Dynamically-Fulfilled Application Constraints through Technical Services

Towards Flexible Component Deployments

Denis Caromel, Christian Delbé, Alexandre di Costanzo, and Matthieu Morel
INRIA Sophia - I3S - CNRS - Université de Nice Sophia Antipolis
Email: First.Last@sophia.inria.fr

***Abstract*— We propose in this paper, a mechanism for Grid computing frameworks, for specifying environmental requirements that may set and be optimized by deployers. Specified by designers by parameterizing deployment abstractions, the constraints can be dynamically mapped onto the infrastructure. This work is integrated in the ProActive middleware with the concept of *technical services*.**

We illustrate this mechanism with a concrete use case: deploying a component-based application with fault-tolerance on an heterogeneous grid provided by the ProActive Peer-to-Peer infrastructure.

I. Introduction

Software engineering defines clear separations between the roles of actors during the development and usage of a software component. In particular, a designer is expected to specify not only the functional services offered or required by a component, but also the conditions on the environment that are required for a correct deployment - so that the deployer can fulfill her or his task. The designer must therefore have a way to specify environmental requirements that must be respected by targeted deployment resources. The deployer, from his knowledge of the target infrastructure, must be able to specify optimized and adequate adaptations or creations of the resources. Programmers of applicative components, who should mostly concentrate on the business logic, may be provided with abstractions for the distribution of the components; deployment requirements can be specified on these abstractions.

In the context of Grid computing, current platforms are falling short to express these deployment requirements, especially dynamically fulfillable ones, i.e. requirements that can be fulfilled in several manners at deployment time. Adding and configuring fault-tolerance, security, load balancing, etc. usually implies intensive modification of the source code.

In this paper, we propose a mechanism for Grid computing frameworks, for specifying environmental requirements that may be optimized by deployers.

These requirements are specified by designers by parameterizing deployment abstractions, and are fulfilled dynamically by the deployers. Practically, we propose a mechanism for specifying deployment constraints and dynamically applying them on deployment infrastructures. Deployment constraints are specified on Virtual Nodes, which are deployment abstractions for the ProActive Grid middleware and have been used in the Grid Component Model proposal from the European Coregrid network, a European academic Network of Excellence (NoE).

This paper is organized as follows: Section 2 introduces the ProActive Grid middleware: it provides an overview of the programming model based on active objects, and briefly presents the component programming capabilities. Section 3 presents the deployment framework, which is based on XML descriptors and virtual nodes. We show that deployment resources may be provided either by creating them or by acquiring them. Section 4 considers the problem of specifying and applying deployment requirements, shows how some requirements may be applied dynamically as technical services and proposes a specification of deployment requirements on virtual nodes. Section 5 sums up the different concepts into a concrete example involving the deployment of components with fault-tolerance requirements on a Peer-to-Peer infrastructure: fault-tolerance is fulfilled dynamically.

II. The ProActive Grid Middleware

ProActive is a Java library for concurrent, distributed and mobile computing originally implemented on top of RMI [1] as the transport layer, now HTTP, RMI/SSH, and Ibis are also usable as transport layer. Besides RMI services, ProActive features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronisation mechanisms, and migration of active objects with pending calls. As ProActive is built on top of standard Java APIs, neither does it require any modification to the standard Java execution environment, nor does it make use of a special compiler, preprocessor or modified Java Virtual Machine (JVM).

A. Base Model

A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous

with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [2]. There is a short rendezvous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee. The ProActive library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call.

B. Component based programming

In addition to the standard object oriented programming paradigm, ProActive also proposes a component-based programming paradigm, by providing an implementation [3] of the Fractal component model [4] geared at Grid computing.

Fractal is a modular and extensible component model, which enforces separation of concerns, and provides a hierarchical structure of component systems. Because it is a simple though extensible model with clear specifications, Fractal has been chosen as a base for the Grid Component Model, currently under specification in the CoreGrid European NoE.

In the implementation of the Fractal model with ProActive, components are implemented as active objects, therefore all underlying features of the library are applicable to components.

The deployment of components is addressed in two ways: with a dedicated and standardized Architecture Description Language (ADL) [5] for describing Fractal components, and with the ProActive deployment framework described in the following section.

