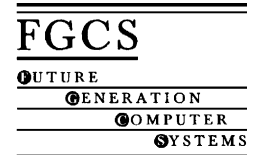




ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE @ DIRECT®

Future Generation Computer Systems 19 (2003) 575–585



www.elsevier.com/locate/future

PadicoTM: an open integration framework for communication middleware and runtimes[☆]

Alexandre Denis^{a,*}, Christian Pérez^b, Thierry Priol^b

^a IRISA/IFSIC, Campus de Beaulieu, 35042 Rennes Cedex, France

^b IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract

Computational grids are seen as the future emergent computing infrastructures. Their programming requires the use of several paradigms that are implemented through communication middleware and runtimes. However some of these middleware systems and runtimes are unable to take benefit of specific networking technologies available in grid infrastructures. In this paper, we describe an open integration framework that allows several communication middleware and runtimes to efficiently share the networking resources. Such framework encourages grid programmers to use the most suited communication paradigms for their applications independently from the underlying networks. Therefore, there is no obstacle to deploy the applications on a specific grid configuration.

© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Communication framework; Middleware integration; Code coupling; High-performance network

1. Introduction

As parallel and distributed systems are merging into a single computational infrastructure called the grid, it is foreseen that the programming of such an infrastructure will require the use of several communication paradigms in a combined and coherent way. Indeed, the availability of grid infrastructures will encourage the development of new applications in the field of scientific computing that was unthinkable some years ago. With the availability of such an amount of computing power, it is now envisaged to simulate more complex physical phenomena. For instance, the sim-

ulation of all physical phenomena that are involved in the design of an aircraft requires the coupling of a large number of simulation codes, in the fields of structural mechanics, computational fluid dynamics, electromagnetism, etc. Each code has its own requirement in term of computing resources (visualization, parallel or vector computers). The codes that compose such an application are generally independently developed. It appears very constraining to require that all codes are based on the same communication paradigm, like MPI, to be able to run on a computational grid. It is more likely that each simulation code has its own requirement in terms of execution support. Some of them are based on message-passing, some others require a shared memory abstraction (either a physical memory or a distributed shared memory). Moreover, the coupling of simulation codes requires the use of specific communication paradigms to transfer both data and control, such as Remote Procedure

[☆] Supported by the incentive concerted action “GRID” (ACI GRID) of the French Ministry of Research.

* Corresponding author.

E-mail addresses: alexandre.denis@irisa.fr (A. Denis), christian.perez@irisa.fr (C. Pérez), thierry.priol@irisa.fr (T. Priol).

Call (RPC) or Remote Method Invocation (RMI). CORBA or Java RMI are good candidates to support the coupling of codes. However, there exists several obstacles that discourage programmers from using the available communication paradigms in their applications. Thus, they are forced to choose one against the others even if it is not the most suitable one.

The first obstacle is that most implementations of the communication paradigms for distributed systems (RPC or RMI) are unable to exploit all the networks available in a grid system, such as those in parallel computers or PC clusters. Existing implementations of such communication paradigms were mainly based on the widely used TCP/IP communication protocol. Implementing TCP/IP on various communication networks could be a solution to solve the problem, but suffers from huge software overhead discouraging the programmers from using distributed programming paradigms within high-performance applications. In such circumstances, the use of RPC or RMI will restrict the deployment of the application on some of the computing resources depending on the availability of networks.

The second obstacle is the design of low-level communication layers for System Area Networks (SANs) in parallel systems or PC clusters (Myrinet [4], SCI [12], etc.) in a grid system. Such communication layers were not designed to be able to share the resources with several communication middleware and runtimes. Usually, these networks are available through a single communication paradigm (message-passing most of the time). Even worse, some communication layers require that the same binary code has to be executed on each node of the parallel computing resource. With such a restriction, it is not possible to execute two different codes on the same parallel system nor to exploit the underlying high-performance network to let the two codes exchange control and data.

Thus there exists a high risk of encouraging the programmers to use a single communication middleware or runtime for both parallel (within a simulation code) and distributed (between simulation codes) programming. For that purpose, one can envisage the use of an MPI [10] implementation for a grid infrastructure. We think that this approach is not suitable for several reasons. First of all, message-based runtimes (e.g. MPI) were not designed to transfer the control; it forces thus the programmer to simulate a RPC on

top of the message-passing runtime. Moreover, there is no way to express the interface of a scientific code. The use of such a code in another application will not be as simple as with a middleware that provides a way to express the interface associated with a code (such as the IDL language of CORBA). Our project aims at removing the two previously mentioned obstacles to allow the programmers to choose the most suitable middleware and runtimes for the design of grid applications.

