Peer-to-Peer and Fault-tolerance: Towards Deployment-Based Technical Services

Denis Caromel, Alexandre di Costanzo, and Christian Delbé

INRIA Sophia - I3S - CNRS - Université de Nice Sophia Antipolis
INRIA, 2004 Rt. des Lucioles, BP 93
F-06902 Sophia Antipolis Cedex, France

Abstract

For effective components, non-functional aspects must be added to the application functional code. Likewise enterprise middleware and component platforms, in the context of Grids, *services* must be deployed at execution in the component containers in order to implement those aspects without application code modifications.

This paper proposes an architecture for defining, configuring, and deploying such *Technical Services* in a Grid platform.

Key words: Grid Deployment, Non-Functional Service, Peer-to-Peer, Fault-Tolerance, Load-Balancing

1 Introduction

The last decade has seen a clear identification of the so called *Non-Functional* aspects for building flexible and adaptable software. In the framework of middleware, e.g. business frameworks such as the EJB [1], architects have been making a strong point at separating the application operations, the *functional aspects*, from services that are rather orthogonal to it: transaction, persistence, security, distribution, etc.

The frameworks, such as Enterprise JavaBeans containers (JBoss, JOnAs, etc.), can further be configured for enabling and configuring such non-functional aspects. Hence, the application logic is subject to various setting and configuration, depending of the context. Overall, it opens the way to effective component codes usable in various contexts, with the crucial feature of parameterizations: choosing at deployment time various *Technical Services* to be added to the application code. In the framework of Grids, current platforms are falling short to provide such flexibility. One cannot really add and configure

Fault-Tolerance, Security, Load Balancing, etc. without intensive modification of the source code. Moreover, there are no coupling with the deployment infrastructures.

In an attempt to solve this shortcoming of current Grid middlewares, this article proposes a framework for defining such Technical Services dynamically, based on the application needs, potentially taking into account the underlying characteristics of the infrastructure.

Section 2 presents a short related work on the field of deploying such kind of non-functional services. Section 3 introduces the ProActive middleware. An overview of the programming model based on active objects, asynchronous communications, and futures is first given, followed with a detailed presentation of the deployment model. The later relies on Virtual Nodes and XML deployment descriptors. For the sake of reasoning in the context of real technical services, Section 4.2.2 presents the Peer-to-Peer infrastructure available in ProActive. Finally, Section 5 proposes a flexible architecture for specifying technical services dynamically at deployment time. Taking Peer-to-Peer and fault-tolerance as basic examples, it is showed how to appropriately combine and configures them together.

2 Related Work

In the field of Grid, the Open Grid Services Architecture OGSA [2] defines a mechanism for creating, managing, and discovering grid services, which are network-enabled entities that provide some capability through the exchange of messages.The OGSA specifies a uniform service semantic that allows users to build their grid applications by assembling some services from enterprises, service providers, and themselves.

Unlike our approach of technical service, the OGSA does not limit services to be only non-functional; for example a grid service can be a cluster for data storage. On the other hand, non-functional services are parts of the architecture model and users cannot configure security or fault-tolerance for their own applications.

The concept of non-functional requirements, i.e. technical services, was first introduce in the field of component models. Such models allows a clear separation between the functional code written by the developer and the non-functional services provided by the framework. In [3] a technical service must be developed by an expert of the field, such as an expert in load-balancing for implementing a load-balancing service, because a field expert can provide a good quality-of-service for a large scale of applications.

In the Enterprise JavaBeans (EJB) [1] framework, technical services are specified by Sun; in the Corba Component Model (CCM) [4], they are provided by CORBA. These services are at the component container level, i.e. they are parts of the framework. For all these frameworks, users specify and configure technical services at the deployment time. Consequently, users have choice between few technical services imposed by the models, thus fault-tolerance expert cannot propose her own solution. Then, users are limited in their choices, they cannot choose between different versions or implementations of the same service.

Unlike component frameworks, we propose an extensible model for technical services that allows programmer experts to propose their own implementation of non-functional services. Like EJB and CCM, users specify and configure the technical service at the deployment time.

3 The ProActive Grid Middleware

ProActive is a Java library for concurrent, distributed and mobile computing originally implemented on top of RMI [5] as the transport layer, now HTTP, RMI/SSH, and Ibis [6] 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 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* [7]. 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.

