Programming the grid with components: models and runtime issues

Alexandre Denis, Sébastien Lacour, Christian Pérez, Thierry Priol, André Ribes

▶ **To cite this version:**

Alexandre Denis, Sébastien Lacour, Christian Pérez, Thierry Priol, André Ribes. Programming the grid with components: models and runtime issues. Beniamino Di Martino and Jack Dongarra and Adolfy Hoisie and Laurence T. Yang and Hans Zima. Engineering The Grid: Status and Perspective, American Scientific Publishers, 2006, 1-58883-038-1. inria-00411008

HAL Id: inria-00411008
https://inria.hal.science/inria-00411008

Submitted on 25 Aug 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PROGRAMMING THE GRID WITH COMPONENTS: MODELS AND RUNTIME ISSUES*

ALEXANDRE DENIS[1] , SÉBASTIEN LACOUR[2] , CHRISTIAN PÉREZ[3] , THIERRY PRIOL[4] , AND ANDRÉ RIBES[5]

[1] INRIA-Futurs, Alexandre.Denis@inria.fr
[2] IRISA/INRIA, Sebastien.Lacour@irisa.fr
[3] IRISA/INRIA, Christian.Perez@irisa.fr†
[4] IRISA/INRIA, Thierry.Priol@irisa.fr
[5] IRISA/INRIA, Andre.Ribes@irisa.fr

Abstract. This paper describes different aspects of our research activities aiming at designing a component-based software infrastructure, called Padico, for computational Grids. Several problems were identified as obstacles to designing such an infrastructure. This paper addresses some of them especially a suitable software component model, capable of encapsulating parallel codes (GridCCM), a communication framework (PadicoTM) allowing software components to communicate between each other taking advantage of various networking technologies available within a grid infrastructure and a network topology description model, permitting an automatic deployment of software components by capturing real-world grid network topologies.

Key words. component model, middleware, runtime, component deployment

AMS subject classifications.

1. Introduction. Programming distributed systems has always been seen as a tedious activity for a programmer. A Grid infrastructure, as the latest incarnation of distributed systems, is no exception to this reality. In addition to the associated coding activity, a programmer often has to deal with low-level programming and runtime issues such as communications between different modules of the application or deployment of modules among a set of available resources. To cope with this problem, several approaches have been pursued to make the programming task easier. Some well known approaches such as *Remote Procedure Call* or *Distributed Objects* allowed usual programming paradigms (function call or objects) to be applied to transparently invoke a function of a remote program or a method of a remote object. Despite some success stories, these two approaches have not been able to reduce the complexity of distributed programming to an acceptable level. Moreover, the automatic deployment of distributed applications is still an important issue.

A third approach based on the composition of software modules has gained acceptance in the last few years. Instead of following an object-oriented approach, and its associated inheritance mechanism, a component approach enforces composition as the main paradigm to develop distributed applications. It offers the advantage of decreasing the design complexity and improves productivity by facilitating software re-use. Such an approach can be applied to programs running on Grid infrastructures. Moreover, a Grid infrastructure, and its associated services to manage distributed resources, is well suited to deploy software components by placing them on available resources taking into account various constraints. It is worth noting that deployment of distributed software components is the missing feature of most of the available distributed component models. Therefore, we think that associating component programming with the Grid is of mutual benefit: making Grid programming easier and

†Corresponding author
*This work was supported by the French ACI GRID initiative

deploying software components in a transparent way. Thus, this will insure a larger success of these two promising technologies.

The focus of this paper is to apply component programming to scientific computing, especially multi-physics applications. Such applications aim to simulate various physics, each of them being implemented by a dedicated code, to increase the accuracy of simulation. It is becoming clear that a radical shift in software development should occur to handle the increasing complexity of such applications. Moreover, the computing infrastructure should provide the level of performance in order to run such applications within a reasonable time frame. A computational grid is by no doubt a computing infrastructure that could deliver this level of performance by combining together high-performance computing resources connected to the Internet.

However, modern software development approaches are often suspected of not providing the level of performance which high-performance computing systems would offer. If we consider the use of a component programming methodology for the design and the implementation of multi-physics applications, and the use of a Grid infrastructure for their execution, several obstacles can be foreseen. The first one is the suitability of existing component models for the encapsulation of scientific simulation codes within software components. It may often be the case that such codes are parallel (mostly SPMD) whereas component models are not designed to encapsulate SPMD parallel codes in an efficient way. The second obstacle is communications between software components. Within a grid infrastructure, there may be several networking technologies from System-Area Networks (SAN) to Wide-Area Networks (WAN). The use of the Internet's lingua franca TCP/IP jointly with a SAN is probably not the best way to exploit all this networking technology. Moreover, several communication middleware or runtime environements will have to work together in a seamless way to ensure communication within a component (parallelism) or between components (distribution). A third obstacle is the deployment of components within a grid infrastructure. Such a deployment should be made transparent to the users taking into account end-user constraints. An end-user should not have to map components onto available grid resources by himself. A grid middleware should manage this operation in an automatic way on behalf of the end-user. We think that these three obstacles represent the major ones and should be addressed by computer scientists.

