Padico: A Component-Based Software Infrastructure for Grid Computing

Alexandre Denis* Christian Pérez† Thierry Priol† André Ribes†

*IRISA/IFSIC †IRISA/INRIA

Campus de Beaulieu, 35042 Rennes Cedex, France

{Alexandre.Denis,Christian.Perez,Thierry.Priol,Andre.Ribes}@irisa.fr

Abstract

This paper describes work in progress to develop a component-based software infrastructure, called Padico, for computational grids based on the CORBA Component Model from the OMG. The objective of Padico is to offer a component model targeting multi-physics simulations or any applications that require the coupling of several codes (simulation or visualization) within a high-performance environment. This paper addresses mainly two issues we identified as important for a grid-aware component model. The first issue deals with the encapsulation of parallel codes into components. We propose an extension to the CORBA *component model called GridCCM. The second issue addresses the problem of the communication between components within a computational grid. We propose a portable runtime, called PadicoTM, able to efficiently support communication in a heterogeneous networking environment.*

1 Introduction

Computational Grids promise to be the next generation of high-performance computing resources. However, programming such computing infrastructures will be extremely challenging. Current grid programming practices tend to be based on existing and well understood models such as message-passing and SPMD (single program multiple data). A computational grid is thus seen as a virtual distributed memory parallel computer; it limits its use to parallel applications, which are only a subset of applications that could benefit from such computing infrastructures. Current efforts, such as Cactus [1], aim at designing problem solving environments (PSE) that offer more flexible programming models based on the idea of modularization. Several codes can be interconnected within a PSE to solve a problem in a specific domain. Component programming models generalize this approach to any domain. Component models were responses to the increasing complexity of the application development processes in business computing including the design phase. The idea behind component programming is to design an application from existing building blocks avoiding the development of codes when they already exist. Component models such as the Enterprise Java Beans (EJB) [25], Microsoft Distributed Component Object Model (DCOM) [18], and more recently the OMG CORBA Component Model (CCM) [19] and Web Services [8] are a few examples. Those component models were mainly designed for business and/or Internet computing and are not well suited for high-performance computing. Most of them do not even run on existing supercomputers. The Common Component Architecture (CCA) initiative [7] aims at defining a component model specification for distributed and parallel scientific computing. It is a minimal specification in that it does not impose a particular runtime environment for the execution of CCA components making the CCA model portable across a wide spectrum of high-performance computing infrastructure including computational grids. However, it does not deal with interoperability issue nor the deployment of components in binary form.

This paper addresses the design of a component-based software infrastructure, called Padico, for computational grids based on the CORBA Component Model from the OMG. It thus takes advantage of all the current efforts to build a component model including all aspects related to the use of components such as discovery, deployment, instantiation, interoperability, etc. CORBA appears as an interesting choice as it is a mature technology which is independent of the operating system, of the languages and of the communication protocols. Moreover, its component model is the most complete, standardized component model. For example, it specifies how to deploy a distributed set of components. Our objective is twofold. First, we aim at extending the CORBA component model in such a way SPMD codes can be encapsulated easily and efficiently. Second, we target to design a portable and efficient runtime environment for computational grids that lets components communicate with each other using the available underlying network, whatever it may be: WAN, LAN or SAN.

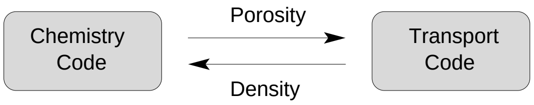

Figure 1. Communication scheme of a basic code coupling application.

The remainder of this paper is divided as follows. Section 2 presents some examples of usage scenarios for which we think that a component model is suitable. Section 3 gives a short overview of the OMG CORBA component model. In Section 4, we propose some extensions to the CCM model, called GridCCM, to support grid applications as well as a portable runtime, called PadicoTM. Section 5 presents some related works. Finally, we provide some concluding remarks in Section 6.

