Programming Metasystems with Active Objects

M. Di Santo¹, F. Frattolillo¹, N. Ranaldo¹, W. Russo², E. Zimeo¹*
¹Department of Engineering – Research Centre on Software Technology
University of Sannio, 82100 Benevento, Italy
²DEIS – University of Calabria, 87030 Rende (CS), Italy

Abstract
The widespread diffusion of metasystems and grid environments makes it necessary to employ programming models able to well exploit a high, variable number of distributed heterogeneous resources. Many software frameworks designed for Grid computing do not address this problem. They only allow the use of existing programming libraries based on explicit message-passing communication models, often not suitable to manage the variability of a Grid. In this paper we present the customization of a component-based middleware for metacomputing, HiMM (Hierarchical Metacomputer Middleware), in order to support distributed programming based on the Active Object model provided by ProActive. This way a meta-system can be efficiently and transparently programmed by unifying the asynchronous remote method invocation model and the reflection provided by meta-objects.

1: Introduction

Today, the presence of a huge amount of computers and mobile devices, such as 3G mobile phones, high-performance laptops and PDAs, results in an increased focus on the interconnection of systems. This evolution of the computer scenario has generated new requirements for distributed application development and deployment and, consequently, promoted two new trends in distributed computing: Grid computing [6, 3], for scientific applications, and Web Services [8], for e-commerce and business applications. In both domains, the heterogeneity and the high, variable number of computing resources involved in a computation makes it important to improve customizability of middleware platforms [7] and to define suitable programming models in order to better exploit large-scale distributed systems.

Currently, many software environments for Grid computing have been released [11, 15, 23], but none of them introduces a new model for programming parallel applications featuring Grid characteristics. These environments, in fact, provide only core services to tackle the typical problems of large-scale, heterogeneous distributed systems, such as connectivity, security, naming, resource information, but they only allow programmers to use libraries for parallel and distributed programming, such as MPI and PVM.

Although the send/receive model used by these libraries has become the de-facto standard for communication in distributed systems, in order to handle the unpredictability of distributed applications, where the topology of their components is not assigned “a priori” and the interaction patterns are highly dynamic and mutable, more dynamic programming models, fully exploiting code mobility [13] and suitable coordination models, are required.

Agents [17] and Active Objects [22] are good candidates since they are able to dynamically create tasks and migrate computational entities and therefore to assure more flexibility in metasystems programming. However, agent programming, by raising the location of software components to the status of first class design concept, completely changes the way distributed applications are designed and deployed. On the other hand, active object models form a common high-level view of objects in distributed object-oriented systems and allow for taking advantage of well-defined theories and design techniques.

Even though many libraries [20, 2] for programming with active objects have been developed in twenty years of research, the emergence of Grids has introduced new issues (such as heterogeneity, scalability, unpredictability and adaptability) to be taken into account during program development. We think that the separation of aspects [19] gives the proper flexibility for programming distributed applications in wide-area, heterogeneous network environments, since computational entities could exploit communication, security and management

---

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
features of a middleware platform without affecting the functional aspects of an application. For these reasons, we have chosen the active object programming model provided by ProActive [5] to develop applications on Grid systems.

ProActive is designed to support parallel programming on a variety of hardware systems, such as SMPs, local area networks of PCs and Internet; it supports the separation of concerns related to functional and non-functional aspects of programming, but it lacks some features that make metasystems programming easier, more flexible and more efficient. For instance, programmers must explicitly manage resource information, resource discovery and object placement without any support from the underlying software platform. ProActive, in fact, is currently implemented on top of RMI that does not offer specialized services for Grid computing.

In this paper, we present the customization of a component-based middleware for metacomputing, HiMM [9, 10], in order to integrate the active object model provided by ProActive [5] with the communication, security and management services needed to handle a Grid. This way, a metasystem can be programmed by exploiting the asynchronous remote method invocation model and the reflective architecture provided by the runtime meta-objects of ProActive. In particular, through meta-objects, programmers can exploit the services of the underlying middleware to manage the distributed virtual architecture provided by HiMM, without affecting the functional code of the application.