The objective of this paper is to present some solutions to overcome these three obstacles. Within the framework of the Padico project, we carried out three research activities, each of them addressing an obstacle. Section 2 addresses the first one: the design of a component model for the grid (GridCCM) based on an existing one (Corba Component Model or Ccm). Section 3 presents a communication framework, called PadicoTM, allowing several communication middleware and runtime environments to work together by isolating them and allowing for various networking technologies to be shared. Section 4 explains the process of deploying components within a Grid and the required extension that should be made to existing Grid middleware such as the Globus Toolkit [9]. Finally, section 5 draws general conclusions and mentions perspectives of this work.

2. A Grid Component Model. The component model we describe in this section is based on the Corba Component Model instead of designing a new one. We think that such a decision offers more advantages than drawbacks. Using an existing component model allows us to take benefit of all the work that has been done both on the design of the model itself and its realization through several open source implementations. We propose here some extensions to the Corba Component Model

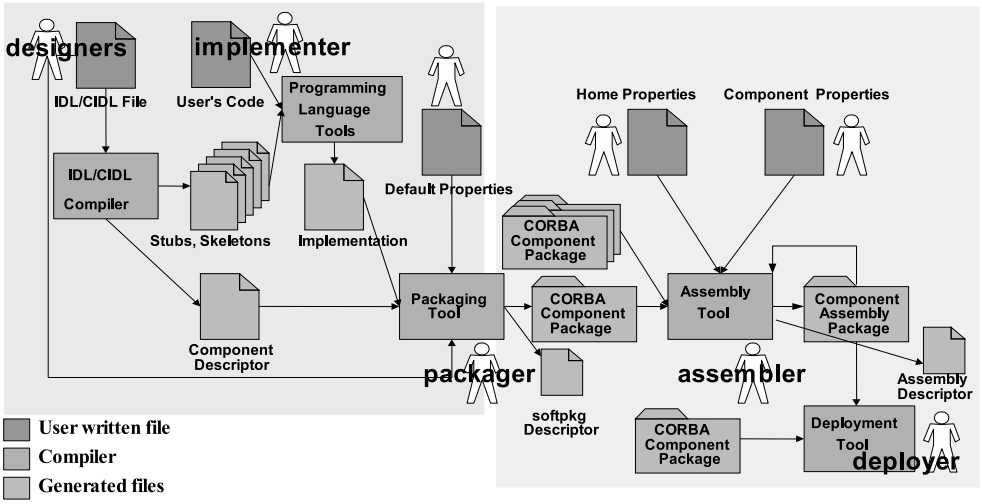


FIGURE 1. Overview of the CORBA Component Model.

that do not require the modification of the OMG specification. Before introducing such extensions, we give a brief overview of CCM in the following sections.

2.1. Overview of the CORBA Component Model. The CORBA Component Model [18] (CCM) is part of the latest CORBA [19] specifications (version 3). The CCM specifications allow the deployment of components into a distributed environment, that is to say that an application can deploy interconnected components on different heterogeneous servers in one operation. Figure 1 presents the general picture of CCM. The component life-cycle is divided into two parts. First, the creation of the component requires to define the component interface, to implement it and then to package it so as to obtain a component package, *i.e.* a component. The second part consists in (optionally) linking together several components into a component assembly and in deploying it. CCM provides a model for all theses phases. For example, the CCM abstract model deals with the external view of a component, while the Component Implementation Framework (CIF) provides a model to implement a component. There are also models for packaging and deploying a component, as well as for the local runtime environment of a component. In this section, we briefly introduce the abstract model, the execution model and the deployment model.

2.2. CCM Abstract Model. A CORBA component is represented by a set of ports described in the Interface Definition Language (IDL) of CORBA 3 defined by the OMG. The IDL of CORBA 3 is an extension of the IDL of CORBA version 2 by the OMG. There are five kinds of ports as shown in Figure 2. *Facets* are named connection points that provide services available as interfaces while *receptacles* are named connection points to be connected to a facet. They describe the component's ability to use a reference supplied by some external agent. *Event sources* are named connection points that emit typed events to one or more interested event consumers, or to an event channel. *Event sinks* are named connection points into which events of a specified type may be pushed. *Attributes* are named values exposed through accessor (read) and mutator (write) operations. Attributes are primarily intended to be used for component configuration, although they may be used in a variety of other

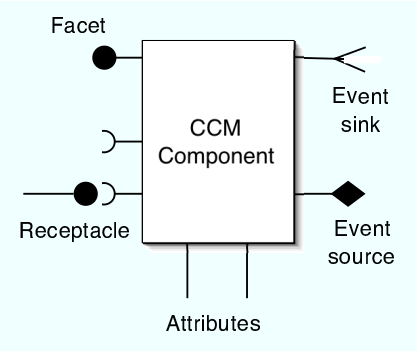


FIGURE 2. A CCM component.

```
// Interface Average definition
typedef sequence<double> Vector;
interface Average {
  double compute(in Vector v);
};
// Component A definition
component aComponent {
  attribute string  name;
  provides  Average avgPort;
  uses      Display dspPort;
};
```

FIGURE 3. A component IDL definition.

```
aComponent ref = ServerComp->provide_avgPort();
ClientComp->connect_avgClientPort(ref);
```

FIGURE 4. Example of code to connect two components.

ways. Figure 3 shows an example of component definition using IDL3.