2 Example of a Grid Application and Usage Scenarios

Many applications, such as code couping application for example, may benefit from the amount of computational power Grids can offer.

Let us consider a simple code coupling application that simulates the transport of chemical products in a porous medium. There are two codes: one code computes the chemical product's density and a second code simulates the medium's porosity. Figure 1 presents the coupling scheme of this application. Both code need to be parallel when simulating 3D media.

This section introduces some typical use cases that such an application may have to face during its life cycle.

Legacy codes. Developing a program is a complex and long time effort. The chemical and the transport codes are probably developed by different teams, each team having its own set of tools: the codes must be assumed to be written in different languages (FORTRAN, C++, etc.) and the parallelism may be based on different paradigms (MPI, PVM, Global Arrays, OpenMP, etc.).

Maintainability. Developers need a simple way to regularly update their code for various reasons: bug fixes, new features, new algorithms, etc.

Deployment: communication flexibility. Two different configurations are available depending on some external conditions. The first configuration is made of two parallel machines connected by a wide area network. Each parallel machine is large enough to only execute one of the two codes. The second configuration is a parallel machine large enough to execute both codes. The features (network, processor, etc) of the machines are known statically. In the first case, the communications between the two codes use the wide are network while they use the parallel machine network in the second case.

Deployment: machine discovery. The users may not have a direct access to some machines. They need a mechanism to find, to deploy and to execute their codes on machines they are get access to. The features of the machines (network technologies, processors, etc.) are not known statically.

Deployment: localization constraints. A company X would like to test the propagation of its patented chemical product. It wants to couple its codes with the transport code. However, the chemistry code (source and binaries) must be on the machines of the company.

Communication security. A grid can be made of secure and insecure networks. The data computed by the simulation need to be secured on insecure networks.

The next section introduces software components as a solution to simply and efficiently support those scenarios.

3 Software Components

3.1 From Objects to Software Components

Object-oriented programming has provided substantial advantages over structured programming. Software component technology is expected to bring a similar evolution to software technology. While object-oriented programming targets application design, component software technology emphasizes component composition and deployment.

Software component technology [26] has been emerging for some years [5] even though its underlying intuition is not very recent [17]. Among all the definitions of software components, here is Szyperski's one [26]: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"

Component applications are naturally modular as each component represents a separate functionality. Code reuse and code maintainability are made easier as components are well-defined and independent. Last, components provide mechanisms to be deployed and connected in a distributed infrastructure. Thus, they appear very well suited for Grid Computing.

3.2 The CORBA Component Model (CCM)

The CORBA Component Model [19] (CCM) appeared in CORBA 3.0 [20]. CCM allows the creation and deployment of components in a distributed environment. CCM is one of the most complete models because it manages the whole life

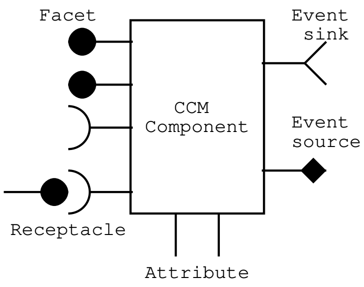

Figure 2. A CCM component.

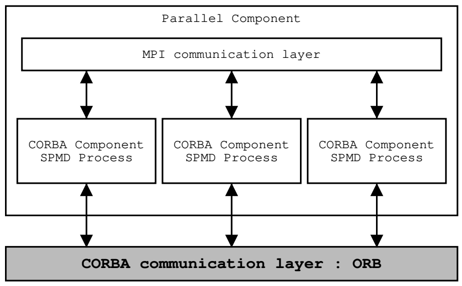

Figure 3. Parallel component concept.

cycle of a component. It specifies four models : abstract, programming, deployment and execution models.

