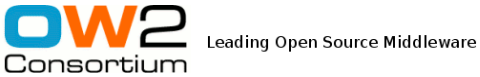
An Open Source Solution for Enterprise Grids & Clouds

ProActive Programming

Advanced Features

Version 2014-02-17

ActiveEon Company, in collaboration with INRIA



Generated on 2014-02-17

ProActive Programming v2014-02-17 Documentation

Legal Notice

Contact: proactive@ow2.org or contact@activeeon.com

Mailing List

proactive@ow2.org

Mailing List Archive

http://www.objectweb.org/wws/arc/proactive

Bug-Tracking System

http://bugs.activeeon.com/browse/PROACTIVE

Contributors and Contact Information

Team Leader

Denis Caromel
INRIA 2004, Route des Lucioles, BP 93
06902 Sophia Antipolis Cedex
France
phone: +33 492 387 631
fax: +33 492 387 971
e-mail: Denis.Caromel@inria.fr

Contributors from OASIS Team	Contributors from ActiveEon Company
Brian Amedro	Vladimir Bodnartchouk
Francoise Baude	Arnaud Contes
Francesco Bongiovanni	Cédric Dalmasso
Florin-Alexandru Bratu	Christian Delbé
Viet Dung Doan	Arnaud Gastinel
Yu Feng	Jean-Michel Guillaume
Imen Filali	Olivier Helin
Fabrice Fontenoy	Clément Mathieu
Ludovic Henrio	Maxime Menant
Fabrice Huet	Emil Salageanu
Elaine Isnard	Jean-Luc Scheefer
Vasile Jureschi	Mathieu Schnoor
Muhammad Khan	
Virginie Legrand Contes	
Eric Madelaine	
Elton Mathias	
Paul Naoumenko	
Laurent Pellegrino	
Guilherme Peretti-Pezzi	
Franca Perrina	
Marcela Rivera	
Christian Ruz	
Bastien Sauvan	
Oleg Smirnov	
Marc Valdener	
Fabien Viale	

Former Important Contributors

Laurent Baduel (Group Communications)
Vincent Cave (Legacy Wrapping)
Alexandre di Costanzo (P2P, B&B)
Abhijeet Gaikwad (Option Pricing)
Mario Leyton (Skeleton)
Matthieu Morel (Initial Component Work)
Romain Quilici
Germain Sigety (Scheduling)
Julien Vayssiere (MOP, Active Objects)

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

In order to make the ProActive Programming documentation easier to read, it has been split into four manuals:

- **ProActive Get Started Manual** - This manual contains an overview of ProActive Programming showing different examples and explaining how to install the middleware. It also includes a tutorial teaching how to use it. This manual should be the first one to be read for beginners.
- **ProActive Reference Manual** - This manual is the main manual where the concepts of ProActive are described. Information on configuration and deployment are described in that manual. It also includes guides for high-level APIs usage such as Master-Worker, SMPD APIs.
- **ProActive Advanced Features Manual** - This manual describes some advanced features like Fault-Tolerance, ProActive Compile-Time Annotations or Web Services Exportation. In Addition, it gives information on some other high-level APIs such as Monte-Carlo or Branch and Bound APIs. Finally, it helps advanced user to extend ProActive.
- **ProActive Components Manual** - ProActive defines a component model called ProActive/GCM suitable to support the development of efficient grid applications. This manual therefore contains all necessary information to be able to understand and use this component model. This model is really linked to ProActive so a ProActive/GCM user may have to refer to the ProActive manuals form time to time.

These manuals should be read in the order defined above. However, this is not essential as these documentations are linked together. Besides, if some links seems to be dead in one of these manuals, make sure that all of them had been built previously (multiple html version). So as to build all the manuals at once, go to your ProActive compile/ directory and type build[.bat] doc.ProActive.manualHtml. This builds all manuals in all formats. You may also need the javadoc documentation since these manuals sometime refer to it. To build all the javadoc documentations, type build[.bat] doc.ProActive.javadoc.complete doc.ProActive.javadoc.published inside the compile/ directory. If you just want to build one of these manuals in a specific format, type build[.bat] to see all the possible targets and chose the one you are interested in.

Part I. Services and Advanced Features

Table of Contents

Chapter 1. Fault-Tolerance

1.1. Overview

ProActive can provide fault-tolerance capabilities through two different protocols: a Communication-Induced Checkpointing protocol (CIC) or a pessimistic message logging protocol (PML). Making a ProActive application, fault-tolerant is **fully transparent**: active objects are turned into fault-tolerant using Java properties that can be set in the deployment descriptor (see Chapter 21. ProActive Grid Component Model Deployment[1]). The programmer can select **at deployment time** the most adapted protocol regarding the application and the execution environment.

Persistence of active objects is obtained through standard Java serialization: a checkpoint thus consists in an object containing a serialized copy of an active object and few information related to the protocol. **As a consequence, a fault-tolerant active object has to be serializable.**

1.1.1. Communication Induced Checkpointing (CIC)

Each active object in a CIC fault-tolerant application has to checkpoint at least every **TTC** (Time To Checkpoint) seconds. When all the active objects have taken a checkpoint, a **global state** is formed. If a failure occurs, the **entire** application has to restart from such a global state. The TTC value depends mainly on the assessed frequency of failures. A little TTC value leads to very frequent global state creation and thus to a little rollback in the execution in case of failure. But a little TTC value leads also to a bigger overhead between a non-fault-tolerant and a fault-tolerant execution. The TTC value can be set by the programmer in the deployment descriptor.

The failure-free overhead induced by the CIC protocol is usually low, and this overhead is quasi-independent from the message communication rate. The counterpart is that the recovery time could be long since all the applications have to restart after the failure of one or more active object.

1.1.2. Pessimistic message logging (PML)

Each active object in a PML fault-tolerant application has to checkpoint at least every TTC seconds and all the messages delivered to an active object are logged on a stable storage. There is no need for global synchronization as with CIC protocol. Each checkpoint is independent: if a failure occurs, only the faulty process has to recover from its latest checkpoint. As for CIC protocol, the TTC value impact 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.

1.2. Making a ProActive application fault-tolerant

1.2.1. Resource Server

To be able to recover a failed active object, the fault-tolerance system must have access to a **resource server**. A resource server is able to return a free node that can host the recovered active object.

A resource server is implemented in ProActive in org.objectweb.proactive.core.body.ft.servers.resource.ResourceServer. This server can store free nodes at deployment time. The user can specify in the deployment descriptor a resource virtual node. Each node mapped on this virtual node will automatically register itself as a free node at the specified resource server.

1.2.2. Fault-Tolerance servers

Fault-tolerance mechanism needs servers for the checkpoints storage, the localization of the active objects, and the failure detection. Those servers are implemented in the current version as a unique server (org.objectweb.proactive.core.body.ft.servers.FTServer),

[1] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/AdvancedFeatures/pdf/../../ReferenceManual/multiple_html/GCMDeployment.html

that implements the interfaces of each server (org.objectweb.proactive.core.body.ft.servers.*.*). This global server also includes a resource server.

This server is a classfile server for recovered active objects. It must therfore have access to all classes of the application, i.e. it has to be started with **all classes of the application in its classpath**.

The global fault-tolerance server can be launched using the ProActive/bin/FT/startGlobalFTServer.[sh|bat] script, with 5 optional parameters:

- the protocol: -proto [cic|pml]. Default value is cic.
- the server name: -name <serverName>. The default name is FTServer.
- the port number: -port <portNumber>. The default port number is 1100.
- the fault detection period: -fdperiod <periodInSec>. This value defines the time between two consecutive fault detection scanning. The default value is 10 sec. Note that an active object is considered as faulty when it becomes unreachable, i.e. when it becomes unable to receive a message from another active object.

The server can also be directly launched in the java source code, using org.objectweb.proactive.core.process.JVMProcessImpl class:

```
JVMProcessImpl jvmProcessImpl = new JVMProcessImpl(
   new org.objectweb.proactive.core.process.AbstractExternalProcess.StandardOutputMessageLogger());
jvmProcessImpl.setClassname(org.objectweb.proactive.core.body.ft.servers.StartFTServer.class
      .getName());

// optional line: Default arguments
jvmProcessImpl.setParameters(Arrays.asList("-proto", "cic", "-name", "FTServer", "-port", "1100",
      "-fdperiod", "30"));

jvmProcessImpl.startProcess();
```

Note

If one of the servers is unreachable when a fault-tolerant application is deploying, fault-tolerance is automatically and transparently disabled for all the application.

Warning

When launching the fault-tolerance server, you have to specify the "java.security.manager" JVM argument as well as the "java.security.policy" JVM argument. Otherwise, you would get an access denied exception.

1.2.3. Configure fault-tolerance for a ProActive application

Fault-tolerance capabilities of a ProActive application are set in the deployment descriptor, using a faultTolerance technical service, defined by the class org.objectweb.proactive.core.body.ft.service.FaultToleranceTechnicalService. This service is defined in the GCMA descriptor: active objects that are deployed with this technical service are turned into fault-tolerant. This service has first to defines the protocol that have to be used for this application. The user can select the appropriate protocol with the entry <property name="protocol" value="[cic|pml]"/> in the definition of the service.

The service also defines **servers URLs**:

- <property name="global" value="rmi://..."/> set the URL of a **global** server, i.e. a server that implements all needed methods for fault-tolerance mechanism (stable storage, fault detection, localization). If this value is set, all others URLs will be **ignored**.
- <property name="checkpoint" value="rmi://..."/> set the URL of the checkpoint server, i.e. the server where checkpoints are stored.

- <property name="location" value="rmi://..."/> set the URL of the location server, i.e. the server responsible for giving references on failed and recovered active objects.
- <property name="recovery" value="rmi://..."/> set the URL of the recovery process, i.e. the process responsible for launching the recovery of the application after a failure.
- <property name="resource" value="rmi://..."/> set the URL of the resource server, i.e. the server responsible for providing free nodes that can host a recovered active object.

Finally, the **TTC** value is set in fault-tolerance service, using <property name="ttc" value="x"/>, where x is expressed in **seconds**. If not, the default value (30 sec) is used.

1.2.4. A deployment descriptor example

Here is an example of GCMA descriptor that deploys 2 virtual nodes: one for deploying fault-tolerant active objects and one as resource for recovery. The two fault-tolerance behaviors correspond to two fault-tolerance services, Workers and Resource. Note that non-fault-tolerant active objects can communicate with fault-tolerant active objects as usual. Chosen protocol is CIC and TTC is set to 5 sec for all the application.

```
<?xml version='1.0' encoding='UTF-8'?>
<GCMApplication
  xmlns='urn:gcm:application:1.0'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='urn:gcm:application:1.0 http://proactive.inria.fr/schemas/gcm/1.0/
ApplicationDescriptorSchema.xsd'>
  <environment>
    <javaPropertyVariable name='proactive.home'/>
    <javaPropertyVariable name='java.home'/>
    <descriptorDefaultVariable name='deploymentDescriptor' value='../../_CONFIG/localhost/generic.xml' />
    <descriptorDefaultVariable name='jvmargDefinedByTest' value=' '/>
  </environment>
  <application>
    <proactive relpath='${proactive.home}' base='root'>
      <configuration>
        <java base='root' relpath='${java.home}/bin/java' />
      </configuration>
      <virtualNode id='Workers' capacity='2'>
        <technicalServices>
          <class name='org.objectweb.proactive.core.body.ft.service.FaultToleranceTechnicalService'>
            <property name='global' value='rmi://localhost:1100/FTServer' />
            <property name='ttc' value='5' />
            <property name='protocol' value='cic' />
          </class>
        </technicalServices>
      </virtualNode>
      <virtualNode id='Resource' capacity='1'>
        <technicalServices>
          <class name='org.objectweb.proactive.core.body.ft.service.FaultToleranceTechnicalService'>
            <property name='global' value='rmi://localhost:1100/FTServer' />
            <property name='ttc' value='5' />
            <property name='protocol' value='cic' />
            <property name='resource' value='rmi://localhost:1100/FTServer' />
          </class>
        </technicalServices>
      </virtualNode>
    </proactive>
```

```
  </application>
  <resources>
    <nodeProvider id='remote'>
      <file path='${deploymentDescriptor}'/>
    </nodeProvider>
  </resources>
</GCMApplication>
```

1.3. Programming rules

1.3.1. Serializable

Persistence of active objects is obtained through standard Java serialization: a checkpoint thus consists in an object containing a serialized copy of an active object and a few information related to the protocol. As a consequence, a fault-tolerant active object **has to be serializable**. If a non serializable object is activated on a fault-tolerant virtual node, fault-tolerance is automatically and transparently disabled for this active object.

1.3.2. Standard Java main method

Standard Java thread, typically main method, cannot be turned into fault-tolerant. As a consequence, if a standard main method interacts with active objects during the execution, consistency can no more be ensured: after a failure, all the active objects will roll back to the most recent global state **but the main will not**.

So as to avoid such inconsistency on recovery, the programmer must minimizes the use of standard main by, for example, delegating the initialization and launching procedure to an active object.

```
public static void main(String[] args){
  Initializer init = (Initializer)(PAActiveObject.newActive(
    'Initializer.getClass.getName()', args);
  init.launchApplication();
  System.out.println('End of main thread');
}
```

The object init is an active object, and as such will be rolled back if a failure occurs: the application is kept consistent.

1.3.3. Checkpointing occurrence

To keep fault-tolerance fully transparent (see the technical report[2] for more details), active objects can take a checkpoint **before the service of a request**. As a first consequence, if the service of a request is infinite, or at least much greater than TTC, the active object that serves such a request can no more take checkpoints. If a failure occurs during the execution, this object will force the entire application to rolls back to the beginning of the execution. The programmer must thus avoid infinite method such as

```
public void infiniteMethod(){
  while (true){
    this.doStuff();
  }
}
```

The second consequence concerns the definition of the runActivity() method (see Chapter 9.5.5. Using A Custom Activity[3]). Let us consider the following example:

[2] http://www-sop.inria.fr/oasis/personnel/Christian.Delbe/publis/rr5246.pdf

[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/AdvancedFeatures/pdf/../../ReferenceManual/multiple_html/ActiveObjectCreation.html#ActiveObjectCreationWithCustomActivity

```
public void runActivity(Body body) {
  org.objectweb.proactive.Service service = new org.objectweb.proactive.Se\
rvice(body);
  while (body.isActive()) {
    Request r = service.blockingRemoveOldest();
    ...
    /* CODE A */
    ...
    /* CHECKPOINT OCCURRENCE */
    service.serve(r);
  }
}
```

If a checkpoint is triggered before the service of r, it characterizes the state of the active object at the point /* CHECKPOINT OCCURRENCE */. If a failure occurs, this active object is restarted by calling the runActivity() method, **from a state in which the code /* CODE A */ has been already executed**. As a consequence, the execution looks like if /* CODE A */ has been executed twice.

The programmer should then avoid to alter the state of an active object in the code preceding the call to service.serve(r) when he redefines the runActivity() method.

1.3.4. Activity Determinism

All the activities of a fault-tolerant application has to be deterministic (see [BCDH04] for more details). The programmer has then to avoid the use of non-deterministic methods such as Math.random().

1.3.5. Limitations and known bugs

Fault-tolerance in ProActive is still not compliant with the following features:

- active objects exposed as Web services (see Chapter 4, *Exporting active objects as Web Services*), or reachable using http protocol
- security (see Chapter 2, *Security Framework*), as fault-tolerance servers are implemented using standard RMI

CIC and PML protocols are not compatible: a fault-tolerance application can use **only** one of these two protocols.

Fault-Tolerance in ProActive is not compliant with on-the-fly RMI stub generation, available since Java 1.5. Even with a JRE 1.5 or greater, ProActive RMI stubs has to be created **before** running the application, with the /ProActive/compile/build.[sh/bat] script.

1.4. A complete example

1.4.1. Description

You can find in ProActive/examples/nbody/nbodyFaultTolerance.[sh|bat] a script that starts a fault-tolerant version of the ProActive NBody[4] example. This script actually call the ProActive/examples/nbody/nbody.[sh|bat] script with the -displayft option. The Java source code is the same as the standard version. The only difference is the 'Execution Control' panel added in the graphical interface, which allows the user to remotely kill Java Virtual Machine so as to trigger a failure by sending a killall java signal. Note that this panel will not work with Windows operating system, since the killall does not exist. But a failure can be triggered for example by killing the JVM process on one of the hosts.

