GRAS: A RESEARCH & DEVELOPMENT FRAMEWORK FOR GRID AND P2P INFRASTRUCTURES

Martin Quinson (Martin.Quinson@loria.fr)
Nancy University / LORIA*

ABSTRACT

Distributed service architectures are mandatory to handle the platform scale and dynamicity hindering the development of grid and P2P applications. These large-scaled distributed applications are difficult to design, develop and tune because of both theoretical and practical issues.

This paper presents the GRAS framework that allows developers to first implement and experiment with such an infrastructure in simulation, benefiting from a controlled environment. The infrastructure can then be deployed *in situ* without code modification.

We detail our design goals, and contrast them with the state of the art. We study the exchange of a message (from the *Pastry* protocol) using either GRAS or several other solutions. We quantify both the code complexity and the performance and find that GRAS performs better according to both metrics.

KEY WORDS

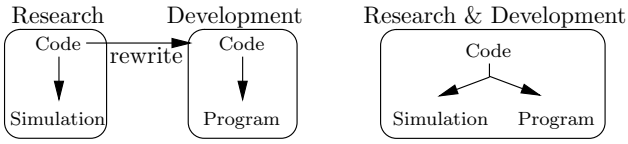
Distributed application development, grid simulation.

1 Introduction

Large scale distributed computing platforms such as grids and peer-to-peer (P2P) systems are very challenging to use because of their scale, dynamicity and heterogeneity (in terms of hardware capacities and software environment). That is why every application developers have to rely on distributed services infrastructures such as the Globus MDS and GIS [7] for resource and data discovery, the NWS ([13]) for resource monitoring, NetSolve [1] for application deployment. These infrastructures are in turn challenging to develop and to tune. Furthermore, the underlying distributed algorithms are generally extremely complex and difficult to study.

None of the classical standards of parallel computing is suited to the development of distributed service infrastructures. Most of the currently deployed infrastructures thus rely on lightweight communication libraries. For instance, NWS and NetSolve use specific communication libraries specifically fitted to their use and hence difficult to reuse in other contexts.

Another difficulty raised by large scale platforms is their dynamicity that prevent reliable reproduction of experiments and hinder faithful algorithm comparisons. As a result, developers typically spend inordinate amount of time to establish evaluation environments. Moreover, due to the changing nature of the platforms, two subsequent executions of the same code will necessary face different conditions, possibly triggering different application behaviors. Simulation constitutes a solution to these problems, but the resulting implementations are typically confined to proof-of-concept prototypes. They would need a complete rewrite to be useable *in situ*.



(a) Classical approach. (b) GRAS approach.

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 distributed 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 seamlessly producing efficient real-life-enabled code. While GRAS does not pretend to address all issues pertaining to large scale computing, we believe that its combined simulation/*in situ* approach is the key to the rapid and easy development of effective adapted infrastructures.

The remainder of this article is organized as follows: §2 presents the state of the art. §3 details the goals of GRAS. §4 provides some experimental results, both in term of performance and code complexity. §5 concludes the paper and presents some future work.

2 Related Work

Our work integrates in a unified framework the development, the simulation, and the execution of distributed applications. Related work thus falls in these three categories, as highlighted below.

2.1 Distributed application development

Solutions such as **Vampir** [10] allow the programmer to explore graphically the communication patterns of

*UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP.

MPI applications and identify *e.g.* the performance bottlenecks. The main advantage of GRAS in this context stands in the use of a simulator, allowing to reproduce the experiments under controlled conditions. **Dimemas** [2] is the performance predictor associated with VAMPIR. Given the application trace recorded by VAMPIR, it interpolates what the application execution would be on another platform. This gives some hints about the code scalability, but does not help to develop and debug the application.

Macedon [11] aims at providing a unified framework to compare large-scale overlay algorithms. They can be specified in a domain-specific language, and run either in live setting or on top of a simulator. The project's goal is to enable fair comparisons of algorithmic merits rather than artifacts of implementations. Therefore, even though algorithms encoded within MACEDON lead to working implementations, they remain prototypes whose 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** [5]. They allows 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 extends this work to heterogeneous platforms.