The CCM abstract model allow developers to define interfaces and properties of components. Components may interact through different types of ports as shown in Figure 2. Facets and receptacles are synchronous ports that express what services a component provides (facet) and/or requires (receptacle). Events are published by its event sources and received by event sinks. The CCM programming model defines the Component Implementation Definition Language (CIDL) which is used to describe the implementation structure of a component and its system requirements: the set of implementation classes, the abstract persistence state, etc. The CCM deployment model is based on the use of software packages, i.e. "ZIP" archives containing component descriptors and implementations. Descriptors are written using the Open Software Description (OSD) language which is an XML vocabulary. The CCM execution model defines containers as runtime environments for component instances. Containers hide the complexity of most of the system services like the transaction, security, persistence, and notification services.

3.3 Revisiting our example with CCM

CCM brings interesting answers to many scenarios of our example. It manages the heterogeneity of the languages, computers and networks: so, legacy codes can be embedded in CORBA components and deployed in a distributed heterogeneous environment thanks to the deployment model. CCM is a dynamic model. It allows components to be dynamically connected and disconnected. Moreover, it manages versioning: CCM answers maintainability issues. Finally, CORBA [20] offers a rich environment for security issues, including authentication and delegation.

CCM solves many issues of our example. But, it lacks some functionalities to manage efficiently parallel codes.

4 CCM and Grids

4.1 Shortcomings of CCM in our example

A limitation of CCM is that it does not provide any support to encapsulate parallel codes. Modifying the parallel code to a master-slave approach so as to restrict CORBA communications to one node (the master) does not appear to be the right solution: it may require non trivial modifications to the parallel code and the master node may become a communication bottleneck. This issue is addressed by GridCCM, a parallel extension to CCM, presented in Section 4.2.

We also consider two other problems. The first one is the management of the network heterogeneity within high-performance constraints. The second one in the cohabitation of different middleware systems inside one process, like CORBA and MPI for example. These problems are addressed by PadicoTM in Section 4.3.

GridCCM and PadicoTM currently define Padico, a component-based software infrastructure for grid computing. The goal is to offer a programming and execution framework to be able to easily and efficiently use Grids.

4.2 Parallel CORBA Components: GridCCM

4.2.1 Introducing Parallelism into CCM

GridCCM extends the CORBA Component Model with the concept of parallel components. Its objective is to allow an efficient encapsulation of parallel codes into GridCCM components. Another goal of GridCCM is to encapsulate parallel codes with as few modifications to parallel codes as possible. Similarly, we target to extend CCM without introducing deep modifications into the model: the CORBA Interface Definition Language (IDL) is not modified and parallel components are interoperable with standard sequential components. We currently limit the model to only embed SPMD (*Single Program Multiple Data*) codes because of two considerations. Many parallel codes are indeed SPMD and the SPMD model is an easily manageable model.

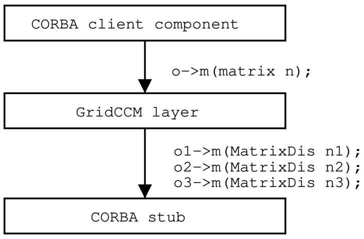

Figure 4. GridCCM intercepts and translates remote method invocations.

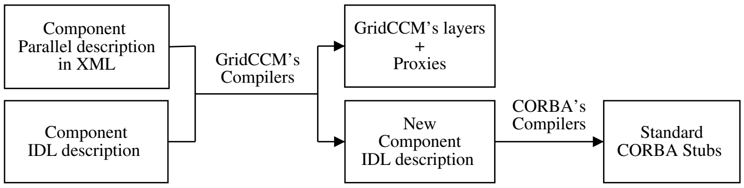

Figure 5. Compilation steps to generate a parallel component.

Figure 3 illustrates a parallel component at runtime. The SPMD code uses MPI for its inter-process communications; it uses CORBA to communicate with other components. To avoid bottlenecks, all processes of a parallel component participate to inter-component communications. The nodes of a parallel component are not directly exposed to other components. We introduced proxies to hide the nodes. More details about parallel CORBA components are presented in [21].

4.2.2 Managing the Parallelism

To introduce parallelism, like data redistribution, without requiring any change to the ORB, we choose to introduce a software layer between the user code (client and server) and the stub as illustrated in Figure 4.