[4] http://proactive.inria.fr/apps/nbody.html

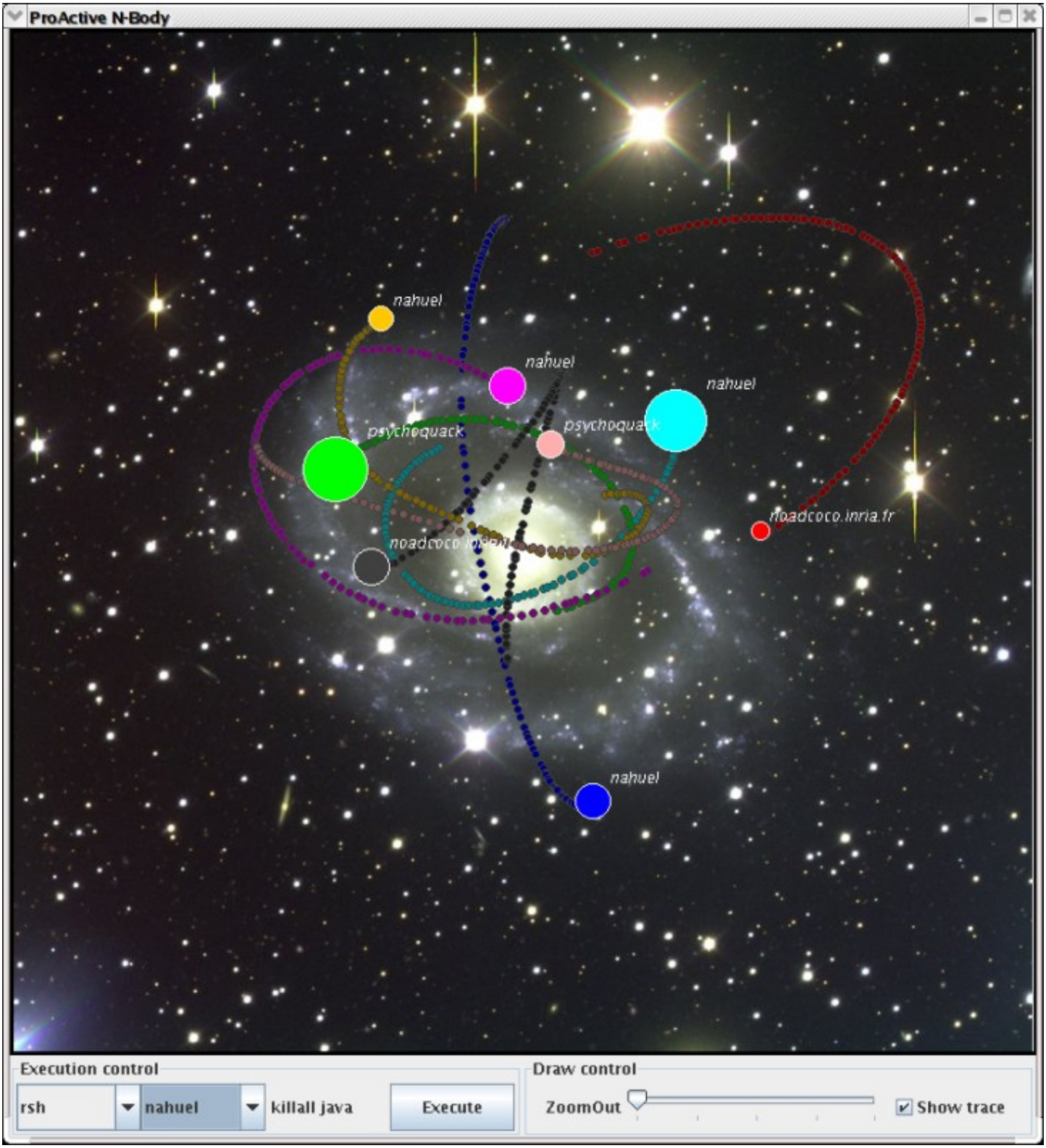


Figure 1.1. The nbody application, with Fault-Tolerance enabled

This snapshot shows a fault-tolerant execution with 8 bodies on 3 different hosts. Clicking on the 'Execute' button will trigger the failure of the host called Nahuel and the recovery of the 8 bodies. The checkbox **Show trace** is checked: the 100 latest positions of each body are drawn with darker points. These traces allow to verify that, after a failure, each body finally reach the position it had just before the failure.

1.4.2. Running NBody example

Before starting the fault-tolerant body example, you have to edit the ProActive/examples/nbody/GCMA_FautlTolerance.xml GCMA descriptor, to load a GCMD with your own hosts, as follow:

```
...
<resources>
  <nodeProvider id='worker'>
    <file path='../TO/YOUR/GCMD.xml' />
  </nodeProvider>
</resources>
...
```

Of course, more than one host is needed to run this example, as failure are triggered by killing all Java processes on the selected host.

The GCMA descriptor must also specify the GlobalFTServer and ResourceServer location by setting the global and resource technical service properties.

Before running the application itself, you should start the Fault-Tolerance server, using ProActive/bin/startGlobalFTServer.[sh|bat] script (see Section 1.2.2, “Fault-Tolerance servers”).

Finally, you can start the fault-tolerant ProActive NBody and choose the version you want to run:

```
~/ProActive/examples/nbody> ./nbodyFaultTolerance.sh
Starting Fault-Tolerant version of ProActive NBody...
--- N-body with ProActive --------------------------------
 **WARNING**: $PROACTIVE/descriptors/FaultTolerantWorkers.xml MUST BE SET \
WITH EXISTING HOSTNAMES !
       Running with options set to 4 bodies, 3000 iterations, display true
 1: Simplest version, one-to-one communication and master
 2: group communication and master
 3: group communication, odd-even-synchronization
 4: group communication, oospmd synchronization
 5: Barnes-Hut, and oospmd
Choose which version you want to run [12345]:
4
Thank you!
 --> This ClassFileServer is reading resources from classpath
Ibis enabled
Created a new registry on port 1099
//tranquility.inria.fr/Node-157559959 successfully bound in registry at //t\
ranquility.inria.fr/Node-157559959
Generating class: pa.stub.org.objectweb.proactive.examples.nbody.common.St\
ub_Displayer
************ Reading deployment descriptor: file:./../../../descriptors/\
FaultTolerantWorkers.xml ********************
```

Chapter 2. Security Framework

In order to use the Proactive Security features, you have to install **the Java(TM) Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files** available at Sun's website[1]. Extract the file and copy jar files to your <jre_home>/lib/security. That will override the *local_policy.jar* and *US_export_policy.jar* files, so if you want to be able to undo this operation, rename these files before copying the new ones.

2.1. Overview

Usually, applications and security are developed for a specific use. We propose here a security framework that allows dynamic deployment of applications and security configuration according to this deployment.

ProActive security mechanism provides a set of security features from basic ones like communications authentication, integrity, and confidentiality to more high-level features including secure object migrations, hierarchical security policies, and dynamically negotiated policies. All these features are expressed at the ProActive middleware level and used transparently by applications.

It is possible to attach security policies to Runtimes, Virtual Nodes, Nodes and Active Objects. Policies are expressed inside an XML descriptor.

2.2. Security Architecture

2.2.1. Base model

A distributed or concurrent application built with **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. All other objects inside the active object are called **passive objects** and cannot be referenced directly from objects which are outside of this active object (see Figure 2.1, “Setting up virtual distributed sandboxes at runtime”). This absence of sharing is important with respect to security.

[1] http://java.sun.com/javase/downloads/index.jsp

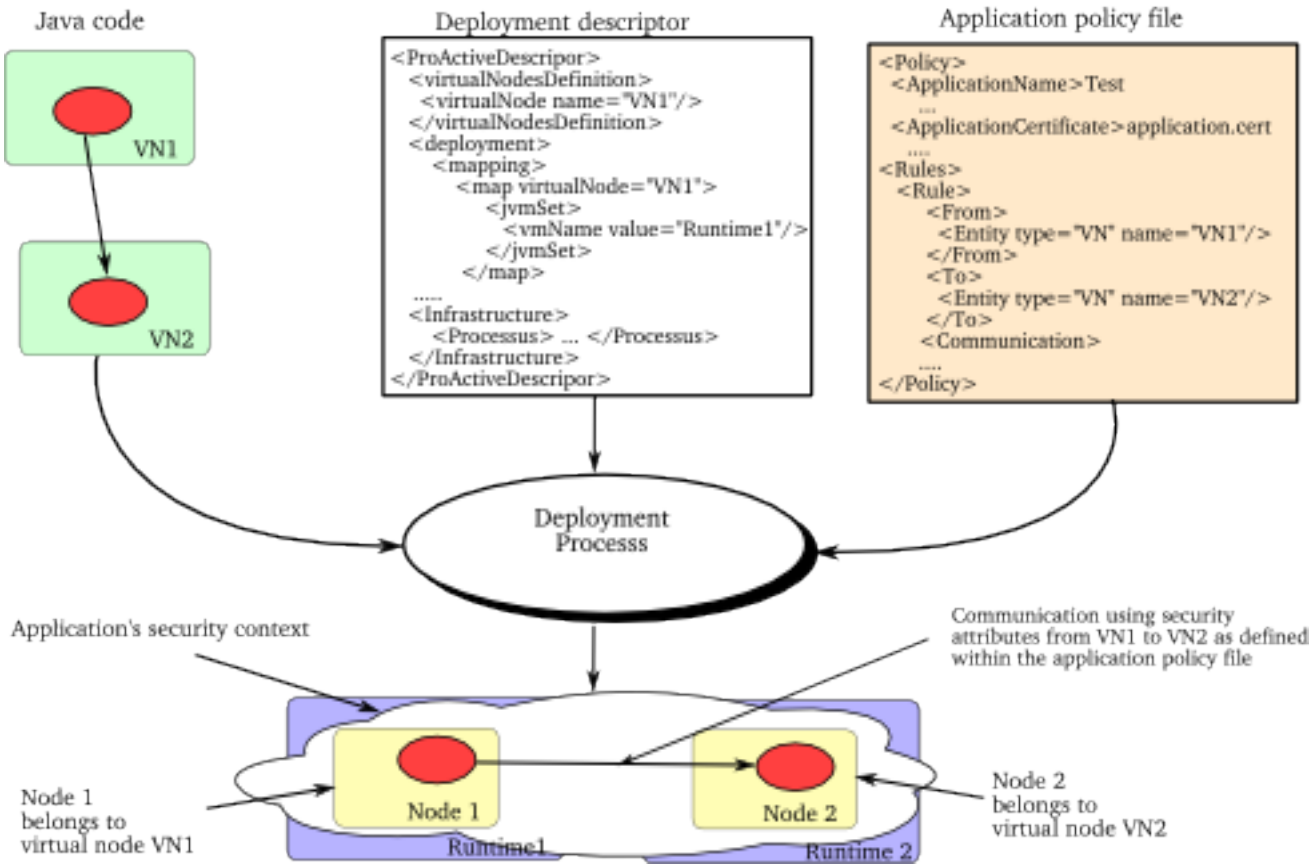


Figure 2.1. Setting up virtual distributed sandboxes at runtime

The security is based on Public Key Infrastructure. Each entity owns a certificate and an private key generated from a user's certificate.

Certificates are generated automatically by the security mechanism. The validity of a certificate is checked by validating its certificate chain. As shown in the next figure, before validating the certificate of an active object, the certificate of the application and the user's one will be checked. If a valid path is found then the certificate of the object is validated.

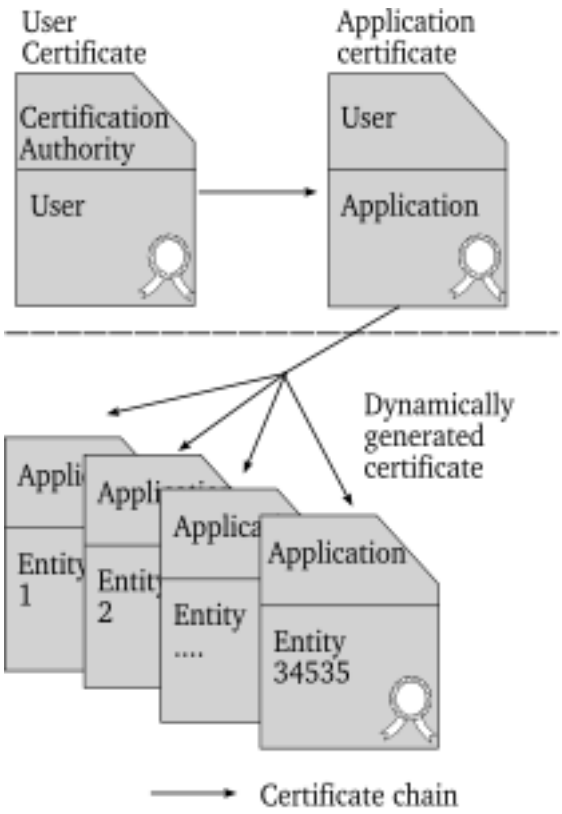


Figure 2.2. Certificate chain

2.2.2. Security is expressed at different levels

Security is expressed at different levels, according to who wants to set policy:

- Administrators set policy at the domain level. It contains general security rules.
- Resource provider set policy for resources. People who have access to a cluster and wants to offer CPU time under some restrictions. The runtime loads its policy file at launch time.
- Application level policy is set when an application is deployed through an XML descriptor.

The ProActive middleware will enforce the security policy of all entities interacting within the system, ensuring that all policies are applied.

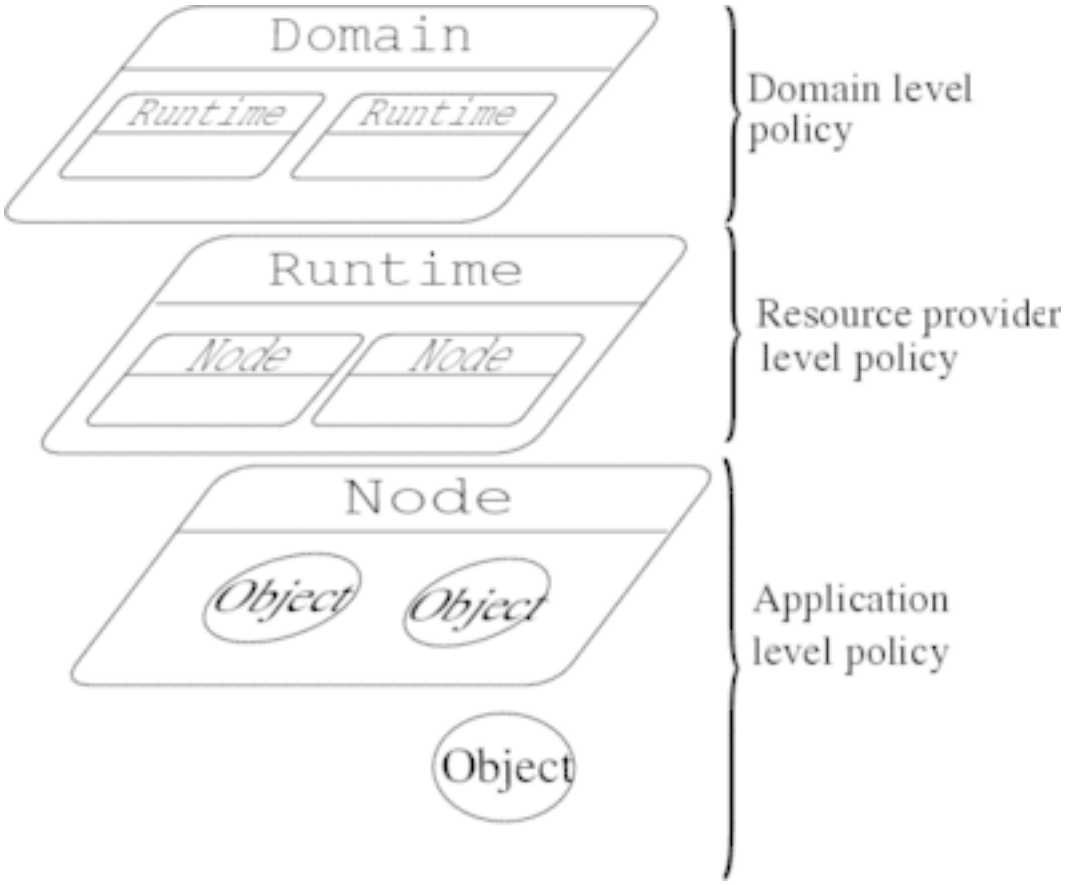


Figure 2.3. Hierarchical security

2.3. Detailed Security Architecture

2.3.1. Nodes and Virtual Nodes

The security architecture relies on two related abstractions for deploying Grid applications: **Node** and **Virtual Node**. A node gathers several objects in a logical entity. It provides an abstraction for the physical location of a set of activities. Objects are bound to a node at creation or after migration. In order to have a flexible deployment (eliminating from the source code machine names, creation protocols), the system relies on **Virtual Nodes** (VNs). A VN is identified as a name (a simple string), used in a program source, defined and configured in an descriptor. The user can attach policy to these virtual nodes. Virtual Nodes are used within application code to structure it. For example, an object which will be used as a server will be set inside a virtual node named "Server_VN" and client objects will be set inside a "Client_VN" virtual node. The user expresses policy between server and client object inside a descriptor file. The mapping between Virtual Nodes and Nodes is done when the application starts.

