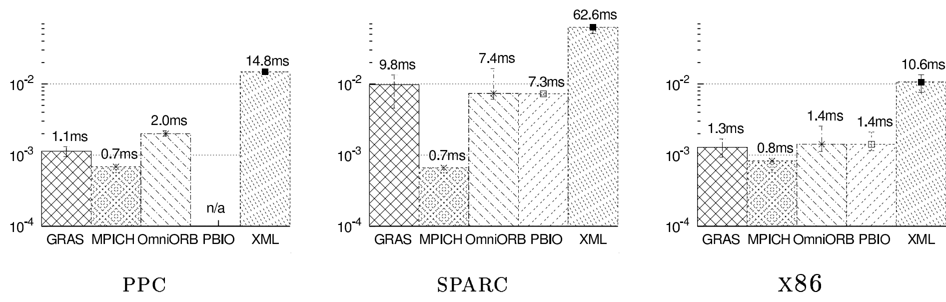

Figure 6: Intra-machine performance.

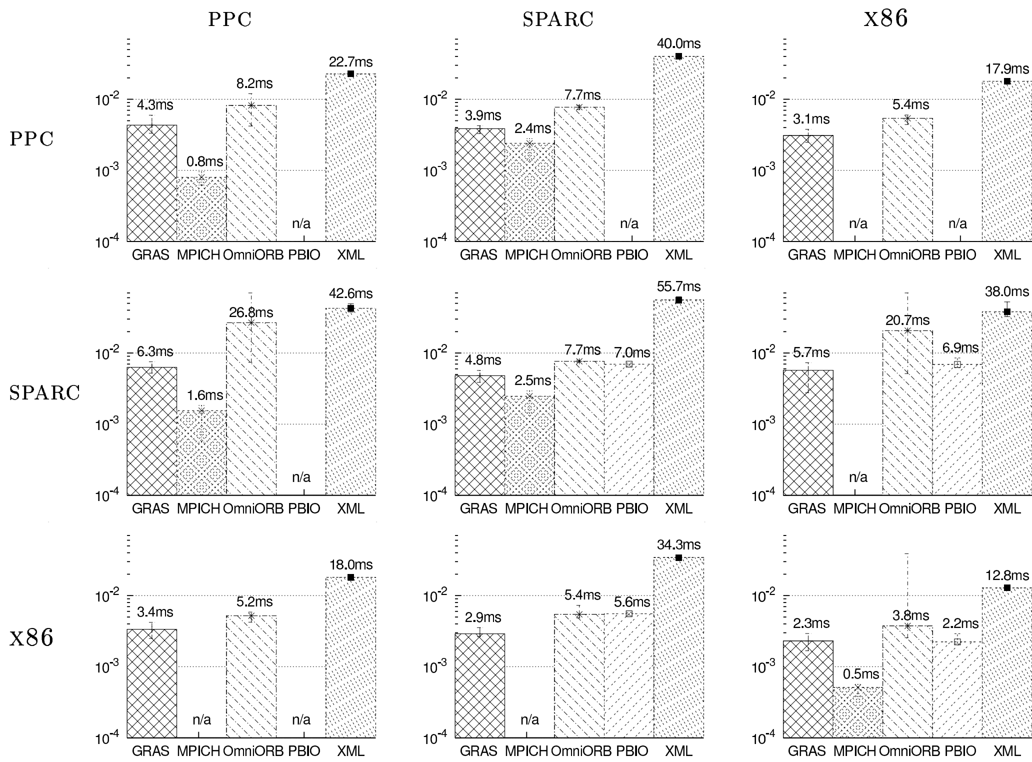

Figure 7: LAN performance (column: sender; row: receiver).

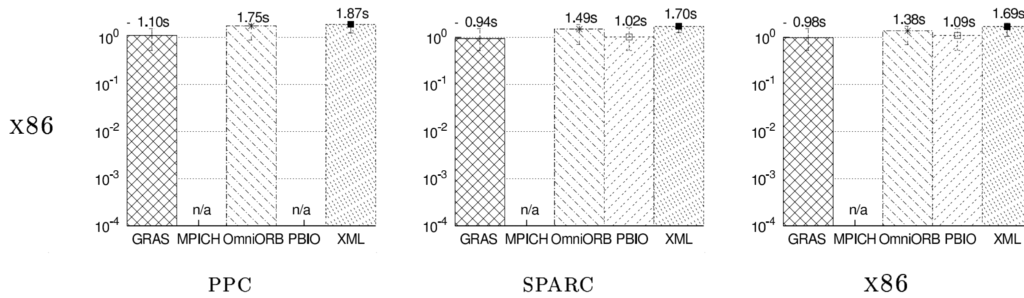

Figure 8: Wide Area Network (WAN) performance.

and big-endian ones. We were also unable to exchange data on the WAN using MPICH.

The most expected result is that the XML based solution suffers from bad performance. The systematic conversion from and to a textual representation of the data induces an extra computation load on the machines while the verbosity of this representation loads the network. The performance is thus worse by at least one order of magnitude.

Another general trend is that when MPICH is usable, it is about twice as fast as the other solutions. Unfortunately, it is only usable in half of the settings, as discussed previously.

One can also remark that the results are not symmetric: it is twice as long to exchange the data with GRAS from PPC to SPARC than from SPARC to PPC. This is naturally due to the NDR data representation (*cf.* §2.3): the receiver is the one doing the conversion, and SPARC hosts are much slower than PPC ones. The same effect can be observed with OmniORB and PBIO. Finally, the differences between solutions tend to be attenuated on WAN. Indeed, the latency becomes more important, masking the optimization done in each solution.