4 Descriptor-based Deployment of Grid Applications

4.1 Deployment Model

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 succeeded in completely avoiding scripting for configuration, getting computing resources, etc. ProActive provides, as a key approach to the deployment problem, an abstraction from the source code such as to gain in flexibility [8]. A first 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.

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 VN 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 VNs and nodes: the function created by the deployment, the mapping. This mapping is specified in the deployment descriptor. 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 VNs.

Figure 1 summarizes the deployment framework provided by the ProActive middleware. Deployment descriptor 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 descriptor and nodes are mapped to physical resources, i.e. to the infrastructure.

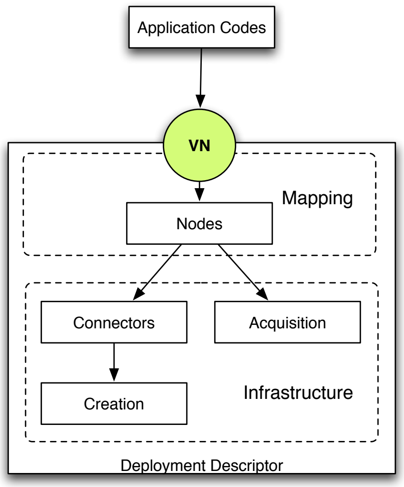

Fig. 1. Deployment descriptor model

4.2 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.

4.2.1 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`, and `Ibis` [6].

4.2.2 Acquisition-based deployment

The main goal of the ProActive Peer-to-Peer (P2P) infrastructure [9] 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 [10]. 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, which are nodes. The main problem for applications to use those resources is that resources are returned via a best-effort mechanism; there are no guarantees that the requested number of resources can be satisfied.

5 Deployment and Technical Services

5.1 Model

Some parts of applications may require specific non-functional services, such as security, load-balancing, or fault-tolerance. These constraints can only be expressed by the deployer of the application because she is the only one that can configure them for the physical infrastructure.

Because the deployment infrastructure is abstracted into virtual nodes, we propose to express these non-functional requirements as contracts [11] in deployment descriptors (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 maintains a clear separation of the roles: the developer implements the application without take into account of non-functional requirements; and the deployer, considering the available physical infrastructure, enforces the requirements when writing the mapping of virtual nodes in the deployment descriptor. Then, the expert implements and provides non-functional services as technical services. Moreover, we propose to leverage the definition of deployment non-functional services with the introduction of dynamically applicable technical service.

5.2 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 expert 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

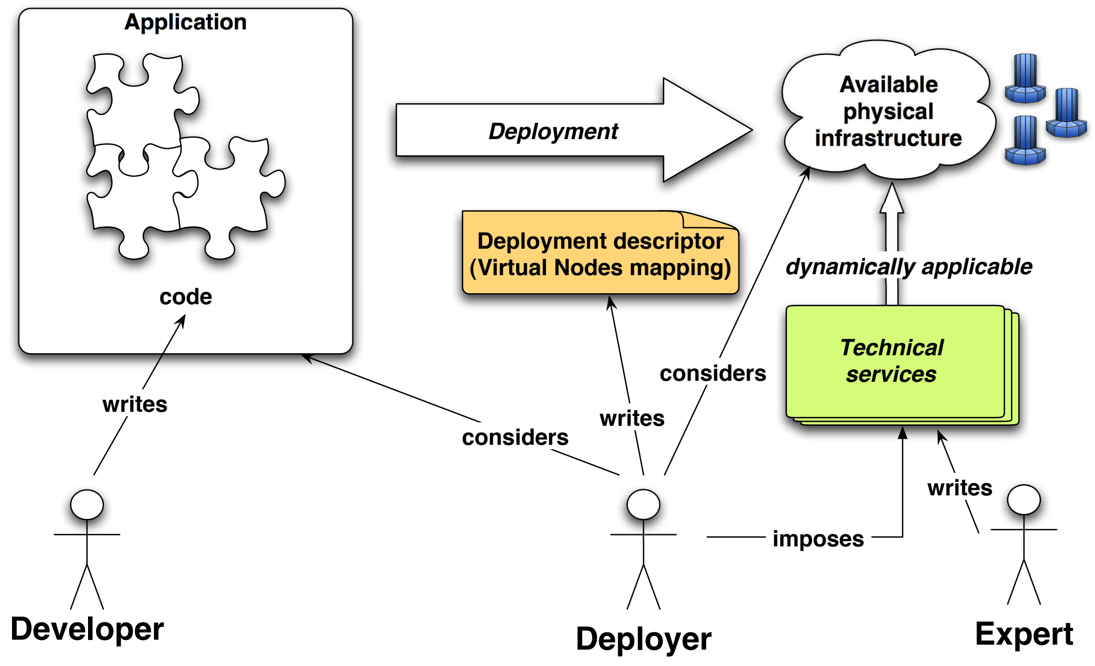

Fig. 2. Deployment roles and artifacts

"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.