A call to a parallel operation of a parallel component is intercepted by this new layer that sends the data from the client nodes to the server nodes. It can perform a redistribution of the data on the client side, on the server side or during the communication between the client and the server. The decision depends on several constraints like feasibility (mainly memory requirements) and efficiency (client network performance versus server network performance).

The parallel management layer is generated by a compiler specific to GridCCM. This compiler uses two files: an IDL description of the component and an XML description of the component parallelism. Figure 5 presents the compilation steps. In order to have a transparent layer, a new IDL interface description is generated. This interface derived from the original interface is internally used by the GridCCM layer to actually invoke operations on the server side. The original IDL interface is used between the user code and the GridCCM layer on the client and the server sides.

In the new IDL interface, the user arguments described as distributed have been replaced by their equivalent distributed data types. This transformation constraints the types that can be distributed. The current implementation requires the user type to be an IDL `sequence` type, that is to say a 1D array. So, one dimension distribution can automatically be applied. This scheme can easily be extended to multidimensional arrays: a 2D array can be mapped to a sequence of sequences and so on. CORBA data constructors may allow memory copies to be avoided.

4.2.3 Preliminary Implementation of GridCCM

We have implemented a preliminary prototype of GridCCM on top of two existing CCM implementations: OpenCCM [27] and MicoCCM [22]. OpenCCM is developed at the research laboratory LIFL (*Laboratoire d'Infomatique Fondamentale de Lille*) and is written in Java. MicoCCM is an *OpenSource* implementation based on the Mico ORB and is written in C++. We are indeed targeting high-performance. However, Java CCM implementations are more complete than C++ implementations. So, we used the Java implementation to show the feasibility and the genericity of the approach. Section 4.4 presents some preliminary performance results.

4.3 Managing Communications: PadicoTM

GridCCM requires several middleware systems at the same time, typically CORBA and MPI. They should be able to efficiently share the resources (network, processor, etc.) without conflicts and without competing with each other. Moreover, we want every middleware systems to be able to use every available resources with the most appropriate method so as to achieve the highest performance. Thus, we propose a three-level runtime layer model which decouples the interface seen by the middleware systems from the interface actually used at low-level: an *arbitration layer* plays the role of resources multiplexer; an *abstraction layer* virtualizes resources and provides the appropriate communication abstractions; a *personality layer* implements various APIs on top of the abstract interfaces. The originality of this model is to propose both parallel and distributed com-

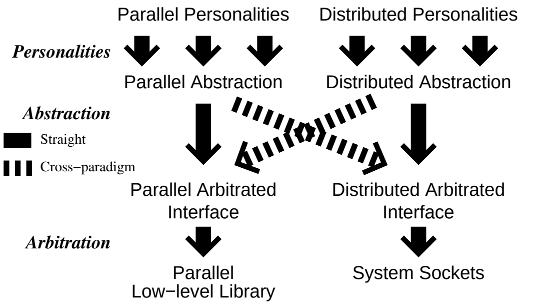

Figure 6. The PadicoTM communication model.

munication paradigms at every level, even in the abstraction layer. There is no "bottleneck of features". This model is implemented in PadicoTM, an overview of which is shown in Figure 6.

4.3.1 Arbitration Issues

Supporting CORBA and MPI, *both running simultaneously* in the same process using the same network, is not straightforward. Access to high-performance networks is the most conflict-prone task when using multiple middleware systems at the same time. There are various conflicts sources: hardware with exclusive access (e.g. Myrinet through BIP), hardware with limited non-shareable physical resources (e.g. SCI mappings), incompatible drivers (e.g. BIP *or* GM on Myrinet). Moreover, it is now common that middleware implementations use multithreading. However, middleware systems are likely to use incompatible thread policies, or simply different multithreading packages. In the worst case, more than one middleware system cannot coexist in the same process nor on the same machine. If ever we are lucky enough and it works, the access to the network is competitive, prone to race conditions, and most likely suboptimal. Resource access should be *cooperative* rather than *competitive*, as described in [11, 12].