These results are quite satisfying for us: beside of MPICH, GRAS is the fastest solution in all settings, but the X86/X86 setting (where PBIO is faster by 0.1ms – 4%) and the SPARC intra-machine setting (where both OmniORB and PBIO are faster by 2.5ms – 25%). This performance, added to the portability of our solution and its simplicity of use shown above constitute strong arguments for the quality of the GRAS framework.

6 Conclusion

This paper introduces a new grid programming framework allowing developers to evaluate and tune their applications easily thanks to a simulator. The resulting code can then be deployed on the target platform without code modification.

We detail the project design goals, and locate GRAS in the state of the art through a quick survey. In addition, we comment a basic example of a remote matrix multiplication tool to show the ease of use of this framework. We then compare several versions of code exchanging one message (part of the Pastry application protocol) on the network using either GRAS or MPICH, OmniORB, PBIO or an XML representation using the expat parser. Finally, the performance of each version are assessed. We find that GRAS is the less demanding solution to the user, and that MPICH outperforms by far every solutions. GRAS offers the second best performance of our test sets. On the other hand, MPICH does not allow to exchange data over the WAN neither between little- and big-endian Linux hosts.

We thus claim that GRAS is an easy to use distributed application development framework resulting in efficient yet portable applications suited to a typical grid platform. It shorten the development cycles by simplifying the user code and allowing the debugging phase to take place on the simulator. We believe that its combined simulation/*in-situ* approach is the key to effective grid infrastructures and applications.

The GRAS source code represents 15,000 lines of C code. It was recently merged in the SIMGRID project, which is freely available from its web page[4] and comes with all relevant information as well as with several example programs.

GRAS currently enables to easily build a distributed application on UNIX platforms. That is fine when designing a distributed services infrastructure for a grid platform. However, we believe that our framework could also be used to develop peer-to-peer applications such as the PASTRY data location overlay. We are therefore working on getting a version that would also run on WINDOWS platforms. This would moreover allow to test peer-to-peer applica-

[4] http://gforge.inria.fr/projects/simgrid/

tions against a much more realistic simulation model than the ones that are generally used. Indeed, most simulations used in the peer-to-peer community use over-simplified network models that do not account for network outage or contention, which may be a problem when large file transfers are taken into account.

Acknowledgments

The author would like to thank Rich WOLSKI and the NWS project for the source of inspiration it constitutes, for the kindness of staff members, as well as for allowing the experiments presented in this paper to take place in their lab. He would also like to thank Henri CASANOVA for his wise advices on the preliminary versions of this paper as well as Arnaud LEGRAND for the friendly and constructive discussions on GRAS.

References

[1] M. Allman. An evaluation of XML-RPC. *ACM Performance Evaluation Review*, 30(4), March 2003.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[3] R. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and S. Müller. Performance Prediction in a Grid Environment. In *Proceedings of the 1st European Across Grids Conference*, Santiago de Compostela, 2003.

[4] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving Simulation for Network Research. Technical Report 99-702, University of Southern California, 1999.

[5] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielson, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1. Technical report, World Wide Web Consortium, May 2000.

[6] E. Caron and F. Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. 2005. To appear.

[7] H. Casanova, A. Legand, and L. Marchal. Scheduling Distributed Applications: the SimGrid Simulation Framework. In *Proceedings of the third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, may 2003.

[8] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *Proceedings of the 9th Heterogeneous Computing workshop (HCW'2000)*, pages 349–363, 2000.

[9] P. Dickens, P. Heidelberger, and D. Nicol. Parallelized direct execution simulation of message-passing parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 7(10):1090–1105, 1996.

[10] G. Eisenhauer, F. Bustamante, and K. Schwan. Native Data Representation: An efficient wire format for high-performance distributed computing. *IEEE transactions on parallel and distributed systems*, 13(12):1234–1246, december 2002.

[11] I. Foster and C. Kesselman (Eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[12] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

[13] B. Henderson-Sellars. Modularization and mccabe's cyclomatic complexity. In *Communications of the ACM*, volume 37, pages 17–19, Dec 1992.

[14] S. Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, February 1998.

[15] S. Kim, P. Ohly, R. Kuhn, and D. Mokhov. A performance tool for distributed virtual shared-memory systems. In Acta Press, editor, *4th IASTED Int. Conf. Parallel and Distributed Computing and Systems*, pages 755–760, Calgary, Canada, 2002.

[16] J. Liu and D. Nicol. *DaSSF 3.1 User's Manual*, April 2001. Available at http://www.cs.dartmouth.edu/~jasonliu/projects/ssf/papers/dassf-manual-3.1.ps.

[17] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[18] S. Prakash and R. Bagrodia. MPI-SIM: using parallel simulation to evaluate MPI programs. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 467–474. IEEE Computer Society Press, 1998.

[19] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *LNCS*, 2218, 2001.

[21] G. Shao, F. Berman, and R. Wolski. Using effective network views to promote distributed application performance. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[22] V. Sunderam, J. Dongarra, A. Geist, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–547, 1994.

[23] G. Tel. *Introduction to distributed algorithms.* Cambridge, 2 edition, 2000.

[24] A. Vahdat, K. Cum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[25] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[26] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems, Metacomputing Issue*, 15(5–6):757–768, Oct. 1999.

[27] H. Xia, H. Dail, H. Casanova, and A. Chien. The MicroGrid: Using Emulation to Predict Application Performance in Diverse Grid Network Environments. In *Workshop on Challenges of Large Applications in Distributed Environments (CLADE'04)*, 2004.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399