III. Deployment framework

A software component is often defined as a unit of deployment. This implies that the deployment phase is fundamental in the component-based programming paradigm. The deployment phase consists of several activities, and usually involves first a packaging activity, where all artifacts and configuration descriptions are integrated (release step). Further activities involve the configuration of the system and components so that the components can be instantiated on selected resources, and the selection and allocation of suitable resources.

In this section, we describe the deployment framework provided with the ProActive library, focusing on the description and on the selection of resources.

A. Deployment model

As Szyperski points out in [6], most industrial component models follow the concept of attribute-based programming, and attributes are factored out and placed in separate XML-based deployment descriptors. This enables a clean object of manipulation for the deployment step, and factors the roles of the designer (who places custom attributes) and of the deployer (who manipulates configuration files). This concept can be used for both applicative description (ADL attributes) and infrastructure description.

In the ProActive library, the deployment model is articulated around three concepts: component assembly through ADL, virtual nodes, and deployment descriptors, which are described in the next section.

B. Descriptor-based Deployment of Grid Applications

The deployment of grid applications is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers and clusters. The commoditization of resources through grids and the increasing complexity of applications are making the task of deploying central and harder to perform.

ProActive succeeds in completely avoiding scripts for configuration, getting computing resources, etc. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code so as to gain in flexibility [7].

1) Principles: A first key principle is to *fully* eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols,

The goal being to deploy any application anywhere without changing the source code. The deployment sites are called *nodes*, and correspond for ProActive to JVMs which contain active objects.

A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities.

To summarize, in order to abstract away the underlying execution platform, and to allow a *source-independent deployment* a framework has to provide the following elements:

- an abstract description of the distributed entities of a parallel program or component,
- an external mapping of those entities to real *machines*, using actual *creation, registry*, and *lookup* protocols.

2) XML deployment descriptors: To answer these requirements, the deployment framework in ProActive relies on XML descriptors. These descriptors use a specific notion, *Virtual Nodes* (VNs):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,
- a VN, after activation, is mapped to one or to a set of *actual ProActive Nodes*, following the mapping defined in an XML descriptor file.

A virtual node is a concept of a distributed program or component, while a node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between virtual nodes and nodes: the function created by the deployment, the mapping. This mapping is specified in the deployment descriptor. There is no automatic mapping between virtual nodes and active objects: the active objects are deployed by the application onto nodes given by a virtual node. By definition, the following operations can be configured in the deployment descriptor:

- the mapping of VNs to nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup JVMs.

Figure 1 summarizes the deployment framework provided by the ProActive middleware. Deployment descriptors can be separated in two parts: mapping and infrastructure. The VN, which is the deployment abstraction for applications, is mapped to nodes in the deployment descriptors, and nodes are mapped to physical resources, i.e. to the infrastructure.

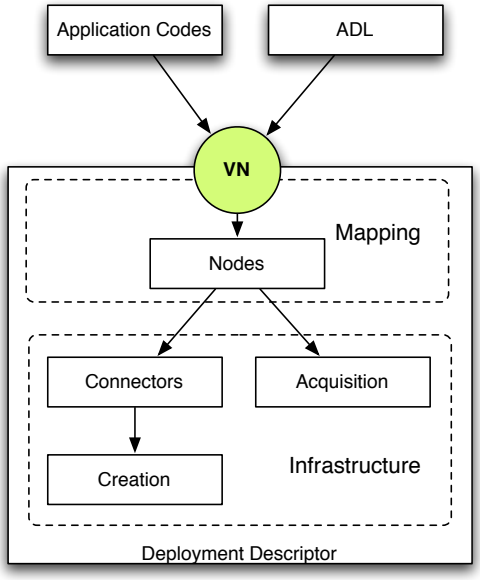

Fig. 1. Deployment descriptor model

C. Retrieval of resources

In the context of the ProActive middleware, nodes designate physical resources from a physical infrastructure. They can be created or acquired. The deployment framework is responsible for providing the nodes mapped to the virtual nodes used by the application. Nodes may be created using remote connection and creation protocols. Nodes may also be acquired through lookup protocols, which notably enable access to the ProActive Peer-to-Peer infrastructure as explained below.