The rest of the paper is organized as follows. Section 2 presents a brief overview of the active object model and of its implementation in ProActive. Section 3 describes HiMM and its customization features. Section 4 describes and discusses the integration of ProActive with HiMM. Section 5 shows and compares performance results. Finally, section 6 concludes the paper and introduces future work.

2: ProActive implementation of the Active Object model

ProActive is a Java library for seamless sequential, multi-threaded and distributed programming. It proposes a heterogeneous programming model characterized by passive and active objects. Passive objects are common objects whereas active objects are implicitly structured as actor-like entities. Actors [1], as well as common objects, unify both data and code in local states, called behaviors, and are dynamically created and referred through system-wide identifiers, called mail addresses. Actors manifest a pure reactive nature and, differently from common objects, interact with other actors solely via asynchronous, one directional message passing.

Messages are guaranteed to be delivered to their destinations with “finite delay” but the transmission order between two messages may not be preserved at delivery. In order to temporally uncouple communicating actors, incoming messages are buffered into mail queues associated to receiving actors, before being serially dequeued and processed. The processing of a message triggers the execution of the actor script, the code in the behavior of the receiver. A script consists of methods which specify the actions to be performed when messages are processed. During processing, new actors can be created, messages asynchronously sent and the current behavior substituted by a new one (replacement behavior). In practice, replacements implement local state changes, which can span from simple updates in the values of state variables to radical changes of the behavior.

The active object model implemented by ProActive introduces some differences to the described actor model: (1) behaviors can not be completely replaced, and only changes in the values of state variables are allowed; (2) mail addresses are handled as common object references and communication among active objects is not based on explicit message passing. Therefore, active objects look like passive ones but can be remotely created and accessed in a transparent way via asynchronous remote method invocations based on transparent future variables (a data driven synchronization mechanism called wait-by-necessity [4]). In addition, the ProActive programming model is hybrid, since both active and passive objects can coexist. This scenario enforces the application to be structured into multiple subsystems, each one composed of single active object and any number of passive objects (possibly zero) private to the subsystems. This property is satisfied by deeply copying passive objects when they are passed as method arguments.

An active object is structured according to the active object pattern [21]. Therefore, it is composed of a proxy, an activation queue, a scheduler and a servant, which are respectively named in ProActive proxy, body, thread of control, and object implementation. Through the use of the Proxy pattern and the automatic generation of stubs (by using the tool BCEL [25]), ProActive allows for invoking a method of an active object with the same syntax of local invocations and without the burden that characterizes the definition of remote objects in Java RMI [27]. Therefore, distributed applications can be programmed by following the same design techniques used for sequential programs and without the need to
explicitly manage active objects. In particular, by exploiting the Meta Object Protocol (MOP) [18] on which ProActive is strongly based, every passive object can be dynamically turn into an active one by invoking a primitive of the library that adds a proxy and a body to the passive object.

The body is responsible of handling the method calls intercepted and reified by the proxy. These calls, once transformed by the proxy in first-class objects, are then buffered in the activation queue, successively extracted by the scheduler and served following a specified scheduling policy. This policy can be programmed at the application level by exploiting the meta-properties offered by some programming interfaces (InitActive, RunActive, EndActive) that allow to customize a particular phase of the life cycle of an active object.

3: A brief introduction to HiMM

In this section, we briefly describe HiMM (for details see [10]) and its main features by using the layer model proposed in [12]. In particular, we introduce the architecture and then present some programming features and customization properties used to integrate ProActive with HiMM.

3.1. Distributed architecture

At the fabric layer, HiMM allows a user to exploit collections of computers (hosts), which can be workstations or computing units of parallel systems, interconnected by heterogeneous networks.