Facets and receptacles allow a synchronous communication model based on the remote method invocation paradigm. An asynchronous communication model based on data transfer is implemented by the event sources and sinks.

A component is managed by an entity named *home*. A home provides factory and finder operations to create and/or find a component instance. For example, a home exposes a `create` operation which locally creates a component instance.

2.3. CCM Execution Model. CCM uses a programming model based on containers. Containers provide the run-time environment for CORBA components. A container is a framework for integrating transactions, security, events, and persistence into a component's behavior at runtime. Containers provide a standard set of services to a component, enabling the same component to be hosted by different container implementations. All component instances are created and managed at runtime by its container.

2.4. CCM Deployment Model. The deployment model of CCM is fully dynamic: a component can be dynamically connected to and disconnected from another component. For example, Figure 4 illustrates how a component **ServerComp** can be connected to a component **ClientComp** through the facet **FacetExample**: a reference is obtained from the facet and then it is given to a receptacle. Moreover, the model supports the deployment of a static application. In this case, the assembly phase has produced a description of the initial state of the application. Thus, a deployment tool can deploy the components of the application according to the description. It is worthwhile to remark that it is just the initial state of the application: the application can change it by modifying its connections and/or by adding/removing components.

The deployment model relies on the functionality of some fabrics to create component servers which are hosting environments of component instances. The issues of determining the machines where to create the component servers and how to actually create them are *out of the scope* of the CCM specifications.

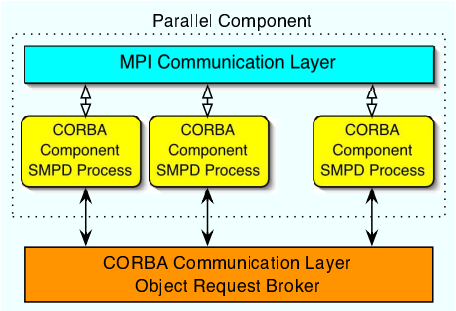


FIGURE 5. Parallel component concept.

2.5. Adapting CCM to the Grid. The main problem 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 the right solution: it may require non trivial modifications to the the parallel code and the master node may become a communication bottleneck in parallel to parallel component communications. This problem is address by GridCCM.

2.5.1. Introducing Parallelism into CCM. This section presents GridCCM, an extension of the CORBA Component Model. It adds the concept of parallel components to CCM. Its objective is to allow an efficient encapsulation of parallel codes into GridCCM components. We currently restrict ourselves to embed SPMD (*Single Program Multiple Data*) codes[1]. 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 to the model. That is why, we do not allow ourselves to do any change to the CORBA Interface Definition Language (IDL). In the same way, a parallel component has to be interoperable with a standard sequential component.

Figure 5 presents a parallel component in the CORBA framework. The SPMD code may use MPI for its inter-process communications; it uses CORBA to communicate with other components. In order 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 are exposed in [21].

2.5.2. Managing the Parallelism. To introduce parallelism support, 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 6.

A call to a parallel operation of a parallel component is intercepted by this new layer. The layer 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,

[1]This choice stems from two considerations. First, many parallel codes are indeed SPMD. Second, SPMD codes bring an easily manageable execution model.

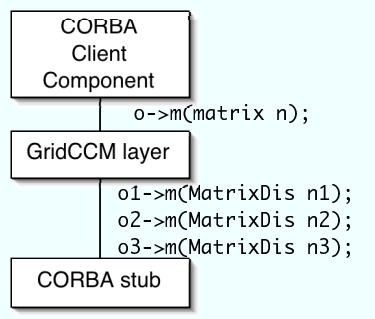


FIGURE 6. GridCCM intercepts and translates remote method invocations.

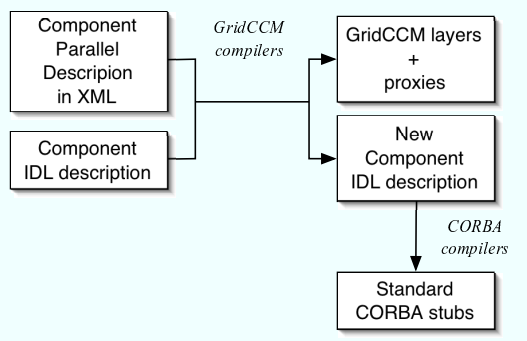


FIGURE 7. Compilation steps to generate a parallel component.

as illustrated in Figure 7. This compiler uses two files: an IDL description of the component and an XML description of the component parallelism. In order to have a transparent layer, a new IDL description is generated during the generation of the component. GridCCM layer internally uses an interface derived from the original interface. The new IDL interface is the interface that is remotely invoked 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. Because of this transformation, there are some constraints about 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 for multidimensional arrays: a 2D array can be mapped to a sequence of sequences.

