Resource Management for Parallel Adaptive Components

Luc Courtrai, Frédéric Guidec, Nicolas Le Sommer, Yves Mahéo

Resource Management for Parallel Adaptive Components

Luc Courtrai, Frédéric Guidec, Nicolas Le Sommer, Yves Mahéo
VALORIA, Université de Bretagne-Sud, France
{Luc.Courtrai|Frederic.Guidec|Nicolas.Le-Sommer|Yves.Maheo}@univ-ubs.fr

Abstract

This paper reports the development of the Concerto platform, which is dedicated to supporting the deployment of parallel adaptive components on clusters of workstations. The current work aims at proposing a basic model of a parallel component, together with mechanisms and tools for managing the deployment of such a component. Another objective of this work is to define and implement a scheme that makes it possible for components to perceive their runtime environment. This environment is modelled as a set of resources. Any component can discover and monitor resources, using the services offered by the platform.

1 Introduction

Clusters of computers are now found in an increasing number of laboratories and companies where they take on various forms. They may for instance play the role of low-cost supercomputers. A group of workstations connected via a high performance network may also be used as a cluster, even though these workstations are still exploited for common purposes. Such clusters can make interesting resources for Grid Computing. In this perspective, the objective may be to deploy parallel applications on computing infrastructures that contains several clusters. However, proposing a software solution that allows the development of applications that take benefit from these architectures is still a challenge.

Several approaches can be envisaged for designing and deploying an application that exploit one or several clusters. Among them, the component approach is worth being studied. It allows complex applications to be designed by assembling available components. We consider in this paper the case where each component is designed as a "parallel code" that is to be deployed on a cluster

Even if it is possible to design a component in an *ad hoc* manner so that it exploits at best a specific architecture, it is preferable to favour the component's portability. Each component should ideally be deployable on a large range of clusters. To achieve this goal, one may think of extending the notion of virtual machine to the entire cluster in order to hide the specificities of the underlying hardware platforms. Another approach, that we explore, consists in enforcing the adaptability of the component so it can allow for the hardware and software specificities of the platforms on which it is deployed.

Component adaptation has several facets. Auto-adaptation occurs when the component itself is responsible for deciding to adapt its behaviour to external conditions. Such a component can be referred to as an *adaptive* component. In other cases, it is the environment, especially the hosting platform, that enforces an adaptation strategy by imposing that a component modify its behaviour according to explicit directives. In this case, the component, must have been designed so as to be able to receive such directives from the platform. Such a component can be referred to as an *adaptable* component. The difference between these two forms of adaptation is not discussed any further in this paper. Instead, we focus on the basic mechanisms that are necessary in both cases to provide information that makes possible the decision process and leads to adaptation.

If the adaptive component has the possibility to obtain information on the target architecture, it will be able to choose an appropriate configuration when it is deployed. The state of the platform can be expressed in the form of qualitative and quantitative information covering aspects such as the number of nodes in the cluster, the computation power of these nodes, the bandwidth offered by the communication links, the availability of a given peripheral device, or that of a specific software library. However, an initial configuration of the component may not always be sufficient. Whenever a component cannot be deployed on a dedicated platform, it may have to share resources with other components, and even with other applications. In such circumstances new conditions may arise at runtime, requiring that components change their behaviour accordingly. A component should thus have means to dynamically gather information pertaining to its environment, so that it can adapt itself, for example by redistributing data, by balancing its load differently, or by choosing a new algorithm. The kind of information

that can help take such decisions is for instance the CPU load observed on a given node, or the bandwidth available on a link.

This paper describes the Concerto software platform. This platform supports the deployment of parallel components on a cluster, and it provides these components with means to adapt themselves. The platform is dedicated to the deployment and the support of parallel components written in Java. Information that feeds adaptation decisions comes from the observation of resources. In this particular context the term "resource" has a broad meaning. We thus envisage to deal with:

- "system" resources such as the memory or the CPU of each cluster node, or a scientific computing library;
- "conceptual" resources related to the application itself, such as the sockets or threads used by a component.