1) Creation-based deployment: Machine names, connection and creation protocols are strictly separated from application code, and ProActive deployment descriptors provide the ability to create remote nodes (remote JVMs). For instance, deployment descriptors are able to use various protocols:

- local
- ssh, gsissh, rsh, rlogin
- lsf, pbs, sun grid engine, oar, prun
- globus (GT2, GT3 and GT4), unicore, glite, arc (nordugrid)

Deployment descriptors allow to combine these protocols in order to create seamlessly remote JVMs, e.g. log on a remote cluster frontend with `ssh`, and then use `pbs` to book cluster nodes to create JVMs on each. All processes are defined in the *infrastructure* part of the descriptor.

In addition, the JVM creation is handled by a special process, *localJVM*, which starts a JVM. It is possible to specify the classpath, the Java install path, and all JVM arguments. In addition, it is in this process that the deployer specifies which transport layer the ProActive node uses. For the moment, ProActive supports as transport layer: `RMI`, `HTTP`, `RMIssh`, `Ibis`, and `SOAP`.

2) Acquisition-based deployment: The main goal of the Peer-to-Peer (P2P) infrastructure is to provide a new way to build and use grids. The infrastructure allows applications to transparently and easily obtain computational resources from grids composed of both clusters and desktop machines. The application deployment burden is eased by a seamless link between applications and the infrastructure. This link allows applications to be communicant, and to manage the resources volatility.

The P2P infrastructure has three main characteristics. First, the infrastructure is decentralized and completely *self-organized*. Second, it is flexible, thanks to parameters for adapting the infrastructure to the location where it is deployed. Last, the infrastructure is portable since it is built on top of JVMs, which run on cluster nodes and on desktop machines. Thus, the infrastructure contributes to ProActive, providing a new way for: deploying applications and acquiring already running JVMs (instead of starting new ones).

The proposed P2P infrastructure is an unstructured P2P network, such as Gnutella [8]. Therefore, the infrastructure resource query mechanism is similar to the Gnutella communication system, which is based on the Breadth-First Search algorithm (BFS). The system is message-based with application-level routing. Messages are forwarded to each acquaintance, and if the message has already been received (looped), then it is dropped.

At the beginning, when a fresh peer joins the network, it only knows acquaintances from a list of potential network members, such as with super-peer architectures. The initially known peers will not be permanently available, and as a consequence peers have to update their list of acquaintances to stay connected in the infrastructure.

The proposed infrastructure uses a specific parameter called *Number of Acquaintances* (NOA): the minimum number of known acquaintances for each peer. Peers update their acquaintance list every *Time to Update* (TTU). NOA and TTU are both configurable, checking their own acquaintance list to remove unavailable peers, i.e. they send heartbeat messages to them. When the number in the list is less than the NOA, a peer will try to discover new acquaintances. To discover new acquaintances, peers send exploring messages through the infrastructure. Note that each peer can have its own parameter values, and that they can be dynamically updated.

Applications use the P2P infrastructure as a pool of resources. The main problem for applications to use those resources is that resources are returned via a best-effort mechanism; there are no guaranties of that the number requested resources can be satisfied. Recently we have improved the resource query mechanism with adding the possibility of filtering requested resources on three operating system properties: the system name, version, and the system architecture.

Those properties are provided by the Java system properties. The filtering mechanism is indeed done by peers of the infrastructure; when a peer gets a resources query, first checks if it is free and then checks OS property constraints.

IV. Constrained deployment

A. Rationale

Some components may require specific non-functional services, such as availability, reliability, security, real-time, persistence or fault-tolerance. Some constraints may also express deployment requirements, for example the expected number of resources (minimum, maximum, exact), or a timeout for retrieving these resources. These constraints can only be expressed by the designers of the components.

Because the deployment infrastructure is abstracted into virtual nodes, we propose to express these non-functional requirements as contracts [9] in a dedicated descriptor of virtual nodes (Fig. 2). This allows a clear separation between the conceptual architecture using virtual nodes and the physical infrastructure where nodes exist or are created; it enforces designer-defined constraints; it maintains a clear separation of the roles: the designer specifies deployment constraints, and the deployer, considering the available physical infrastructure, enforces the requirements when writing the mapping of virtual nodes in the deployment descriptor. Moreover, we propose to leverage the definition of deployment constraints with the introduction of dynamically fulfillable constraints.