The remainder of this paper is divided as follows. [Section 2](#) gives a short description of communication middleware and runtimes that should be integrated into our open integration platform. In [Section 3](#), we sketch the architecture of the PadicoTM platform. [Section 4](#) gives some performance results that were obtained with the PadicoTM platform. [Section 5](#) presents some related works. Finally, we present some concluding remarks in [Section 6](#).

2. Communication middleware and runtimes

This section aims at giving a brief overview of several communication middleware systems and runtimes we would like to integrate into an open framework, and draws a list of problems that such an open framework has to solve.

2.1. Message-passing

Message-passing has been widely adopted as the communication paradigm in the programming of distributed memory parallel systems. Although in the past there were various message-passing based runtimes provided by the parallel systems vendors, several projects aimed at designing a common message-passing interface. PVM [27] and MPI [9] are examples of such projects. Such runtimes allow the sending and receiving of messages through explicit *send* and *receive* operations with various semantics (blocking or non-blocking). Messages are usually associated with a type to allow a selection at the receiving side. Nowadays most of the parallel programs designed for distributed memory parallel systems are based on MPI. However, MPI was mainly designed for parallel programming and not for distributed programming.

2.2. Distributed shared memory (DSM)

DSM systems [15,18] are seen as an alternative for the programming of distributed and/or parallel systems. It gives the illusion of a single address space in a computational infrastructure in which each node has its own local physical memory. Although this paradigm has had few success, we think that the availability of a single address space in a grid infrastructure could simplify the programming of irregular applications for which data distribution is extremely challenging, or even impossible. Current DSM implementations are built on existing or specific message-passing libraries.

2.3. Distributed objects and components

CORBA [20] is a specification from the Object Management Group (OMG) to support distributed object-oriented applications. An application based on CORBA can be seen as a collection of independent software components or CORBA objects. RMI are handled by an Object Request Broker (ORB) which provides a communication infrastructure independent of the underlying network. An object interface is specified with the Interface Definition Language (IDL). An IDL compiler is in charge of generating a stub for the client side and a skeleton at the server side. Stubs and skeletons aim at connecting a client of a particular object to its implementation through the ORB. Within the ORB, several protocols exist to handle specific network technologies. The most important protocol is Internet Inter-ORB Protocol (IIOP) which is used to support IP-based networks. However, IIOP was designed for interoperability and offers limited performance. Fortunately, CORBA provides the ability to write an Environment-Specific Inter-ORB Protocol (ESIOP) which can handle other network technologies. However, there are very few ESIOP implementations for specific network technologies such as those in PC clusters or parallel computers. Moreover, the problem is more complex as we may think. A high-performance CORBA implementation will typically utilize SAN with a dedicated high-performance protocol. It needs to be interoperable with other standard ORBs, and thus should implement both high-speed protocol for SAN and standard IIOP for interconnecting with other ORBs

over TCP/IP. From the application designer perspective, such as high-speed ORB must behave as any other ORB.

2.4. Supporting several communication middleware and runtimes

Supporting CORBA and MPI, *both running simultaneously*, is not straightforward. Several access conflicts for networking resources may arise. For example, only one application at a time can use Myrinet [4] through BIP [22]. If both CORBA and MPI try to use it without being aware of each other, there are access conflicts and reentrance issues. If each middleware (e.g. CORBA, MPI, DSM, etc.) has its own thread dedicated to communications, with its own policy, communication performance is likely to be sub-optimal. If ever we are lucky enough and there is no resource conflict, there is probably a more efficient way than putting side by side pieces of software that do not see each other and that act in an “egoistic” fashion. In a more general manner, resource access should be cooperative rather than competitive.

3. Padico™ architecture

Padico is our research platform to investigate the problems of integrating several communication middleware and runtimes. Padico™, standing for Padico Task Manager, is the runtime of Padico. The role of Padico™ is to provide a high-performance infrastructure to *plug in* middleware like CORBA, MPI, Java Virtual Machine (JVM), DSM, etc. It offers a framework that deals with communication and multi-threading issues, allowing different middlewares to efficiently cohabit within the same process. Its strength is to offer the same interface to very different networks. Such platform is being used as a runtime for code coupling applications based on the concept of parallel CORBA objects [8,23], that need to simultaneously use a middleware (CORBA) and a runtime (MPI). Fig. 1 shows a typical use of Padico™: an application uses both MPI and CORBA at the same time. The following sections focus on the description of Padico™.