Our aim is to develop mechanisms that make it possible to collect information related to a non-limited set of such resources. The infrastructure we propose is extensible, as it is designed in such a way that new types of resources may be easily allowed for when needed. The Concerto platform can thus evolve so as to take into account new hardware and software specificities of clusters.

The remaining of this paper is organized as follows. Section 2 introduces the basic model of parallel components we propose, and it describes the implementation of the deployment mechanisms. The modelling of resources within the platform and the tools that permit the observation of their state are presented in section 3. Section 4 concludes the paper.

2 Parallel Components

Component-based application development is already possible through the use of component technologies proposed by industry, such as Microsoft COM [10] or Sun's Enterprise Java Beans. OMG (Object Management Group) also develops its own solution with CORBA Component Model [11]. However theses technologies have not been designed to support parallel components, that is, components that involve parallel activities. Some preliminary work has been carried out on models or platforms that target parallel components [1, 5]. In this work the main objective is to reuse large scientific code relying on data parallelism.

The Concerto platform is dedicated to adaptive parallel components. Although the concept of parallel component is central to our project, our aim is not to propose a new component model, but to provide an infrastructure that enforces the adaptability of components. In this perspective, we propose below a minimal definition of what we mean by parallel component in Concerto.

A programmer who wishes to develop a parallel component for Concerto must design his component as a set of cooperating threads. He must also define the factory part of the component (name, interface, implementation). The platform offers facilities for managing non-functional aspects of the component.

Component interface

A parallel component hosted by the Concerto platform has three interfaces.

- *Factory interface*: No constraints are put on the type of the factory interface. The programmer may for example propose an interface based on Java RMI. The component is then an object implementing the *Remote* interface, whose methods will be called remotely by the clients of the component. The component may also be a server listening to a port of the machine on which a client must open a socket. One may also design a distributed interface (that is to say, an interface associated with several objects that each implement a part of the interface) in order to be connected in parallel with another parallel component.
- *Life cycle interface*: Through this interface, the different steps of the life of the component can be controlled. To date, this mainly covers deploying the component on the cluster, and stopping this component. In the future it should be possible to distinguish between several phases within the deployment process, and to propose persistence services.
- *Resource interface*. The component exhibits a resource interface through which the user accesses information related to the resources used by the component. More precisely, the component itself is considered as a resource in the Concerto platform. Hence Concerto components are required to implement an *observe()* method that returns an observation report (see Section 3 for more details). By default, the observation report generated by a component aggregates the observation reports on all the resources it uses. If the component programmer finds it useful (for example for security reasons), he may restrict the amount of information disclosed to the component's client by defining a particular type of observation report.

Internal structure of a component

When building a parallel component, the programmer develops a set of Java threads (actually a set of classes implementing the *Runnable* interface). These threads cooperate to perform the methods of the component's factory interface.

Threads are gathered into placement (or distribution) entities called fragments. A fragment is a set of threads that belong to a single component, and that are bound to run within the same virtual machine on the same cluster node. These threads will thus be able to share a common object space. Communication and synchronization between the threads of a fragment are performed just like in any other multi-threaded Java program. On the other hand, threads that belong to different fragments must rely on external communication and synchronization mechanisms, such as sockets and RMI.

Component deployment

In order to deploy a component on a cluster, one must provide a description file for this deployment. This file describes:

- the component's structure (expressed in terms of the fragments and threads to be deployed);
- placement directives for the fragments. One can for example duplicate some fragments on all cluster nodes, or assign a fragment to a specific node.
- constraints imposed by the component in order for its deployment to be feasible (availability of a specific version of the JVM, of a RMI registry, etc.)

We have developed an XML dialect that permits the specification of such directives. The Concerto platform can parse this dialect so as to ensure the deployment of components on a cluster.