B. Constraints

Expressing deployment constraints at the level of virtual nodes enforces a strict separation of non-functional requirements from the code. By using a dedicated descriptor of virtual nodes, the constraints may easily be modified or adapted by the designer to express new requirements. Virtual nodes descriptors also allow a strict separation between the non-functional requirements and the description of the application. Because virtual nodes are abstractions that may be used in component ADLs or in application codes, constrained deployment through virtual nodes descriptors is applicable for both component-based and object-based applications.

We distinguish *statically* fulfilled requirements, which may not usually change in selected nodes (for instance the operating system), from *dynamically* fulfilled requirements, which may be applied at runtime by configuring the nodes (for instance the need for fault-tolerance or load-balancing). There are many ways to specify static constraints, as proposed in OLAN [10], in Corba Software Descriptor files [11], or more specifically in the context of Grid computing, as recently proposed by the Global Grid Forum in the Job Submission Description Language (JSDL) [12]. The JSDL could be extended for defining constraints that may be dynamically fulfilled.

C. Dynamically fulfilled constraints

The deployer may decide to use an acquisition-based deployment, which means retrieving existing nodes from a given infrastructure (for instance a P2P infrastructure). In that case, available nodes exhibit static configurations as chosen by the administrator when deploying the P2P infrastructure. The deployer or deployment tool filters available nodes based on these requirements and should only propose matching nodes. In general, the deployment of a given application currently takes place on pre-configured infrastructures.

This selection process is unfortunately restrictive, as when using an existing node infrastructure, one may not find any matching resource. Deployment on this existing infrastructure is therefore impossible in such a case.

Moreover, some requirements that are usually considered as static, may actually be dynamically fulfilled. An example being the operating system: for instance, when deploying on the french Grid'5000 infrastructure, the operating system can be installed at deployment time [13].

Lastly, different strategies may be applied to fulfill non-functional requirements and the most adequate strategies may depend on the characteristics of the infrastructure at runtime, for example the topology (see next section for an example).

In order to allow such dynamic adaptation, we introduce the concept of *Technical Service*.

D. Technical Services

A technical service is a non-functional requirement that may be dynamically fulfilled at runtime by adapting the configuration of selected resources.

This section describes our proposal for a simple and unique specification for the configuration of technical services.

From the programmer point of view, a technical service is a class that implements the `TechnicalService` interface. This class defines how to configure a node. From the deployer point of view, a technical service is a set of "variable-value" tuples, each of them configuring a given aspect of the application environment.

For example, for configuring fault-tolerance, a `FaultToleranceService` class is provided; it defines how the configuration is applied from a node to all the active objects hosted by this node. The deployer of the application can then configure in the deployment descriptor the fault-tolerance using the technical service XML interface.

Technical services are defined in deployment constraints, and may be overridden in deployment descriptors.

In deployment constraints, a technical service is attached to a virtual node (it belongs to the virtual node container tag) and is defined as follows:

```
<technical-service id = "myService" class="
    services.Service1">
  <arg name="name1" value="value1"/>
  <arg name="name2" value="value2"/>
</technical-service>
```

The `class` attribute defines the implementation of the service, a class which must implement the `TechnicalService` interface:

```
public interface TechnicalService {
  public void init(HashMap argValues);
  public void apply(Node node);
}
```

