A flexible middleware for metacomputing in multi-domain networks

Franco Frattolillo

Research Centre on Software Technology
Department of Engineering, University of Sannio
Benevento, Italy

Summary
Middlewares are software infrastructures able to harness the enormous, but often poorly utilized, computing resources existing on the Internet in order to solve large-scale problems. However, most of such resources, particularly those existing within "departmental" organizations, cannot often be considered actually available to run large-scale applications, in that they cannot be easily exploited by most of the currently used middlewares for grid computing. In fact, such resources are usually represented by computing nodes belonging to non-routable, private networks and connected to the Internet through publicly addressable IP front-end nodes. This paper presents a flexible Java middleware able to support the execution of large-scale, object-based applications over heterogeneous multi-domain, non-routable networks. In addition, the middleware has been also designed as a customizable collection of abstract components whose implementations can be dynamically installed in order to satisfy application requirements.

Key words:
Middleware, Metacomputing, Grid computing.

Introduction

Middlewares are the special software infrastructures that enable the enormous, but often poorly utilized, computing resources existing on the Internet to be harnessed in order to solve large-scale problems. They extend the concept, originally introduced by PVM [1], of a "parallel virtual machine" restricted and controlled by a single user, making it possible to build computing systems, called *metacomputers*, composed of heterogeneous computing resources of widely varying capabilities, connected by potentially unreliable, heterogeneous networks and located in different administrative domains [2, 3]. However, this forces middlewares to deal with highly variable communication delays, security threats, machine and network failures, and the distributed ownership of computing resources, if they want to give a support to programmers in solving problems of configuration and optimization of large-scale applications in this new computational context. Therefore, middlewares can build on most of the current distributed and parallel software technologies, but they require further and significant advances in mechanisms, techniques, and tools to bring together computational resources distributed over the Internet to efficiently solve a single large-scale problem according to the new scenario introduced by *grid computing* [2, 3, 4]. In fact, in such a scenario, *cluster computing* still remains a valid and actual support to high performance parallel computing, since clusters of workstations represent high performance/cost ratio computing platforms and are widely available within "departmental" organizations, such as research centres, universities, and business enterprises [5]. However, workstation clusters are usually exploited within trusted and localized network environments, where problems concerning security, ownership, and configuration of the used networked resources are commonly and easily solved within the same administrative domain. Furthermore, it is also worth noting that most of the computing power existing within departmental organizations cannot often be considered actually available to run large-scale applications, in that it cannot be easily exploited by most of the currently used middlewares for grid computing [2, 3]. In fact, such computing power is often represented by computing nodes belonging to non-routable private networks and connected to the Internet through publicly addressable IP front-end nodes [5].

The considerations reported above suggest that a middleware should be able to exploit computing resources across multi-domain, non-routable networks and to abstractly arrange them according to a hierarchical topology atop the physical interconnection network. Furthermore, the need for adaptability requires that a middleware is also characterized by a flexible implementation as well as developed according to a component-based, reflective architecture [6] in order to facilitate dynamic changes in the configuration of the built metacomputers.

Java can be considered an interesting language widely used to develop middlewares for metacomputing. It has been designed for programming in heterogeneous computing environments, and provides a direct support to

Manuscript received February 25, 2006.
Manuscript revised February 28, 2006.

multithreading, code mobility and security, thus facilitating the development of concurrent and distributed applications. However, most of the middlewares exploiting Java do not often adequately support the programming and execution of dynamic parallel applications on wide-area complex network architectures, such as multi-domain, non-routable networks. Furthermore, many solutions proposed in literature often exploit tightly-coupled interaction and communication paradigms based on message-passing or the Java RMI package, and lack specific coordination mechanisms able to manage the resource variability in the configuration of metacomputers.