3 Resource Modelling and Control

Motivation and main principles

The main objective of project Concerto is to provide software components with means to perceive their runtime environment so that they can adapt their behaviour to the state of this environment, and to its variations. We are developing in Java a software platform in which the runtime environment of a component is modelled using objects that reify the various resources offered.

As a general rule, we qualify as "resource" any hardware or software entity a software component may use during its execution. The resources considered to date in the Concerto platform include system resources (CPU, system memory, swap, graphical user interface, network interface, etc.) that characterize chiefly the underlying hardware platform, as well as "conceptual resources" (sockets, processes, threads, directories, files, RMI server, etc.) that rather pertain to the applicative environment considered.

Since a software component is liable to use all or some of the resources available in its environment, the Concerto platform must provide mechanisms that make it possible for the component:

- to check the availability of any resource (or kind of resource) in its environment;
- to discover the existence of a specific resource (or kind of resource);
- to ask for the status of a specific resource;
- to ask that the platform notifies the component when a given condition is reached regarding the status of a specific resource.

Since the Concerto platform is dedicated to the deployment of parallel software components on a cluster of machines, the dissemination of resources over the nodes of this cluster must be allowed for. The above-mentioned mechanisms must thus make the distribution of resources transparent for components. A thread belonging to a component must be able to gather information not only about the resources available on the node it runs on but also about distant resources and resources distributed over whole cluster.

Modelling of system and conceptual resources

Any kind of resource liable to be used by components deployed on the Concerto platform must be reified as Java objects. We have thus started the development of a class hierarchy in order to model these resources. This hierarchy is partially reproduced in Figure 1. It is meant to be extended as new resource types are allowed for in the platform.

Some of the classes shown in Figure 1 model resources that pertain to the hardware level. Classes *CPU*, *Memory*, and *NetworkInterface* belong to this category. The class *ClusterNode* is used to aggregate the three former classes, so that any cluster node can be modelled as a single resource object.

Other classes shown in Figure 1, such as classes *Socket* and *Thread*, are standard classes defined in the JDK *(Java Development Kit)*. Their implementation was revised in project Concerto in such a way that the state of the conceptual resources they model can be observed at runtime.

Classes *Fragment* and *Component* were introduced in order to define functionalities that are specific to project Concerto. With these classes, a parallel component and a component fragment can be perceived as resources in the cluster they are deployed on. One can therefore benefit from the services implemented in Concerto to manage the components deployed on a cluster, as well as the fragments deployed on any cluster node.

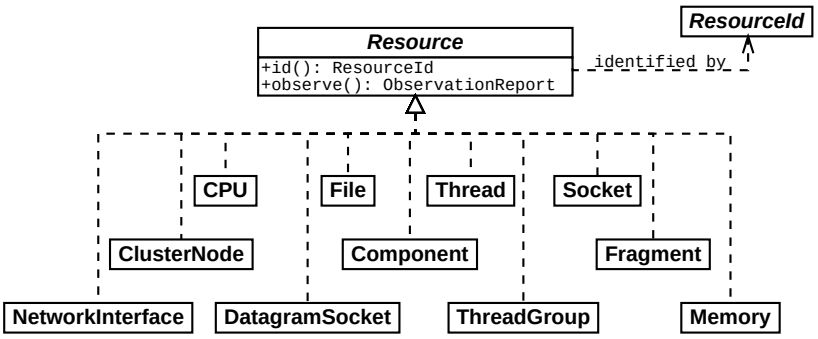

Figure 1. Object-based modelling of a few resource types.