2.3.2. Hierarchical Security Entities

Grid programming is about deploying processes (activities) on various machines. The final security policy that must be set for those processes depends on many factors: that is dictated by the application's policy, but other factor such as the machine locations, the security policies of their administrative domain, and the network being used to reach those machines has to be considered too.

The previous section defined the notions of **Virtual Nodes** and **Nodes**. Virtual Nodes are application abstractions, and nodes are only a runtime entity resulting from the deployment: a mapping of Virtual Nodes to processes and hosts. A first decisive feature allows the definition of application-level security on those abstractions:

Definition 1. *Virtual Node Security*
Security policies can be defined at the level of Virtual Nodes. At execution, that security will be imposed on the Nodes resulting from the mapping of Virtual Nodes to JVMs, and Hosts.

Thus, virtual nodes are the foundation for intrinsic application level security. If, at design time, it appears that a process always requires a specific level of security (e.g. authenticated and encrypted communications all the time), then that process should be attached to a virtual node on which those security features are imposed. It is the designer responsibility to structure his/her application or components into virtual node abstractions compatible with the required security. Whatever deployment occurs, those security features will be maintained. We expect this case to occur infrequently, for instance in very sensitive applications where even an intranet deployment calls for encrypted communications.

The second decisive feature deals with a major Grid aspect: deployment-specific security. The issue is actually twofold:

1. allowing organizations (security domains) to specify general security policies,
2. allowing application security to be specifically adapted to a given deployment environment.

Domains are a standard way to structure (virtual) organizations involved in a Grid infrastructure; they are organized in a hierarchical manner. This logical concept allows the expression of security policies in a hierarchical way.

Definition 2. *Declarative Domain Security*
Fine grain and declarative security policies can be defined at the level of Domains. *A Security Domain is a domain to which a certificate and a set of rules are associated.*

This principle deals with the two issues mentioned above:

- The administrator of a domain can define specific policy rules that must be obeyed by the applications running within the domain. However, a general rule expressed inside a domain may prevent the deployment of a specific application. To solve this issue, a policy rule can allow a well-defined entity to weaken it. As we are in a hierarchical organization, allowing an entity to weaken a rule means allowing all included entities to weaken the rule. The entity can be identified by its certificate;
- a Grid user can, at the time he runs an application, specify additional security based on the domains being deployed onto.

The Grid user can specify additional rules directly in his deployment descriptor for the domains his application is deployed on. Note that those domains are actually dynamic as they can be obtained through external allocators, or even Web Services in an OGSA infrastructure (see [FKTT98]). Catch-all rules might be important in that case to cover all cases, and to provide a conservative security strategy for unforeseen deployments.

Finally, as active objects are active and mobile entities, there is a need to specify security at the level of such entities.

Definition 3. *Active Object Security*
Security policies can be defined at the level of Active Object. Upon migration of an activity, the security policy attached to that object follows.

In open applications, e.g. several principals interacting in a collaborative Grid application, a JVM (a process) launched by a given principal can actually host an activity executing under another principal. The security of this latter principal has to be retained in such a case. Moreover, it can also serve as a basis to offer, in a secure manner, hosting environments for mobile agents.

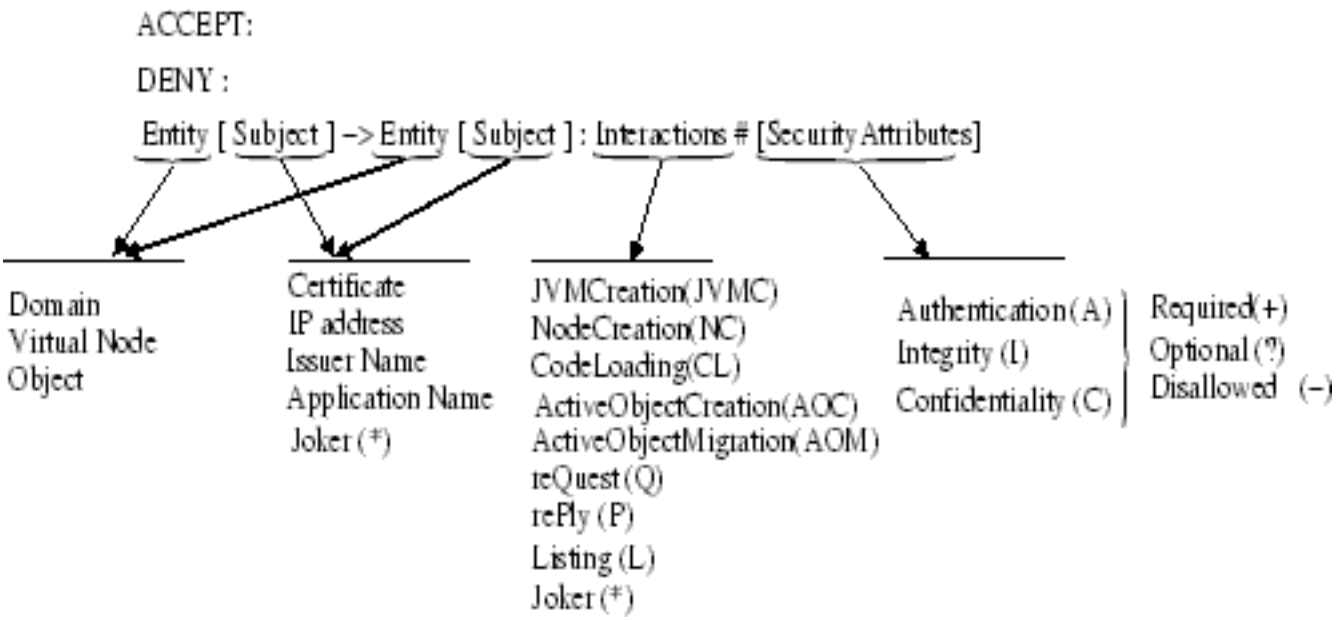


Figure 2.4. Syntax and attributes for policy rules

2.3.3. Resource provider security features

Prior to an application starting on a grid, a user needs to acquire some resources (CPU time, disk storage, bandwidth) from the grid. A **resource provider** is an individual, a research institute or an organization who wants to offer some resources under a certain security policy to a restricted set of people. According to our definition, the resource provider will set up one or more runtimes where clients will be able to perform computation. Each runtime is set with its own policy. Theses runtimes could be globally distributed.

2.3.4. Interactions, Security Attributes

Security policies are able to control all the **interactions** that can occur when deploying and executing a multi-principals Grid application. With this goal in mind, interactions span the creation of processes, to the monitoring of activities (Objects) within processes, including of course the communications. Here is a brief description of those interactions:

- RuntimeCreation (RC): creation of a new Runtime process
- NodeCreation (NC): creation of a new Node within a Runtime (as the result of Virtual Node mapping)
- CodeLoading (CL): loading of bytecode within a Node, used in presence of object migration.
- ObjectCreation (OC): creation of a new activity (active object) within a Node
- ObjectMigration (OM): migration of an existing activity object to a Node
- Request (Q), Reply (P): communications, method calls and replies to method calls
- Listing (L): list the content of an entity; for Domain/Node provides the list of Node/Objects, for an Object allows to monitor its activity.

For instance, a domain is able to specify that it accepts downloading code from a given set of domains, whose transfer has been authenticated and guaranteed not to be altered. As a policy might allow un-authenticated communications, or because a domain (or even country) policy may specify that all communications are un-encrypted, the three security attributes Authentication (A), Integrity (I) and Confidentiality (C) can be specified in three modes: Required (+), Optional (**?**), Disallowed (-)

For example, the tuple [+A,?I,-C] means that authentication is required, integrity is accepted but not required, and confidentiality is not allowed.

As grids are decentralized, without a central administrator controlling the correctness of all security policies, these policies must be **combined**, **checked**, and dynamically **negotiated**. The next two sections discuss how this is done.

2.3.5. Combining Policies

As the proposed infrastructure takes into account different actors of the grid (e.g. domain administrator, grid user), even for a single-principal single-domain application, there are potentially several security policies to be considered. This section deals with the combination of those policies to obtain the final security tuple of a single entity. An important principle being is **a sub-domain cannot weaken the rules of its super-domains**.

During execution, each activity (Active Object) is always included in a **Node** (due to the Virtual Node mapping) and at least in one **Domain**, the one used to launch a JVM (D_0). Figure 2.5, "Hierarchical Security Levels" hierarchically represents the security rules that can be activated at execution: from the top, hierarchical domains (D_1 to D_0), the virtual node policy (VN), and the Active Object (AO) policy. Of course, such policies can be inconsistent, and there must be clear principles to combine the various sets of rules.

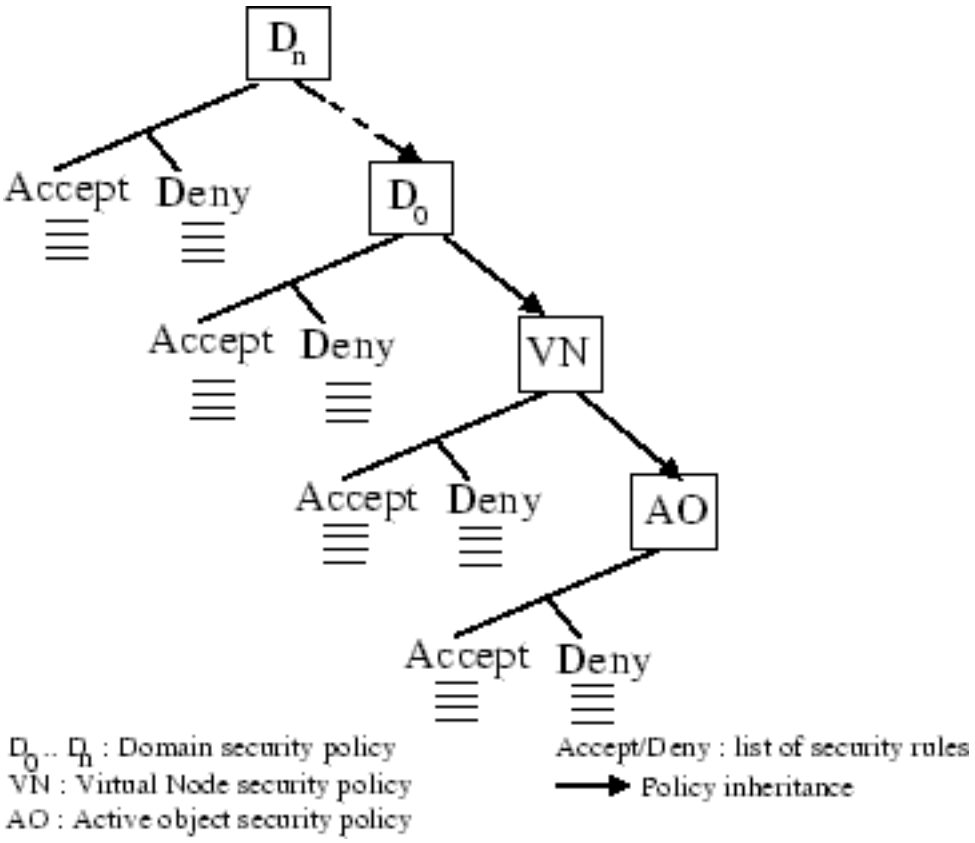


Figure 2.5. Hierarchical Security Levels

There are three main principles:

1. choosing the **most specific rules** within a given domain (as a single grid actor is responsible for it),
2. an interaction is valid only if all levels accept it (absence of weakening of authorizations),
3. the security retained attributes are the most constrained based on a partial order (absence of weakening of security).

Let us consider the following example, where the catch-all rule specifies that all Requests (Q) and Replies (P) must be authenticated, integrity checked, and confidential, but within the specific "CardPlus" domain integrity and confidentiality will be optional.

```
Domain[*] -> Domain[*]: Q,P: [+A,+I,+C]
Domain[CardPlus] -> Domain[CardPlus]: Q,P: [+A,?I,?C]
```

This means that for any activity taking place within the CardPlus domain, the second rule will be chosen (integrity and confidentiality will be optional), as the catch-all rule is less-specific than the "CardPlus" domain rule, and there is no hierarchical domain relationship

between the two rules. Of course, comparison of rules is only a partial order, and several incompatible most specific rules can exist within a single level (e.g. both ACCEPT and DENY most specific rules for the same interaction, or both +A and -A).

Between levels, an incompatibility can also occur, especially if a sub-level attempts to weaken the policy on a given interaction (e.g. a domain prohibits confidentiality [-C] while a sub-domain or the Virtual Node requires it [+C], a domain D_i prohibits loading of code while D_j (j <= i) authorizes it). In all incompatible cases, the interaction is not authorized and an error is reported.

2.3.6. Dynamic Policy Negotiation

During execution, entities interact in a pairwise fashion. For each interaction (JVM creation, communication, migration, ...), each entity will want to apply a security policy based on the resolution presented in the previous section. Before starting an interaction, a **negotiation** occurs between the two entities involved. Figure 2.6, "Result of security negotiations" shows the result of such negotiation. For example, if for a given interaction, entity A's policy is [+A,?I,?C], and B's policy is [+A,?I,-C], the negotiated policy will be [A,?I,-C]. If both policies specify an attribute as optional, the attribute is not activated.

The other case which leads to an error is when an attribute is required by one, and disallowed by the other. In such a case, the interaction is not authorized and an error is reported. If the two entities security policies agree, then the interaction can occur. In the case that the agreed security policy includes confidentiality, the two entities negotiate a session key.

		ENTITY A		
		Required (+)	Optional (?)	Disallowed (–)
ENTITY B	Required (+)	+	+	Error
	Optional (?)	+	?	–
	Disallowed (–)	Error	–	–

Figure 2.6. Result of security negotiations

2.3.7. Migration and Negotiation

In large scale grid applications, migration of activities is an important issue. The migration of Active Objects must not weaken the security policy being applied.

When an active object migrates to a new location, three scenarios are possible:

- the object migrates to a node belonging to the same virtual node and included in the same domain. In this case, all negotiated sessions remain valid.
- the object migrates to a known node (created during the deployment step) but which belongs to another virtual node. In this case, all current negotiated sessions become invalid. This kind of migration requires to reestablish the object security policy, and if it changes, to renegotiate with interacting entities.

- The object migrates to an unknown node (not known at the deployment step). In this case, the object migrates with a copy of the application security policy. When a secured interaction takes place, the security system retrieves not only the object's application policy but also policies rules attached to the node on which the object is to compute the policy.

2.4. How to create policy files and certificates

A GUI has been created to facilitate certificate generation. It is a RCP plugin shipped with the IC2D tool. Once IC2D started, go to **Window -> Open Perspective -> Other...** and select **Security**.