This paper presents a flexible middleware able to support the execution of large-scale, object-based applications over heterogeneous multi-domain, non-routable networks. In particular, the middleware, called *Java Multi-domain Middleware* (*JMdM*), can exploit computing resources hidden from the Internet, but connected to it through publicly addressable IP front-end machines, as computing nodes of a unique metacomputer. This way, all the computing power available within departmental organizations and non-publicly IP addressable can be harnessed to run applications without having to exploit low-level, "ad hoc" software libraries or specific systems or resource managers for grid computing, which could turn the development of parallel applications into a burdensome activity as well as penalize application performance. Finally, *JMdM* has been also developed according to the Grid reference model originally introduced in [7], and has been designed as a customizable collection of abstract components whose implementations can be dynamically installed to satisfy application requirements.

The outline of the paper is as follows. Section 2 presents *JMdM* and describes the architecture of the metacomputers that can be built by using the middleware. Section 3 describes the main implementation details of *JMdM*. Section 4 reports on some experimental results. Finally, in section 5 a brief conclusion is available.

2. The Architecture of the Middleware

JMdM is a Java-based middleware for metacomputing designed according to the reference model described in [7] and by which it is possible to build and dynamically reconfigure a metacomputer as well as program and run large-scale, object-based applications on it. In particular, the metacomputer can harness computing resources available on the Internet as well as those belonging to multi-domain, non-routable private networks, i.e. computing nodes not provided with public IP addresses, but connected to the Internet through one publicly addressable IP front-end node.

2.1 Services

The core services supplied by *JMdM* are: remote process creation and management, dynamic allocation of tasks, information service, directory service, authentication of remote commands, performance monitoring, communication model programmability, transport protocol selection and dynamic loading of components.

All services are accessible by user applications at run-time, whereas only some of them are accessible also through a specific graphical tool, called the *Console*. In fact, the services accessible through the *Console* mainly belong to three of the different layers defined in the Grid reference model [7]: *connectivity*, *resource* and *collective* layer.

The services belonging to the *connectivity* layer enable a programmer to discover the resources available on the network and to get information about their computing power. Such information can be then exploited to select a subset of the available resources and build a metacomputer. In particular, the run-time architecture of the metacomputer can be customized by selecting both the software components to be installed on each computing node and the transport protocol to be used for the communication among nodes.

The services belonging to the *resource* layer enables a programmer to automatically discover and reserve the required resources as well as configure the transport protocols to be used. This can be accomplished by issuing a specific resource reservation query through an XML file.

Finally, the services belonging to the *collective* layer make it possible to start an application and negotiate its computing needs with the available resources in order to correctly allocate application tasks on them.

2.2 Architecture of the Metacomputer

JMdM makes it possible to exploit collections of computing resources, called *hosts*, to build a metacomputer. *Hosts* can be PCs, workstations or computing units of parallel systems interconnected by heterogeneous networks. However, *JMdM* supplies all the basic services that enable a programmer to exploit the features of the underlying physical computing and network resources in a transparent way. To this end, the metacomputer appears to be abstractly composed of computational *nodes* interconnected by a virtual network based on a multi-protocol transport layer. In particular, the networked *nodes* can be arranged according to a hierarchical virtual topology in order to better exploit the

performances of dedicated interconnects or to harness the computing resources belonging to multi-domain, non-routable networks (see Figure 1). In fact, in such an organization, the *nodes* allocated onto *hosts* hidden from the Internet or connected by dedicated, fast networks can be grouped in *macro-nodes*, which thus abstractly appear as single, more powerful, virtual computing units. To this end, it is worth noting that a metacomputer is assumed to be made up by at least one macro-node, called the *main* macro-node, which groups all the nodes allocated on the publicly addressable IP hosts taking part in the metacomputer.