Observation reports

Observation reports make it possible to gather information about the state of the many kinds of resources distributed in a cluster in a homogeneous way. A hierarchy of Java classes was developed in order to allow the generation, the collection, and the management of such reports (see Figure 2). Any Java object that models a resource in the Concerto platform implements method *observe()*, which returns a report about the current state of the resource considered. A report is modelled as a Java object that implements the *ObservationReport* interface. Of course the actual content of the report depends on the type of the resource considered. Hence, when method *observe()* is called on a *Thread* object, this object returns a report of type *ThreadReport*. The class *ThreadReport* provides pieces of information that characterize the state of the thread object considered (current priority level, amount of CPU and memory consumed since this thread was started, etc.). Likewise, calling method observe() on a Memory object returns a *MemoryReport*, which provides information about the current state of the system memory.

Resource identification and tracking

In the Concerto platform, all resources are modelled as Java objects that can be created and destroyed at any time. Nevertheless, any resource object must be identified unambiguously. Therefore, the platform implements mechanisms for identifying and tracking resources at runtime. These mechanisms rely on a naming system that gives any resource a unique name. Whenever a resource object is created on a cluster node, this object is given a unique identifier (object of type *ResourceId*, see Figure 1). Moreover, the resource object created is immediately registered within a resource manager (object of type *ResourceManager*), whose function is to identify and to keep track of any existing resource object.

An instance of class *ResourceManager* is created on each cluster node whenever a new component is deployed. This resource manager permits the identification, location, and collection of observation reports for:

- the conceptual resources used by the component it is associated with;
- the system resources of the cluster (which are considered as global resources shared by all components);
- the other components that have been deployed on the cluster (remember that each component is perceived as a resource, and can therefore produce an observation report on demand).

Search patterns make it possible to search for resources, and to collect observation reports from these resources selectively. They model search strategies as Java objects. For example, one can create a search pattern object that describes a local search strategy (i.e. search limited to a specific node of the cluster), and another object that describes a global search strategy (i.e. search performed on all the nodes of the cluster). The interface *SearchPattern* serves as the root of a hierarchy of classes that each describe a specific search strategy (Figure 3).

The following segment of code shows how a resource manager can be called when looking for specific resources. In this example several kinds of search patterns are used in order to specify that the search should apply (1) to local resources only; (2) to the resources located on a remote node whose identity is specified as an argument; (3) to the whole cluster.

```
ResourceManager manager = ResourceManager.getManager();
Set localIds = manager.getResourceIds(new LocalSearch()); // (1)
Set remoteIds =
   manager.getResourceIds(new LocalSearch(remoteNodeId)); // (2)
Set allIds = manager.getResourceIds(new GlobalSearch()); // (3)
```

Once the identity of a resource object has been obtained, one can require that the resource manager collects and re-

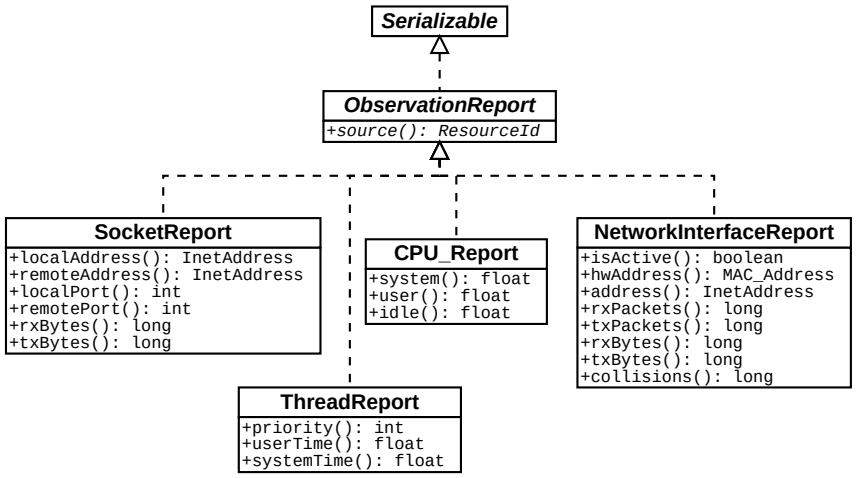

Figure 2. Observation reports modelling.

turns an observation report concerning this object. Whether the resource object considered is local or remote remains transparent for the caller.