2.5.3. Preliminary Implementation of GridCCM. We have implemented a preliminary prototype of GridCCM on top of two existing CCM implementations: OpenCCM [23] and MicoCCM [20]. Our first prototype has been derived from OpenCCM [23]. OpenCCM is developed at the research laboratory LIFL (*Laboratoire d'Infomatique Fondamentale de Lille*) and is written in Java. The second prototype has been derived from MicoCCM [20]. MicoCCM is an *OpenSource* implementation based on the Mico ORB and is written in C++. Considering the second prototype, we have shown that GridCCM is able to efficiently aggregate the bandwidth allowing parallel components to best use the underlying network. Some experiments have been done with a WAN, called VTHD [24], a French high-bandwidth WAN that connect several clusters located in several INRIA research units. A bandwidth of 103 MB/s (820 Mb/s) was obtained (using a 1 Gbit/s link) between two clusters, each of them running a parallel component encapsulating a parallel code running on 11 cluster nodes.

3. A communication Framework for Software Components. 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. Unfortunately, existing CORBA

implementations are able neither to use a wide range of networks nor to be used beside MPI. Therefore we designed a communication framework to cope with these issues. The important features which should be supported by grid-enabled middleware systems are:

Transparency — The middleware systems used by an application should be able to transparently and efficiently use the available resources. For example, a MPI, PVM, Java or CORBA communication should be able to utilize high speed networks (SAN) as well as local area networks (LAN) and wide area networks (WAN). Moreover, they should adapt their security requirements to the characteristics of the underlying network, *e.g.* if the network is secure, it is useless to cipher data.

Flexibility — There is a diversity of middleware systems, and we can assume there will always be. It seems important not to tie grid applications to a specific grid framework but instead to ease the "gridification" of middleware systems.

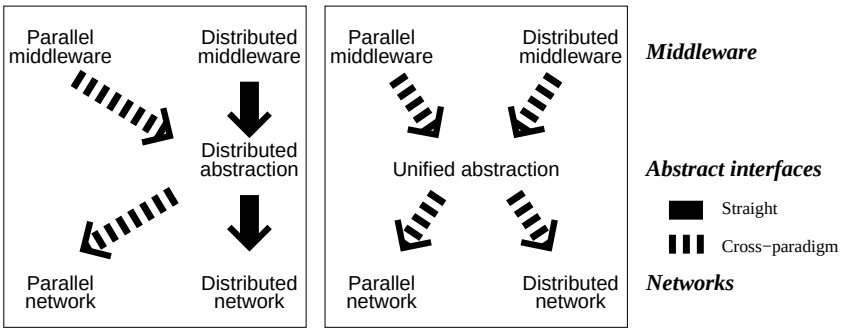
Interoperability — Grids are not a closed world. Grid applications will need to be accessible using standard protocols. So, there is a high need to keep protocol interoperability.

Support Multiple Communication Paradigms — GridCCM requires several middleware systems, *e.g.* MPI and CORBA. Thus, it is important to allow different middleware systems to be used simultaneously.

We introduce the PadicoTM [7, 6] communication framework which is able to deal with these problems. PadicoTM is based on 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 communication paradigms at every level, even in the abstraction layer. There is therefore no "bottleneck of features" as depicted in Figure 8. The following sections give a presentation of the different layers of PadicoTM as shown in Figure 9.

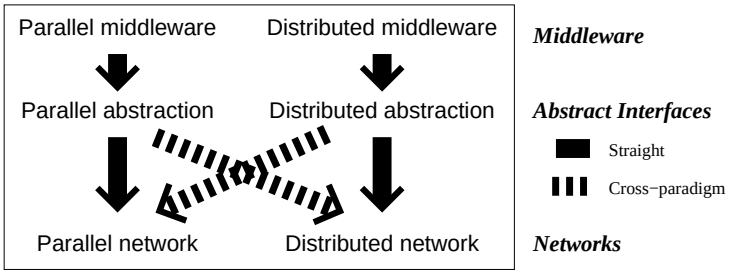
3.1. Arbitration Issues. Supporting CORBA and MPI, *both running simultaneously* in the same process using the same network, is not straightforward. Access to high-performance networks is the most conflict-prone task when using multiple middleware systems at the same time. We propose that arbitration should be dealt for at the lowest possible level, so as to build more advanced abstractions atop a fully reentrant system. Arbitration is performed by a layer which provides a consistent, reentrant and multiplexed access to every networking resources, each resource is utilized with the most appropriate driver and method. The arbitrated interfaces are designed for efficiency and reentrance. Thus, we propose these API to be callback-based (*à la* Active Message). For true arbitration, this layer is the only client of the system-level resources: all accesses to the network should be performed through the arbitration layer. It provides also arbitration between different networks (*e.g. Myrinet* against Ethernet) so that they do not bother each other, and between different middleware systems even if the communication library does not provide multiplexing. More details about cooperative access rather than competitive are given in [5].

The arbitration layer in PadicoTM is called *NetAccess*, which contains two subsystems: *SysIO* for access to system I/O (sockets, files), and *MadIO* for multiplexed access to high-performance networks. The *core* of *NetAccess* manages the threads with the polling loops and enforces fairness between *SysIO* and *MadIO*. The interleaving



(a) Everything expressed through a single abstraction (distributed) — two cross-paradigm translations are needed for a parallel middleware atop a parallel network.

(b) A unified abstraction makes compromises in all cases! — gives up most possible optimizations and imposes compromises to everything.



(c) Dual-abstraction model: use different abstract interfaces for different paradigms — only required compromises are done.