These problems are dealt with in the *arbitration layer*. The arbitration layer aims at providing an intelligent and multiplexed access to every networking hardware. Then, we will be able to provide more advanced abstractions on top of a fully multiplexed and reentrant communication system. The arbitration layer provides an access method for the available networking hardware; each type of network is used with the most appropriate paradigm. We believe that the low-level interface should respect the differences between the parallel and distributed paradigms; trying to bend them to a common API would lead to an awkward model and sub-optimal performance. Thus, we utilize *distributed oriented* links (WAN, LAN) with plain sockets, and *parallel oriented* networks (Myrinet, SCI, high-performance networks inside a parallel machine) with a low-level library optimized for parallelism. For good I/O reactivity [6] and portability over high-performance networks, we have chosen Marcel [10] (multithreading library) and Madeleine [3] (parallel oriented high-performance network library) as foundations. This layer is the only client of the low-level resources. Then, it should be the unique entry-point for low-level access. All accesses to the networks, multithreading, Unix signals, libraries or memory allocation should be performed through the arbitration layer. It contains a subsystem for each low-level paradigm (one for Madeleine, one for sockets), and a core which handles the interleaving between the different paradigms to avoid competition, and enforces a coherent multithreading policy among the concurrent polling loops.

4.3.2 Abstraction Layer

On top of the arbitration layer, there is an abstraction layer which provides higher level services, independent of the hardware. Its goal is to provide various abstract interfaces well suited for their use by various middleware systems.

A wide-spread design consists in providing a unique abstraction on which several middleware systems may be built. However, if this unique abstract interface is parallel-oriented (*à la* MPI: message-based, SPMD, logical numbering of processes), dynamicity and link-per-link management are not easy. On the other hand, if this unique abstract interface is distributed-oriented (*à la* sockets: streams, fully dynamic), the performance is likely to be poor. Thus we propose an abstraction layer with both parallel- and distributed-oriented interfaces; these abstract interfaces are provided on top of every method provided by the arbitration layer. A given abstract interface should be the same whatever the underlying network is. The abstract layer should be fully transparent: a middleware system built on top of the abstract layer should not have to know whether it uses Myrinet, a LAN or a WAN; it always uses the same API and does not even choose which hardware it uses. The abstraction layer is responsible for automatically and dynamically choosing the best available service from the low-level arbitration layer according to the available hardware; then it should map it onto the right abstraction. This mapping could be *straight* (same paradigm at low and abstract levels, e.g. parallel abstract interface on parallel hardware) or *cross-paradigm*– e.g. distributed abstract interface on parallel hardware, as shown in Figure 6. PadicoTM implements a parallel-oriented API called *Circuit* and a distributed-oriented API called *VLink*.

4.3.3 Personality Layer

The abstraction layer provides abstract interfaces, which are generic interfaces for parallel and distributed paradigms. However, for a better flexibility and for seamless integration of legacy codes, it is better to provide standard APIs. This is achieved through the use of *personalities* on top of the abstract interfaces. Personalities are thin adapters which adapt a generic API to make it look like another close API. They do not do protocol adaptation nor paradigm translation; they only adapt the syntax. We have implemented personality modules which make *Circuit* look like Madeleine or FastMessages, and *VLink* look like standard BSD sockets or Posix.2 Asynchronous Input/Output interface (*Aio*).