The following window should then appear:

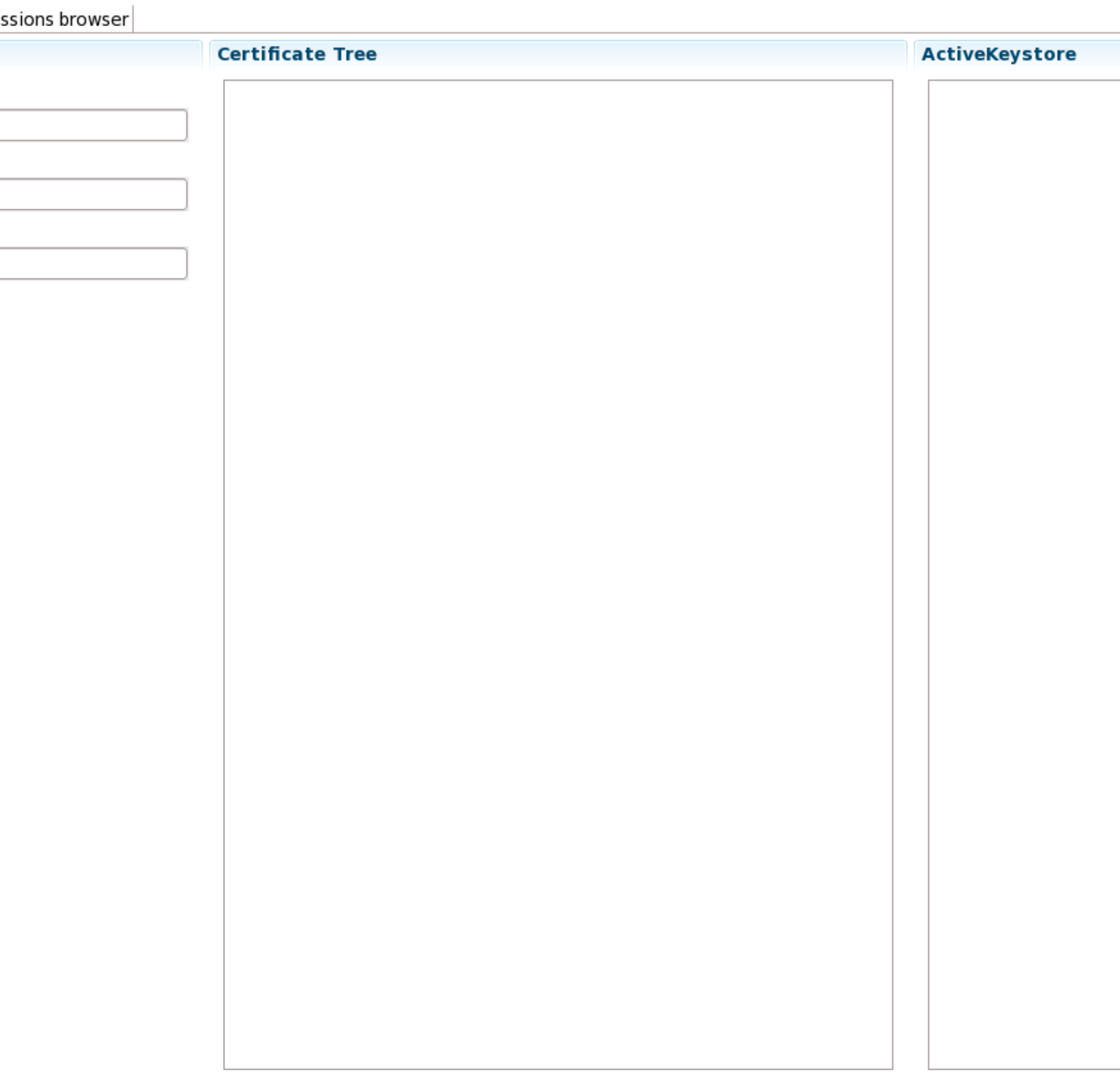


Figure 2.7. The IC2D Security Perspective

From this tab, you can generate a root certificate filling the certificate information column and clicking on **Generate self signed certificate**.

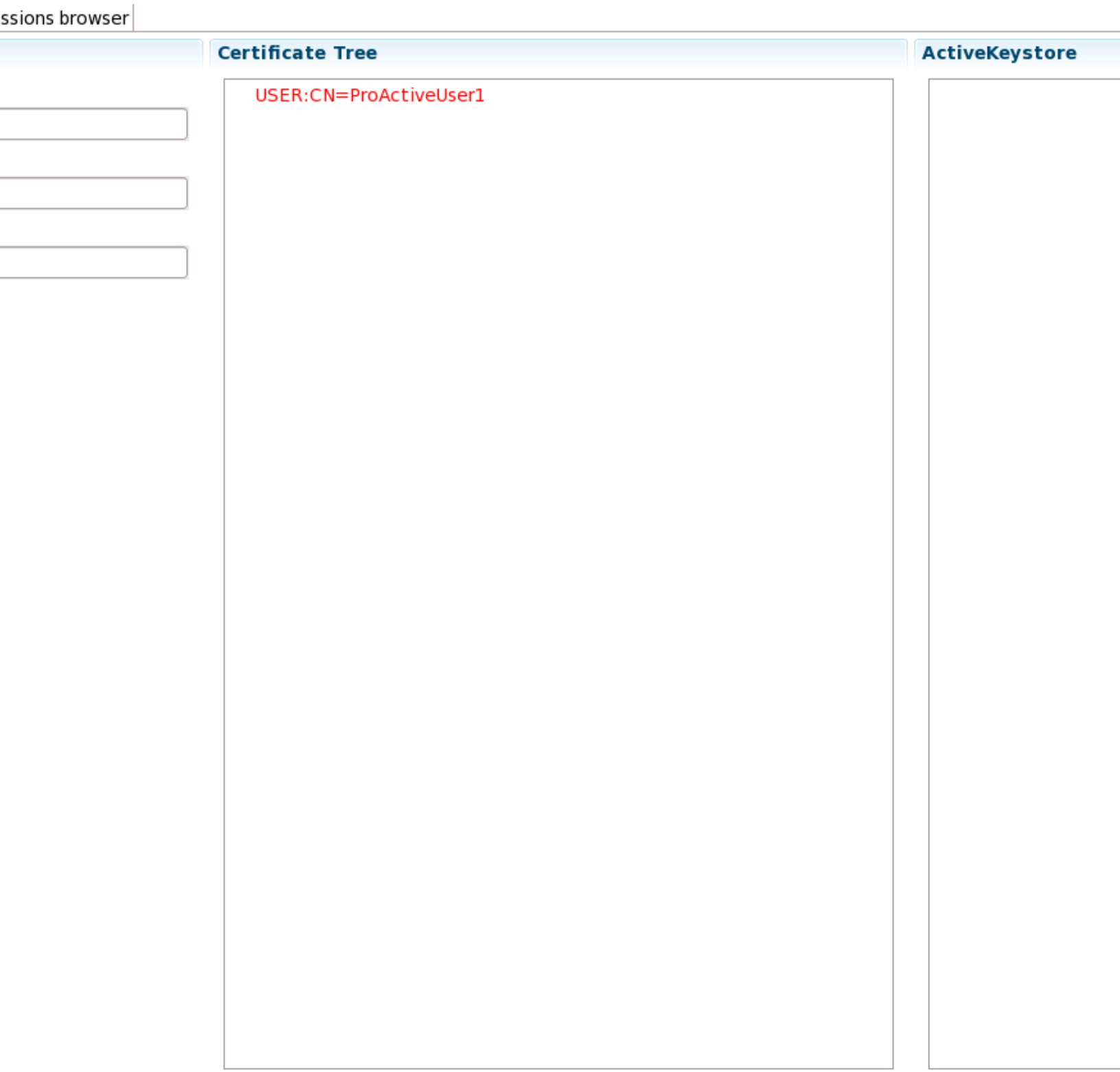


Figure 2.8. Creating a root certificate

Then, you can generate a child certificate filling the certificate information column and clicking on **Generate child certificate** after having selected the parent certificate on the second column.

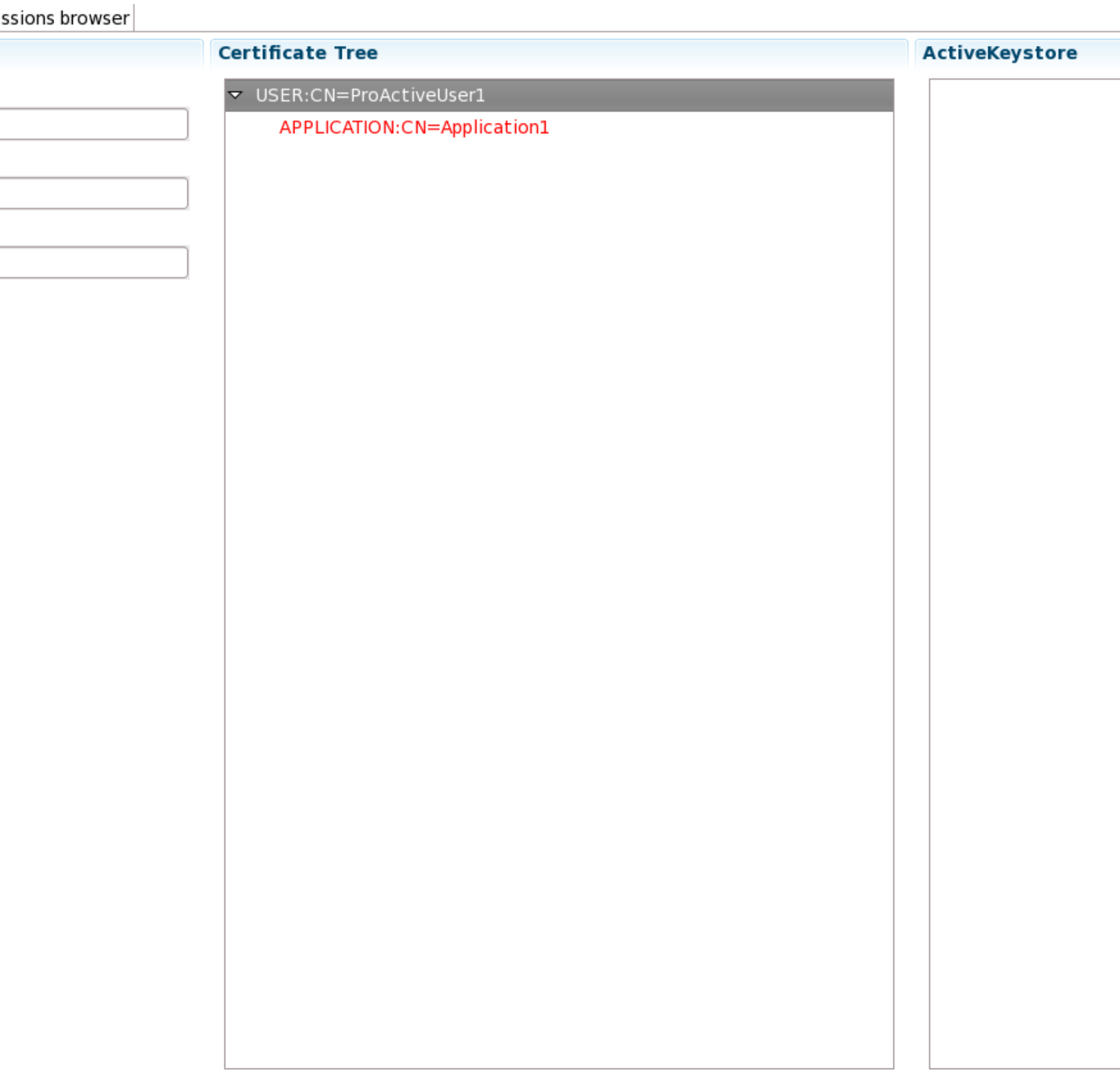


Figure 2.9. Creating a child certificate

Once you have generated all the certificates you need, you can drag and drop the ones you want to activate from the second column to the last one. Activating certificates allows you to use them afterwards for rule creation. It also allows you to save them into a file.

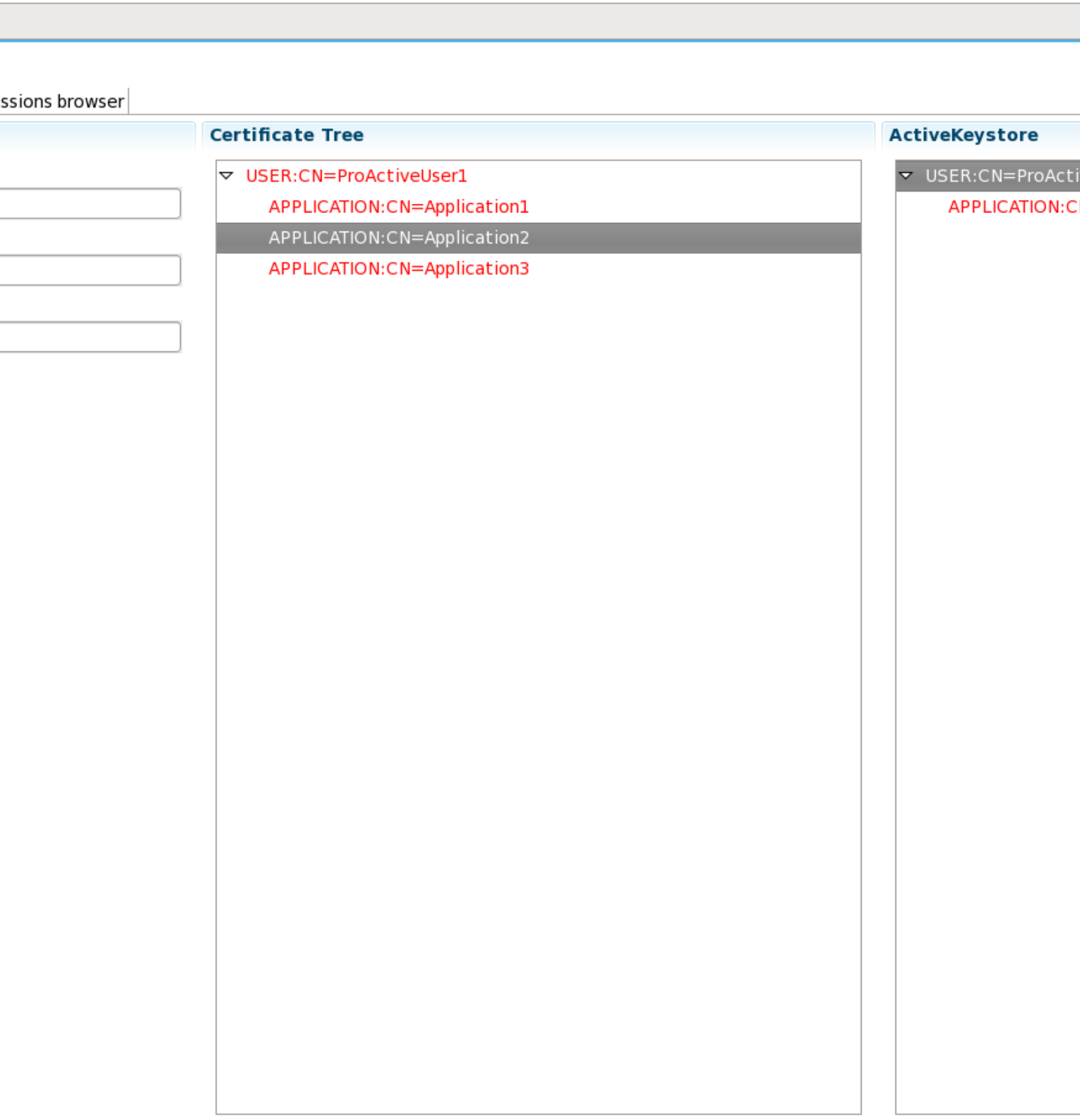


Figure 2.10. Activating a keystore

Now, let us have a look at the *Keystore Editor* tab. This tab allows you to get some information on the generated keytores, to save them into files or to load some others to be used after for rule edition. Certificates are saved under a PKCS12 format (extension .p12). This format is natively supported by the ProActive Security mechanism.

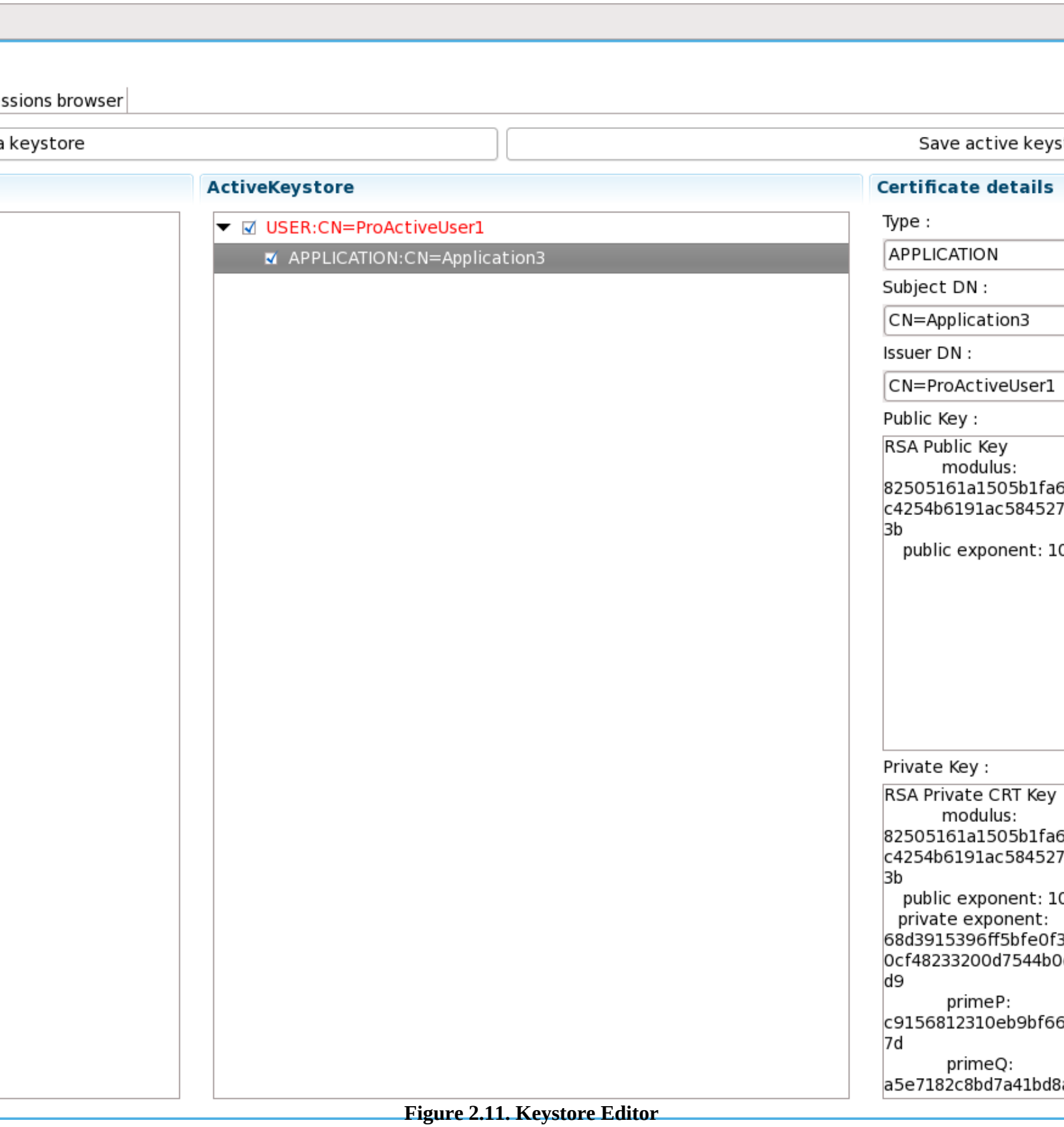


Figure 2.11. Keystore Editor

Finally, you can use the *Rule Editor* tab to edit your rules. These rules can be saved into a policy file using the *Save rules* button. To get familiar with the generated policy file, please refer to Section 2.6, “The XML Security Descriptor in details”. To create a new rule, you just have to click on the *New* button and then fill the right panel. The *From* and *To* part can be filled by drag and drop from the *ActiveKeystore* column.