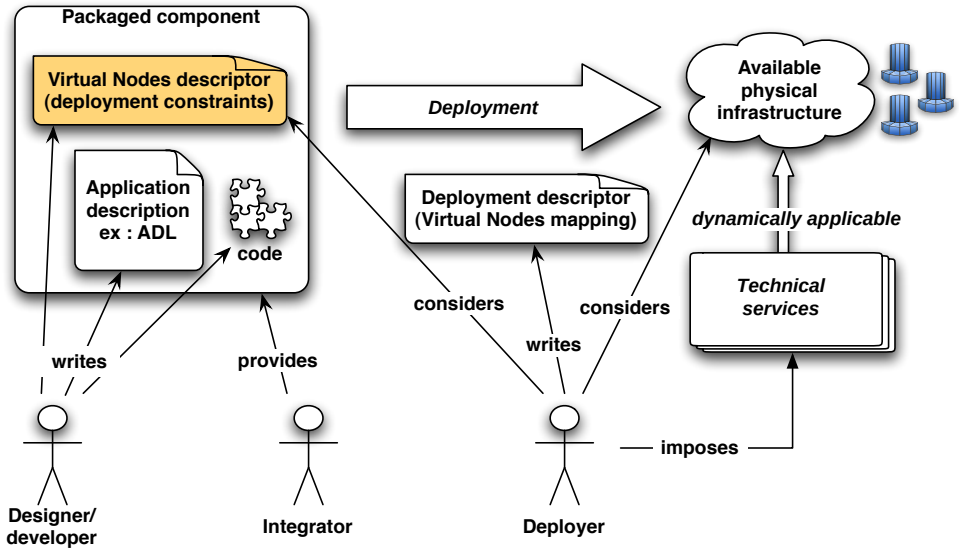

Fig. 2. Deployment roles and artifacts

The configuration parameters of the service are specified by `arg` tags in the deployment descriptor. Those parameters are passed to the `init` method as a map associating the name of a parameter as a key and its value. The `apply` method takes as parameter the node on which the service must be applied. This method is called after the creation or acquisition of a node, and before the node is used by the application.

Two or several technical services could be combined if they touch separate aspects. Indeed, two different technical services, which are conceptually orthogonal, could be incompatible *at source code level*.

In practice, we have noticed such an incompatibility in our implementation of fault-tolerance and load balancing services, developed by two different programmers. That is why a virtual node can be configured by only *one* technical service. However, combining two technical services can be done at source code level, by providing a class extending `TechnicalService` that defines the correct merging of two concurrent technical services.

If two separate and packaged components define incompatible constraints on homonymous virtual nodes, they *cannot* be deployed on the same target nodes. Fortunately, this problem can be solved by creating a composite component containing these components and performing renaming of the virtual nodes: deployment can then be performed on a disjoint set of nodes, which eliminates the incompatibility issue. Fig. 3 provides an illustration of this method: component 1 and component 2 are packaged components that both define constraints on a virtual node named VN1, but these constraints are incompatible. By wrapping these components into the composite component named component 3, it is possible to remap the deployment of components 1 and 2 onto the separate virtual nodes VNA and VNB. The remapping takes place in the ADL of the composite component; we provide an extension of the ADL for this purpose.

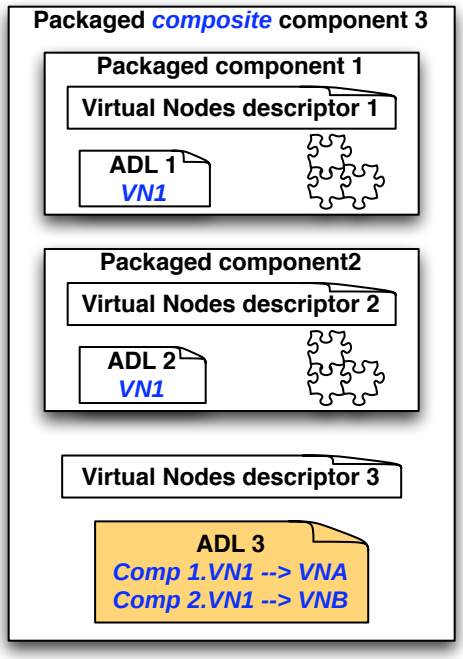

Fig. 3. Composition of components with renaming of virtual nodes

E. Virtual Nodes Descriptors

The description of virtual nodes is expressed in a dedicated virtual nodes descriptor, in XML format :

```
<virtual-nodes>
  <virtual-node name="VN1">
    <technical-service type="services.Service1
        "/>
    <processor architecture="x86"/>
```

```
    <os name="linux" release="2.6.15"/>
  </virtual-node>
<virtual-nodes>
```

Non-functional contracts are here expressed in a simple way. The tag `technical-service` specifies the technical service (more precisely the type (class or idealy interface) defining the technical service), which has to be applied on the virtual node `VN1` at deployment time. Regarding static constraints, we are also considering adopting the JSDL naming conventions for defining static constraints on virtual nodes.