At the connectivity layer, HiMM allows for building a metacomputer composed of abstract nodes interconnected by a virtual network able to exploit the features of the underlying physical networks in a transparent way. To meet the constraints of the Internet organization, to better exploit the performances of dedicated interconnects and to assure scalability, the networked nodes can be virtually arranged according to a hierarchical topology which we refer to as HiM (Hierarchical Metacomputer). In this 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.

Each macro-node is controlled by a special node, the Coordinator (C), which: (1) creates the macro-node by activating nodes onto available hosts; (2) takes charge of updating the status information of each node grouped by the macro-node; (3) monitors the liveness of nodes to dynamically change the configuration of the macro-node; (4) causes, when it crashes, the automatic "garbage collection" of the nodes in the macro-node; (5) acts, from the application point of view, as a sort of gateway, in that it allows for communication among nodes of two adjacent levels. The coordinator of the highest hierarchical level is the root of a HiM. Differently from the other coordinators, it sees only one level, since, at the upper level, it is interfaced directly with the user through the Console.

Each host wanting to donate its computing power runs a special server, the Host Manager (HM), which receives creation commands by the console, authenticates them and, if the user is authorized, creates the required nodes as processes running the JVM. It is worth noting that the console creates nodes only at the highest hierarchical level of a HiM. However, if a coordinator receives a creation command, it takes charge of creating nodes inside the macro-node it controls, according to the configuration information stored in an XML [28] file provided by the domain administrator.

![Figure 1. The distributed architecture of a HiM](image)

At the resource layer, HiMM provides an information service based on the interaction between two distributed components: the Resource Manager (RM) and the HM. A RM, allocated on one of the hosts taking part in a macro-node, is periodically contacted by the HMs running on the hosts belonging to the same macro-node and wanting to publish information about: (1) the CPU power and its utilization; (2) the available memory; (3) the communication performance, if the host allocates the coordinator of a lower level macro-node. Information is collected by the RM and made available to subscribers. Thus, the macro-node coordinator can know the maximum computing/communication power made available by its level and considered at the upper level as...
the power of the macro-node. At the highest level, the
user can know the globally available computing power
and reserve a part of it by issuing, through the console, a
subscription request to the RM of the level. Currently, the
HiMM collective layer is under development.

3.2. Node structure and its programming features

To support adaptability and to facilitate a future
extension, HiMM implements its services in several
software components whose interfaces are designed
according to a Component Framework approach [24, 16].
In particular, the proposed framework is a set of co-
operating interfaces that define an abstract design useful
to provide solutions for metacomputing problems. The
abstract 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.

Both nodes and coordinators are implemented as
processes in which a set of software components are
loaded either at start-up or at run-time. The main
components are: (1) the Node Manager (NM), whose
main task is to interact with the coordinator in order to
guarantee macro-node consistency, to provide users with
services for writing distributed applications and to take
charge of some system tasks, such as the creation of new
nodes at run-time; (2) the Node Engine (NE), which
contains and integrates a collection of configurable
components that allow the node behavior to be
customized in order to provide applications with the
necessary programming or run-time support.

![Figure 2: Node components](image)

The NE components can be installed in the framework
by using a specific component of the NM that directly
interfaces the node to the network level: the Level
Manager (LM). After having loaded and initialized the
components, an LM invokes the methods that
characterize their life cycle in response to system
generated events. Therefore, application components do
not directly access the NE, but can use its features or
change its behavior through the NM.

The configurable components of the NE are: (1) the
Execution Environment (EE), the Level Sender (LS) and
the Message Consumer (MC). The EE defines the node
behavior. Each node can handle a different EE
implementation. Thus, HiMM can run MIMD
applications by distributing their components wrapped in
the implementations of the EE of each node. The EE of a
node may contain either the application components or
the data structures necessary to run parallel and
distributed applications according to a specific
programming model. Anyway, for the EE to control the
virtual machine and to use services offered by the other
node components, it must have access to the NM. This
way, the running application can both evolve on the basis
of configuration information related to a level (such as the
level size) and dynamically control the virtual
machine (for instance, by adding a node if more
computing power is necessary).