2.2 Simulation and emulation tools

With these tools, one can conduct tests on platform they do not have access to. For instance, they allow to quantify how the application behaves when a process is placed on an host with a slower CPU than the others.

One of the classical *emulation* solutions is **MicroGrid** [14]. It allows to run applications on a virtual platform by trapping every network-related call and mediating them. The computing resources are simulated by a local scheduler, which allocates CPU time slices to each process according to a predetermined rate. Communications are mediated according to the results of an ad-hoc simulator.

The **SimGrid** [4] 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 can simulate thousands of processes on a single workstation and is several orders of magnitudes faster than complete simulators or emulators.

2.3 Execution and communication layers for distributed applications

The classical message passing libraries such as **PVM** or **MPI** were designed for cluster computing. They are thus particularly adapted to applications presenting a regular communication and execution patterns. On the opposite, GRAS is designed for loosely coupled applications with potentially highly irregular patterns. Moreover, GRAS aims at offering efficient communication of structured data while PVM or heterogeneous implementations of MPI (such as MPICH on Linux, see §4) use XDR for data encoding, impacting badly the performance.

The **AMPiC** library[1] 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 using sockets, secure connexions, MPI or Globus communication libraries. It only lacked an efficient wire protocol to become an appealing grounding layer for the GRAS runtime environment (*cf.* §3.3).

The main goal of the **CORBA** standard is the interoperability between software environments while we aim at performance and interoperability of a given software environment across hardware settings. The target of CORBA is thus different from ours.

Nowadays, **XML** constitutes the *de facto* standard for interoperability. It is used in technologies such as **SOAP**[3], used in turn in **OGSA** [8]. Being a human-readable protocol, it allows for easy debugging and parsing. However, performance greatly suffers from the required data conversion to and from a textual representation. This is why we preferred an efficient binary representation to XML (*c.f.* §4).

3 The GRAS project

GRAS is designed to build well-tested distributed infrastructures offering a specific service to large-scale distributed applications and middlewares. For instance, one could use GRAS to build grid computational servers comparable to NETSOLVE [1], platform monitoring sensors like the NWS ones [13] or Distributed Hash Tables like Pastry [12]. 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.

3.1 Offered interface

The targeted applications are more easily described using an *event-driven* model than with a *SPMD* model. The GRAS framework relies on the *active message* paradigm and provides a high level message passing interface. The main concepts are thus:

Agents. They are the individual processes taking part to the distributed application.

Sockets. These communication end-points are very similar to the semantic of the BSD sockets.

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

Messages. They represent what gets exchanged between the agents. The type of each message gives semantic informations about the application. The actual payload format must be precisely described to allow GRAS to convert it to another representation automatically on need. Any messages of the same type must share the same payload format.

Any valid C type can be used as payload, including structures and pointers. The datatype format can be parsed directly from the C structure definition automatically in most cases.

Callbacks. These are user-defined functions attached to a given message type. They get run automatically to handle any incoming message of the matching type.

It is still possible to explicitly wait for a message matching some criteria (right message type, or right expediter, *etc.*). Messages received in the meanwhile will be queued for future use. GRAS also allows RPC-like messages, where the receiver is supposed to return an answer through a second message. Failures on the RPC receiver are automatically reported to the caller.

GRAS offers several additional features such as classical data containers (dynamic arrays, hash tables), a distributed logging service, a unit testing framework and an exception mechanism, all in C ANSI.

3.2 Development Framework for Distributed Applications

Typical distibuted programming issues. Concurrent algorithms introduce specific difficulties like race conditions and 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. To alleviate these problems, the main idea of GRAS is to allow simulator-assisted development of distributed application. The simulator constitutes a fast and controlled environment for developers, who produce seamlessly efficient real-life-enabled code.