The segment of code shown below is a continuation of the former example. The caller calls the resource manager and ask for an observation report on a specific resource object (assuming that the value of *resId* was extracted from one of the three id sets collected in the former example).

```
[...]
ObservationReport report = manager.observe(resId);
```

In the current implementation of the Concerto platform, resource managers rely on the RMI mechanism for exchanging information. The objects modelling search patterns and observation reports are all serializable, so they can be transmitted between two nodes using the RMI mechanism. This characteristic shows in the class hierarchies reproduced in Figures 2 and 3.

Resource classification and selection.

The resources registered within a resource manager can be of various types (eg *CPU*, *Memory*, *Socket*, *Thread*, *File*, etc.). The Concerto platform implements mechanisms for classifying and selecting resources based on the notion of "resource pattern".

The interface *ResourcePattern* (see Figure 3) defines a function *isMatchedBy()*, which takes a resource object as a parameter, and returns a boolean whose value depends on whether this object satisfies the considered selection criterion or not. In a simple case resource selection can rely on the actual type of the resource object which is submitted to the test. Hence, in the class *CPU_Pattern* (which implements interface *ResourcePattern*), the method *isMatchedBy()* simply checks that the object passed as a parameter is a *CPU* object. But one can also implement more sophisticated selection mechanisms. For example the class *SocketPattern* is implemented so as to achieve the selection of socket objects based on criteria that take into account not only the type of the resource (this object must be of type *Socket*), but also the local and remote IP address and ports associated with this socket, as well as the number of bytes sent and received via this socket.

The following example shows the creation of three resource patterns. The first pattern permits the search for and selection of parallel components. The second pattern permits to select specifically resource objects that model the CPUs in a cluster. The third pattern makes it possible to select only those resource objects that model sockets resources, and that additionally satisfy the following selection criteria: the IP address of the remote host must belong to the 195.83.160/24 network, and the remote port must be in the range 0 to 1023. On the other hand, the local IP address and port the socket is bound to can take any value.

```
ResourcePattern componentPattern = new ComponentPattern();
ResourcePattern cpuPattern = new CPU_Pattern();
ResourcePattern socketPattern =
    new SocketPattern(InetAddress.AnyAddress, "195.83.160/24",
            PortRange.AnyPort, new PortRange(0, 1023));
[...]
```

The resource manager can handle requests that take a *ResourcePattern* object as a parameter. One can thus request that the resource manager locate a specific set of resources, or collect and return observation reports from these resources. For example, assume that the resource manager receives a search request with the *componentPattern* de-

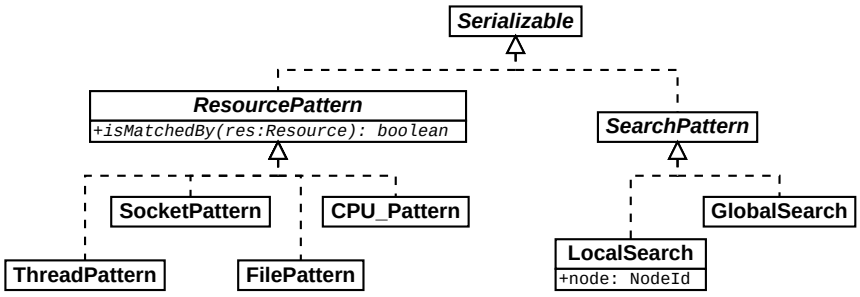

Figure 3. Modelling of the patterns that can be used to select resources (left-hand side of the hierarchy) and to describe search strategies (right-hand side of the hierarchy).

fined in the former example. It will thus search for those resources that match this pattern, and return only the identities of component resources. On the other hand, if the resource manager is required to search for those resources that match the socket pattern, it will only consider the sockets created by the calling component, and will select among these sockets those whose characteristics (IP addresses, port numbers, etc.) match the *SocketPattern*.