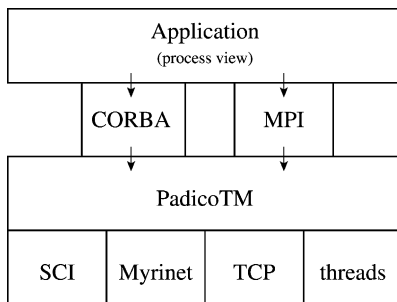


Fig. 1. Example of a typical PadicoTM application which uses both MPI and CORBA.

3.1. PadicoTM overview

The design of PadicoTM, derived from the software component technology is very modular. Every module is represented as a component: a description file is attached to the binary files. PadicoTM is composed of *core* modules and *service* modules. PadicoTM core implements module management, network multiplexing and thread management. PadicoTM core comprises three modules: *Puk*, *TaskManager* and *NetAccess*. Services are plugged in PadicoTM core. The available services are:

- advanced network API (*VSock* described in Section 3.5 and *circuit* in Section 3.6) on top of native PadicoTM network API;
- middleware and runtimes, namely CORBA, MPI, and a JVM (Section 4);
- gatekeepers (Section 3.7) which enable the user to remotely steer the processes on every nodes.

3.2. Dynamicity

There is a network model discrepancy between the “distributed world” (e.g. CORBA) and the “parallel world” (e.g. MPI). Communication layers dedicated to parallelism typically use a static topology:¹ nodes cannot be inserted or removed into the communicator while a session is active. On the other hand, CORBA has a distributed approach: servers may be dynamically started, clients may dynamically contact servers. The network topology is dynamic. High-performance networks API are mostly biased toward the parallel

¹ PVM and MPI2 address this problem but do not allow network management on a link-per-link basis.

model; thus, it is challenging to map the distributed communication model of CORBA onto SAN such as Myrinet [4] or SCI [12].

Since most communication libraries for SAN (e.g. BIP [22], Madeleine [2] or vendor’s MPI on most machines) require the processes on all nodes to be started at the same time, we chose that PadicoTM bootstraps a unique binary on each node. It satisfies the SPMD requirement of the communication library. Since we do not want all nodes to actually run the same application, we chose to store applications into *dynamically loadable modules*. Thanks to this mechanism, different binaries can be dynamically loaded into the different nodes of a cluster or a parallel computer that participates to a grid system. For example, we can load a CORBA server on one node and CORBA clients on other nodes. In PadicoTM, we call this bootstrap binary *Padico μ -Kernel*, or in shorter *Puk*. Once the *Puk* module is bootstrapped on each node, it loads the other modules and starts them. *Puk* is able to do only three things: load, start and unload modules on the node it manages. It knows nothing about threads nor about the network—these tasks are delegated to the *TaskManager* and *NetAccess* modules described below.

We want the *module* concept to be open. We do not restrict ourselves to binary dynamically loadable libraries. Actually, modules are described in a file written in XML. This description file contains the name of a *driver* able to load this module, references to other modules for dependency checking, *units* and *attributes*. A driver is a set of functions which tell *Puk* how to load, start and unload a given type of unit. Different drivers may be seen as module types. For example, the *binary* driver defines units as binary shared objects (“*.so*” libraries on Unix), the *java* driver defines units as Java classes, or the *pkg* driver defines units as being modules. Attributes are environment variables aimed at configuring modules. Fig. 2 shows the description for the *ORB* module: it should be loaded by the *binary* driver, requires the *VSock* module, contains the *libORB.so* unit and an attribute for referencing the CORBA name service.

3.3. Coherent thread management

It is now common that middleware implementations use multi-threading. However, middleware systems which are not designed to run together in the

```

<mod name="ORB" driver="binary">
  <requires>VSocket</requires>
  <attr label="NameService">
    corbaloc:iiop:paraski.irisa.fr:2809/NameService
  </attr>
  <unit>libORB.so</unit>
</mod>

```

Fig. 2. XML description for the ORB service.

same process are likely to use incompatible thread policies, or simply different multi-threading packages. An application runs into trouble when mixing several kinds of threads. That is why PadicoTM must provide the plugged in middleware with a portability layer for multi-threading.

At first look, it may seem attractive to use Posix threads (known as `pthread`) as a foundation. However, it has been shown [5] that MPI and current implementations of Posix threads do not stack up nicely. To deal with portability as well as performance issues, we choose the Marcel [7] multi-threading library. Marcel is a multi-threading library in user space. It implements an N:M thread scheduling on SMP architectures. Marcel has been designed to guarantee a good reactivity of the application to network I/O when used in conjunction with the Madeleine [3] communication layer.

The *TaskManager* module of PadicoTM is based on Marcel. Every PadicoTM modules which use multi-threading are supposed to use Marcel and no other multi-threading library. This is not very constraining: Marcel API is very similar to Posix threads API.