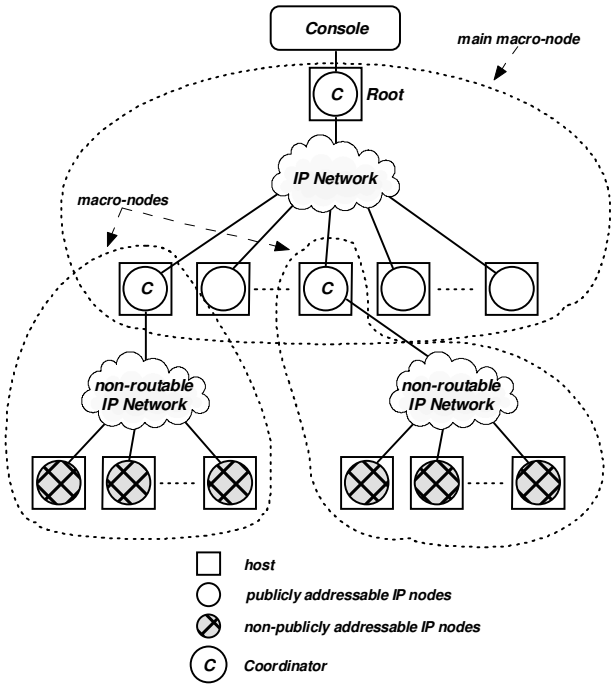

Fig. 1 The architecture of a metacomputer.

The internal nodes of a macro-node are fully interconnected. This means that they can directly communicate with the nodes belonging to the other macro-nodes making up the metacomputer.

Each node maintains status information about the dynamic architecture of the metacomputer, such as information about the identity and liveness of the other nodes. To this end, it is worth noting that the hierarchical organization of the metacomputer allows each node to keep and update only information about the configuration of the macro-node which it belongs to, thus promoting scalability, since the updating information has not to be exchanged among all the nodes of the metacomputer.

Each macro-node is managed by a special node, called *Coordinator* (C), which:

- is allocated onto the publicly addressable IP host of each non-routable, private network interconnecting other non-directly addressable IP hosts;
- creates the macro-node by activating nodes onto the available hosts within the private network;
- takes charge of updating the status information of each node grouped by the macro-node;
- monitors the liveness of nodes to dynamically change the configuration of the macro-node;
- carries out the automatic "garbage collection" of the crashed nodes in the macro-node;
- acts as an application gateway enabling nodes belonging to different macro-nodes of the metacomputer to directly communicate.

The metacomputer is controlled by the *Coordinator* of the *main* macro-node, called *Root*, which is directly interfaced with the user through the *Console*, by which the configuration of the metacomputer can be dynamically managed (see Figure 1). To this end, each host wanting to make its computing power available to *JMdM* runs a special server, called *Host Manager* (HM), which receives creation commands by the *Console*, authenticates them and creates the required *nodes* as processes running the Java Virtual Machine (see Figure 2).

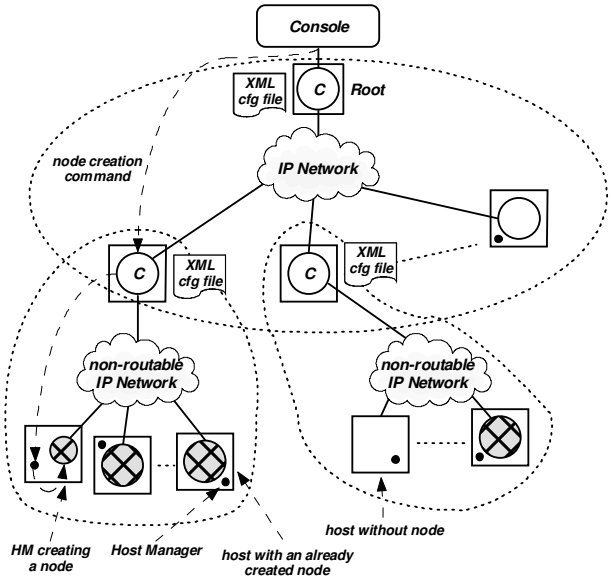

Fig. 2 The execution of a *node* creation command.

It is worth noting that the *Console* can create *nodes* only on publicly addressable IP hosts, i.e. the *nodes* that will belong to the *main* macro-node. However, when a *Coordinator* receives a creation command, it can create nodes inside the macro-node that it manages according to the configuration information stored in a specific XML

file provided by the administrator of the computing resources grouped by the macro-node and hidden from the Internet (see Figure 2).