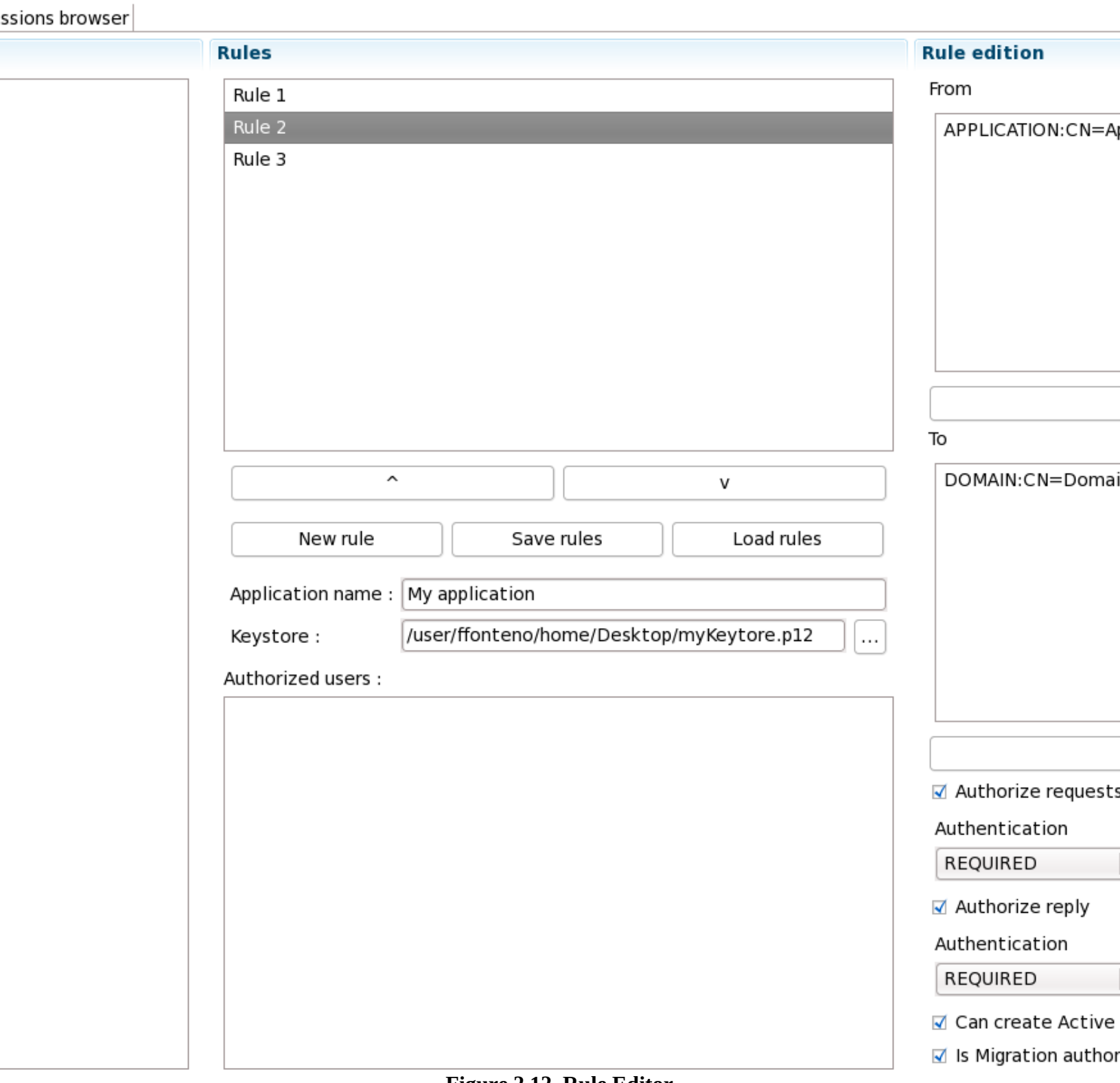


Figure 2.12. Rule Editor

The last tab, *Sessions browser*, cannot be used directly. It is used to see certificate details at runtime. To do this, you have to monitor your application with IC2D and then, you can click right on an active object and select **Export SM to Security view**. You will then have access to a window looking as follows:

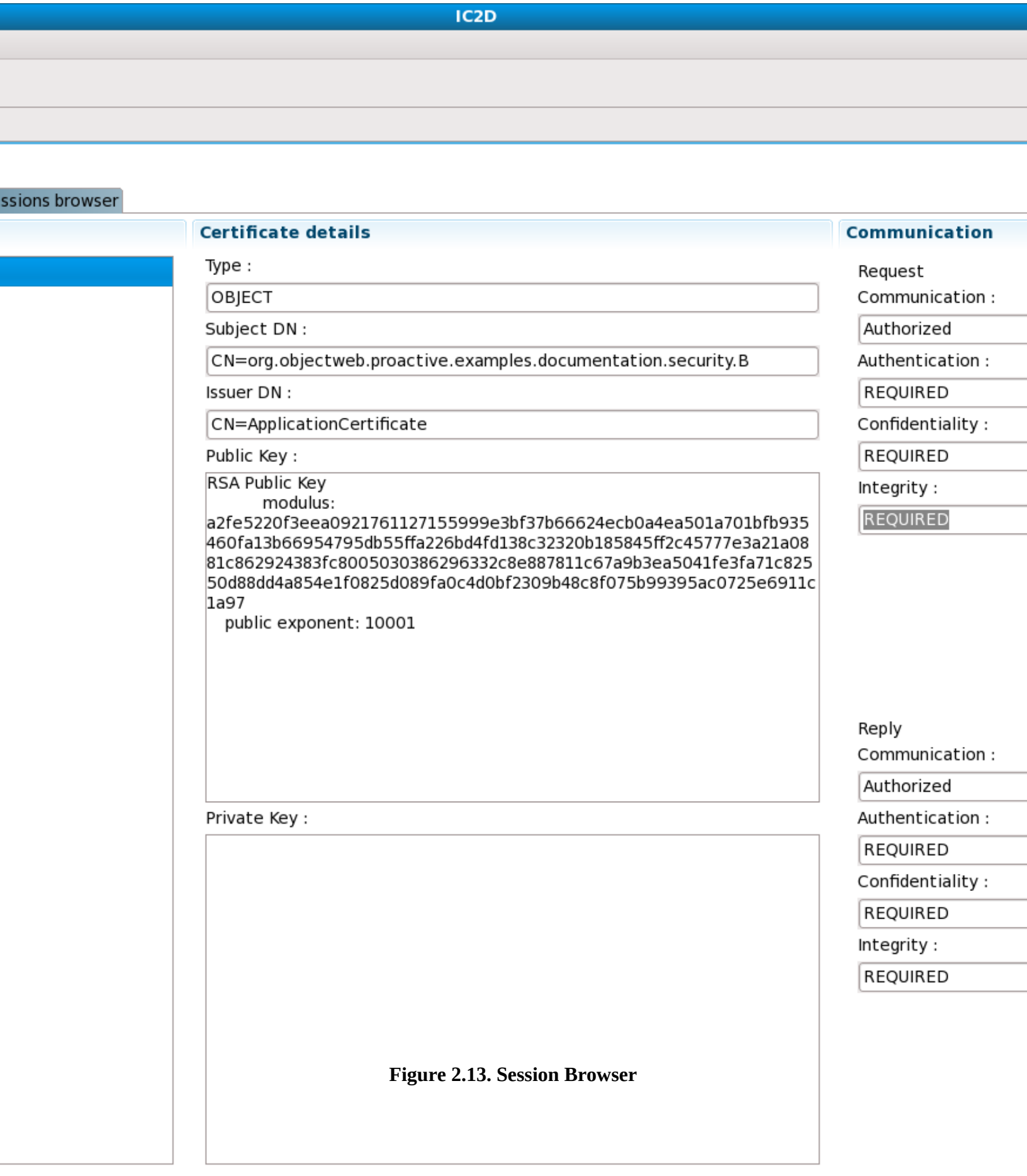


Figure 2.13. Session Browser

2.5. Activating security mechanism

Within the deployment descriptor, the <security> tag is used to specify the policy for the deployed application. It will be the policy for all the created Nodes and Active Objects. The following example shows an example of a deployment descriptor with such a <security> tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<ProActiveDescriptor xmlns="urn:proactive:deployment:3.3"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:proactive:deployment:3.3 http://www-sop.inria.fr/oasis/ProActive/schemas/
deployment/3.3/deployment.xsd">

   <!-- Variable Definitions -->
   <variables>
      <descriptorVariable name="PROACTIVE_HOME" value="/user/ffonteno/home/proactive-git/programming" />
      <descriptorVariable name="JAVA_HOME" value="/user/ffonteno/home/src/java/jdk1.6.0_11/jre/bin/java" />
   </variables>

   <!-- Security Policy Definitions -->
   <security>
       <file uri='ApplicationPolicy.xml'/>
   </security>

   <!-- Virtual Node Definitions -->
   <componentDefinition>
      <virtualNodesDefinition>
         <virtualNode name="VN_A"/>
         <virtualNode name="VN_B"/>
      </virtualNodesDefinition>
   </componentDefinition>

   <deployment>

      <!-- Mappings between Virtual Nodes and JVMs -->
      <mapping>
         <map virtualNode="VN_A">
            <jvmSet>
               <vmName value="jvm_A" />
            </jvmSet>
         </map>
         <map virtualNode="VN_B">
            <jvmSet>
               <vmName value="jvm_B" />
            </jvmSet>
         </map>
      </mapping>

      <!-- Mappings between JVMs and process references. -->
      <!-- Process references are used hereafter (within the infrastructure element)
          to describe the process used to create the JVMs. -->
      <jvms>
         <jvm name="jvm_A">
            <creation>
               <processReference refid="localJVM_A" />
            </creation>
```

```
        </jvm>
        <jvm name="jvm_B">
          <creation>
            <processReference refid="localJVM_B" />
          </creation>
        </jvm>
      </jvms>
    </deployment>
    <infrastructure>
      <processes>

        <!-- Process Definitions -->
        <processDefinition id="localJVM_A">
          <jvmProcess
            class="org.objectweb.proactive.core.process.JVMNodeProcess">
            <classpath>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
            </classpath>
            <javaPath>
              <absolutePath
                value="/user/ffonteno/home/src/java/jdk1.6.0_11/jre/bin/java" />
            </javaPath>
          </jvmProcess>
        </processDefinition>

        <!-- Process Definitions -->
        <processDefinition id="localJVM_B">
          <jvmProcess
            class="org.objectweb.proactive.core.process.JVMNodeProcess">
            <classpath>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/ProActive.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/bouncycastle.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/fractal.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/trilead-ssh2.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/javassist.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/log4j.jar"/>
              <absolutePath value="${PROACTIVE_HOME}/dist/lib/xercesImpl.jar"/>
            </classpath>
            <javaPath>
              <absolutePath
                value="/user/ffonteno/home/src/java/jdk1.6.0_11/jre/bin/java" />
            </javaPath>
          </jvmProcess>
        </processDefinition>

      </processes>
    </infrastructure>
</ProActiveDescriptor>
```

2.6. The XML Security Descriptor in details

Inside the policy file, you can express policy between entities (domain, runtime, node, active object).

The entity tag can be used to:

- express policies on entities described inside the descriptor (lines 13, 15)
- express policies on existing entities by specifying theirs certificates (line 32).

2.6.1. Construction of an XML policy:

A policy file must begin with:

```
<Policy xmlns="urn:proactive:security:1.1" xmlns:schemaVersion="1.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="urn:proactive:security:1.1 http://www-sop.inria.fr/oasis/ProActive/
schemas/security/1.1/security.xsd">
```

Next, application specific information is given. <*ApplicationName*> sets the application name. This allows easy identification of which application an entity belongs to.

```
<ApplicationName>My Application</ApplicationName>
```

<*PKCS12KeyStore*> describes the path to the keystore created using the graphical interface. The path could be either absolute or relative. An absolute path contains the whole path (starting with / under Unix systems or a letter under Windows systems). A relative path describes the path to follow from the policy file in order to find the keystore file. For instance, setting just the name of the keystore file indicates that the file is located under the same directory as the one of the policy file.

```
<PKCS12KeyStore>ApplicationCertificate.p12</PKCS12KeyStore>
```

<*CertificationAuthority*> contains all trusted certificate authority certificates.

```
<CertificationAuthority>
   <Certificate>ca.cert</Certificate>
</CertificationAuthority>
```

Then we can define policy rules. All rules are located within the <*Rules*> tag. A <*Rule*> is constructed according to the following syntax:

- <*From*> tag contains all entities from which the interaction is made (source). It is possible to specify many entities in order to match a specific fine-grained policy.

  ```
  <From>
     <Entity type="VN" name="CN=VN_A"/>
  </From>
  ```

  <*Entity*> is used to define an entity. the 'type' parameter can be 'VN', 'certificate', or 'DefaultVirtualNode'.
  - 'VN' (Virtual Node) referrers to virtual nodes defined inside the deployment descriptor.
  - 'DefaultVirtualNode' is a special tag. This is taken as the default policy. The "name" attribute is ignored.
  - 'certificate' requires that the path to the certificate is set inside the 'name' parameters.
- <*To*> tag contains all entities onto which the interaction is made (targets). As with the <*From*> tag, many entities can be specified.

  ```
  <To>
     <Entity type="VN" name="CN=VN_B"/>
  </To>
  ```
- The <*Communication*> tag defines security policies to apply to requests and replies.

  <*Request*> sets the policy associated with a request. The *'value'* parameter can be:

- *'authorized'* means a request is authorized.
- *'denied'* means a request is denied.

Each attribute of the <*Attributes*> tag (*authentication*, *integrity*, *confidentiality*) can be set to *'required'*, *'optional'* or *'denied'*.

<*Reply*> tag has the same parameters as <*Request*>

```
<Communication>
  <Request value="authorized">
    <Attributes authentication="required" confidentiality="required" integrity="required"/>
  </Request>
  <Reply value="authorized">
    <Attributes authentication="required" confidentiality="required" integrity="required"/>
  </Reply>
</Communication>
```

- <*Migration*> controls migration between <*From*> and <*To*> entities. Values can be *'denied'* or *'authorized'*.

```
<Migration>authorized</Migration>
```

- <*OACreation*> controls creation of active objects by <*From*> entities onto <*To*> entities.

