Open access • Proceedings Article • DOI:10.1109/CCGRID.2002.1017122

PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes — Source link

Alexandre Denis, Christian Pérez, Thierry Priol

Institutions: French Institute for Research in Computer Science and Automation

Published on: 21 May 2002 - Cluster Computing and the Grid

Topics: Grid computing, Middleware, Grid, Concurrent computing and Application software

Related papers:

- Optimisation of component-based applications within a grid environment
- Developing a user-level middleware for out-of-core computation on Grids
- Adaptive Distributed Computing Middleware for Computational Finance Applications
- A virtual file system interface for computational grids
- Multi-domain grid/cloud computing through a hierarchical component-based middleware

Share this paper:

View more about this paper here: https://typeset.io/papers/padicotm-an-open-integration-framework-for-communication-291i3a2khs

PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes

Alexandre Denis, Christian Pérez, Thierry Priol

To cite this version:

Alexandre Denis, Christian Pérez, Thierry Priol. PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes. IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002), May 2002, Berlin/Germany, Germany. pp.144-151. inria-00000132

HAL Id: inria-00000132
https://hal.inria.fr/inria-00000132

Submitted on 24 Jun 2005

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.

PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes

Alexandre Denis[1] Christian Pérez[2]
Thierry Priol[2]
[1]IRISA/IFSIC, [2]IRISA/INRIA,
Campus de Beaulieu - 35042 Rennes Cedex, France
{Alexandre.Denis, Christian.Perez, Thierry.Priol}@irisa.fr

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 the presence 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 available in a computational grid. 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.

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 simulation 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 for example MPI, to be able to run on a computational grid. It is more likely that each simulation code has its own requirement in term 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 RPC (Remote Procedure Call) or RMI (Remote Method Invocation). 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 (SAN) in parallel systems or PC clusters (Myrinet, SCI, ...) in a grid system. Such communication layers were not designed to be able to share the networking resources with several communica-

tion 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 [8] implementation for a grid infrastructure. We think that this approach is not suitable for several reasons. First of all, message-based runtimes (eg. 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 [20] and MPI [7] 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.

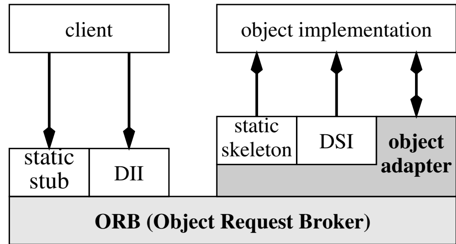

Figure 1. CORBA Architecture

2.2 Distributed Shared Memory

Distributed shared memory systems [13, 11] 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 [15] is a specification from the OMG (Object Management Group) to support distributed object-oriented applications. Figure 1 describes its architecture. An application based on CORBA can be seen as a collection of independent software components or CORBA objects. Remote method invocations 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 IIOP (Internet Inter-ORB Protocol) 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 ESIOP (Environment-Specific Inter-ORB Protocol) 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 a 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 through BIP [17]. 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 (eg. CORBA, MPI, a 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 PadicoTM Architecture

Padico is our research platform to investigate the problems of integrating several communication middleware and runtimes. PadicoTM, standing for Padico Task Manager, is the runtime of Padico. The role of PadicoTM is to provide a high performance infrastructure to *plug in* middleware like CORBA, MPI, JVM (Java Virtual Machine), DSM (Distributed Shared Memory), 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 [18, 6] for which there is a need to simultaneously use a middleware (CORBA) and a runtime (MPI). Figure 2 shows a typical use of PadicoTM:

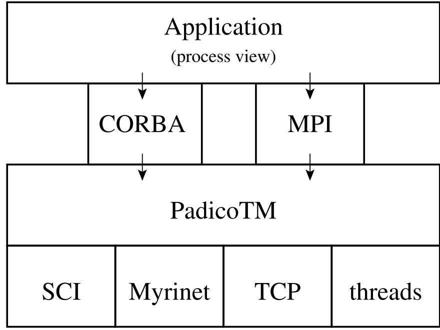

Figure 2. Example of a typical PadicoTM application which uses both MPI and CORBA

an application uses MPI and CORBA at the same time. The following sections focus on the description of PadicoTM.

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* described in Section 3.6) on top of native PadicoTM network API;
- middleware and runtimes, namely a CORBA module (Section 4.2), a MPI module (Section 4.1) and a Java Virtual Machine;
- gatekeepers (Section 3.7) which enable the user to remotely steer the processes on every nodes.

Currently, we have a functional prototype with all these modules available.

3.2 Dynamicity

Static vs. Dynamic. There is a network model discrepancy between the "distributed world" (eg. CORBA) and the "parallel world" (eg. MPI). Communication layers dedicated to parallelism typically use a static topology[1]: nodes

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

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

Loadable modules. Since most communication libraries for SAN (eg. BIP, 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.