All the nodes of the metacomputer can exchange messages through a communication interface independent of the transport protocol used by the hosts. In particular, this interface adopts a communication semantics based on the "one-sided" model [8] and has been designed so that each node can directly send objects. According to this model, objects can be either simple messages or tasks, and can be sent also to the nodes that do not store the object code. To this end, each node is provided with a code loader, which can retrieve the object code from a distributed code repository, called *Distributed Class Storage System* (DCSS), purposely developed to ensure scalability to the proposed middleware (see Figure 3).

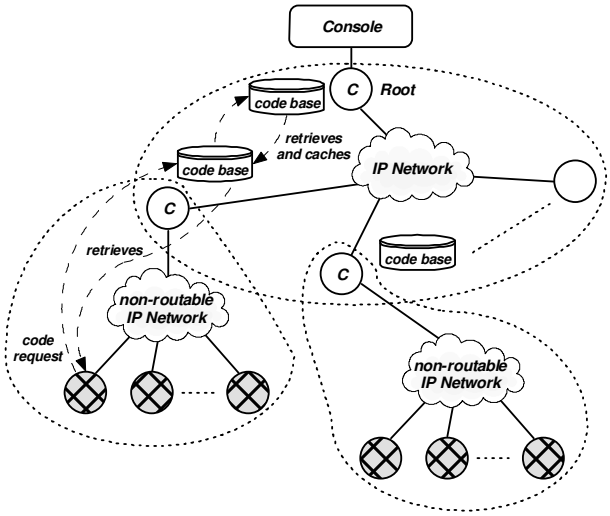

Fig. 3 The *Distributed Class Storage System*.

The architecture of the DCSS follows the global architecture of the metacomputer. Therefore, each macro-node is provided with a dedicated *code base* managed by the *Coordinator*. When a *node* needs a code, it requires the *Coordinator* of its macro-node to retrieve it. If the *Coordinator* lacks the required code, it asks the *Root* for it, which stores all the code of the running application. However, whenever a *Coordinator* obtains the required code, it stores it in a local cache, in order to reduce the loading time of successive code requests coming from other nodes of the macro-node.

JMdM also provides a "publish/subscribe" information service implemented by two distributed components: the *Resource Manager* (RM) and the HM (see Figure 4). To this end, each macro-node has an RM, which can be allocated on one of the hosts belonging to the macro-node. A RM is periodically contacted by the HMs of the hosts belonging to the macro-node and wanting to publish information about the CPU power and its utilization or the available memory or the communication performance. Information is collected by the RM and made then available to "subscribers", which are the *Coordinators* belonging to the metacomputer.

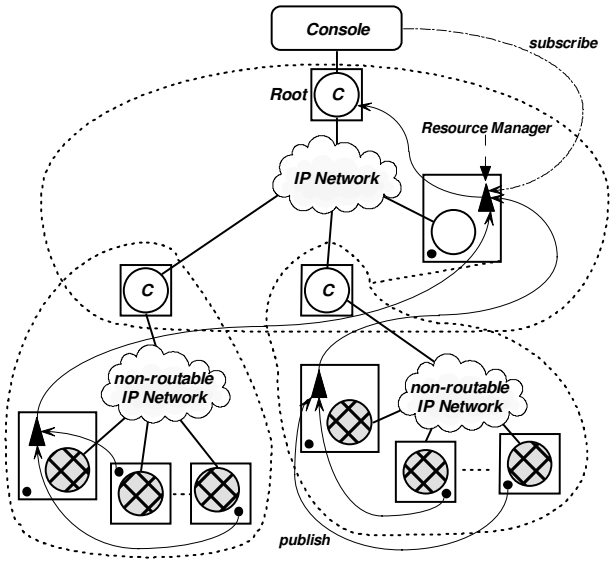

Fig. 4 The "publish/subscribe" information service.

Thus, each *Coordinator* can know the maximum computing/communication power made available by its macro-node. Furthermore, this information is also made available to the *Root*, which can thus know the power of all the macro-nodes making up the metacomputer. This allows the user to know the globally available computing power and reserve a part of it by issuing a subscription request to the *Console*. Then, the *Console* can ask the RM of the *main* macro-node for selecting and reserving only the computing resources required. The result of this process is an XML file containing all the current system information to create a metacomputer without having to consult anew the RM.