4.3.4 Middleware Systems on PadicoTM

One of the strengths of PadicoTM is that it is straightforward to port existing middleware systems on PadicoTM personalities. Most of the time, it is required no change in their source code nor in their makefiles, thanks to wrappers used at link stage. This is very interesting when considering the complexity of developing a middleware system like MPI or CORBA. We have ported an MPI implementation on PadicoTM derived from MPICH/Madeleine [4] with very few changes. Various CORBA implementations have been seamlessly used on top of PadicoTM with no code change thanks to the use of wrappers: omniORB 3 [2], omniORB 4, ORBacus 4.0, and Mico 2.3. The SOAP implementation gSOAP has also been seamlessly used on top of PadicoTM. We have ported Kaffe 1.0 (Java virtual machine) on PadicoTM for integration of Java codes. Moreover, we have ported Certi 3.0 (HLA implementation) on PadicoTM. The middleware systems, like any other PadicoTM module, are dynamically loadable. Thus, any combination of them may be used at the same time and can be dynamically changed.

4.4 Performance Evaluation of PadicoTM and GridCCM

This section presents some basic performance results showing that Padico is able to achieve high performance. The performance of MPI and CORBA on top of PadicoTM are introduced before the performance of GridCCM.

The test platform consists of dual-Pentium III 1 GHz with 512 MB RAM, switched Ethernet-100, Myrinet-2000 and Linux 2.2.

MPI and CORBA performance in PadicoTM. The raw bandwidth of MPI and various CORBA implementations in PadicoTM over Myrinet-2000 is shown in Figure 7. For MPI and omniORB, the peak bandwidth is excellent: 240 MB/s, which is 96 % of the maximum Myrinet-2000 hardware bandwidth. The latency is 11 μs for MPI and 20 μs for omniORB.

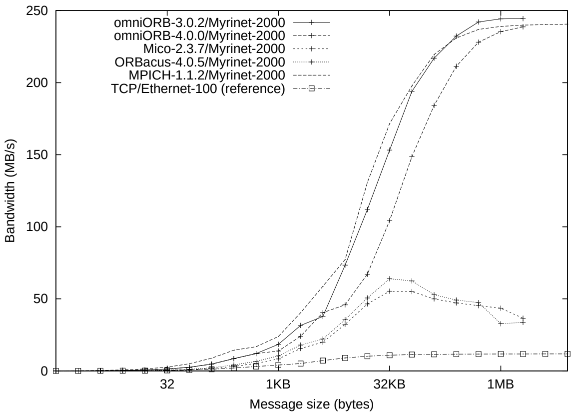

Figure 7. CORBA and MPI bandwidth on top of PadicoTM.

Other CORBA implementations get poor results. Mico peaks at 55 MB/s with a latency of 62 μs, and ORBacus gets 63 MB/s with a latency of 54 μs. These numbers are consistent with theory [11]: unlike omniORB, Mico and ORBacus always copy data for marshalling and unmarshalling.

OmniORB is as fast as MPI regarding the bandwidth, and slightly slower for latency. This latency could be lowered if we used a specific protocol (called ESIOP) instead of the general GIOP protocol in the CORBA implementation.

The MPI performance in PadicoTM is very similar to MPICH/Madeleine [4] from which PadicoTM's MPI implementation is derived; PadicoTM adds no significant overhead neither for bandwidth nor for latency. Concurrent benchmarks (CORBA and MPI at the same time) show the bandwidth is efficiently shared: each gets 120 MB/s.

Preliminary GridCCM performance. The performance of a preliminary implementation of GridCCM based on MicoCCM 2.3.7 has been measured between two parallel components. A first parallel component invokes an operation on a second parallel component with a vector of integers as an argument. The invoked operation only contains a `MPI_Barrier`. Both parallel components are instantiated on the same number of nodes. The latency and the aggregate bandwidth over PadicoTM/Myrinet-2000 is shown in Figure 8: the bandwidth is efficiently aggregated. The latency is the sum of the Mico latency and the `MPI_Barrier`; the experiments show the expected behavior.

The behavior of GridCCM on top a Fast-Ethernet network based on MicoCCM (resp. on OpenCCM (Java)) is similar: the bandwidth scales from 9.8 MB/s (resp. 8.3 MB/s) to 78.4 MB/s (resp. 66.4 MB/s).

Number of nodes	Latency (μs)	Aggregate bandwidth (MB/s)
1 to 1	62	43
2 to 2	93	76
4 to 4	123	144
8 to 8	148	280

Figure 8. Performance between two parallel components over Myrinet-2000.