The *TaskManager* module provides handy queues for asynchronous processing of *Puk* operations. All *Puk* operations are performed in the same thread to avoid reentrance issues at low-level. The modules outside the PadicoTM core are not supposed to perform direct calls to *Puk*; they should use it through the *TaskManager* API instead. The *TaskManager* module manages system calls so that they do not block the whole process. It provides hooks for polling loops so that they do not compete with each other. As the *TaskManager* knows the threads of every modules, it is able to chose a coherent policy.

3.4. Cooperative access to the network

Access to high-speed networks is the more conflict-prone task when using multiple middleware systems at the same time. Some access methods require an exclusive access to the hardware (e.g. Myrinet through BIP) thus only one library can use it at the same time, i.e. CORBA or MPI, not both; some networks have limited resources which can be exhausted if different libraries open separate connections (e.g. SCI); on some network hardware, several drivers are available but only one is usable at a time (e.g. BIP or GM on Myrinet).

In the worst case, middleware cannot coexist in the same process nor on the same machine, due to network access conflict. In the best case, if middleware systems do not know each other, each would run its own polling thread so that the access to the network is competitive and prone to race conditions.

To deal with low-level, portability, and performance issues, we chose to use Madeleine [3] as a foundation for the *NetAccess* module of PadicoTM. The Madeleine communication layer was designed to bridge the gap between low-level communication interfaces (such as BIP [22], SBP or UNET) and middleware. It provides an interface optimized for *RPC-like* operations that allows zero-copy data transmissions on high-speed networks, and is best used with Marcel threads. A unique polling loop managed by the PadicoTM *NetAccess* module dispatches incoming messages to modules that want access to high-speed networks. Thus, every module use the network through *NetAccess*: there is no access conflict. Moreover, there is no competition thanks to the unique polling loop.

In order to allow several middleware to use the network, there is a need for multiplexing in some layer.

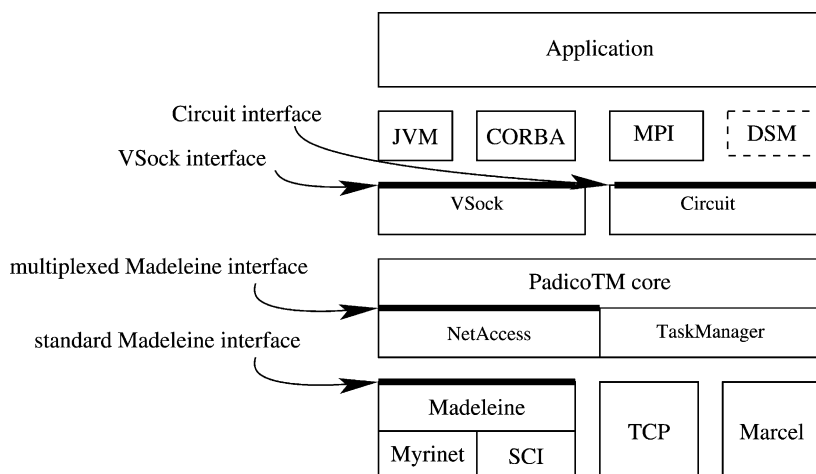


Fig. 3. PadicoTM modules.

Madeleine provides no more multiplexing channels than what is allowed by the hardware; it means two channels on Myrinet, and only one channel on SCI. However, to deploy an arbitrary number of communication middlewares in a PadicoTM process, we need an unbound number of logical communication channels. The *NetAccess* module multiplexes logical "PadicoTM channels" on top of Madeleine hardware channels. Practically, *NetAccess* uses one Madeleine channel with one polling loop listening on it. The modules that want to use Madeleine register callback functions which are called when a message arrives. To guarantee that the communications are deadlock-free, callbacks are not allowed to block nor to send directly a message on the network. However, if they need to send a reply or to wait on a condition, the *TaskManager* can do it in another thread. This mechanism requires very few changes to existing Madeleine applications. Moreover, user's applications do not want to use Madeleine directly; they use CORBA or MPI instead. Only developers of middleware for PadicoTM need to use these callbacks.

Multiplexing on top of Madeleine adds a header to all messages. This can increase significantly the latency if not done properly. We implement "headers combining" which enables most messages to contain only one combined header plus the body. Headers of all logical layers are aggregated into a single low-level packet. For each outgoing message, *NetAccess* allocates a buffer for headers; on top of *NetAccess*, each

layer adds its headers in the buffer. Thus, multiplexing on top of Madeleine adds virtually no overhead compared to middleware built on top of regular Madeleine. We measured that the overhead is negligible.