```
<OACreation>authorized</OACreation>
```

Here is a complete security policy:

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="urn:proactive:security:1.1" xmlns:schemaVersion="1.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="urn:proactive:security:1.1 http://www-sop.inria.fr/oasis/ProActive/
schemas/security/1.1/security.xsd">
  <ApplicationName>My Application</ApplicationName>
  <PKCS12KeyStore>ApplicationCertificate.p12</PKCS12KeyStore>

  <CertificationAuthority>
    <Certificate>ca.cert</Certificate>
  </CertificationAuthority>
  <Rules>
    <Rule>
      <From>
        <Entity type="VN" name="CN=VN_A"/>
      </From>
      <To>
        <Entity type="VN" name="CN=VN_B"/>
      </To>
      <Communication>
        <Request value="authorized">
          <Attributes authentication="required" confidentiality="required" integrity="required"/>
        </Request>
        <Reply value="authorized">
          <Attributes authentication="required" confidentiality="required" integrity="required"/>
        </Reply>
      </Communication>
      <Migration>authorized</Migration>
      <OACreation>authorized</OACreation>
    </Rule>
    <Rule>
```

```
        <From>
          <Entity type="Node" name="CN=VN_B"/>
        </From>
        <To>
          <Entity type="Node" name="CN=VN_A"/>
        </To>
        <Communication>
          <Request value="authorized">
            <Attributes authentication="required" confidentiality="required" integrity="required"/>
          </Request>
          <Reply value="authorized">
            <Attributes authentication="required" confidentiality="required" integrity="required"/>
          </Reply>
        </Communication>
        <Migration>authorized</Migration>
        <OACreation>authorized</OACreation>
      </Rule>
  </Rules>
  <AccessRights/>
</Policy>
```

Note that the JVM that reads the deployment descriptor should be started with a security policy. In order to start a secure JVM, you need to use the property proactive.runtime.security and give a path a security file descriptor.

Here is an example:

```
java -Dproactive.runtime.security=descriptors/security/jvmlocal.xml TestSecureDeployment secureDeployment.xml
```

Chapter 3. Message Tagging

3.1. Overview of Message Tagging

Message Tagging allows you to add tags on the ProActive messages such as requests and replies. Each tag has an identifier and a unique value, and they can take a data of object type. Theses tags have a method called at each propagation. This method is the only needed method in your implementation of the abstract class Tag.

3.2. API

To retrieve the tag list of a request or a reply being served in your application, you can use the following API:

- **PAMessageTagging.getCurrentTags()**: Return the current MessageTags object of the current request served.

From this MessageTags object, you have access to the following API to interact with tags:

- **addTag(Tag tag)**: appends a new Tag to the tag list of this message.
- **checkTag(String id)**: checks whether a tag with this **id** already exists on this message.
- **getAllTagsId()**: retrieves a **Set of String** with all the tag IDs on this message.
- **getData(String id)**: retrieves the **data object** of the tag whose ID is **id**.
- **getTag(String id)**: retrieves the tag instance of the tag whose ID is **id**.
- **removeTag(String id)**: removes the tag whose ID is **id** from the current tag list of this message.

Once you have retrieved the tag instance, you have access to the following methods:

- **getData()**: gets the data object of this tag.
- **getID()**: gets the identifier of this tag.
- **getLocalMemory()**: gets the LocalMemory object of this tag.
- **setData()**: sets a new data object for this tag.
- **apply()**: executes the job of the tag at the propagation.

3.3. Local Memory Space

All the tags have access to a **LocalMemory** object, which is a local memory space on the Active Object on which the request is currently being served. They can save or retrieve data in this space (key/value). This is useful when the tag needs information on a previous passage of this Active Object. This local memory space is created for a specified lease time. The **max lease time** is specified by the **proactive.tagmemory.lease.max** PAProperty which represents the max lease value time in second.

You can use this **local memory space** with the following API in your Tag implementation:

- **createLocalMemory(int lease)**: creates the local space for your tag on the current active object with this **lease** time value (in second), and returns the LocalMemory instance created.
- **getLocalMemory()**: gets a reference to the local memory of your tag.
- **clearLocalMemory()**: clears the local memory of your tag.

Once you have your LocalMemory instance, you have access to:

- **put(String key, Object value)**: to put a value into that space.

  get(String key): to retrieve a value previously putted.

Each time a value is accessed, the current lease value of the LocalMemory space is incremented by half of the initial value. A thread runs periodically (**proactive.tagmemory.lease.period**) to check the lease value of each LocalMemory, and decrements their current lease.

3.4. Example of a user Tag implementing the abstract class Tag

This section presents a tag example which is able to give information on the request depth. Thus, we can know for example how many requests have been served before serving the current one.

Here is the implementation of this tag:

```
public class IncrementingTag extends Tag {

   private Integer depth;

   public IncrementingTag(String id) {
      super(id);
      this.depth = 0;
   }

   @Override
   public Tag apply() {
      this.depth++;
      return this;
   }

   public Integer getDepth() {
      return this.depth;
   }

}
```

Each time this tag is propagated, that is, each time the apply method is called, the depth field is incremented.

In order to test this tag, we have written a loop between three active objects:

- **A** which has a reference to an active object **B**
- **B** which has a reference to an active object **C**
- **C** which has a reference to the first active object **A**

All this classes define a propagate method which have the same principle:

1. Gets the *"TAG_00"* tag or creates it if it does not exist (only for class A).
2. Displays the depth of this tag.
3. Gets the data of this tags where the current path has been stored and adds the current class.
4. Gets or creates the local memory and stores the current round number, that is, the number of has been visited.

Here is the propagate method of the class **A**:

```
public void propagate(Integer maxDepth) {

   MessageTags tags = PAMessageTagging.getCurrentTags();

   // Gets or creates the tag 'Tag_00'
   IncrementingTag t = (IncrementingTag) tags.getTag("TAG_00");
   if (t == null)
      t = (IncrementingTag) tags.addTag(new IncrementingTag("TAG_00"));

   // Displays the tag depth
```

```
    System.out.println("\n----------------------");
    System.out.println("A: Tag depth = " + t.getDepth());

    // Gets the tag data and adds the current class to the path
    String str = (String) t.getData();
    if (str == null)
        str = "A";
    else
        str += " -> A";
    t.setData(str);
    System.out.println("A: Path = " + t.getData());

    // Gets or creates the memory 'MEM_00'
    // and stores in it the round number.
    LocalMemoryTag mem = t.getLocalMemory();
    if (mem == null) {
        t.createLocalMemory(10).put("MEM_00", new Integer(0));
        mem = t.getLocalMemory();

    } else {
        Integer round = (Integer) mem.get("MEM_00");
        mem.put("MEM_00", ++round);
    }
    System.out.println("A: Round number = " + (Integer) mem.get("MEM_00"));
    System.out.println("----------------------");

    // Propagates to the active object 'B'
    if (t.getDepth() < maxDepth)
        activeB.propagate(maxDepth);
}
```

This example is available in the **org.objectweb.proactive.examples.documentation.messagetagging** package.

3.5. Distributed Services Flow Tag

The Distributed Services Flow Tag (DSF Tag) allows the following of all messages (requests and replies) so as to know to which services flow these messages belong to. This tag is necessary if you want to do a graphical analysis in IC2D of the execution of your application.

This tag can be enable or disable with the **proactive.tag.dsf** ProActive property. Set this property to **true** if you want to enable the DSF Tag propagation in the execution of the application. It is set to false by default.

Chapter 4. Exporting active objects as Web Services

4.1. Overview

This feature allows to expose an active object as a web service and thus, to call it from any client written in any foreign language.

Indeed, applications written in C#, for instance, cannot communicate with ProActive applications. We have chosen web services technology to enable interoperability because they are based on XML and HTTP. Thus, any active object can be accessible from any enabled web services language.

4.2. Principles

A **web service** is a software entity, providing one or several functionalities, that can be exposed, discovered and accessed over the network. Moreover, web services technology allows heterogeneous applications to communicate and exchange data in a remotely way. In our case, the useful elements of web services are:

- **The SOAP Message**

  The SOAP message is used to exchange XML based data over the internet. It can be sent via HTTP and provides a serialization format for communicating over a network.

- **The HTTP Server**

  HTTP is the standard web protocol generally used over the 80 port. In order to receive SOAP messages, you can either install an HTTP server that will be responsible of the data transfer or use the default HTTP server which is a Jetty server. However, This server is not sufficient to treat a SOAP request. For this, you also need a SOAP engine.

- **The SOAP Engine**

  A SOAP Engine is the mechanism responsible of making transparent the unmarshalling of the request and the marshalling of the response. Thus, the service developer doesn't have to worry with SOAP. In our case, we use Apache CXF which can be installed into any HTTP server.

- **The client**

  Client's role is to consume a web service. It is the producer of the SOAP message. The client developer does not have to worry about how the service is implemented. The CXF java library provides classes to easily invoke a service (see examples).

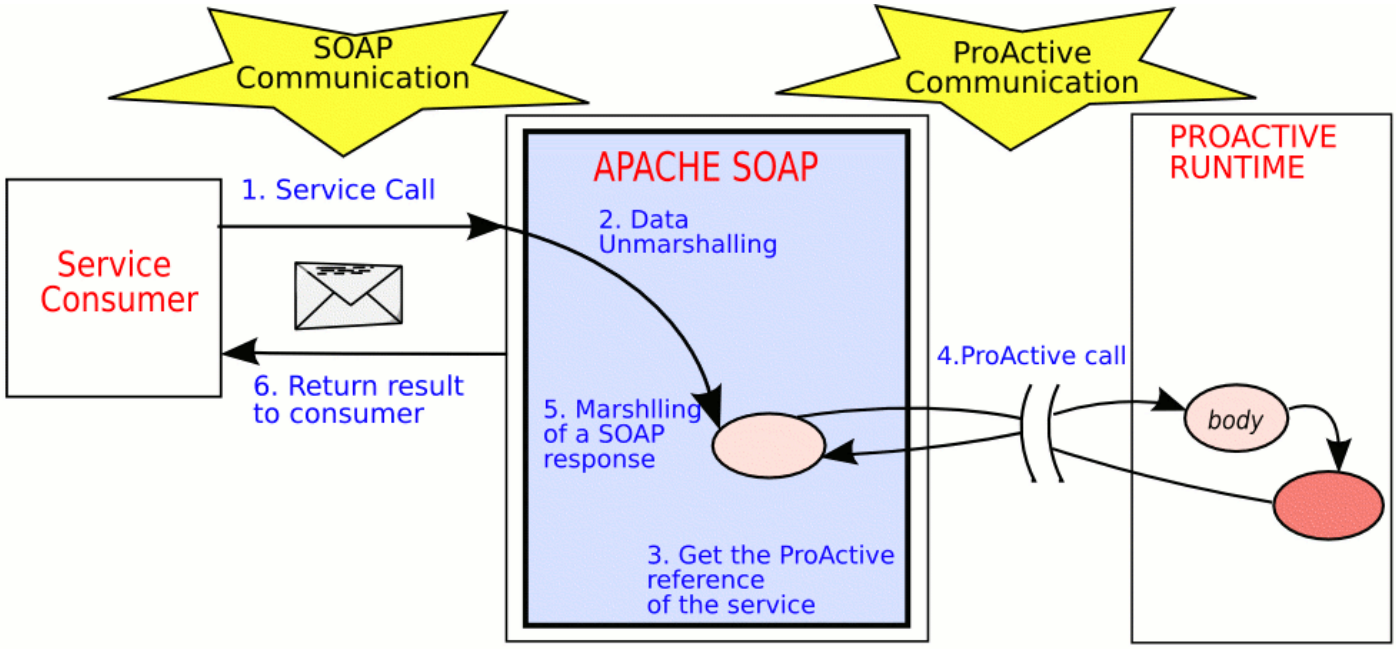


Figure 4.1. Steps taken when an active object is called via SOAP

4.3. Pre-requisite: Installing the Web Server and the SOAP engine

If you want to expose your active objects on the local embedded Jetty server or on Jetty servers deployed during your deployment, you just have to deploy proactive using the build deploy command into the compile directory. If you want to expose them on an other http server, you have to build the proactive.war archive. For this, you have to go into your compile directory of your ProActive home and type build proActiveWar. The proactive.war file will be built into the dist/ directory.

If you have chosen to use the default Jetty server, then you have nothing else to do. Jetty server will automatically take files it needs.

If you have chosen to use your own HTTP server, then you just have to copy the proactive.war file into the webapp/ directory of your HTTP server. Some HTTP servers need to be restarted to take into account this new web application but some others like Tomcat can handle hot deployment and, thus, do not need to be restarted.

Warning

If you use your own HTTP server, you have to be aware that the Jetty server will be launched. Most of ProActive examples are launched with the option '-Dproactive.http.port=8080' which specifies Jetty port. So, if you want to use your own server on port 8080, you have to modify the jetty port in launching scripts or to change the port of your own server.

4.4. Steps to expose or unexpose an active object as a web services

The steps for exporting and using an active object as a web service are the following:

- Write your active object in a classic way as below:

```
B b = (B) PAActiveObject.newActive(B.class.getName(), new Object[] {});
```

- Once the element created and activated, a **WebServicesFactory** object has to be instantiated:

```
WebServicesFactory wsf;
wsf = AbstractWebServicesFactory.getDefaultWebServicesFactory();
```

or

```
WebServicesFactory wsf;
wsf = AbstractWebServicesFactory.getWebServicesFactory("cxf");
```

- Then, you have to instantiate a **WebServices** object as follows:

```
WebServices ws = wsf.getWebServices(myUrl);
```

If you want to use the local Jetty server to expose your active object, you can use the following line to retrieve the good URL:

```
String myUrl = AbstractWebServicesFactory.getLocalUrl();
```

This will get the Jetty port which is a random port and build the url.

- And finally, you can used one of the WebServices methods:

```
/**
 * Expose an active object as a web service with the methods specified in <code>methods</code>
 *
 * @param o The object to expose as a web service
 * @param urn The name of the service
 * @param methods The methods that will be exposed as web services functionalities
 *                If null, then all methods will be exposed
 * @throws WebServicesException
 */
public void exposeAsWebService(Object o, String urn, String[] methods) throws WebServicesException;

/**
 * Expose an active object as a web service with the methods specified in <code>methods</code>
 *
 * @param o The object to expose as a web service
 * @param urn The name of the service
 * @param methods The methods that will be exposed as web services functionalities
 *                If null, then all methods will be exposed
 * @throws WebServicesException
 */
public void exposeAsWebService(Object o, String urn, Method[] methods) throws WebServicesException;

/**
 * Expose an active object with all its methods as a web service
 *
 * @param o The object to expose as a web service
 * @param urn The name of the service
 * @throws WebServicesException
 */
public void exposeAsWebService(Object o, String urn) throws WebServicesException;

/**
 * Undeploy a service
 *
 * @param urn The name of the service
 * @throws WebServicesException
 */
public void unExposeAsWebService(String urn) throws WebServicesException;
```

where:

- **o** is the active object
- **urn** is the service name which will identify the active object on the server.
- **methods** is a String or a Method array containing the methods you want to make accessible. If this parameter is null (or is absent), all the public methods will be exposed.

4.5. Exposing as a web service on remote Jetty servers launched during the deployment

When you call the exposition of your active object, the CXF servlet is deployed on the local Jetty server. It is therefore possible to expose your object locally without doing anything else.

Now, let us assume that your active object is deployed on a remote node. In that case, you have a running jetty server on the remote host containing this node but no servlet has been deployed on it. So, if you want to expose your active object on the remote host, you will have to make a small modification of the previous piece of code.

The web service extension provides classes that implements the InitActive interface which are in charge of deploying the CXF servlet on the host where your object has been initialized. Thus, you just have to instantiate your active object using this implementation of InitActive.

```
// Get the CXF InitActive in charge of deploying the CXF Servlet
InitActive cxfInitActive = WebServicesInitActiveFactory.getInitActive("cxf");
HelloWorld helloWorld = (HelloWorld) PAActiveObject.newActive(HelloWorld.class.getName(), null,
      new Object[] {}, myNode, cxfInitActive, null);
```

Now, let us assume that your active object is deployed on a node or locally and that you want to expose it on a host where another node is deployed. On this host, there is only a running Jetty server without any web services servlet. In order to deploy the CXF servlet on it, you can use the following code:

```
WebServicesInitActiveFactory.getInitActive("cxf").initServlet(myNode);
```

Then, you can expose your active object as a web service as in Section 4.4, “Steps to expose or unexpose an active object as a web services”.

4.6. Accessing the services

Once the active object deployed, you can access it via any web service client (such as C#, SoapUI, Apache CXF API, Apache Axis2 API, ...) or with your favorite browser (only if the returned type of your service methods are primitive types).

First of all, the client will get the WSDL file matching this active object. This WSDL file is the 'identity card' of the service. It contains the web service public interfaces and its location. Generally, WSDL files are used to generate a proxy to the service. For example, for a given service, say 'compute', you can get the WSDL document at http://localhost:8080/proactive/services/compute?wsdl.

Now that this client knows what and where to call the service, it will send a SOAP message to the web server. The web server looks into the message and performs the right call. Then, it returns the response into another SOAP message to the client.