FIGURE 8. Several abstraction models may be envisaged.

policy between *SysIO* and *MadIO* is dynamically user-tunable through a configuration API to give more priority to system sockets or high performance network depending on the application. *NetAccess* is open enough so as to allow the integration of other subsystems beside *MadIO* and *SysIO* for other paradigms such as Shmem on SMP for example.

3.1.1. *NetAccess MadIO*: API for Accessing Parallel-oriented Hardware. For good I/O reactivity and portability over high performance networks, we have chosen the high-performance network library Madeleine [1] as a foundation. Madeleine is used for high-performance networks such as *Myrinet*, SCI, VIA. Madeleine provides no more multiplexing channels than what is allowed by the hardware (*e.g.* 2 over *Myrinet*, 1 over SCI). *MadIO* adds a logical multiplexing/demultiplexing facility which allows an arbitrary number of communication channels. Multiplexing on top of Madeleine adds a header to all messages. This can significantly increase the latency if not done properly. We implement *headers combining* to aggregate headers from several layers into a single packet. Thus, multiplexing on top of Madeleine adds virtually no overhead to middleware systems which send headers anyway. We actually measure that the overhead of *MadIO* over plain Madeleine is less than 0.1 μs which is imperceptible on most current networks.

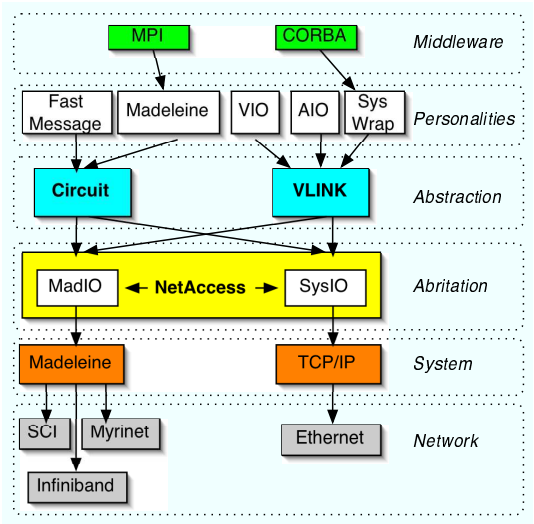


FIGURE 9. The PadicoTM communication framework.

3.1.2. *NetAccess SysIO*: API for Accessing Distributed-oriented Hardware. Contrary to a widespread belief, using directly the socket API from the OS does not bring full reentrance, multiplexing and cooperation. Several middleware systems not designed to work together may get into troubles when used simultaneously, even with only plain TCP/IP. There are reentrance issues for signal-driven I/O (used by middleware systems designed to deal with heavy load), which results in an incorrect behavior, or worst, in a crash. If a middleware system uses blocking I/O and another uses active polling, the one which does active polling holds near 100 % of the CPU time; it will result in inequity or even deadlock. To solve these conflicts, *SysIO* manages a unique receipt loop that scans the opened sockets and calls user-registered callback functions when a socket is ready. The callback-basedness guarantees that there is no reentrance issue nor signals to mangle with.

3.2. Abstraction Layer. On top of the arbitration layer, the abstraction layer 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.

3.2.1. A dual-abstraction model. A wide-spread design for communication frameworks consists in providing a unique abstraction on which several middleware systems may be built (see Figure 8 *a* and *b*). 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 (Figure 8 *c*). 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.

The abstract interfaces in PadicoTM are called *VLink* for distributed computing, and *Circuit* for parallelism.

3.2.2. Distributed abstract interface: *VLink*. The *VLink* interface is designed for distributed computing. It is client/server-oriented, supports dynamic connections, and streaming. In order to easily allow several personalities —both synchronous and asynchronous personalities—, *VLink* is based on a flexible asynchronous API. This API consists in five primitive operations —`read`, `write`, `connect`, `accept`, `close`. These functions are asynchronous: when they are invoked, they initiate (*post*) the operation and may return before completion. Their completion may be tested at any time by polling the *VLink* descriptor; a handler may be set which will be called upon operation completion. Such a set of functions is called a *VLink*-driver. *VLink* drivers have been implemented on top of: *MadIO*, *SysIO*, Parallel Streams for WAN, *AdOC* [14], *loopback*.

3.2.3. Abstract interface for parallelism: *Circuit*. The *Circuit* interface is designed for parallelism. It manages communications on a definite set of nodes called a *group*. A *group* may be an arbitrary set of nodes, *e.g.* a cluster, a subset of a cluster, may span across multiple clusters or even multiple sites. *Circuit* allows communications from every node to very other node through an interface optimized for parallel runtimes: it uses incremental packing with explicit semantics to allow on-the-fly packet reordering, like in Madeleine [1]. Collective operations in *Circuit* still needs to be investigated. *Circuit* adapters have been implemented on top of *MadIO*, *SysIO*, *loopback* and *VLink* (to use the *alternates VLink* adapters); a given instance of *Circuit* can use different adapters for different links.