Puk, *TaskManager* and *NetAccess* modules compose PadicoTM core. Other modules are called services. They are plugged in the PadicoTM core. Fig. 3 sums up the available modules in PadicoTM.

3.5. Virtual sockets

The TCP/IP network protocol is designed for use over a WAN. It is not well suited for use over a SAN. Moreover, system calls add a significant latency to the data path. That is why we avoid as much as possible kernel-level communication libraries. However, the widespread socket interface from Berkeley is fairly well suited for networking. Most networking middleware use sockets; some of them heavily rely on the concept of sockets and would require very deep changes to use another communication paradigm. Thus, we chose to implement a socket-like interface on top of the native *NetAccess* interface, like Fast Socket [24] on top of Active Messages. Our approach relies on the concept of *virtual socket*, that we call *VSocket*. It implements a subset of the standard socket functions in user space on top of *NetAccess*, for achieving high-performance. It performs zero-copy datagram transfer with a socket-like connection handshake mechanism.

VSocket is a multi-protocol communication layer with auto-selection. It automatically selects the adequate protocol according to the available hardware. For interoperability issues, *VSocket* is able to communicate with *VSocket*-unaware applications using standard TCP/IP protocol. It determines by itself whether an address (a pair of standard IP address–port number) is reachable using Madeleine or if it should revert to standard TCP. From the application point of view, *VSocket* behaves exactly as regular sockets, even if the data path is bypassed through *NetAccess*/Madeleine instead of TCP/IP when possible.

Then, it is straightforward to port on top of *VSocket* existing middleware based on sockets like CORBA or a JVM.

3.6. Groups and circuits

The *NetAccess* module is a low-level communication layer of PadicoTM. It creates communication channels which comprise every nodes of a cluster. However, one may want for example to deploy two MPI codes coupled with CORBA on a cluster. In this case, each MPI code spans across only a group of nodes, though the low-level communication library spans across all nodes.

To handle such cases, PadicoTM provides the concept of logical groups of nodes, which we define as a set of nodes. We define a *circuit* as a *NetAccess* communication channel restricted to a group. Thus, higher level communication libraries such as MPI or DSM run on a *circuit*. The logical topology does not have to match the hardware topology. This is different from creating MPI groups inside the MPI communicator: the group is handled by PadicoTM, thus the middleware library is loaded only on the required nodes, and the other nodes may load any other middleware (e.g. DSM).

To manage modules on groups, we provide an additional driver for *Puk* called *multi*. The *multi* driver is aimed at running SPMD codes and middleware (e.g. MPI, DSM) on PadicoTM groups. Basically, the *multi* driver transforms the modules it contains into SPMD modules. For example, when the user loads a *multi* module, the driver forwards the request to the nodes of the group, performs synchronization, gathers and aggregates the return codes. All *Puk* operations (load, start, unload) are *SPMDized*, with appropri-

ate synchronization. For the *multi* driver, units are modules and the group name is given through an attribute.

3.7. Remote control

For dynamically monitoring and managing modules on each node, Padico comprises *PadicoControl*, a set of applications to remotely steer PadicoTM processes. Currently, there are two such applications: a GUI, and a command-line tool for more advanced users. Communications between these tools and PadicoTM rely on CORBA or an XML-based RPC (the use of SOAP is being investigated), thus allowing the design of third-party tools.

A PadicoTM service called *gatekeeper*, loaded in PadicoTM processes, listens to incoming requests and handles them (for example, load a module, return the list of running modules, etc.). It is mostly a remote interface for the *TaskManager* (see Section 3.3).

For the moment, we use a single-user security policy. Security is managed through the use of session keys. When PadicoTM processes are launched, the same session key is given to the user and to the gatekeeper. All requests to *PadicoControl* must contain a session key which matches the one known by the gatekeeper. If keys do not match, the request is not taken into account. Thus, only the user who launched the processes is authorized to steer them.

4. Experiments with middleware and runtimes with PadicoTM

The MPI implementation in PadicoTM is derived from MPICH/Madeleine [2] with very few changes (use *circuit* instead of Madeleine and replace the polling thread with a callback). The CORBA implementation in PadicoTM is based on OmniORB3 [1] from AT&T. The porting of OmniORB on top of *VSocket* and Marcel threads is straightforward. We also ported another implementation of CORBA, namely MICO, to show the ability of PadicoTM to support various CORBA-based middleware. However, the best performance was obtained using OmniORB. The JVM module is based on Kaffe [14], on top of *VSocket* and Marcel.