3. The Implementation of *JMdM*

In order to follow the rapid progress of technology and to better manage the heterogeneity of both computing and network resources in the context of metacomputing, *JMdM* has been designed according to a component-based, reflective architecture that enables the middleware to dynamically adapt to changes in the configuration of physical components [6].

3.1 Design Principles

All the services supplied by *JMdM* are implemented by several software components whose interfaces have been designed according to a component framework approach [9] in order to support adaptability and to make it possible to easily add further extended services.

A component is an encapsulated software module defined by its public interfaces, which follows a strict set of behavior rules defined by the environment in which it runs.

A component framework is a software environment able to simplify the development of complex applications by defining a set of rules and contracts governing the interaction of a targeted set of components in a constrained domain [10].

The framework proposed to develop *JMdM* is a set of co-operating interfaces that define an abstract design able to provide solutions for metacomputing problems. In particular, such design is concretized by the definition of classes, which implement the interfaces of the abstract design and interact with the basic classes representing the *skeleton* of the framework. In fact, this approach makes the metacomputer management easier, in that it gives the possibility of modifying or substituting a component implementation without affecting other parts of the middleware. In addition, it allows a component to have different implementations, each of which can be selected and dynamically loaded and integrated in the system. Therefore, *JMdM* has been designed as a collection of loosely-tied components, whose customization enables programmers to extend the basic services. To this end, the *JMdM*'s architecture supplies a set of integrated and interacting components, some of which represent the skeleton of the framework, whereas the others can be customized by the programmer in order to adapt the system to the specific needs of an application. Furthermore, to facilitate component integration, two design patterns [11] have been exploited: *inversion of control* and *separation of concerns*.

The former suggests that the execution of application components has to be controlled by the framework, and everything a component needs to carry out its tasks has to be provided by the hosting environment. Thus, the flow of control is determined by the framework, which coordinates and serializes all the events produced by the running application.

The latter allows a problem to be analyzed from several points of view, each of which can be addressed independently of the others. Therefore, an application can be implemented as a collection of well-defined, reusable, easily understandable modules. The clear separation of interest areas, in fact, reduces the coupling among modules, thus helping their cohesion.

3.2 Implementation

Coordinators and *nodes* are all implemented as processes able to load a set of software components either at start-up or at run-time. In particular, the main loaded components are (see Figure 5): the *Node Manager* (NM) and the *Node Engine* (NE).

The NM has the main task to store system information and to interact with the *Coordinator* in order to guarantee macro-node consistency. The NE takes charge of receiving application messages from the network and processing them.

More precisely, the NM implements some system tasks, and so it:

- monitors the liveness within the metacomputer by periodically sending control messages to the *Coordinator* of its macro-node;
- kills the *node* when the macro-node coordinator is considered crashed;
- creates the NE by using the configuration information supplied the *Coordinator* of its macro-node.

In addition, the NM loaded by a *Coordinator* takes also charge of setting-up the macro-node sub-network according to information retrieved either from the RM of the macro-node or from an XML file locally stored.

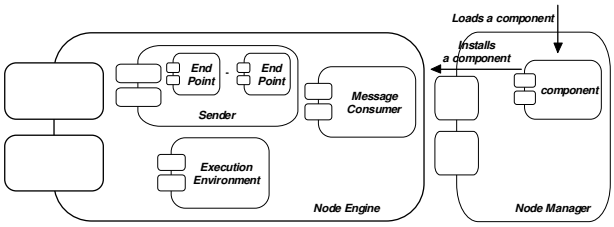

Fig. 5 The components of a *node* and their interactions.

The NE implements the node behavior by installing a set of components, which are dynamically loaded by a specific component of the NM, called the *Loader* (LD). In fact, the components loaded by NE are all configurable, and make it possible to customize the node behavior in order to provide applications with the necessary programming or run-time support. However, applications are not allowed to directly access the NE, but they can exploit its features or change its behavior through the NM. As a consequence, the NM also represents the interface between the application and the services provided by the middleware, such as the metacomputer reconfiguration and management.