Module type. 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. Figure 3 is 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 running on the `paraski` machine and listening on port 10000.

```
<mod name="ORB" driver="binary">
  <requires>VSock</requires>
  <attr label="NameService">
    corbaname::paraski.irisa.fr:10000
  </attr>
  <unit>libORB.so</unit>
</mod>
```

Figure 3. XML description for the *ORB* service.

3.3 Thread Management

Common thread library. It is now common that middleware implementations use multi-threading. However, middleware systems which are not designed to run together in the 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 [4] 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 [5] 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 [2] communication layer.

Coherent thread management. 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 (described in Section 3.2). 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

High performance networks. 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 (eg. Myrinet through BIP) thus only one library can use it at the same

time – ie. CORBA *or* MPI, not both; some networks have limited resources which can be exhausted if different libraries open separate connections (eg. SCI); some network hardware can be used through several drivers, but it causes conflicts if more than one driver is used to access the same hardware at the same time (eg. on Myrinet, all middleware systems must agree on the driver to use: BIP *or* GM).

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 [2] 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 [17], SBP or UNET) and middleware. It provides an interface optimized for *RPC-like* operations that allows zero-copy data transmissions on high-speed networks such as Myrinet or SCI, 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.

Multiplexing over Madeleine. In order to allow several middleware to use the network, there is a need for multiplexing in some layer. Madeleine provides no more multiplexing channels than what is allowed by the hardware. For example, Madeleine provides two channels on top of BIP, and only one channel on top of SCI. However, we want to be able to deploy an arbitrary number of communication middlewares in a PadicoTM process. Therefore, we need an arbitrary 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. Figure 4 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 described in the previous section, like Fast Socket [19] on top of Active Messages. Our approach relies on the concept of *virtual socket*, that we call *VSock*. 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.

VSock is a multi-protocol communication layer with auto-selection. It automatically selects the adequate protocol according to the available hardware. For interoperability issues, *VSock* is able to communicate with *VSock*-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, *VSock* 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 *VSock* existing middleware based on sockets like CORBA or a Java Virtual Machine.

3.6 Groups and circuits

The *NetAccess* module is a low-level communication layer of PadicoTM. It creates communication channels

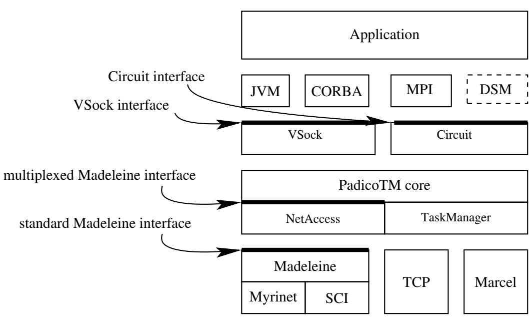

Figure 4. PadicoTM modules

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. A group is a set of nodes of a cluster or of a parallel machine. We define a *circuit* as a *NetAccess* communication channel restricted to a group. Thus, higher level communication libraries such as MPI or a DSM run on a circuit. The logical topology has not to match the hardware topology. This is different from creating MPI groups inside the high-level MPI communicator: there is no need to change an existing application which expects to use `MPI_COMM_WORLD`, the middleware library (eg. MPI) is loaded only on nodes which actually run an MPI application, and finally it is available for any other middleware such as a Distributed Shared Memory (DSM).

To manage modules on groups and circuits, we provide an additional driver for *Puk* called `multi`. The `multi` driver is aimed at running SPMD codes and SPMD middleware (such as MPI or a DSM) on PadicoTM groups and circuits. Basically, the `multi` driver transforms the modules it contains into SPMD modules. For example, when the user loads a `multi` module (with only one request from the user), the driver forwards the request to the given group of nodes, performs synchronization, and aggregates the return codes. All *Puk* operations (load, start, unload) are performed on a group of nodes instead of a single node, with appropriate synchronization. For the `multi` driver, units are modules. 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 a PadicoTM process. Currently, there are two such applications: a GUI written in Java for portability, 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 specific 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 [3] 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 *VSock* 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 Java Virtual Machine module is based on Kaffe [10], on top of *VSock* and Marcel.

Our benchmark machines are “old” dual-Pentium II 450MHz machines, with Ethernet-100, SCI and Myrinet-1, and “more recent” dual-Pentium III 1GHz with Myrinet-2000.

4.1 MPI

The MPI module in PadicoTM gets the bandwidth shown on Figure 5. The peak bandwidth is excellent: 240 MB/s on Myrinet-2000 and 75 MB/s on SCI. The latency is 11 μs on Myrinet-2000 and 23 μs on SCI. This performance is very similar to MPICH/Madeleine [3] from which PadicoTM MPI implementation is derived; PadicoTM adds no noticeable overhead neither for bandwidth nor for latency.