Large scale distribution. The scale of the target platforms can range from a dozen of hosts to several hundreds for grids, or even millions for P2P systems. The platforms are then naturally highly heterogeneous. Since the resources are often shared between «grid users» and «local users» owning it, the characteristics usually vary with time. On the other hand, increasing the number of hosts dramatically increases the probability that at least one host is unavailable at a given time. Also, 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 the processes compiled on each site taking the library location and compiler settings into account can become difficult.

These technical difficulties often distract the developers from the algorithmic challenges raised by grid and P2P applications, making the development even more challenging. We expect users to run their applications quite frequently during the debugging cycle to test them in a representative set of scenarios and platforms. This need for a very fast evaluation scheme leads us to base our work on the SimGrid simulator, which is the fastest solution to our knowledge.

Simulation/*in-situ* dual approach. To allow the same code to run both on top of a real platform and in simulation mode, GRAS virtualizes the operating system and provides explicit system call wrappers. Indeed, `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.

Moreover, the computation durations have to be reported into the simulator. When the user code needs W Mflop, the corresponding simulated process has to be blocked for W/ρ virtual seconds if its virtual host delivers ρ Mflop/s. GRAS provides a mechanism to automatically benchmark W.

3.3 Distributed Runtime Environments

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. Since GRAS does not interfere with the computation and storage facilities and because of the distributed settings of the targeted applications, the communication layer deserves a lot of attention.

The *Native Data Representation* (NDR) constitutes an efficient data representation first demonstrated by PBIO [6] and used in GRAS. Data structures are sent as they are represented in memory on the sender side. If the receiver architecture matches the sender one, the data can be placed in memory without any analysis, completely avoiding the encoding costs. When architectures do not match, the receiver converts the remote data representation to the local one.

Portability. The operating system and hardware heterogeneity issue has to be addressed. 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.

The framework itself is ported to Linux (x86, amd64, ia64, Alpha, Sparc, Hppa and PPC); Mac OS X; Solaris (Sparc and x86); Irix and Aix.GRAS have no external dependency to ensure its usability everywhere.

```
1  typedef struct { /* message payload */
2    int id, row_count;
3    double time_sent;
4    row_t *rows; /* array, size=8 */
5    int leaves[MAX_LEAFSET];
6  } welcome_msg_t;
7
8  typedef struct { /* helper structure */
9    int which_row;
10   int row[COLS][MAX_ROUTESET];
11 } row_t;
```

Figure 1. C definition of the exchanged message.

4 Experimental Evaluation

This section evaluates quantitatively the GRAS framework, according to code complexity and execution performance. In the GRAS context, the simulator is seen as a development tool. We thus do not consider simulation performance here and refer the reader to [4] for further information on the used simulator.

For these experiments, we implemented a simple example using several communication libraries. The code simplicity was then measured using classical metrics and the performance was compared in different settings. The chosen message is involved in the Pastry application protocol [12]. Figure 1 presents the C definition of this data type, which is 5236 bytes long.

In this experiments, we compare GRAS to the following solutions: the MPICH implementation (version 1.2.5.3) of the MPI standard; the OmniORB implementation (version 4.0.5) of the CORBA standard; PBIO (presented in §2.3) 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.

4.1 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 metric shown on the first line assesses the code complexity and its maintenance difficulty [9]. The second line reports the number of lines of code (not counting blank lines nor comments).

	GRAS	MPI	PBIO	CORBA	XML
McCabe	8	10	10	12	35
Lines	48	65	84	92	150

Table 1. Comparision of code complexities and sizes.

The XML solution is by far the most complicated solution. It may be due to the expat parser we use, but this is the fastest XML parser. 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. GRAS automatically marshals the data according to the type description, which is automatically parsed from the C structure declaration. This allows GRAS to be the least demanding solution from the developer perspective, according to both metrics.

4.2 Code performance

The GRAS runtime only mediates the communications, leaving the computational code unchanged. We thus only evaluate communications.