The configurable components of the NE are: the *Execution Environment* (EE), the *Sender* (SD) and the *Message Consumer* (MC) (see Figure 5). They implement the abstractions that define the execution behavior of a *node*: a message, coming from the network, is passed to the MC, which is the interface between an incoming communication channel and the EE. Then, the message is synchronously or asynchronously delivered to the EE, which processes it according to the strategy coded in the EE implementation. As a result of message processing, new messages can be created and sent to other *nodes* by using the SD.

The EE defines the node behavior. It can contain either application components or data structures necessary to run parallel and distributed applications according to a specific programming model. As a consequence, since each *node* can handle a different EE implementation, MIMD applications can be run by simply distributing their components wrapped in the implementations of the EE of each *node*. Furthermore, EE can also dynamically control the configuration of the metacomputer as well as exploit the services implemented by the other node components, since it has access to the NM. Thus, the running application can both evolve on the basis of the configuration information characterizing the metacomputer and dynamically configure it, by adding, for example, a *node*, if more computing power is necessary.

The SD implements services for routing the messages generated on a *node* towards all the other *nodes* of the metacomputer. In particular, *JMdM* provides a default implementation for this component, called *Default Sender* (DS), which is loaded if any other SD is not installed by the user application.

The DS implements basic communication mechanisms able to exploit the multi-domain network organization of a metacomputer and to support the development of more sophisticated communication primitives, such as the ones based on synchronous or collective messaging. It exploits some components of the NE not accessible to the user, called *End Points* (EPs).

An EP is the local image of a remote *node* belonging to the macro-node. It manages a link to the *transport module* selected during the configuration phase of the metacomputer to enable communications, through a specific protocol, towards the remote *node* represented by the EP. To this end, it is worth noting that transport modules are characterized by a communication interface independent of the underlying transport protocol used to manage communications. Thus, a transport module implementing a new communication protocol or exploiting a native communication library can be easily integrated in *JMdM* by only developing a specific module, called *adapter*, able to abstract from the implementation details characterizing the low-level communications. In fact, every *adapter* has to carry out the serialization of the objects sent by the application according to the specific features implemented by the corresponding transport module. In particular, *JMdM* implements, by default, three *adapters*. The first is based on TCP, the second extends UDP by adding reliability, and the third is based on "Fast Messages" for Myrinet networks.

An MC is a component whose reference is passed to all the transport modules installed on the node. It implements the actions that have to be performed on the node, according to the strategy of message consuming defined by the programmer, whenever a message is received from the network.

4. Experimental results

This Section reports on some preliminary performance experiments conducted by exploiting two different clusters of PCs connected by a Fast Ethernet network.

The first cluster is composed of 8 PCs interconnected by a Fast Ethernet hub and equipped with Intel Pentium IV 3 GHz, hard disk EIDE 60 GB, and 1 GB of RAM. The second cluster is composed of 16 PCs connected by a Fast Ethernet switch and equipped with Intel Xeon 2.8 GHz, hard disk EIDE 80 GB, and 1 GB of RAM. All the PCs use the release 5.0 of the SUN JDK.

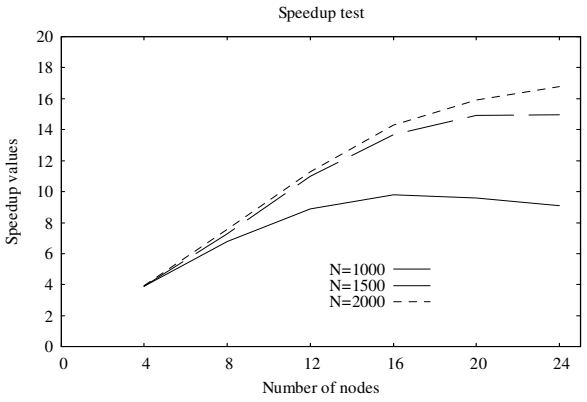

Fig. 6 Speedup values obtained by using all publicly addressable IP nodes.