Implementation details

The mechanisms we use in Concerto for modelling resources and allowing their observation have a larger scope than that of adaptive parallel components. These mechanisms have been gathered in an environment called RAJE *(Resource-Aware Java Environment)*. RAJE permits the reification and the observation of system resources. Moreover, it defines the main schemes for identifying, localizing and observing resources. The Concerto platform is based on the RAJE environment which is extended in order to allow for specific notions such as parallel components or fragments.

The RAJE environment and the Concerto platform are presently implemented on Linux and rely on a variant of the Kaffe 1.0.6 JVM. Details on RAJE can be found in [7]. This paper describes how resource observation is implemented and how consumption shares are imputed to Java threads. Article [9] presents also RAJE (and the JAMUS platform built upon RAJE). Namely, the services provided by RAJE are compared to those offered by others tools such as JRes [4], GVM [3], KaffeOS [2]. Naccio [6] Ariel [8], etc.

4 Conclusion

This article presents the Concerto platform, which aims at allowing the deployment and the support of parallel components on clusters of workstations. Ongoing work focuses on proposing a basic parallel component model, as well as tools for the deployment of such components. Our objective is, in a first stage, to adopt as simple and versatile a model as possible. We rather put the emphasis on the development of mechanisms useful to adaptation. Indeed, the clusters targeted by the Concerto platform are mainly non-dedicated clusters composed for example of several workstations that are shared by several components, and even several applications and users. The runtime environment of parallel components is heterogeneous and may vary along the component's execution. So we have designed tools that provide the components with means to perceive their execution environment and its variations. The components' environment is likened to a set of resources. Each component can discover the existence of a particular resource and observe its state thanks to the services the platform offers.

The development of the Concerto platform is still in progress. The deployment tool, that includes a graphical frontend, should be extended. Besides, we intend to make interaction mechanisms available to the components so that they can command the platform to inform them on changes in the state of resources. This implies the definition of a formalism that can be used by components to describe interesting events and to implement a notification scheme that allows for the distributed aspects of the components.

Acknowledgment

This work is supported by the French Ministry of Research in the framework of the ACI GRID program.

References

[1] R. Amstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Towards a Common Component Architecture for High-Performance Scientific Computing. In *Proc. of the 8th International Sym-*

posium on High-Performance Computing, Redondo Beach, California, Aug. 1999.

[2] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *4th Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[3] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Techniques for the Design of Java Operating Systems. In *USENIX Annual Technical Conference*, June 2000.

[4] G. Czajkowski and T. von Eicken. JRes: a Resource Accounting Interface for Java. In *ACM OOPSLA Conference*, 1998.

[5] A. Denis, C. Pérez, and T. Priol. Towards High Performance CORBA and MPI Middleware for Grid Computing. In *Proc. of 2nd International Workshop on Grid Computing*, Denver, Colorado, Nov. 2001.

[6] D. Evans and A. Twyman. Flexible Policy-Directed Code Safety. In *IEEE Security and Privacy*, May 1999.

[7] F. Guidec and N. Le Sommer. Towards Resource Consumption Accounting and Control in Java: a Practical Experience. In *ECOOP'2002, Workshop on Resource Management for Safe Languages*, Malaga, Spain.

[8] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular Real-Time Resource Management in the Rialto Operating System. In *5th Workshop on Hot Topic in Operating System (HotOS-V)*, May 1995.

[9] N. Le Sommer and F. Guidec. A Contract-Based Approach of Resource-Constrained Software Deployment. In *Proc. of the 1st International IFIP/ACM Working Conference on Component Deployment (CD'2002)*, Berlin, Germany.

[10] Microsoft. The Component Object Model Specification. Technical report, Microsoft Corporation, Oct. 1995.

[11] OMG. CORBA Components. Technical Report OMG-orbos-99-07-01, OMG TC Documents, July 1999.