4.7. Limitations

Contrary to the previous version of the active object exposition using Apache SOAP, this new version which uses CXF can handle complex types. That is, with this version, you can implement a service which exposes a method returning or taking in argument an instance of any class. By using CXF and JAX-WS annotations, there is no limitation on the type of this class. It only implies to use the appropriate adapters (see the CXF documentation [1] for more details). However, if you do not want to use JAX-WS annotations or CXF, the class of this instance has to respect some criteria due to the serialization and CXF/Axis2 restrictions:

[1] http://cxf.apache.org/docs/index.html

- This class has to be serializable
- This class has to supply a no-args constructor and preferably empty
- All the fields of this class has to be private or protected
- This class has to supply public getters and setters for each field. These getter and setter methods have to be in this class and not in one of its super classes.

4.8. A simple example: Hello World

4.8.1. Hello World web service code

Let's start with a simple example, an Hello world active object exposed as a web service:

```
HelloWorld hw = (HelloWorld) PAActiveObject
        .newActive(HelloWorld.class.getName(), new Object[] {});

// As only one framework is supported for the moment,
// you can use either one of the two following methods
WebServicesFactory wsf;
wsf = AbstractWebServicesFactory.getDefaultWebServicesFactory();
wsf = AbstractWebServicesFactory.getWebServicesFactory("cxf");

// If you want to use the local Jetty server, you can use
// AbstractWebServicesFactory.getLocalUrl() to get its url with
// its port number (which is random except if you have set the
// proactive.http.port variable)
WebServices ws = wsf.getWebServices("http://localhost:8080/");

ws.exposeAsWebService(hw, "MyHelloWorldService", new String[] { "putTextToSayAndConfirm",
        "putTextToSay", "sayText" });
```

The active object hw has been deployed as a web service on the web server located at http://localhost:8080/proactive/services/ and its service name is "MyHelloWorldService". The accessible service method are putTextToSayAndConfirm, putTextToSay and sayText. You can see its wsdl file through the http://localhost:8080/proactive/services/MyHelloWorldService?wsdl and you call the helloWorld method with your browser through, for example, http://localhost:8080/proactive/services/MyHelloWorldService/sayText.

If you want to expose a method which requires arguments, you can call using the same link as previously but adding ?[arg0]=[value]&[arg1]=[value]... where [argn] is the name of the n-th argument as written in the wsdl file.

You can also call a webservice using the ProActive client which uses a CXF client. Here is an example:

```
// Instead of using "cxf", you can also use webServicesFactory.getFrameWorkId().
// As only one framework is supported for the moment, use the following method
// is equivalent to ClientFactory.getDefaultClientFactory()
ClientFactory clientFactory = AbstractClientFactory.getClientFactory("cxf");

// Instead of using "http://localhost:8080/", you can use ws.getUrl() to
// ensure to get to good service address.
Client client = clientFactory.getClient("http://localhost:8080/", "MyHelloWorldService",
        HelloWorld.class);

// Call which returns a result
Object[] res = client.call("putTextToSayAndConfirm", new Object[] { "Hello World!" },
        String.class);
```

```
System.out.println((String) res[0]);

// Call which does not return a result
client.oneWayCall("putTextToSay", new Object[] { "Hi ProActive Team!" });

// Call with no argument
res = client.call("sayText", null, String.class);

System.out.println((String) res[0]);

// Call with no argument
res = client.call("sayText", null, String.class);

System.out.println((String) res[0]);
```

Chapter 5. Making ProActive programming easier: the ProActive Java Annotations System

5.1. Compile-time annotations

Compile-time annotations provide a way to check at compile-time the constraints imposed on Java code by the ProActive library. Usually, these constraints are documented in the ProActive manual, and it is left to the responsibility of the programmer to obey these rules. If the rules are broken, the effects are usually seen at runtime, translated into specific runtime exceptions, which are difficult to interpret by the inexperienced ProActive programmers. The compile-time annotation system makes it possible to verify these rules at compile-time, in order to avoid long debugging sessions made out of stack traces eyeballing. Ideally, the runtime errors that the programmer will receive will be more related to the logic of the application, and less to the misuse of the ProActive library.

5.1.1. General usage

First of all, note that the annotation system is **non-intrusive**. You can easily program using the ProActive library without bothering about annotations. They only serve as helpers for developers, as mentioned previously.

Secondly, the compile-time annotations are implemented using tools provided in the standard JDK distribution, starting from version 1.5. The following backward-compatibility goal was kept in mind while developing the annotation system: the code written by the ProActive library user should compile using JDK 1.5. But if you want the "full" power of the annotation system, it is advised to use JDK 1.6 or more. If you (still) develop using JDK 1.5, some of the checks will not be performed. This documentation specifies which checks are disabled for 1.5.

The compile-time checks are integrated into the compilation process. This means that the checks will be performed when you will compile your code, if you specify several command-line switches for the compilation command. Depending on the JDK version you are using, different compilation commands must be used. If you use JDK 1.5, you must compile using apt[1]. If you use JDK 1.6, you can simply use **javac**, with the appropriate command-line switches. These will be described in the following paragraphs.

The general way to use annotations is the following. For every annotation @Annotation provided by the ProActive library:

- Annotate the applicable programming language constructed with the @Annotation
- When compiling, use one of the following compilation commands
  - For JDK 1.5:

    ```
    $apt -cp $CLASSPATH:$JAVA_HOME/lib/tools.jar:$PROACTIVE_HOME/dist/lib/ProActive.jar [input_files]
    ```
  - For JDK 1.6:

    ```
    $javac -processor org.objectweb.proactive.extensions.annotation.common.ProActiveProcessorCTree -cp
     $CLASSPATH:$JAVA_HOME/lib/tools.jar:$PROACTIVE_HOME/dist/lib/ProActive.jar [input_files]
    ```

  The above snippets assume that the ProActive distribution library and its dependencies are in the classpath, and that $JAVA_HOME environment variable points to your JDK distribution.
- Correct the compile-time errors.

You can also use the annotation processing inside IDEs. Following is a description on how you can use it with the main IDEs:

- **Eclipse** To configure Eclipse, the following steps must be undertaken:
  1. Right-click on the project you are working on, and select Properties to bring on the Project Properties property page
  2. Go to Java Compiler -> Annotation Processing

[1] http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html

3. Check the 'Enable project specific settings', 'Enable annotation processing' and 'Enable processing in editor' checkboxes
4. Go to Factory Path under Annotation Processing
5. Select 'Add External JARs' and navigate to the location of ProActive.jar
6. Make sure the checkbox next to the ProActive jar is checked
7. Click on apply. You will be prompted with a dialog box that will ask you whether you want to do a full project rebuild. If this is what you want, click Yes, otherwise just hit No.
8. Click on the OK button.

Then, you can use the annotations, and when starting a compilation inside Eclipse, the violations of the Active Object constraints will appear as error markers inside the IDE. Note that this will only verify the constraints checked using apt - this is the position of the Eclipse development team[2].

- **NetBeans** cannot be configured to use APT proactive annotation processor without manual editing of project's build.xml file.

  In order to setup annotation processing using javac (applicable for JDK 6) go to

  Project Properties -> Compiling -> Additional compiler options and add the following option

  ```
  -processor org.objectweb.proactive.extensions.annotation.common.ProActiveProcessorCTree
  ```

  Then build the project.

- **IntelliJ IDEA**

  To setup javac's annotation processor first make sure that javac compiler is chosen in Project Settings -> Compiler.

  Then add the following option to Additional command line parameters

  ```
  -processor org.objectweb.proactive.extensions.annotation.common.ProActiveProcessorCTree
  ```

5.1.2. @ActiveObject

The @ActiveObject annotation is applicable on a Java class definition, and it tests whether the class can be the type of an active object or not. As you can see in the Active Object section of the manual, there are some constraints that a class definition should satisfy in order to be used as an active object type:

- non private no-argument constructor should be explicitly defined
- the class must not be final
- the class must be public
- the class must not have final non-private methods
- a public field without setter and getter is not recommended
- an Active Object method should not return null, as this literal cannot be checked on the caller side

If you use JDK 1.5, the last point will not be checked.

5.1.3. @RemoteObject

The @RemoteObject annotation is applicable on a Java class definition and it verifies limitations of remote objects. The list of limitations is exactly the same as for active objects (see above) with only one exception - remote object methods can return null.

5.1.4. @MigrationSignal

@MigrationSignal is applied on a method definition in order to mark the fact that the annotated method should be used as a migration signal. What is a migration signal? The most common way to implement migration using ProActive is for the active object to export a public method inside which to call **PAMobileAgent.migrateTo()**, and this call should be the last statement inside the method body.

[2] http://dev.eclipse.org/newslists/news.eclipse.tools.jdt/msg22872.html

This kind of method will be called henceforth a *migration signal*. The MigrationSignal annotation enforces the migration guidelines specified in the ProActive manual, section on mobility. At compilation, the annotated method is checked to see if migrateTo() is called, and if this is indeed the last statement. Migration signals can be also defined by calling other methods that perform migration - and not directly through calling migrateTo().

5.1.5. @Migratable

@Migratable is applied on an object in order to specify that it will be used as a migratable object. This means that the object has to be an active object - it must be annotated using the @ActiveObject annotation - and also that it has to implement the java.io.Serializable interface.

5.1.6. Migration Strategy

When implementing mobile agents using ProActive, one can specify methods that should be executed when arriving to or departing from a site. The signature of these methods must be:

```
void methodName()
```

namely, the method must be parameterless and with no return value. For further details you can refer to this paper[3]. This constraint is checked using two annotations: **@OnDeparture** and **@OnArrival**. Both of them are applicable on method definitions:

- **@OnDeparture** marks the method that should be executed when the mobile agents depart from a site
- **@OnArrival** marks the method that should be executed when the mobile agents arrive to a site

At compilation, the methods are checked to match the above-mentioned signature.

5.1.7. @NodeAttchedCallback

When a client subscribes to Node attachment notification, the method passed as parameter is invoked each time a Node is attached to the GCMVirtualNode. The method must have the following signature:

```
void method(Node, String)
```

This constraint is checked by using @NodeAttchedCallback annotation with a method definition. For more information about node attachment notification, please refer to org.objectweb.proactive.gcmdeployment.GCMVirtualNode.subscribeNodeAttachment(...)[4]

5.1.8. @VirtualNodesReadyCallback

When a client subscribes to isReady notification, the method passed as parameter is invoked when the Virtual Node becomes Ready. The method must have the following signature:

```
void method(String)
```

This constraint is checked by using @VirtualNodesReadyCallback annotation with a method definition. For more information about isReady notification, please refer to org.objectweb.proactive.gcmdeployment.GCMVirtualNode.subscribeIsReady(...)[5]

5.1.9. @ImmediateService

When a method is tagged with the @ImmediateService annotation, all method calls on this particular method are treated as an immediate service. The legacy way to make a method to be served as an immediate service was by calling the PAActiveObject.setImmediateService(...) method.

[3] http://proactive.inria.fr/userfiles/file/papers/Mobility.pdf
[4] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/AdvancedFeatures/pdf/../../ReferenceManual/api_complete/org/objectweb/proactive/gcmdeployment/GCMVirtualNode.html#subscribeNodeAttachment(java.lang.Object,%20java.lang.String,%20boolean)
[5] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/AdvancedFeatures/pdf/../../ReferenceManual/api_complete/org/objectweb/proactive/gcmdeployment/GCMVirtualNode.html#subscribeIsReady(java.lang.Object,%20java.lang.String)

Chapter 6. ProActive on top of OSGi

6.1. Overview of OSGi — Open Services Gateway initiative

OSGi is a corporation that works on the definition and promotion of open specifications. These specifications mainly aim at packaging and delivering services among all kinds of networks.

OSGi Framework

The OSGi specification defines **a framework** that allows a diversity of services to be executed in a service gateway. Thus, many applications can **share a single JVM**.

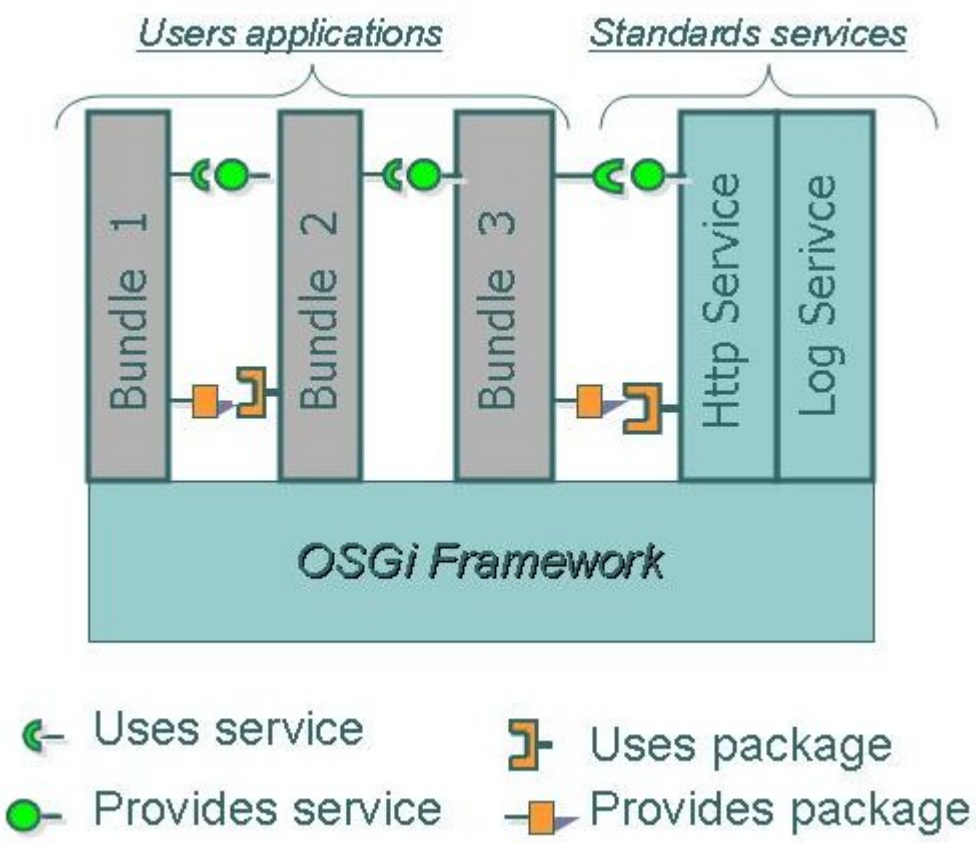


Figure 6.1. The OSGi framework entities

Bundles

In order to be **delivered and deployed** on OSGi, each piece of code is packaged into bundles. Bundles are functional entities offering **services and packages**. They can be delivered dynamically to the framework. Concretely, a bundle is a Java jar file containing:

- The application classes, including the so-called bundle **Activator**.
- The **Manifest file** that specifies properties about the application. For instance, this **Manifest file** specifies the bundle Activator and packages which are required by the application.

- Other resources the application could need (images, native libraries, or data files...).

Bundles can be plugged dynamically and their so-called **lifecycle** can be managed through the framework (start, stop, update...).

Manifest file

This important file contains crucial parameters for the bundle. It specifies which Activator (entry point) the bundle has to use, the bundle classpath, the imported and exported packages and so on...

Services

Bundles communicates with each other thanks to **services and packages sharing**. A **service** is an object registered into the framework in order to be used by other applications. The definition of a service is specified in a Java interface. OSGi specifies a set of standard services like Http Service, Log Service...

We currently use the Apache Felix[1] implementation. For more information on OSGi, please visit the OSGi[2] website.

6.2. ProActive bundle and service

In order to use ProActive on top of OSGi, we have developed the **ProActive Bundle** that contains all classes required to launch a ProActive runtime.

The **ProActive bundle offers a service**, the **ProActiveService** that has almost the same interface that the ProActive static class. When installed, the ProActive bundle starts a new runtime, and clients that use the ProActive Service will be able to create active object on this runtime.

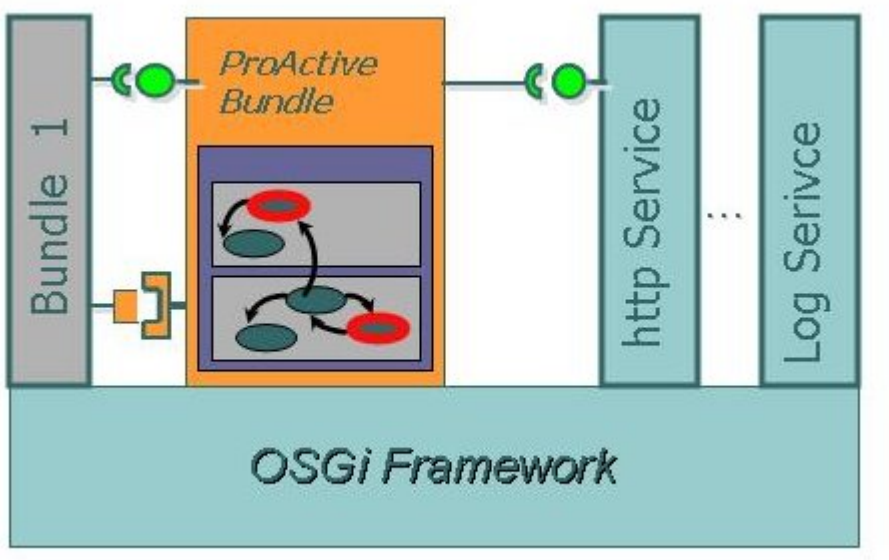


Figure 6.2. The Proactive Bundle uses the standard Http Service

The active object will be accessible remotely from any java application, or any other OSGi gateway. The communication can be either **rmi** or **http**. In case of using http, the ProActive bundle requires the installation of the **Http Service** that will handle http communications through a Servlet. We show in the example section how to use the ProActive service.