For this, we conduct experiments involving computers of different architectures (PPC, SPARC and x86), and at different scales. On Figure 2, the first part presents the timings measured when data are exchanged between processes placed onto the same host. The second part 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). The last part 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 was 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 and big-endian ones. We were also unable to use MPICH on the WAN.

The performance of the XML based solution is worse than any other by one order of manitude. The systematic data conversions from and to a textual representation induce an extra computation load while the verbosity of this representation stresses the network. When MPICH is usable (half of the settings), it is about twice as fast as the other solutions. It seems that the apparent poor performance of our solution comes from the extra analysis performed: we interpret the data description at runtime to perform the data exchange automatically while MPICH requires to write the exchange code manually. This is a tradeoff between code simplicity *vs.* speed. The solution we are currently working on in order to alleviate this consists

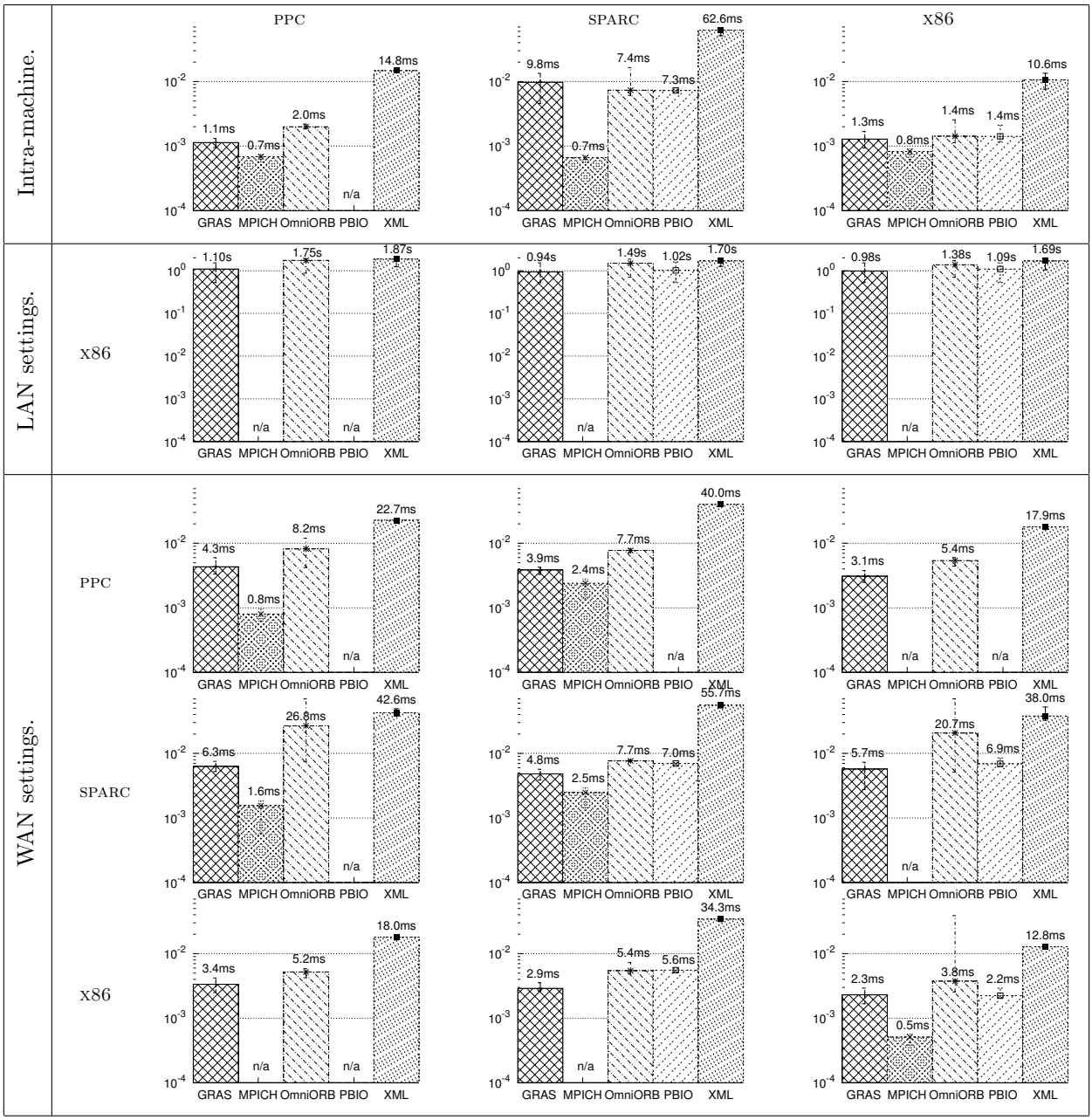


Figure 2. Performance comparison.

in analyzing the data description at compilation time to generate automatically the exchange code and thus bypass the analyze costs at runtime. Finally, the differences between solutions tend to be attenuated on WAN since the latency masks any optimization.

These results are quite satisfying for us: beside of MPICH, GRAS is the fastest solution in all settings, but the LAN 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.

5 Conclusion

This paper introduces a new large scale distributed 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 present the state of the art through a quick survey and detail our design goals. We then compare the complexity and performance of several codes exchanging one message of Pastry over the network using either GRAS, MPICH, OmniORB, PBIO or an XML representation using the expat parser. We find that GRAS is the least demanding solution to the user, and is only outperformed by MPICH. On the other hand, MPICH does not allow to exchange data over

the WAN neither between little- and big-endian Linux.

GRAS is thus an easy to use distributed application development framework resulting in efficient yet portable applications suited to a typical grid and P2P platforms. It shortens the development cycles by simplifying the user code and allowing the debugging phase to take place on the simulator. We think that this approach is the key to effective distributed infrastructures.