GridCCM and PadicoTM allow binary components to be deployed on different sorts of networks and to transparently and efficiently use the available network.

5 Related Works

There exist few research activities dealing with the design of component models for high-performance computing. The most well known project in that area is the Common Component Architecture (CCA) [7] that aims at defining a specification for a component model for distributed and parallel scientific computing. It is a set of specifications that describe various aspects of the model like a scientific interface definition language (SIDL) and the concept of ports that define the communication model. CCA does not impose a runtime environment for the execution of CCA components making the CCA model portable across a wide spectrum of high-performance computing infrastructure including computational grids. However, interoperability is only at source level and there is no support in the model for parallel components.

Web Services [8] is a component model which is gaining large acceptance. If they appear interesting to build Grid Services, they do not appear well suited to build grid-aware *high-performance* applications. There is no support for deploying applications and their performance is poor.

Several middleware environments for managing the network communications have emerged. The ADAPTIVE Communication Environment (ACE) [24] is the closest to PadicoTM. It aims at providing a C++ high level abstract and portable interface for system features such as network and multithreading. It targets realtime – i.e. predictability – rather than high performance. It does not support high-performance networks and offers a specific API for tight integration with a middleware built on top of it. Recent works (PACE) add a Posix.1 API to ACE for seamless integration into existing codes. However, it only deals with portability on various operating systems, not with arbitration neither with automatic selection of the protocol. Similarly, Panda [23] is a framework which deals with networking and multithreading. It is mainly a portability layer to build runtime environments dedicated to parallel languages.

Harness [16] is a framework that targets high-performance distributed computing. It is built on Java. Like PadicoTM, it considers middleware systems as plugins. Currently, there is only a PVM plugin and published performance mentions only plain TCP. Proteus [9] is a system for integrating multiple message protocols such as SOAP and JMS within one system. It aims at decoupling application code from protocol, which is an approach quite similar to our separation of arbitration level/abstraction level, but at a much higher level in the protocol stack. Nexus [14] used to be the communication subsystem of Globus. It was based on the concept of global pointers. Nowadays, it becomes accepted that MPICH-G2 [13] built on Globus-IO is a popular communication mechanism for grids. However, it is appropriate only to deploy parallel applications on grids, which is a too limiting a model for certain grid applications.

6 Conclusion

Computational grids allow new kinds of applications to be developed. For example, code coupling applications can benefit from the very huge computing, networking and storage resources provided by computational Grids. Software component technology appears to be a very promising technology to handle such applications. However, software component models do not offer an adequate support to embed parallel codes into components.

This paper introduces Padico, a component-based software infrastructure for grid computing; it comprises GridCCM and PadicoTM. GridCCM introduces parallelism inside CORBA components, thus allowing parallel numerical simulation codes to be embedded in CORBA components. PadicoTM enables the deployment of parallel CORBA based applications on grids; it allows them to use several middleware systems (such as CORBA and MPI at the same time) and enables them to transparently utilize all available networks with the appropriate method.

Some issues have not been solved yet. For instance, currently the security is managed through the use of the CORBA security infrastructure which is sometimes too coarse-grained. For example, if two components are placed inside the same parallel machine, we can assume that communications are secure and thus can be optimized by disabling the encryption. However, this issue has still to be investigated. Deployment mechanisms should still be improved. In particular, we investigate the relationship between CCM and Globus [15]: component servers could be deployed within a grid-wide authentication mechanism.

GridCCM is still work in progress; basic examples with GridCCM are working though. Its performance is interesting but we expect a CCM implementation on top of omniORB to achieve truly high performance.

PadicoTM is implemented and works with MPICH, various CORBA 2 and CORBA 3 implementations, gSOAP and the Certi HLA implementation. It is Open Source software and is available at `http://www.irisa.fr/paris/Padico`.