The deployer in charge of writing the deployment descriptor is aware of the requirements by looking at the virtual nodes descriptor, and must ensure the infrastructure matches the requirements. There is no contract management module, such as in [14], nor deployment planner such as in [15]. Indeed, contracts are verified when retrieving nodes from the physical infrastructure, resulting in runtime errors if contracts are not respected. This ensures a simple framework in terms of specification and verification, eludes resource planning issues, and could still be plugged to a resource allocator framework such as Globus' GARA [16].

F. Deployment process: summary

The specification of non-functional constraints as well as the dynamic fulfilling of the constraints expressed as technical services imply a new deployment process, which is summed up in a standard case in Fig. 2. In this figure, roles and artifacts are explicitly expressed :

- the designer and developer provide the code and the description of the component system (ADL), as well as the non-functional constraints in a virtual node descriptor
- the integrator gathers compiled sources, ADL and virtual nodes descriptor into a deliverable package
- the deployer writes or adapts a deployment descriptor so that the deployment of the component system verifies the virtual nodes descriptor with respect to the available infrastructure. Technical services are applied at runtime if needed.

Two other scenarios are possible:

- a designer wants to use a given set of Grid resources, and the interface to these resources is a deployment descriptor provided by the system administrator/provider of the resources.
- a deployer wants to use available packaged components, and deploy them on a given infrastructure for which a deployment descriptor is available.

In all scenarios, the specification of non-functional constraints in the virtual nodes descriptor ensures that the application requirements and the physical infrastructure are compatible, and this compatibility is possibly attained by dynamically updating the configuration of the physical resources.

V. Use case: deployment on a peer-to-peer infrastructure with fault-tolerance requirements

This section illustrates the concept of dynamically fulfilled deployment constraints through technical services: it presents a use case involving the deployment of a component system with some fault-tolerance requirements on a P2P infrastructure; it demonstrates how the proposed approach helps resolving deployment and QoS requirements in the most suitable way. Beforehand, we provide an explanation of the fault-tolerance mechanism and configuration in ProActive, which is essential to the comprehension of this use case.

A. Fault-tolerance in ProActive

As the use of desktop grids goes mainstream, the need for adapted fault-tolerance mechanisms increases. Indeed, the probability of failure is dramatically high for such systems: a large number of resources imply a high probability of failure of one of those resources. Moreover, public Internet resources are by nature unreliable.

Rollback-recovery [17] is one solution to achieve fault-tolerance: the state of the application is regularly saved and stored on a stable storage. If a failure occurs, a previously recorded state is used to recover the application. Two main approaches can be distinguished : the *checkpoint-based* [18] approach, relying on recording the state of the processes, and the *log-based* [19] approach, relying on logging and replaying inter-process messages.

Fault-tolerance in ProActive is achieved by rollback-recovery; two different mechanisms are available. The first one is a Communication-Induced Checkpointing protocol (CIC): each active object has to checkpoint at least every *TTC* (Time To Checkpoint) seconds. Those checkpoints are synchronized using the application messages to create a *consistent* global state of the application [20]. If a failure occurs, *every active object*, even the non faulty one, must restart from its latest checkpoint. The second mechanism is a Pessimistic Message Logging protocol (PML): the difference with the CIC approach is that there is no need for global synchronization, because all the messages delivered to an active object are logged on a stable storage. Each checkpoint is independent: if a failure occurs, only the faulty process has to recover from its latest checkpoint.

Basically, we can compare those two approaches based on two metrics: the failure-free overhead, i.e. the additional execution time induced by the fault-tolerance mechanism without failure, and the recovery time, i.e. the additional execution time induced by a failure during the execution. The failure-free overhead induced by the CIC protocol is usually low [21], as the synchronization between active objects relies only on the messages sent by the application. Of course, this overhead depends on the TTC value, set by the programmer; the TTC value depends mainly on the assessed frequency of failures. A small TTC value leads to very frequent global state creation and thus to a small rollback in the execution in case of failure. But a small TTC value leads also to a higher failure free

overhead. The counterpart is that the recovery time could be high since all the application must restart after the failure of one or more active object.

As for CIC protocol, the TTC value impacts on the global failure-free overhead, but the overhead is more linked to the communication rate of the application. Regarding the CIC protocol, the PML protocol induces a higher overhead on failure-free execution. But the recovery time is lower as a single failure does not involve all the system: only the faulty has to recover.