The GRAS source code represents 15,000 lines of C code. It was recently merged with the SimGrid project, which is freely available from its web page[2] and comes with all relevant information as well as with several example programs and tutorials.

GRAS currently enables to build a distributed application on UNIX platforms. We are porting our runtime to Windows since it is used by most of the end-users. This would increase our impact on the P2P community and test their algorithms against a much more realistic simulation model than the ones that are generally used. At the same time, we are implementing several grounding services (such as platform monitoring or application deployment) using GRAS. This will demonstrate the efficiency of our tool while constituting building bricks for others. The simulation / *in situ* dual approach demonstrated in GRAS also makes it possible to validate the simulator itself. The results obtained so far are encouraging, and open the way to more work on adapted platform models providing a good tradeoff between efficiency and realism.

Acknowledgments

The author would like to thank Rich Wolski and the NWS project for the source of inspiration it constitutes, 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] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Shi, and Vadhiyar. NetSolve v1.4.1. Technical Report 02-05, U. of Tennessee, 2002.

[2] R. Badia, F. Escale, E. Gabriel, J. Gimenez, R. Keller, J. Labarta, and S. Mueller. Performance Prediction in a Grid Environment. *European Across Grids Conference*, 2003.

[3] 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, 2000.

[4] H. Casanova, A. Legrand, and L. Marchal. Scheduling Distributed Applications: the SimGrid Simulation Framework. *IEEE Int. Symp. on Cluster Computing and the Grid*, 2003.

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

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

[7] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *Int. Conf. on Network and Parallel Computing*, pages 2–13. LNCS 3779, 2005.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. *Open Grid Service Infrastructure, GGF*, 2002.

[9] B. Henderson. Modularization and McCabe's Cyclomatic Complexity. *Communications of the ACM*, 37(12):17–19, 1992.

[10] S. Kim, P. Ohly, R. Kuhn, and D. Mokhov. A performance tool for distributed virtual shared-memory systems. In *IASTED Int. Conf. Parallel and Distributed Computing and Systems*, 2002.

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

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

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

[14] 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*, 2004.

[2] http://simgrid.gforge.inria.fr/