Two classes of experiments have been conducted. In the former, all the PCs belonging to the clusters have been provided with public IP addresses, and this has made it possible to build a metacomputer characterized solely by the *main* macro-node. In the latter, the PCs belonging to the clusters have been configured so as to form two private, non-routable networks. Therefore, two macro-

nodes have been configured within the built metacomputer.

In both the experiments, the parallel product of two square matrices, whose size is *N*, has been executed. In particular, the product has been implemented by using the "striped partitioning": the right matrix is transferred and allocated on each node of the metacomputer, whereas the left matrix is split in groups of rows each one transferred to a different node.

The product has been programmed according to two different programming models: the "Send/Receive" (S/R) model and the "Active Objects" (AO) model. The former is the well-known message-passing model used to program parallel and distributed applications, whereas the latter is the model proposed in [11, 12] to overcome some limitations of the message-passing and RPC models.

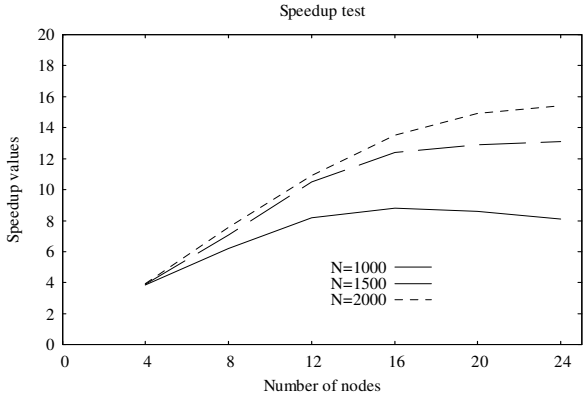

Fig. 7 Speedup values obtained by exploiting two PC clusters configured as non-routable private networks.

Both models have been implemented by customizing the NE components, according to what reported in the previous Section. Furthermore, each test program has been structured according to the SPMD computational model, in order to differentiate the instructions to be run by *nodes* from those to be run by *Coordinators*. Finally, the TCP transport protocol has been used among all the nodes of the metacomputer.

Both in the first and in the second experiment we have measured the speedup factor when the number of *nodes* building the metacomputer changes.

The results of the first test are reported in Figure 6. The curves show that the speedup values tend to reach a maximum at a number of nodes depending on the size of the matrices.

The second test has been conducted to exploit the peculiar characteristics of *JMdM*. In particular, the two clusters have been interconnected by a Fast Ethernet network and have been interfaced to a multi-homed PC used to only run the *Console*.

Figure 7 shows that the speedup values achieved with the configuration characterizing the second test are essentially similar to the ones obtained in the first test. In fact, Figure 8 shows that the penalization of the performance determined by the overhead caused by the management of the macro-nodes at run-time is rather limited and within 10%.

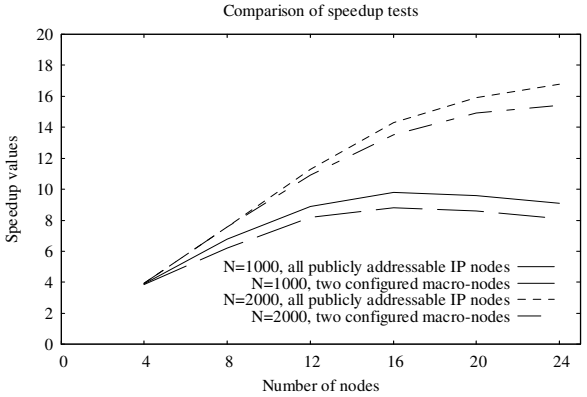

Fig. 8 Comparison of the speedup values shown in Figure 6 and in Figure 7.

Finally, it is worth noting that all the Figures shown above report on the performance achieved by *JMdM* in executing the product of two matrices programmed according to the S/R model. However, the same product programmed according to the AO model obtains a similar performance in the conducted tests, with differences within 7%. This demonstrates that the proposed middleware is characterized by a good flexibility that is achieved without reducing performance.