The LS exports services for routing the messages
generated on a node toward the other nodes reachable
through the controlled level. If a user does not install any
specific LS, HiMM provides a default implementation,
called Default Level Sender (DLS). The DLS implements
basic communication mechanisms able to exploit the
hierarchical organization of a HiM and to support the
development of more sophisticated communication
primitives, such as the ones based on synchronous or
collective messaging. More precisely, the DLS allows an
application to asynchronously send a message to: (1) a
node at the same level of the sender (send); (2) all the
nodes at the same level of the sender (limited broadcast);
(3) all the nodes at the same level of the sender and all
the other nodes recursively met in each macro-node
belonging to the level of the sender (deep broadcast).

Currently, the DLS uses the Java standard serialization
to send objects on the network through sockets or other
communication interfaces (such as Fast Messages for
Myrinet networks). In particular, in order to support
multiple transport protocols, the DLS uses a common
transport layer based on a one-way communication
interface for sending objects (in unicast, multicast and
broadcast mode). This way, the object serialization is
performed at the HiMM transport layer. In this layer,
object serialization can be optimized by using specific
features of the used transport protocol, while multicast
and broadcast communication can be improved by
serializing only once each sent message. In addition,
following this organization, it is possible to plug in a
more efficient version of Java serialization by only
modifying a transport module of the middleware. Currently, HiMM does not integrate a transport module supporting reliable IP multicast.

The Message Consumer implements the actions which have to be performed by a node whenever a message is received from the network according to the policy specified by the programmer.

4: ProActive over HiMM

In this section, we describe the library obtained merging ProActive with HiMM. The library has been called ProActive/HiMM (P/H) to distinguish it from the native one that we call ProActive/RMI (P/R).

The implementation of P/H has been easily performed thanks to the particular organization of both middleware platforms used. ProActive is heavily based on the Adapter Pattern [14], whereas HiMM is implemented following the Component Framework approach.

We have implemented specific adapters in order to substitute RMI with the HiMM communication facilities (see the HBodyAdapter in Figure 3). Moreover, through adapters, programs can use HiMM services at run-time by accessing the NM.

Figure 3 shows a piece of the P/H architecture composed of two nodes interconnected by the HiMM transport layer (which supports TCP, RUDP and Fast Messages). The node structure shows the three main components involved in a computation (EE, LS, and MC) and the binding between an active object and the NM. Each active object lives inside the EE of a node and it is accessible from a remote side through a series of components: the specific HiMM body adapter, the LS and the MC. Figure 3 also shows the critical path of a typical asynchronous remote method invocation on a remote active object and the return of the produced result.

P/H follows the same approach of P/R to remotely create an active object: it introduces an abstraction of a physical computational resource, named node, indviduated through an URL. Therefore an active object can be remotely created by using the following instruction:

```
Node n = NodeFactory.getNode("himm://1");
B b = (B) Proactive.newActive("B", params, n, …);
```

Differently from P/R, P/H uses a close distributed system whose nodes are identified by integers dynamically assigned to physical resources by the underlying middleware platform.

In the following we show a simple programming example used both to describe how to program ProActive applications and to analyze the P/H performance. The example is the well-known multiplication of square matrices implemented by using the strip partitioning of the left matrix. The right matrix is turned into an active object and replicated on each node of the metasystem.