4.2 CORBA

The bandwidth of the high-performance CORBA implementation is shown on Figure 5. The benchmark consists in a remote invocation of a method which takes an *inout*

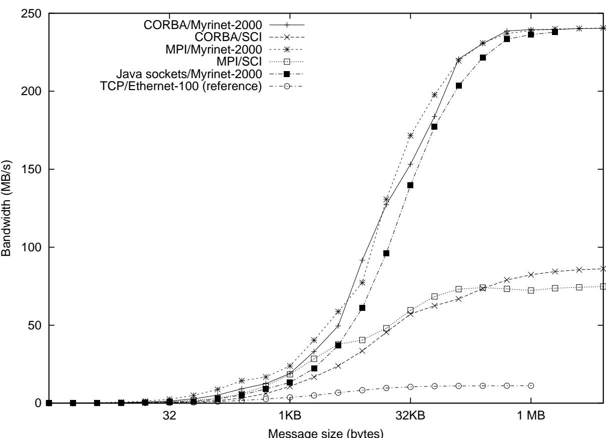

Figure 5. CORBA and MPI bandwidth on top of PadicoTM

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 on 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 μs. It is a good point when compared to the 160 μs latency of the ORB over TCP/Ethernet-100. On the "more recent" machines (Pentium III 1GHz, Myrinet-2000), the latency of CORBA is 20 μs where MPI gets 11 μs.

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.

4.3 Java

Padico provides a Java Virtual Machine module based on Kaffe [10]. 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. Figure 5 shows the bandwidth of Java sockets over Myrinet-2000.

5 Related Works

From our knowledge, there exist very few research works dealing with the design of an open integration framework for communication middleware and runtimes. Most of the works focused on the performance optimization of a single middleware or runtime. Since high-performance MPI is well known, we focus here on high-performance CORBA. TAO [12] (the ACE ORB) 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 network protocols found on clusters of PCs such as BIP or SISCI. 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 [16]. The bandwidth can be as low as 0.75 MB/s for structured types. On SCI, results are quite good [14] (156 μs, 37.5 MB/s) for messages of raw bytes; figures for structured types on SCI are not published. CrispORB [9], 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 platform 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 such as SCI and Myrinet. 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. Concerning the status of the project, all the functionality described in this paper has been implemented. It is expected to distribute this platform during spring 2002.

Acknowledgments

This work was supported by the Incentive Concerted Action "GRID" (ACI GRID) of the French Ministry of Research. 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, L. Bougé, J.-F. Méhaut, and R. Namyst. Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, March 2001. To appear.

[3] 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)*, page 51, San Francisco, April 2001. IEEE.

[4] L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Parallel and Distributed Processing. Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–482, San Juan, Puerto Rico, April 1999. In conj. with IPPS/SPDP 1999. IEEE TCPP and ACM SIGARCH, Springer-Verlag.

[5] V. Danjean, R. Namyst, and R. Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, Cancun, Mexico, May 2000. In conjunction with IPDPS 2000. IEEE TCPP and ACM, Springer-Verlag.

[6] A. Denis, C. Pérez, and T. Priol. Portable parallel CORBA objects: an approach to combine parallel and distributed programming for grid computing. In *Proc. of the Intl. Euro-Par'01 conf.*, pages 835–844, Manchester, UK, 2001. Springer.

[7] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[8] Ian Foster, Jonathan Geisler, William Gropp, Nicholas Karonis, Ewing Lusk, George Thiruvathukal, and Steven Tuecke. Wide-area implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, November 1998.

[9] Yuji Imai, Toshiaki Saeki, Tooru Ishizaki, and Mitsushiro Kishimoto. CrispORB: High performance CORBA for system area network. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 11–18, 1999.

[10] Kaffe: an OpenSource implementation of a Java Virtual Machine. http://www.kaffe.org.

[11] P. Keleher, D. Dwarkadas, A. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.

[12] F. Kuhns, D. Schmidt, and D. Levine. The design and performance of a real-time I/O subsystem. In *Proceedings of the 5th IEEE Real-Time Technology and Applicati ons Symposium (RTAS99)*, Vancouver, Canada, June 1999.

[13] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *Proceedings of the 1986 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[14] Sai-Lai Lo and S. Pope. The implementation of a high performance ORB over multiple network transports. Technical report, Olivetti & Oracle Laboratory, Cambridge, March 1998.

[15] Object Management Group. The Common Object Request Broker: Architecture and Specification (Revision 2.2), February 1998.

[16] S. Pope and Sai-Lai Lo. The implementation of a native ATM transport for a high performance ORB. Technical report, Olivetti & Oracle Laboratory, Cambridge, June 1998.

[17] L. Prylli and B. Tourancheau. Bip: a new protocol designed for high performance networking on myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, Lect. Notes in Comp. Science, pages 472–485. Springer-Verlag, apr 1998. In conjunction with IPPS/SPDP 1998.

[18] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, August 1999.

[19] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *USENIX '97*, pages 257–274, January 1997.

[20] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.