Our benchmark machines are “old” dual-Pentium II 450 MHz machines, with Ethernet-100, Dolphin SCI

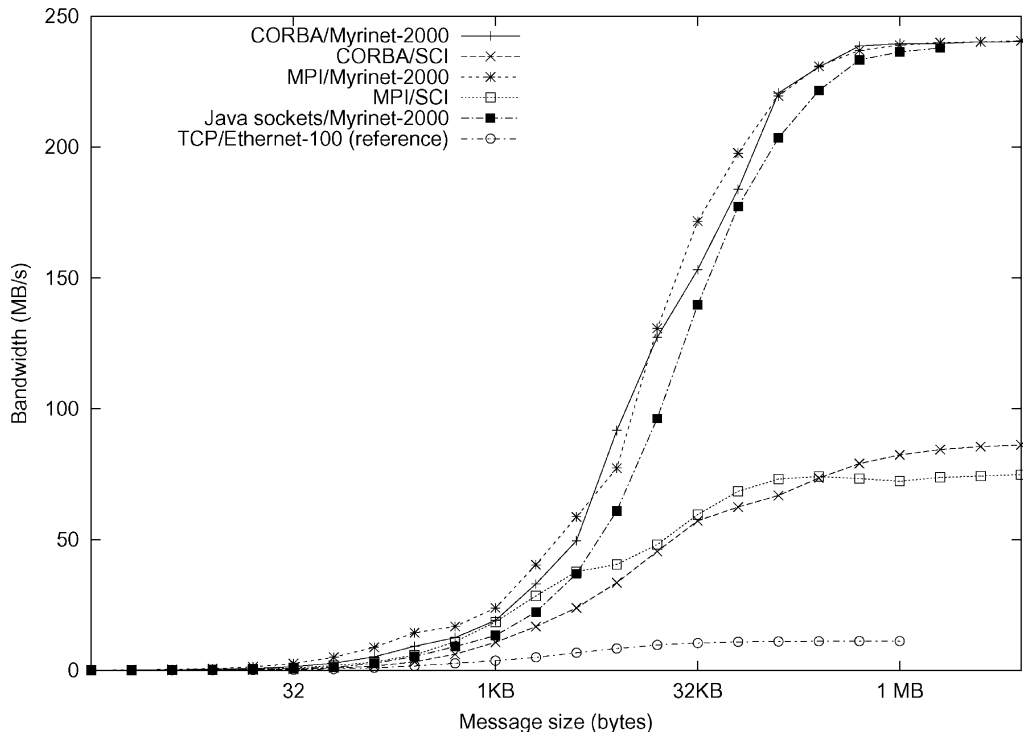


Fig. 4. CORBA and MPI bandwidth on top of PadicoTM.

and Myricom Myrinet-1 adapters, and “more recent” dual-Pentium III 1 GHz with Myricom Myrinet-2000.

The MPI module in PadicoTM gets the bandwidth is shown in Fig. 4. The peak bandwidth is excellent: 240 MB/s on Myrinet-2000 and 75 MB/s on SCI. The latency is 11 μ s on Myrinet-2000 and 23 μ s on SCI. This performance is identical to MPICH/Madeleine [2] from which PadicoTM MPI implementation is derived; PadicoTM adds no noticeable overhead neither for bandwidth nor for latency.

The bandwidth of the high-performance CORBA implementation is shown in Fig. 4. The benchmark consists in a remote invocation of a method which takes an *inout* parameter of variable size (sequence of `long`). The peak bandwidth is 240 MB/s on Myrinet-2000, 89 MB/s on SCI, and 101 MB/s on Myrinet-1 (not shown in figure). This performance is very good. We reached more than 96% of the maximum achievable bandwidth with Madeleine.

On the old machines (Pentium II 450, SCI or Myrinet-1), the latency of CORBA for an empty remote invocation is around 55 μ s. It is a good point

when compared to the 160 μ s latency of the ORB over TCP/Ethernet-100. On the more recent machines (Pentium III 1 GHz, Myrinet-2000), the latency of CORBA is 20 μ s where MPI gets 11 μ s.

CORBA is as fast as MPI regarding the bandwidth, and slightly slower than MPI for latency. This latency could be lowered if we used a specific protocol (called ESIOP) instead of the all-purpose GIOP protocol in the CORBA implementation. This performance is very good, though. As far as we know, OmniORB in PadicoTM is the fastest CORBA implementation.

Padico provides a JVM module based on Kaffe [14]. It has been modified to use Marcel threads and *VSock*. Thus, Java sockets can reach very good performance when a high-speed network is available. Fig. 4 shows the bandwidth of Java sockets over Myrinet-2000.