A technical service is defined as a stand-alone block in the deployment descriptor. It is attached to a virtual node (it belongs to the virtual node container tag); the configuration defined by the technical service is applied to all the nodes mapped to this virtual node. A technical service is defined as follows:

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

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);
}
```

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.

A technical service is attached to a virtual node as following:

```
<virtualNodesDefinition>
 <virtualNode name="virtualNode1" serviceRefid="myService"/>
</virtualNodesDefinition>
```

Figure 3 summarizes the deployment framework with the added part for non-functional aspects.

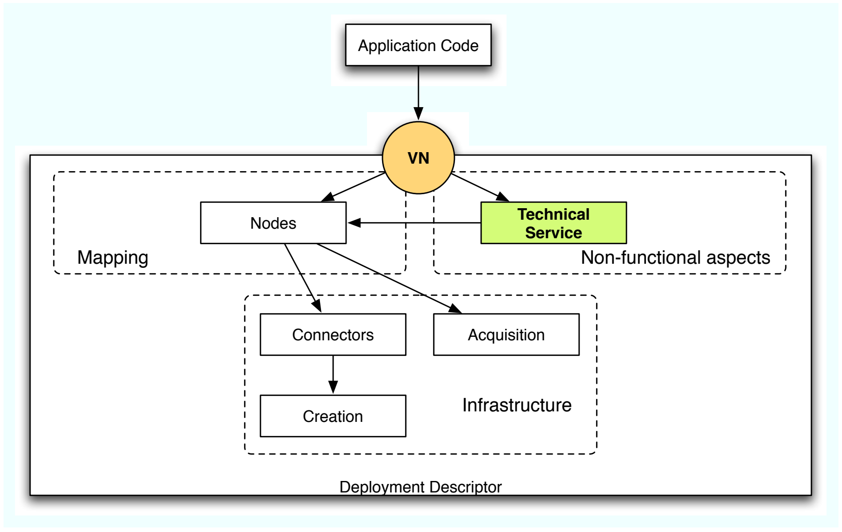

Fig. 3. Deployment descriptor model with the technical service part

ProActive provides a mechanism for load-balancing active objects [12], the drawback of this work is that the activation of the mechanism is at the code level. So, we have implemented a technical service for activing and configuring load-balancing at deployment time. Figure 4 shows the implementation of our load-balancing technical service.

The implementation does the same thing of what the developer does for activating load-balancing at source code level.

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*.

```java
public class LoadBalancingTS implements TechnicalService {

        public void init(Map argValues) {
                String metricFactoryName = (String) argValues.get
                    ("MetricFactory");
                MetricFactory mf = (MetricFactory) Class.forName(
                    metricFactoryName).newInstance();
                LoadBalancing.activate(mf);
        }

        public void apply(Node node) {
                LoadBalancing.addNode(node);
        }
}
```

Fig. 4. Load-balancing technical service code

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.

6 Example: Fault-Tolerant Flow-Shop on Peer-to-Peer

This section illustrates the concept of dynamically deploying and configuring technical services: it presents a use case involving the deployment of an appliation with some fault-tolerance requirements on a P2P infrastructure; it demonstrates how the proposed approach helps resolving deployment 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.

6.1 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 [13] 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* [14] approach, relying on recording the state of the processes, and the *log-based* [15] 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 [16]. 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 [17], 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.

6.1.1 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 5. 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.

6.2 Example

To illustrate our mechanism of technical services, we consider a master-slaves application for solving Flow-Shop problems. A Flow-Shop 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.

Figure 5 shows a complete example of a deployment descriptor based on the P2P infrastructure.

This descriptor defines two virtual nodes: one for hosting the masters and one