Acknowledgments

This work was supported by the Incentive Concerted Action "GRID" (ACI GRID) of the French Ministry of Research.

References

[1] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The cactus code: A problem solving environment for the grid. In *HPDC*, pages 253–260, 2000.

[2] OmniORB Home Page. AT&T Laboratories Cambridge, `http://www.omniorb.org`.

[3] O. Aumage, L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4):607–626, Apr. 2002.

[4] O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine: a true multi-protocol MPI for high-performance networks. In *Proc. 15th Intl. Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, Apr. 2001. IEEE.

[5] L. Barroca, J. Hall, and P. Hall. *Software Architectures: Advances and Applications*, chapter An Introduction and History of Software Architectures, Components, and Reuse. Springer Verlag, 1999.

[6] L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *LNCS*, pages 468–482, San Juan, Puerto Rico, Apr. 1999. In conj. with IPPS/SPDP 1999, Springer-Verlag.

[7] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *HPDC*, pages 51–59, 2000.

[8] E. Cerami. *Web Services Essentials*. O'Reilly & Associates, 1st edition, Feb. 2002.

[9] K. Chiu, M. Govindaraju, and D. Gannon. The proteus multiprotocol library. In *Proceedings of the 2002 Conference on Supercomputing (SC'02)*, Baltimore, USA, Nov. 2002.

[10] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *LNCS*, pages 1160–1167, Cancun, Mexico, May 2000. In conj. with *IPDPS 2000*, Springer-Verlag.

[11] A. Denis, C. Pérez, and T. Priol. Towards high performance CORBA and MPI middlewares for grid computing. In C. A. Lee, editor, *Proc. of the 2nd Intl. Workshop on Grid Computing*, number 2242 in LNCS, pages 14–25, Denver, Colorado, USA, Nov. 2001. Springer-Verlag. In conj. with *SuperComputing 2001*.

[12] A. Denis, C. Pérez, and T. Priol. PadicoTM: An open integration framework for communication middleware and runtimes. In *IEEE Intl. Symposium on Cluster Computing and the Grid (CCGRID2002)*, 2002.

[13] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-area implementation of the message passing interface. *Parallel Computing*, 24(12):1735–1749, 1998.

[14] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, 1997.

[15] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[16] D. Kurzyniec, V. Sunderam, and M. Migliardi. On the viability of component frameworks for high performance distributed computing: A case study. In *IEEE Int. Symposium on High Performance Distributed Computing (HPDC-11)*, Edimburg, Scotland, July 2002.

[17] M. D. McIlroy. Mass Produced Software Components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO.

[18] Microsoft. Distributed component object model. http://www.microsoft.com/com/.

[19] OMG. CORBA Component Model V3.0. OMG Document formal/02-06-65, June 2002.

[20] OMG. The Common Object Request Broker: Architecture and Specification V3.0. OMG Document formal/02-06-33, June 2002.

[21] C. Pérez, T. Priol, and A. Ribes. A parallel CORBA component model for numerical code coupling. In C. A. Lee, editor, *Proc. of the 3rd Intl. Workshop on Grid Computing*, number 2536 in LNCS, pages 88–99, Baltimore, Maryland, USA, Nov. 2002. Springer-Verlag.

[22] F. Pilhofer. The MICO CORBA component project. http://www.fpx.de/MicoCCM.

[23] T. Rühl, H. Bal, R. Bhoedjang, K. Langendoen, and G. Benson. Experience with a portability layer for implementing parallel programming systems. In *Proceedings of the Intl. Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, USA, Aug. 1996.

[24] D. C. Schmidt. An architectural overview of the ACE framework: A case-study of successful cross-platform systems software reuse. *USENIX login magazine, Tools special issue*, Nov. 1998.

[25] Sun. Enterprise java beans. http://java.sun.com/products/ejb/.

[26] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.

[27] M. Vadet, P. Merle, R. Marvie, and J.-M. Geib. The OpenCCM platform. http://www.objectweb.org/openccm/index.html.