3.3. Personality Layer and Middleware Systems. The middleware systems likely to be used by grid-enabled applications are various: MPI, CORBA, SOAP, HLA, JVM, PVM, etc. Moreover, for each kind of middleware, there are several implementations which have their own specific properties. Developing a middleware system is a heavy task —for example, MPICH contains 200,000 lines of C— and requires very specific skills. Moreover, the standards —and thus, the middleware systems themselves— are ever-changing. It does not seem reasonable to re-develop an implementation of each one of these middleware systems specifically for a given communication framework. Instead of adapting the middleware systems to our communication framework, we adapt our communication framework to the expectation of the existing middleware systems. Thus it is easy to follow the new versions and to use specific features of a given implementation.

To seamlessly re-use existing implementations of middleware systems, we choose to virtualize networking resources. It consists in giving the middleware system the illusion that it is using the usual resource it knows, even if the *real* underlying resource is completely different. For example, we show a "socket" API to a CORBA implementation so as to make it believe it is using TCP/IP, even if it is actually using another protocol/network behind the scene. This is performed through the use of thin wrappers on top of the appropriate abstract interface to make it look like the required API. We call these small wrappers *personalities*. It is possible to give several personalities to an abstract interface.

PadicoTM provides several well-known API through simple "cosmetics" adapters

over the *VLink* and *Circuit* abstract interfaces. These thin API wrappers are called *personalities*. The personalities for *VLink* are: *Vio* for an explicit use through a socket-like API; *SysWrap* supplies a 100 % socket-compliant API through wrapping at runtime, binary-compatible with C, C++ or FORTRAN legacy codes without even recompiling. Thus, legacy applications are able to transparently use all PadicoTM communication methods without losing interoperability with PadicoTM-unaware applications on plain sockets. We implement an AIO personality on top of *VLink* which provides a plain Posix.2 Asynchronous I/O (AIO) API. Thin adapters on top of *Circuit* provides a FM 2.0 API, and a (virtual) Madeleine API.

Thanks to *SysWrap*, various middleware systems have been seamlessly ported on PadicoTM with absolutely no change in their code: CORBA implementations (omniORB 3, omniORB 4, ORBacus 4.0, all Mico 2.3.x including CCM-enabled versions), an HLA implementation (*Certi* from the *Onera*), and a SOAP implementation (gSOAP 2.2). A Java virtual machine (Kaffe 1.0.7) has been slightly modified for use within PadicoTM, with some changes in its multi-threading management code. Thanks to the virtual Madeleine personality, the existing MPICH/Madeleine [2] implementation can run in PadicoTM. The middleware systems are dynamically loadable into PadicoTM. Arbitration guarantees that any combination of them may be used at the same time.

4. Deploying Components on a Computational Grid. One of the long run goals of computational grids is to provide computer power in the same way as the electric power grid supplies electric power [11], *i.e.* transparently. Here, *transparency* means that the user does not know what particular resources provide electric or computational power. So the user should just have to submit his or her application to a computational grid and get back the result of the application without worrying about resource selection, resource location, or mapping processes on resources. In other words, application deployment should be as *automatic* and easy as plugging an electric device into an electric outlet.

Automatic deployment of component-based applications is crucial for better acceptance of the component-based programming model as well as for the success of computational grids which can host various types of applications (parallel, distributed, *etc.*). One of the advantages of the CORBA Component Model (CCM, [18]) is that it specifies both a *packaging model* and *deployment model*. However, CCM does not say how execution hosts may be selected, nor how processes may be launched on computers from a practical viewpoint.

To really achieve automatic deployment, we need both a description of the computational grid which we have access to plus a packaged application (Figure 10). Those two pieces of information are given to a *deployment tool* which selects resources and actually launches the application on the selected, distributed resources of the grid. The following sections provide further details on the different steps of automatic deployment: application and resource information description, deployment planning, actual execution and configuration of the application.

4.1. Application and Resource Information Description. Two pieces of information are required for automatic deployment: a description of the component-based application to be deployed and a description of the grid resources in which the application may be deployed.

4.1.1. Component-Based Application Description. Within the context of CCM, an application is made of a set of components, called a component assembly

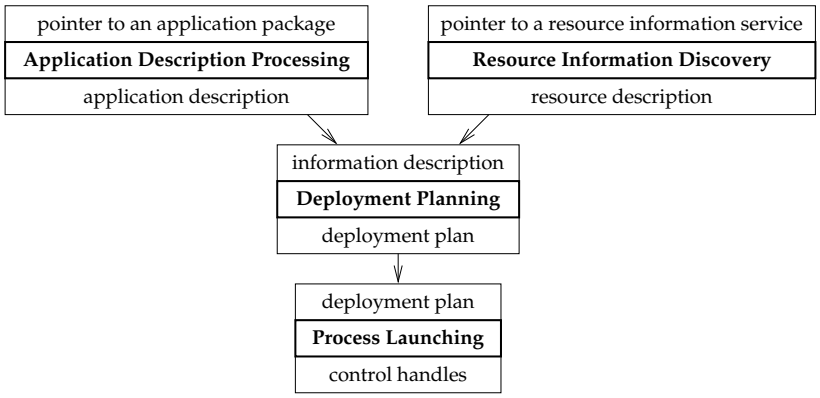


FIGURE 10. Overview of the deployment architecture.

package. This is a compressed archive provided by the user to the deployment tool. It includes, among other files, the assembly description which describes all the components of the assembly and their interconnections, as well as initial configuration parameters.