To start the application, a specific EE component has to be implemented. The method start of this EE takes charge of reading input parameters and creating matrix objects on each node of the first level of the metacomputer. At this point the computation starts with the invocation of the method multiply that invokes the same method on each clone of the right matrix distributed in the level.
private Matrix[] multiply (Matrix mSx, Matrix[] gDx) {
    Method active, each specific request for the invocation of the remote active object is running. So, while the object is coordinator (nodeMgr.isCoordinator()), the method initActivity is executed when the active object is remotely created to verify if the node on which the object has been created is a coordinator (nodeMgr.isCoordinator()). In such case, the right matrix is encapsulated in an active object and replicated on each node inside the macro-node managed by the coordinator (ProActive.newActive).

    The method runActivity is executed when the remote active object is running. So, while the object is active, each specific request for the invocation of the method multiply is reified by the meta-object Service and served in different ways depending on the nature of the node. If the node is simple, the request is immediately served. If the node is a coordinator, the request is immediately served if the node is a coordinator, the request is immediately served.

5: Performance results

In this section we analyze the performance results shown by P/H and compare them with the ones obtained with P/R. Currently, we have conducted the performance analysis only on a flat architecture composed of six machines, each equipped with two Pentium II 350 MHz and a RAM of 128 MB, interconnected by a Fast Ethernet LAN. The software packages used are: Windows 2000, Java 2 SDK 1.3.1, ProActive rel. 0.9.3 [29], and HiMM rel. 1.0 [26].
We have conducted a first analysis to compare RMI with HiMM and P/R with P/H. In particular, in Figure 4 we report some measures of the time (RTT) between the invocation of a remote method (with one parameter and an empty body) and the return of an integer value, for a varying size of the parameter. The figure shows that for a small size (1 byte) of the parameter, RMI RTT (0.9 ms) is smaller than HiMM RTT (1.3 ms), but P/H RTT (7.8 ms) is smaller than P/R RTT (9.2 ms). In fact, even if the HiMM transport layer is not optimized, P/H behaves better than P/R due to the use of asynchronous messaging that allows low-level ProActive operations to be overlapped with communication.

A similar behavior is reported in Figure 5. This figure shows the number of remote methods that can be invoked in a second, for a varying size of the parameter. In this case, the improvement of P/H (126 m/s compared to 105 m/s) is higher because the asynchronous messaging exploits the communication bandwidth better than RMI.

A further analysis has aimed to measure the speedup factor by running a simple benchmark.

The benchmark, which is the product of two square matrices described in the previous section, has been executed with both P/H and P/R to measure the speedup factor with different sizes of the matrices.

As Figure 6 shows, the performance obtained with P/H is slightly better than the one obtained with P/R, especially when the size of the matrices is small. This is mainly due to the improvement of remote method invocation implemented by HiMM. When the size of the matrices is large, the execution time is dominated by the time of the matrix serialization, which is the same in P/H and P/R, so the improvement of latency for the remote method invocation that characterizes HiMM compared to RMI is irrelevant.

Figure 7 shows the execution times obtained by executing the matrix product with P/H, P/R and HiMM implementing the basic send/receive model (S-R/H). The overhead introduced by P/H compared to S-R/H is significant only when the matrix size is small.
6: Conclusions and future work

The paper describes the integration of ProActive with HiMM in order to create a new framework for easily programming Grid applications by using the active object model. The separation of concerns and the use of meta-objects allow programmers to exploit the underlying features of HiMM to program distributed aspects of applications without affecting functional code. This property enables code reuse and makes the object oriented approach effective for the development of distributed and parallel applications. An example validates our approach and shows how meta-objects can be effectively used to manage the hierarchical architecture of a metasystem at run-time. The same example is used also as a simple benchmark to demonstrate that HiMM assures better performance to ProActive and that the overhead introduced by the new framework is very small. In the future, we will conduct further experiments with a larger, hierarchical distributed system composed of at least two clusters of PCs characterized by different computational power, communication hardware and protocols. To this end, we will use the heterogeneous programming hints module integrated in HiMM (not used in the example shown above) to dynamically manage resources in order to minimize the computing time.

7: References

[29] http://www-sop.inria.fr/oasis/ProActive/.