B. Fault-tolerance Configuration

Choosing the adapted protocol depends on the characteristics of the application, and of the underlying hardware that are known at deployment time; we then design the fault-tolerance mechanism such that making a ProActive application fault-tolerant is automatic and transparent to the developer; there is no need to consider fault-tolerance concerns in the source code of the application. The fault-tolerance settings are actually contained in the nodes: an active object deployed on a node is configured by the settings contained in this node.

Fault-tolerance is a technical service as defined in Section IV-D. The designer can specify in the virtual nodes descriptor the needed reliability of the different parts of the application, and the deployer can choose the adapted mechanism to obtain this reliability by configuring the technical service in the deployment descriptor. The deployer can then select the best mechanism and configuration:

- the protocol to be used (CIC or PML), or no protocol if software fault-tolerance is not needed on the used hardware,
- the Time To Checkpoint value (TTC),
- the URLs of the servers.

C. Example

To illustrate our mechanism of constrained deployment, we consider a *master-slaves* application for solving Flowshop problems. A Flowshop problem aims to find the optimal schedule of a set of jobs on a set of machines in order to minimize the total execution time; this problem can be solved by exploring a solution tree. The whole solution tree is explored in parallel, and while exploring the tree, the current best solution is shared within the application, which allows the elimination of bad tree branches.

The solution tree of the problem is divided by a master in a set of sub-tasks, these sub-tasks are allocated to a number of sub-managers, which can also be at the top of a hierarchy of sub-managers. Sub-managers manage sub-task allocation to the workers and also perform communications between them to synchronize the best current solution. Sub-managers handle dynamic acquisition of new workers and also handle worker failures by reallocating failed tasks. As a consequence, there is *no* need for applying an automatic fault-tolerance mechanism (then to pay an execution-time overhead) on the workers. On the contrary, the manager and the sub-managers *must* be protected against failures by the middleware since there is no failure-handling at application level for them.

In this case, the designer of the application specifies in the virtual nodes descriptor that a fault-tolerance technical service must be applied on the virtual node that hosts manager components, while there is no such constraint on a worker component:

```
<virtual-nodes>
  <virtual-node name="managers">
    <technical-service type="services.
        FaultTolerance"/>
    <processor architecture="x86"/>
  </virtual-node>
<virtual-nodes>
```

When deploying the application, the deployer can choose the most adapted fault-tolerance mechanism depending on the environment by configuring the technical service. This technical service must fit, i.e. extends or implements, the type specified in the virtual node descriptor. For example, suppose that the application is deployed on a desktop grid provided by the ProActive P2P infrastructure. Such resources being strongly prone to failure, the chosen fault-tolerance mechanism must deal with very frequent failures, and thus provide a reactive and fast recovery, even at the expense of a weighty overhead on execution time. Using a lighter but weaker mechanism in this case could lead the system to continuously recovering. Finally, the deployer chooses the PML approach, with a small TTC value (60 sec) as in the following deployment descriptor:

```
<ProActiveDescriptor>
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="managers" property="
       multiple" serviceRefid="ft-master"/>
   <virtualNode name="workers" property="
       multiple"/>
  </virtualNodesDefinition>
 </componentDefinition>
...
 <aquisition>
  <aquisitionDefinition id="p2pservice">
   <P2PService nodesAsked="1000">
    <peerSet>
     <peer>rmi://peer.registry:3000</peer>
    </peerSet>
   </P2PService>
 </acquisitionDefinition>
...
 <technicalServiceDefinitions>
  <service id="ft-master" class="services.
      FaultTolerance">
   <arg name="proto" value="pml"/>
   <arg name="server" value="rmi://host/
       FTServer"/>
   <arg name="TTC" value="60"/>
  </service>
 </technicalServiceDefinitions>
</ProActiveDescriptor>
```

D. Analysis

In this example, the concept of technical service has allowed to apply the *necessary and sufficient* fault-tolerance mechanism when deploying the application:

- *necessary* thanks to the virtual nodes descriptor; the designer has specified the minimum fault-tolerance requirements for its application. Without this specification, the deployer could have unnecessarily applied fault-tolerance on *all* the application.
- *sufficient* thanks to the possibility to choose *at deployment time* how the constraint specified by the designer should be fulfilled to take into account the characteristics of the available resources. If the fault-tolerance aspect had been fully specified by the designer at development time, the chosen fault-tolerance mechanism could have been too weak to be able to deploy on a desktop grid.