```
<ProActiveDescriptor>
  <componentDefinition>
    <virtualNodesDefinition>
      <virtualNode name="master" serviceRefid="master" />
      <virtualNode name="slaves" />
    </virtualNodesDefinition>
  </componentDefinition>
  <deployment>
    <mapping>
      <map virtualNode="master">
        <vmName value="localJVM" />
      </map>
      <map virtualNode="salves">
        <vmName value="p2plookup" />
      </map>
    </mapping>
  </deployment>
  <infrastructure>
    <processes>
      <processDefinition id="localJVM">
        <jvmProcess class="org.objectweb.proactive.core.process.
            JVMNodeProcess" />
      </processDefinition>
    </processes>
    <aquisition>
      <P2PService id="p2plookup" nodesAsked="1000">
        <peer>rmi://registry1:3000</peer>
      </P2PService>
    </aquisition>
  </infrastructure>
  <technicalServiceDefinitions>
    <service id="ft-master" class="services.FaultTolerance">
      <arg name="protocol" value="pml" />
      <arg name="server" value="rmi://host/FTServer1" />
      <arg name="TTC" value="60" />
    </service>
  </technicalServiceDefinitions>
</ProActiveDescriptor>
```

Fig. 5. P2P and Fault-Tolerance: Deployment Descriptor

for hosting the slaves. Only the master virtual node is configured by a technical service defining the most adapted fault-tolerance configuration regarding the underlying hardware; here, the protocol used is PML, set with a short TTC value as we are in P2P with volatile nodes.

Figure 6 shows the full implementation of our fault-tolerance technical service. The functional code of the fault-tolerance is in fact implemented in the ProActive core code for logging messages. Thus the technical service just sets some global properties for starting the logging of messages by the fault-tolerance.

```java
public class FaultToleranceTS implements TechnicalService {
    private String SERVER;
    private String TTC;
    private String PROTOCOL;

    public FaultToleranceTS() {
    }

    public void apply(Node node) {
        ProActiveRuntime par = node.getProActiveRuntime();
        par.setSystemProperty("proactive.ft.server.global", this
            .SERVER);
        par.setSystemProperty("proactive.ft.ttc", this.TTC);
        par.setSystemProperty("proactive.ft.protocol", this.
            PROTOCOL);
    }