5. Related works

Several middleware environments for managing the network communications and multi-threading have

emerged. However, very few take both parallel and distributed paradigms into account and thus are not designed for general middleware integration. The ADAPTIVE Communication Environment (ACE) [26] is distributed-oriented; it aims at providing a C++ high level abstract and portable interface for system features such as network and multi-threading. It does not support high-performance networks (Myrinet, etc.) and offers a specific API for tight integration with a middleware built on top of it. A CORBA implementation called TAO [16] has been built on ACE. TAO focuses on high-performance and real-time aspects. Its main concern is predictability. It may utilize TCP or ATM networks, but it is not targeted to high-performance networks found on clusters of PCs such as Myrinet or SCI. Panda [25] is a framework which deals with networking and multi-threading; it is built to support runtimes dedicated to parallel languages. Harness [17] is a framework for high-performance distributed computing; it is built on Java. Like PadicoTM, it considers middleware systems as plugins. For the moment, there is only a PVM plugin in Harness and published performance mention only plain TCP—no Myrinet nor WAN-optimized protocols. Proteus [6] is a system for integrating multiple message protocols such as SOAP and JMS within one system. It aims at decoupling applications from protocols, which is an approach quite similar to ours, but at a much higher level in the protocol stack. Nexus [11] used to be the communication subsystem of Globus. It was based on the unique concept of global pointers which was too restricting. Nowadays, it becomes accepted that MPICH-G2 [10] built on Globus-IO is a popular communication mechanism for grids, but only supports one API, namely MPI.

Moreover, there are a few works about high-performance CORBA. OmniORB2 had been adapted to ATM and SCI networks. Since the code is not publicly available, we only report published results. On ATM, there is a gap of bandwidth between raw bytes and structured data types [21]. The bandwidth can be as low as 0.75 MB/s for structured types. On SCI, results are quite good [19] (156 μ s, 37.5 MB/s) for messages of raw bytes; figures for structured types on SCI are not published. CrispORB [13], developed by Fujitsu labs, is targeted to VIA in general and Synfinity-0 networks in particular. Its latency

is noticeably better, up to 25% than with standard IIOp.

6. Summary and conclusion

In this paper we have presented an open framework that is able to incorporate various communication runtimes and middleware. This platform enables the execution of applications that are based on both distributed and parallel programming paradigms on grid infrastructures, independently from the underlying networking resources. Such an approach encourages grid programmers to use the most suited communication middleware and runtimes for their applications. Although this platform adds one more layer between the applications and the networking resources, we showed that the additional overhead is insignificant. Moreover, we showed that middleware, such as CORBA, for distributed computing can take benefit from high-performance network. We also showed that CORBA can achieve roughly the same level of performance than MPI sweeping away prejudice concerning the performance of such a middleware.

PadicoTM has been implemented and is distributed as free software. Check out the Padico web site at <http://www.irisa.fr/paris/Padico/> for more information and to download the software and its documentation.

Acknowledgements

We would like to thank the PM2 developers team (<http://www.pm2.org/>) for their efficient support of Madeleine and Marcel.

References

- [1] AT&T Laboratories Cambridge, OmniORB Home Page. <http://www.omniorb.org>.
- [2] O. Aumage, G. Mercier, R. Namyst, MPICH/Madeleine: a true multi-protocol MPI for high-performance networks, in: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), IEEE, San Francisco, April 2001, p. 51.
- [3] O. Aumage, L. Bougé, A. Denis, J.-F. Méhaut, G. Mercier, R. Namyst, L. Prylli, A portable and efficient communication library for high-performance cluster computing,