The assembly and component descriptors can express various requirements such as the processor architecture and the operating system required by a component implementation. A component may have environmental or other dependencies, like libraries, executables, Java classes, *etc.* Another possible requirement is component collocation: components may be free or partitioned to a single process or a single host, meaning that a group of component instances will have to be deployed in the same process or on the same compute node.

The component deployment tool must make sure that those constraints and dependencies will be satisfied at execution time.

4.1.2. Grid Resource Information Description. Before *automatically* deploying the processes of a distributed application on a computational grid, the compute nodes on which the application will be run must be selected *automatically*. In order for the deployment tool to make wise decisions in selecting computers, grid resources must be described precisely.

Information about grid resources includes not only compute and storage resource information, but also *network* description. Network information is important for high-performance applications in particular: resource selection may be constrained by such computer-level and network-level requirements as "I want 32 computers connected by a network of at least 2 Gb/s, like Myrinet".

Compute and storage resource description is rather well mastered (computer architecture, number and speed of CPUs, operating system, memory size, storage capacity, *etc.*), as exemplified by MDS2 (Monitoring and Discovery Service, [3]), the Grid Information Service of the Globus Toolkit version 2 [12]. However, network description received less attention. Simple networks should be described in a simple way, but the description model should allow for the description of complex networks including firewalls, NAT (Network Address Translation), asymmetric links (like asymmetric bandwidths), non-hierarchical topologies, connection to multiple networking technologies (Myrinet, Ethernet, *etc.*). We have proposed [15] a scalable description model of grid network topology (as shown in Figure 11) and have implemented it on top

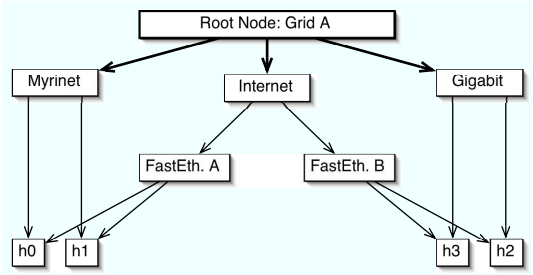


FIGURE 11. Network graph describing the topology of a sample Grid.

of MDS2. The main idea is to group compute nodes together within network groups where network characteristics are roughly similar (bandwidth, latency, jitter, loss rate, *etc.*). This results in a synthetic description of grid resources, including both compute nodes and network.

4.2. Deployment Planning. Once application and resource information has been retrieved, the deployment planner [16] is responsible for 1) selecting resources to run the application, 2) selecting the network links (or network technology) to interconnect the application components, and 3) mapping the application processes (or CORBA component servers) onto the selected resources. The input of the deployment planning algorithm is made of the application description and the resource description. The application description represents a set of constraints which must be satisfied by the selected resources.

The output of the deployment planner is a deployment plan which describes the mapping of the components onto component servers and the mapping of the component servers onto the selected compute nodes of the computational grid. The deployment plan should also specify 1) in what order processes must be launched by the deployment tool, 2) how data must flow from the output of certain processes to the input of other processes, 3) what network connections must be established between every pair of processes. For instance, items 1) and 2) are necessary for CORBA applications, where a Naming Service needs to be launched, and its reference needs to be passed to the component servers launched afterwards.

4.3. Actual Deployment of Components Using the Globus Toolkit. Once a deployment plan has been obtained from the previous step, the component-based application is launched and configured according to the CORBA component model. The technical point is that the selected computers are assumed not to run any component activator or component server. That is the reason why a job submission method is needed to launch an initial process on the selected compute nodes.

This step is fully compatible with the CCM deployment model as explained in [17]. For example, we have developed a prototype called ADAGE: Automatic Deployment of Applications in a Grid Environment. It is able to deploy standard CORBA component using the Globus Toolkit version 2 [12, 10]. The Globus Toolkit is an open source software toolkit used for building grids. It includes software for security enforcement, resource, information, and data management. This middleware is wide-spread and well-established, as exemplified by many projects relying on the Globus Toolkit, such as GriPhyN (the Grid Physics Network, [13]), the American DOE Science Grid [8], the European DataGrid project [4], TeraGrid [22].

As shown on Figure 10, the deployment tool manages two sorts of handles: CORBA

references and handles returned by the grid access middleware (the Globus Toolkit). Both are useful to control application processes, like cancel, suspend, or restart their execution.

5. Conclusion. The deployment of high bandwidth wide-area networks has led computational grids to offer a very powerful computing resource. In particular, this inherently distributed resource is well-suited for multi-physics applications. To face the complexity of such applications as well as the heterogeneity and volatileness of grids, the software component technology appears to be a very adequate programming model. We choose to work with the CORBA component model because its deployment model is very complete: it specifies the deployment of a set of components on a set of distributed (component) servers. However, it does not handle very well some aspects that are inherent to Grid infrastructures: managing parallelism within a component, heterogeneity of networks and automatic deployment of components onto available compute nodes. It specifies neither how to select resources, nor how to initiate component servers on the selected resources. On the other hand, a grid access middleware, such as the Globus Toolkit, deals with security enforcement, resource, information, data management, and portability.