5. Conclusions

This paper has presented *JMdM*, a flexible, customizable and reflective middleware able to run distributed applications based on objects across heterogeneous, multi-domain non-routable networks. In particular, the middleware enables all the computing power available within departmental organizations and non-directly IP addressable to be harnessed to run applications without having to exploit low-level, "ad hoc" software libraries or specific systems or resource managers for grid computing, which could turn the development of parallel applications into a burdensome activity as well as penalize application performance.

The middleware also enables the building of metacomputers made up by abstract computing nodes, each of which is able to dynamically install interacting components that can be purposely customized in order to

adapt the programming model of the built metacomputer to the application needs.

Finally, *JMdM* has been tested by conducting some preliminary experimental tests programmed by implementing two different programming models. In fact, the obtained results demonstrate that flexibility can be achieved without reducing performance.

References

[1] V. Sunderam, J. Dongarra, A. Geist, and R. Manchek, The PVM concurrent computing system: Evolution experiences and trends, *Parallel Computing*, 20(4), pp. 531–547, 1994.

[2] F. Berman, G. Fox, and T. Hey, (editors), Grid Computing: Making the Global Infrastructure a Reality, Wiley, New York, 2003.

[3] I. Foster, and C. Kesselman, (editors), The Grid: Blueprint for a New Computing Infrastructure, 2nd edition, Morgan Kaufmann, San Mateo, CA, 2004.

[4] J. Joseph, and C. Fellenstein, Grid Computing, Prentice-Hall, Englewood Cliffs, NJ, 2003.

[5] F. Frattolillo, Running Large-Scale Applications on Cluster Grids, *Int'l Journal of High Performance Computing Applications*, 19(2), pp. 157–172, 2005.

[6] N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair, Towards a reflective component based middleware architecture, *Proceedings of the Workshop on Reflection and Metalevel Architectures*, Sophia Antipolis and Cannes, France, June 2000.

[7] I. Foster, C. Kesselman, and S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organizations, *Int'l Journal of Supercomputer Applications*, 15(3), pp. 200–222, 2001.

[8] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, Active messages: A mechanism for integrated communication and computation, *Proceedings of the 19th ACM Int'l Symposium on Computer Architecture*, Gold Coast, Queensland, Australia, pp. 256–266, May 1992.

[9] C. Szyperski, Component Software. Beyond Object-Oriented Programming, Addison Wesley, 1997.

[10] R. Johnson, and B. Foote, Designing reusable classes, *Journal of Object-Oriented Programming*, 1(2), pp. 22–35, 1988.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison Wesley, 1995.

[12] M. Di Santo, F. Frattolillo, *et al.*, An Approach to Asynchronous Object-Oriented Parallel and Distributed Computing on Wide-Area Systems, *Proceedings of the Int'l Workshop on Java for Parallel and Distributed Computing*, LNCS, vol. 1800, pp. 536–543, Cancun, Mexico, May 2000.

[13] M. Di Santo, F. Frattolillo, *et al.*, A portable middleware for building high performance metacomputers, *Proceedings of the PARCO Int'l Conference*, North-Holland, Naples, Italy, September 2001.

Franco Frattolillo received the Laurea degree "cum laude" in Electronic Engineering from the University of Napoli "Federico II", Italy, in a. y. 1989/90, and a Ph.D. in Computer Engineering, Applied Electromagnetics and Telecommunications from University of Salerno, Italy. He was a researcher at the Department of Electrical and Information Engineering of the University of Salerno from 1991 to 1998. In 1999 he joined the Faculty of Engineering of the University of Sannio, Italy, where he currently teaches "High performance computing systems", "Advanced topics in computer networks", "Network security", and "System Programming". He is also a research leader at the Research Centre on Software Technology of the University of Sannio and at the Competence Centre on Information Technologies of the Campania Region, Italy. He has written several papers in the areas of parallel programming systems and of cluster and grid computing. His research interests also include network security, digital watermarking, and DRM systems.