    public void init(Map argValues) {
        this.SERVER = (String) argValues.get("proactive.ft.
            server.global");
        this.TTC = (String) argValues.get("proactive.ft.ttc");
        this.PROTOCOL = (String) argValues.get("proactive.ft.
            protocol");
    }
}
```

Fig. 6. P2P and Fault-Tolerance: Fault-tolerance service implementation

6.3 Experimentation

This section aims to show that our Technical Service mechanism is implemented and works. For more benchmarks on the fault-tolerance itself, we invite you to look up this article [18] and this work [9] for experimentation about the P2P and Flow-Shop .

In order to run our experiments, we use the desktop infrastructure fully described in [9]. This infrastructure is a permanent desktop grid managed by our P2P infrastructure (section 4.2.2). All these desktops configuration and hardware are heterogeneous.

Figure 7 shows the Flow-Shop application deployed with the technical service for fault-tolerance on the P2P infrastructure. The instance of the Flow-Shop problem is 15 jobs / 20 machines. The fault-tolerance protocol used is PML with a TTC of 60 seconds. We observe that the computation time decreases with the number of CPUs.

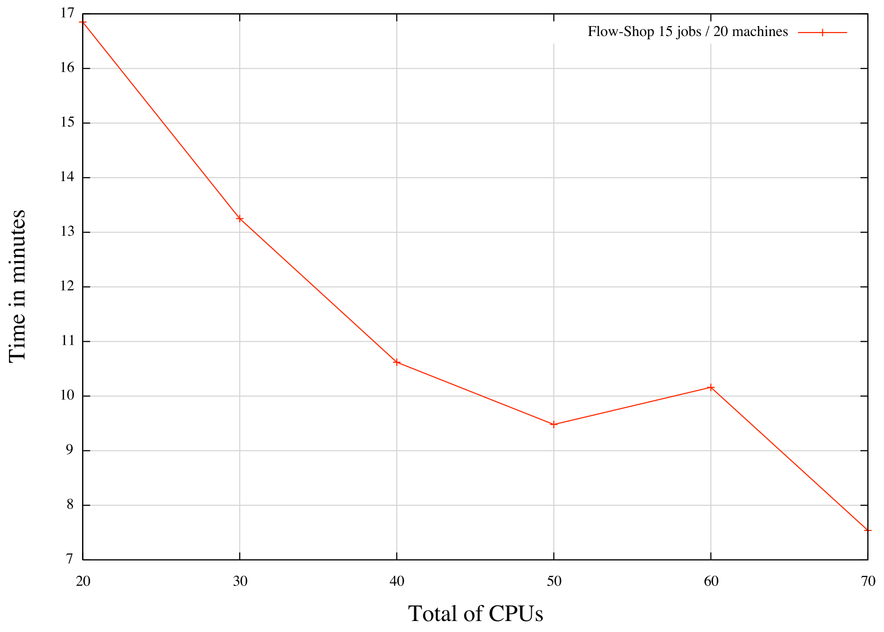

Fig. 7. Benchmarks with Flow-Shop and Fault-tolerance

The increase from 50 to 60 CPUs is due to the fact that some tasks of the problem run on more slower machines than for 50 or 70 CPUs. Benchmarks run on a desktop grid and use a different set of machines at each run. It is hard to control expected machines with the peer-to-peer aspects of our benchmarks.

7 Conclusion and Future Work

This article proposes a way to attach Technical Services to Virtual Nodes, mapping non-functional aspects to the containers, dynamically at deployment and execution. More investigations are needed to look into the fit of the architecture with respect to the complexity of Grid platforms, and to the large number of technical services to be composed and deployed.

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

And in the short term, we are planning to explore the combination of two technical services: fault-tolerance and load balancing.

References

[1] Enterprise JavaBeans Specication. Version 2.1 , Technical Report, 2001, Sun MicroSystems.

[2] Grid Services for Distributed System Integration. I. Foster, C. Kesselman, J. Nick, S. Tuecke. Computer, 35(6), 2002.

[3] J. Kinzley and R. Guerraoui, AOP does it make sense? The Case of Concurrency and Failures, in European Conference on Object-Oriented Programming (ECOOP 2002), Malaga, June 2002.

[4] CORBA Components. Specication. OMG TC Document orbos/99-02-05, Technical Report, 1999, Object Management Group.

[5] Sun Microsystems, Java remote method invocation specification, (Oct. 1998).

[6] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a exible and efficient java-based grid programming environment. Concurrency - Practice and Experience, 17(7-8):10791107, 2005.

[7] D. Caromel, Towards a Method of Object-Oriented Concurrent Programming, Communications of the ACM 36 (9) (1993) 90102.

[8] F. Baude, D. Caromel, L. Mestre, F. Huet, 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, IEEE Computer Society, Edinburgh, Scotland, 2002, pp. 93102.

[9] D. Caromel, A. di Costanzo, and C. Mathieu, Peer-to-Peer for Computational Grids: Mixing Clusters and Desktop Machines, in Parallel Computing Journal on Large Scale Grid, to appear, 2007.

[10] Gnutella, Gnutella peer-to-peer network, http://www.gnutella.com (2001).

[11] S. Frolund and J. Koistinen, Quality-of-service specifications in distributed object systems, in Distributed Systems Engineering, IEEE, vol. 5, 1998, pp. 179-202.

[12] J. Bustos-Jimenez, D. Caromel, A. di Costanzo, M. L. nd Jose M. Piquer, Balancing active objects on a peer to peer infrastructure, in: Proceedings of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005), IEEE, Valdivia, Chile, 2005.

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

[14] D. Manivannan, M. Singhal, Quasi-synchronous checkpointing: Models, characterization, and classification, in: IEEE Transactions on Parallel and Distributed Systems, Vol. 10, 1999, pp. 703713.

[15] L. Alvisi, K. Marzullo, Message logging: Pessimistic, optimistic, causal, and optimal, Software Engineering 24 (2) (1998) 149159.

[16] K. M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, in: ACM Transactions on Computer Systems, 1985, pp. 6375.

[17] F. Baude, D. Caromel, C. Delbe, L. Henrio, A hybrid message logging-cic protocol for constrained checkpointability, in: Proceedings of EuroPar2005, 2005.

[18] F. Baude, D. Caromel, C. Delbé, and L. Henrio. “ A hybrid message logging-cic protocol for constrained checkpointability. ”. In *Euro-Par*, pages 644–653, 2005.