This paper presents some solutions to those problems. We have shown that managing parallelism within a CORBA component, while maintaining scalable connection between components, can be done without modifying the OMG specification. Then, grid programmers can rely on an existing model, from the OMG, which is widely accepted in some application fields, but not yet within the grid user community, to be honest. Concerning network resources, we propose a framework that is capable of virtualizing various networking technologies and their associated protocols. It allows components to communicate between each other without taking care of the underlying networks. Thus a distributed application based on software components can be deployed independently of network resources. It can be executed anywhere while fully taking advantage of the performance characteristics of the underlying network. Moreover, several communication middleware or runtime systems can be used simultaneously within a component without suffering from any side effect or unexpected behavior. Our framework is able to share network resources even if they were not designed to be shared. This will give much flexibility for the deployment of components on Grid infrastructures. As for instance two components exchanging a large amount of data can be mapped onto a set of compute nodes interconnected over a very high-performance network such as Myrinet. However to do so, it requires that the Grid middleware managing the Grid infrastructure be aware of the presence of such networks as well as the topology of these networks. We proposed to extend existing information services to store information related to network resources (topologies, network technologies, *etc.*). Using such extensions, it is possible to allocate resources and to propose a mapping of components to those resources in an automatic way depending on the user's constraints and requirements. Most of our efforts are now devoted to integration of the results presented in this paper with the Globus Toolkit.

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

REFERENCES

[1] O. AUMAGE, L. BOUGÉ, A. DENIS, J.-F. MÉHAUT, G. MERCIER, R. NAMYST, AND L. PRYLLI, *A portable and efficient communication library for high-performance cluster computing*, in

IEEE Intl Conf. on Cluster Computing (CLUSTER 2000), Technische Universität Chemnitz, Saxony, Germany, Nov. 2000, pp. 78–87.

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

[3] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, *Grid information services for distributed resource sharing*, in Proc. of the 10th IEEE International Symp. on High-Performance Distributed Computing (HPDC-10'01), San Francisco, California, Aug. 2001, pp. 181–194.

[4] *The DataGrid Project web site*. `http://www.eu-datagrid.org/`.

[5] A. Denis, C. Pérez, and T. Priol, *Towards high performance CORBA and MPI middlewares for grid computing*, in Proc of the 2nd International Workshop on Grid Computing, G. A. Lee, ed., no. 2242 in LNCS, Denver, Colorado, USA, Nov. 2001, Springer-Verlag, pp. 14–25. In conjunction with *SuperComputing 2001 (SC'01)*.

[6] A. Denis, C. Pérez, and T. Priol, *Network communications in grid computing: At a crossroads between parallel and distributed worlds*, in 18th International Parallel and Distributed Processing Symposium (IPDPS2004), Santa Fe, New Mexico, Apr. 2004, IEEE Computer Society, p. 95a.

[7] A. Denis, C. Pérez, and T. Priol, *Padicotm: An open integration framework for communication middleware and runtimes*, Future Generation Computer Systems, 19 (2003), pp. 575–585.

[8] *The DOE Science Grid web site*. `http://DOEScienceGrid.org/`.

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

[10] I. Foster and C. Kesselman, *The Globus Project: a status report*, in Proc. of the 7th Heterogeneous Computing Workshop, held in conjunction with IPPS/SPDP'98, Orlando, FL, Mar. 1998, pp. 4–18.

[11] I. Foster and C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, Inc, 1998.

[12] The Globus Alliance: `http://www.globus.org/`.

[13] *The Grid Physics Network (GriPhyN) web site*. `http://www.GriPhyN.org/`.

[14] E. Jeannot, B. Knutsson, and M. Bjorkmann, *Adaptive online data compression*, in IEEE High Performance Distributed Computing (HPDC'11), Edinburgh, Scotland, July 2002.

[15] S. Lacour, C. Pérez, and T. Priol, *A network topology description model for grid application deployment*, in Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID 2004), Pittsburgh, PA, USA, Nov. 2004. Held in conjunction with Supercomputing 2004 (SC2004).

[16] S. Lacour, C. Pérez, and T. Priol, *A software architecture for automatic deployment of CORBA components using grid technologies*, in Proceedings of the 1st Francophone Conference On Software Deployment and (Re)Configuration (DECOR'2004), Grenoble, France, Oct. 2004, pp. 187–192.

[17] S. Lacour, C. Pérez, and T. Priol, *Deploying CORBA components on a computational grid: General principles and early experiments using the Globus Toolkit*, in Proceedings of the 2nd International Working Conference on Component Deployment (CD 2004), W. Emmerich and A. L. Wolf, eds., vol. 3083 of LNCS, Edinburgh, Scotland, UK, May 2004, Springer-Verlag, pp. 35–49. Held in conjunction with the 26th International Conference on Software Engineering (ICSE 2004).

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

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

[20] F. Pilhofer, *The MICO CORBA component project*. `http://www.fpx.de/MicoCCM`.

[21] C. Pérez, T. Priol, and A. Ribes, *A parallel CORBA component model for numerical code coupling*, in Proc. of the 3nd International Workshop on Grid Computing, C. A. Lee, ed., LNCS, Baltimore, Maryland, USA, Nov. 2002, Springer-Verlag. to appear.

[22] *The TeraGrid web site*. `http://www.TeraGrid.org/`.

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

[24] *The VTHD project*. http://www.vthd.org.