- in: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000), Technische Universität Chemnitz, Saxony, Germany, November 2000, pp. 78–87.
- [4] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.-K. Su, Myrinet: a gigabit-per-second local area network, *IEEE-Micro* 15 (1) (1995) 29–36.
- [5] L. Bougé, J.-F. Méhaut, R. Namyst, Efficient communications in multi-threaded runtime systems, in: *Parallel and Distributed Processing, Proceedings of the Third Workshop on Runtime Systems for Parallel Programming (RTSPP'99)*, vol. 1586 of *Lecturer Notes in Computer Science*, San Juan, Puerto Rico, In conjunction with IPPS/SPDP 1999, IEEE TCPP and ACM SIGARCH, Springer, Berlin, April 1999, pp. 468–482.
- [6] K. Chiu, M. Govindaraju, D. Gannon, The proteus multi-protocol library, in: *Proceedings of the 2002 Conference on Supercomputing (SC'02)*, Baltimore, USA, November 2002.
- [7] V. Danjean, R. Namyst, R. Russell, Integrating kernel activations in a multi-threaded runtime system on Linux, in: *Parallel and Distributed Processing, Proceedings of the Fourth Workshop on Runtime Systems for Parallel Programming (RTSPP'00)*, vol. 1800 of *Lecturer Notes in Computer Science*, Cancun, Mexico, In conjunction with IPDPS 2000, IEEE TCPP and ACM, Springer, Berlin, May 2000.
- [8] A. Denis, C. Pérez, T. Priol, Portable parallel CORBA objects: an approach to combine parallel and distributed programming for grid computing, in: *Proceedings of the International Euro-Par'01 Conference*, Springer, Manchester, UK, 2001, pp. 835–844.
- [9] Message Passing Interface Forum, MPI: a message-passing interface standard, Technical Report UT-CS-94-230, 1994.
- [10] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, S. Tuecke, Wide-area implementation of the message passing interface, *Parall. Comput.* 24 (12) (1998) 1735–1749.
- [11] I. Foster, J. Geisler, C. Kesselman, S. Tuecke, Managing multiple communication methods in high-performance networked computing systems, *J. Parall. Distribut. Comput.* 40 (1) (1997) 35–48.
- [12] IEEE, Standard for Scalable Coherent Interface (SCI), Standard no. 1596, August 1993.
- [13] Y. Imai, T. Saeki, T. Ishizaki, M. Kishimoto, CrispORB: high performance CORBA for system area network, in: *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999, pp. 11–18.
- [14] Kaffe: an OpenSource implementation of a Java Virtual Machine. <http://www.kaffe.org>.
- [15] P. Keleher, D. Dwarkadas, A. Cox, W. Zwaenepoel, Treadmarks: distributed shared memory on standard workstations and operating systems, in: *Proceedings of the 1994 Winter USENIX Conference*, January 1994, pp. 115–131.
- [16] F. Kuhns, D. Schmidt, D. Levine, The design and performance of a real-time I/O subsystem, in: *Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, Vancouver, Canada, June 1999.
- [17] D. Kurzyniec, V. Sunderam, M. Migliardi, On the viability of component frameworks for high performance distributed computing: a case study, in: *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, Edimburg, Scotland, July 2002.
- [18] K. Li, P. Hudak, Memory coherence in shared virtual memory systems, in: *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, 1986, pp. 229–239.
- [19] S.-L. Lo, S. Pope, The implementation of a high performance ORB over multiple network transports, Olivetti and Oracle Laboratory, Cambridge, March 1998.
- [20] Object Management Group, The Common Object Request Broker: Architecture and Specification (Revision 2.2), February 1998.
- [21] S. Pope, S.-L. Lo, The implementation of a native ATM transport for a high performance ORB, Technical Report, Olivetti and Oracle Laboratory, Cambridge, June 1998.
- [22] L. Prylli, B. Tourancheau, BIP: a new protocol designed for high performance networking on Myrinet, in: *Proceedings of the First Workshop on Personal Computer based Networks of Workstations (PC-NOW'98)*, *Lecturer Notes in Computer Science*, In conjunction with IPPS/SPDP 1998, Springer, Berlin, April 1998, pp. 472–485.
- [23] C. René, T. Priol, MPI code encapsulating using parallel CORBA object, in: *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, August 1999, pp. 3–10.
- [24] S.H. Rodrigues, T.E. Anderson, D.E. Culler, High-performance local area communication with fast sockets, in: *USENIX'97*, January 1997, pp. 257–274.
- [25] T. Rühl, H. Bal, R. Bhoedjang, K. Langendoen, G. Benson, Experience with a portability layer for implementing parallel programming systems, in: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, CA, USA, August 1996, pp. 1477–1488.
- [26] D.C. Schmidt, An architectural overview of the ACE framework: a case-study of successful cross-platform systems software reuse, in: *USENIX Login Magazine*, Tools special issue, November 1998.
- [27] V.S. Sunderam, PVM: a framework for parallel distributed computing, *Concurrency Pract. Exp.* 2 (4) (1990) 315–340.



Alexandre Denis is a PhD student at IRISA/IFSIC. His research interests include the design of communications frameworks aiming at integrating various middleware and runtime systems within a grid environment.



Christian Pérez is a research scientist at IRISA/INRIA. His research interests include parallel and distributed computing. His current research project focuses on the programming of grids, their execution models and runtimes for code coupling applications. He received a PhD in Computer Science from the Ecole Normale Supérieure of Lyon, France in 1999.



Thierry Priol is a senior research scientist at IRISA/INRIA and leader of the PARIS project-team. His research interests include the design of programming environments for high-performance computing systems such as parallel computers, clusters or computational grids. He received a PhD in Computer Science from the University of Rennes 1 in 1989. He also received an “Habilitation à diriger des recherches” from the University of Rennes 1 in 1995.