VI. Conclusion

This paper proposes a mechanism for specifying environmental requirements that may be defined by developers, and specified by deployers by parameterizing deployment abstractions. This mechanism is integrated in the ProActive middleware, and allows flexible component deployments thanks to easily configurable technical services. Application designers can specify minimum deployment requirements, and deployers are able to apply the optimal configuration that fulfills those requirements.

We illustrate the pertinency of this mechanism in a concrete use-case: deploying an component-based application with Fault-Tolerance on an heterogeneous grid provided by the ProActive P2P infrastructure.

References

[1] Sun Microsystems, "Java remote method invocation specification," Oct. 1998, ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.

[2] D. Caromel, "Towards a Method of Object-Oriented Concurrent Programming," *Communications of the ACM*, vol. 36, no. 9, pp. 90–102, September 1993.

[3] F. Baude, D. Caromel, and M. Morel, "From distributed objects to hierarchical grid components," in *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, vol. 2888. Springer Verlag: Lecture Notes in Computer Science, LNCS, 2003, pp. 1226–1242.

[4] E. Bruneton, T. Coupaye, and J.-B. Stefani, "Recursive and dynamic software composition with sharing," in *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.

[5] "Fractal adl tutorial," http://fractal.objectweb.org/tutorials/adl/index.html.

[6] C. Szyperski, *Component Software : Beyond Object-Oriented Programming*. Addison-Wesely, 2002.

[7] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière, "Interactive and descriptor-based deployment of object-oriented grid applications," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*. Edinburgh, Scotland: IEEE Computer Society, July 2002, pp. 93–102.

[8] Gnutella, "Gnutella peer-to-peer network," 2001, http://www.gnutella.com.

[9] S. Frolund and J. Koistinen, "Quality-of-service specifications in distributed object systems," in *Distributed Systems Engineering*, IEE, Ed., vol. 5, 1998, pp. 179,202.

[10] L. Bellissard, M.-C. Pellegrini, and M. Riveill, "Integration and Distribution of Legacy Software with Olan," *Object-Based Parallel and Distributed Computation, France-Japan Workshop*, Toulouse, October 15-17, 1997.

[11] "Corba component model, v3.0," specification, Object Management Group, 2002.

[12] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, A. S.McGough, D. Pulsipher, and A. Savva., "Job submission description language (jsdl) specification, version 1.0." http://forge.gridforum.org/projects/jsdl-wg, 2005.

[13] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard, "Gridõ5000: a large scale and highly reconfigurable grid experimental testbed," in *6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[14] O. Loques and A. Sztajnberg, "Customizing component-based architectures by contract," in *Second International Working Conference on Component Deployment (CD 2004)*, ser. Lecture Notes in Computer Science, vol. 3083. Edinburgh, UK: Springer-Verlag, May 2004, pp. 18–34.

[15] S. Lacour, C. Pérez, and T. Priol, "Generic application description model: Toward automatic deployment of applications on computational grids," in *6th IEEE/ACM International Workshop on Grid Computing (Grid2005), Seattle, WA, USA*. Springer-Verlag, november 2005.

[16] I. Foster, A. Roy, and V. Sander, "A quality of service architecture that combines resource reservation and application adaptation," in *Proceedings of the Eight International Workshop on Quality of Service (IWQOS 2000)*, June 2000, pp. 181–188.

[17] M. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message passing systems," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep. CMU-CS-96-181, oct 1996.

[18] D. Manivannan and M. Singhal, "Quasi-synchronous checkpointing: Models, characterization, and classification," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, 1999, pp. 703–713.

[19] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[20] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," in *ACM Transactions on Computer Systems*, 1985, pp. 63–75.

[21] F. Baude, D. Caromel, C. Delbé, and L. Henrio, "A hybrid message logging-cic protocol for constrained checkpointability," in *Proceedings of EuroPar2005*, August-September 2005.