[1] http://felix.apache.org/site/index.html
[2] http://www.osgi.org

6.3. Yet another Hello World

The example below is **a simple hello world that uses ProActive Service**. It creates an Hello active Object and registers it as a service. We reuse the basic hello example provided within the ProActive's examples. We have to write a **bundle Activator** that will create the active object and register it as a OSGi service.

The HelloActivator has to implements the BundleActivator interface.

```
public class HelloActivator implements BundleActivator {
}
```

The start() method is the one executed when the bundle starts. When the hello bundle starts, we need to get the reference to the ProActive service and use it. Once we have the reference, we can create our active objects thanks to the ProActiveService.newActive() method. In this example, we use this method to create two active objects: One HelloSystemOut object and one HelloLogInfo object. This two classes implement the HelloService class which defines two methods: sayHello() and saySomething(String something). Finally, we register these two objects as a HelloService with two different out properties that will enable us to differentiate them.

```
public class HelloActivator implements BundleActivator {

  private BundleContext context;
  private ProActiveService proActiveService;

  /*
   * (non-Javadoc)
   * @see org.osgi.framework.BundleActivator#start(org.osgi.framework.BundleContext)
   */
  public void start(BundleContext context) throws Exception {
    this.context = context;

    /* gets the ProActive Service */
    ServiceReference sr = this.context.getServiceReference(ProActiveService.class.getName());

    if (sr == null) {
      System.out.println("Couldn't start this bundle: The service '" +
        ProActiveService.class.getName() + "' is not started");
      return;
    }

    this.proActiveService = (ProActiveService) this.context.getService(sr);
    HelloService h = (HelloService) this.proActiveService.newActive(HelloSystemOut.class.getName(),
        new Object[] {});

    /* Register the service */
    Properties props = new Properties();
    props.put("name", "HelloSystemOut");
    props.put("out", "system");

    this.context.registerService(HelloService.class.getName(), h, props);

    h = (HelloService) this.proActiveService.newActive(HelloLogInfo.class.getName(), new Object[] {});

    /* Register the service */
    props = new Properties();
    props.put("name", "HelloLogInfo");
```

```
        props.put("out", "logger");

        this.context.registerService(HelloService.class.getName(), h, props);
    }

    /*
     * (non-Javadoc)
     * @see org.osgi.framework.BundleActivator#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
    }

}
```

Now that we have created the hello active service, we have to **package the application** into a bundle. First of all, we have to write a **Manifest File** where we specify:

- The name of the bundle: Hello World ProActive Service
- The symbolic name of the bundle (required by felix): Hello World ProActive Service
- The class of the Activator: org.objectweb.proactive.examples.osgi.hello.HelloActivator
- The packages our application requires: org.objectweb.proactive...
- The packages our application exports: org.objectweb.proactive.examples.osgi.hello
- We can also specify other information like author, version, description, ...

Here is how the Manifest looks like:

```
Bundle-Name: ProActive Hello World Bundle
Bundle-SymbolicName: ProActive Hello World Bundle
Bundle-Description: Bundle containing Hello World ProActive example
Bundle-Vendor: OASIS - INRIA Sophia Antipolis
Bundle-version: 1.0.0
Export-Package: org.objectweb.proactive.examples.osgi.hello
DynamicImport-Package: *
Bundle-Activator: org.objectweb.proactive.examples.osgi.hello.HelloActivator
Require-Bundle: org.objectweb.proactive.extensions.osgi
```

Installing the ProActive Bundle and the Hello Bundle.

In order to run the example, you need to install an OSGi framework. Proactive uses felix as an OSGi framework but you can choose another one.

- **Generation of the ProActive Bundle as well as other required bundles**

  The proActiveBundle is the jar archive created when deploying ProActive. Yet, this bundle requires the librariesBundle which encompasses packages needed by ProActive. To generate these bundles, you have to go to your $PROACTIVE_HOME/compile and type:

  ```
  ./build deploy librariesBundle
  ```

  or, if you want to directly create all the ProActive bundles that you may need latter on, type:

  ```
  ./build OSGiBundles
  ```

  The ProActive bundle jar file will be generated in **$PROACTIVE_HOME/dist/lib/ProActive.jar** and all the other bundles will be generated in the **$PROACTIVE_HOME/dist/bundle/** directory.

In order to start the Felix framework with the ProActive and the libraries bundles started, the simplest way is to go to the **$PROACTIVE_HOME/examples/osgi** directory and launch the felix.[sh|bat]script. This script will launch felix with the appropriate arguments and it will automatically start the ProActive and the libraries bundles. It will also try to start the ProActive connector bundle as well as the JMX remote management bundle (see below for more information on these bundle). You may have some errors if you have not used the second build target (./build OSGiBundles). This is normal since these two bundles have not been built but it should not pose a problem for the following.

```
felix.[sh|bat]
```

- **Generation of the Hello World example bundle and its client bundle**

  To generate the Hello World bundle as well as its client bundle, you have to go to your $PROACTIVE_HOME/compile and type:

  ```
  ./build OSGiBundles
  ```

  The bundle jar files will be generated in the **$PROACTIVE_DIR/dist/bundle/** directory.
- **Installation and starting of the Hello World example bundle**

  We need now to install and start the hello bundle into the OSGi Framework:

  ```
  --> start file:./../../dist/bundle/helloBundle.jar
  ```

  The command will install and start the Hello active service. The hello service is now an OSGi service and can be accessed remotely.
- **Installation and starting of the Hello World Client example bundle**

  The helloClientBundle shows you how to use the services registered by the helloBundle. Here is its Activator and its manifest file:

```
public class HelloClientActivator implements BundleActivator {

  public void start(BundleContext arg0) throws Exception {

    String filter = "(out=system)";
    ServiceReference[] sr = arg0.getServiceReferences(HelloService.class.getName(), filter);

    if (sr == null) {
      System.out.println("Couldn't start this bundle: The service '" + HelloService.class.getName() +
        "' [filter = " + filter + "]is not started");
      return;
    }

    HelloService h = (HelloService) arg0.getService(sr[0]);

    h.sayHello();
    h.saySomething("Hello ProActive Team!");

    filter = "(out=logger)";
    sr = arg0.getServiceReferences(HelloService.class.getName(), filter);

    if (sr == null) {
      System.out.println("Couldn't start this bundle: The service '" + HelloService.class.getName() +
        "' [filter = " + filter + "]is not started");
      return;
    }

    h = (HelloService) arg0.getService(sr[0]);
```

```
        h.sayHello();
        h.saySomething("Hello ProActive Team!");

    }

    public void stop(BundleContext arg0) throws Exception {
        // TODO Auto-generated method stub
    }

}
```

```
Bundle-Name: ProActive Hello World Bundle
Bundle-SymbolicName: ProActive Hello World Bundle
Bundle-Description: Bundle containing Hello World ProActive example
Bundle-Vendor: OASIS - INRIA Sophia Antipolis
Bundle-version: 1.0.0
Export-Package: org.objectweb.proactive.examples.osgi.hello
DynamicImport-Package: *
Bundle-Activator: org.objectweb.proactive.examples.osgi.hello.HelloActivator
Require-Bundle: org.objectweb.proactive.extensions.osgi
```

You can install it and start it into your framework typing:

```
--> start file:./../../dist/bundle/helloClientBundle.jar
```

6.4. Current and Future works

- We are working on a management application that remotely monitors and manages a large number of OSGi gateways. It uses standard Management API such as JMX (Java Management eXtension). We are writing a ProActive Connector in order to access these JMX enabled gateways and uses Group Communications to handle large scale. Moreover, this application will be graphically written as an Eclipse plugin.
- We plan to deploy remotely active objects and fractal components on OSGi gateways.

Chapter 7. An extended ProActive JMX Connector

7.1. Overview of JMX — Java Management eXtension

JMX [1] (Java Management eXtensions) is a Java technology providing tools and APIs for managing and monitoring Java applications and their resources. Resources are represented by objects called Managed Beans (MBeans).

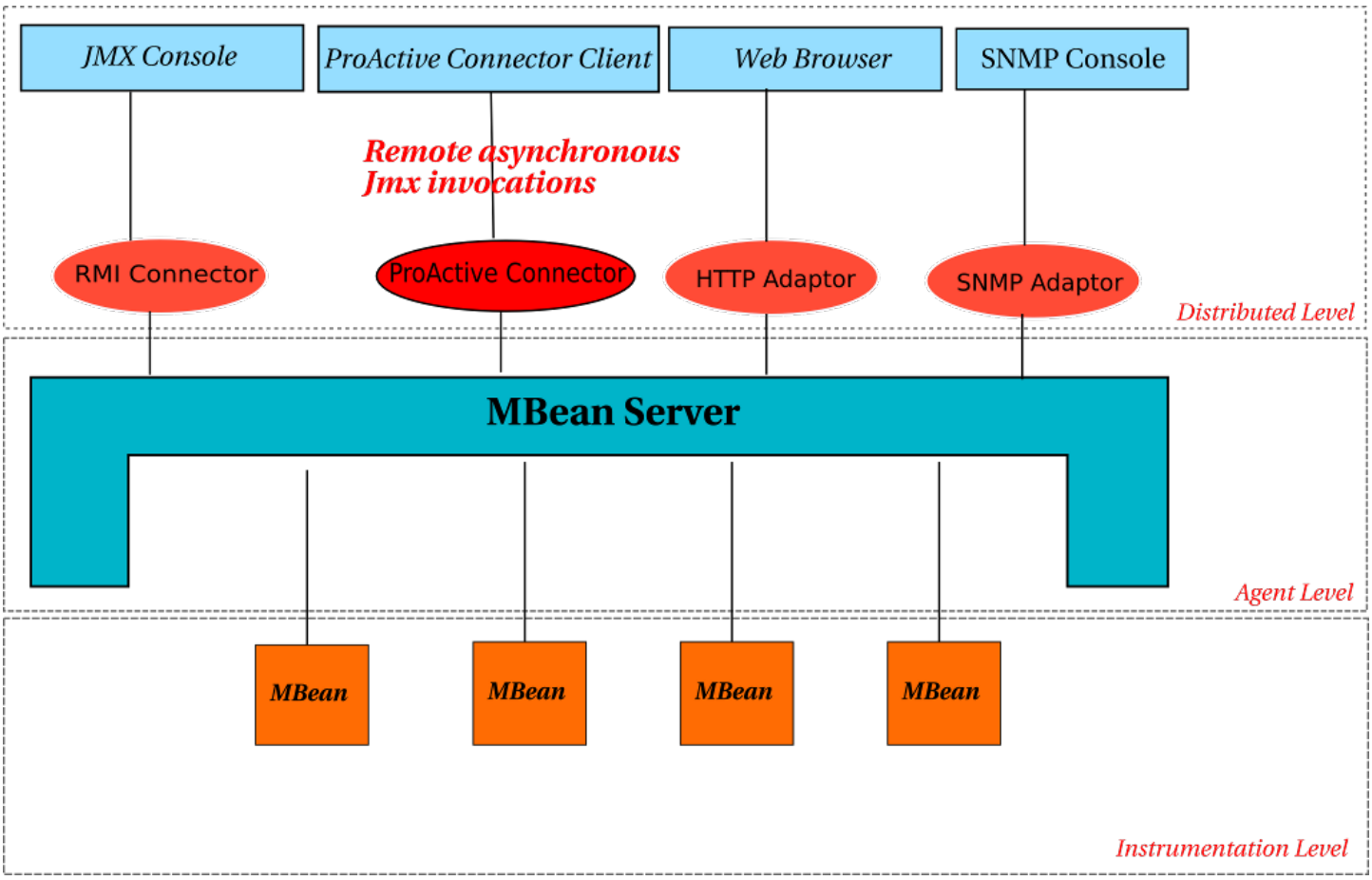


Figure 7.1. This figure shows the JMX 3-level architecture and the integration of the ProActive JMX Connector.

JMX defines a 3-layer management architecture:

- **The Instrumentation Level** contains MBeans and their manageable resources. A Mbean is a Java Object implementing a specific interface and pattern. They contain and define the manageable attributes, the management operations that can be performed onto resources and the notifications that can be emitted by the resources.
- **The Agent Level** acts as a MBeans containers (the MBeanServer) and controls them. This level represents the main part in the JMX specification since it gives access to the managed resources from the clients. The agent level is the central level of the architecture.
- **The Distributed Level** enables the remote management of Java applications. In order to remotely access to managed applications, JMX specification defines two types of remote access: protocol adaptors and protocol connectors. Connectors allow a manager

[1] http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

to perform method calls onto a distant agent's MBeanServer (for example RMI). Adaptors are components that ensure binding between a specific protocol (for example for SNMP or HTTP) and the managed resources. Indeed, they enable Mbeans to be accessed by existing approaches.

7.2. Asynchronous ProActive JMX connector

The JMX technology defines a connector based on RMI. The RMI connector allows the manager to connect to an MBean in a MBeanServer from a remote location and performs operations on it.

We defined a ProActive Connector according to the JMX Remote API JSR 160 [2] that enables asynchronous remote access to a JMX Agent thanks to ProActive. This connector is based on a call via an active object. When invoking the standard API specification methods, the access to the managed application is synchronous, because the JMX remote API provides non-reifiable methods. For example, the invoke() method, allowing to invoke a Mbean's method, throws exceptions:

```
public Object invoke(ObjectName name, String operationName, Object[] params, String[] signature)
  throws InstanceNotFoundException, MBeanException, ReflectionException, IOException;
```

We have extended the API in order to provide asynchronous access thanks to additionnal reifiable methods. The additional invoke() method looks like as follows:

```
public GenericTypeWrapper invokeAsynchronous(ObjectName name, String operationName, Object[] params, String[]
 signature)
```

Thus, all requests sent to the MBean are put in the active object requests queue and a future object is returned to the client.

7.3. How to use the connector?

The ProActive connector allows you to connect an MBean in an MBean server to a remote location, and perform operations on it, exactly as if the operations were being performed locally.

To perform such a call, you have first to create the server connector on the application you wish to manage. This is simply done by adding one line of code in the application:

```
ServerConnector serverConnector = new ServerConnector("MyServerName");
serverConnector.start();
```

Once the connector server part launched, any ProActive JMX connector client can connect to it and manage the application thanks to the ClientConnector class.

```
ClientConnector cc = new ClientConnector("//localhost/", "MyServerName");
```

To perform remote operations on a given MBean, you have to get the reference of the current MBeanServerConnection, which is actually a ProActiveConnection:

```
// Connects the client connector
cc.connect();

// Retrieves the ProActive connection
ProActiveConnection pc = cc.getConnection();

// Creates an ObjectName refering to an MBean
ObjectName beanName = new ObjectName("SimpleAgent:name=hellothere");

// Asynchronously invoke the "concat" method of this mbean with two Strings as parameters
```

[2] http://jcp.org/en/jsr/detail?id=160

```
GenericTypeWrapper<?> returnedObject = pc.invokeAsynchronous(beanName, "concat", new Object[] {
      "FirstString ", "| SecondString" }, new String[] { String.class.getName(),
      String.class.getName() });
```

Note

To know all available methods on a MBeanServerConnection, have a look at the MBeanServerConnection javadoc[3].

7.4. JMX Notifications through ProActive

The JMX specification defines a notification mechanism, based on java events and allowing to alert client management applications. To use JMX notifications, one has to use a listener object that is registered within the MBean server. On the server side, the MBean has to implement the NotificationBroadcaster interface. As we work in a distributed environment, listeners are located remotely and thus, have to be joined remotely. Hence, the **listener must be a serializable active object** and added as a NotificationListener:

```
/* Creates an active listener MyListener */
MyListener listener = (MyListener) PAActiveObject.newActive(MyListener.class.getName(), null);

/* Adds the listener to the Mbean server where we are connected to */
pc.addNotificationListener(beanName, listener, null, null);
```

Note

More informations on JMX on: Getting Started with Java Management Extensions (JMX): Developing Management and Monitoring Solutions [4]

7.5. Example: a simple textual JMX Console

The example available in the $PROACTIVE_HOME/examples/jmx/ directory, is a simple textual tool to connect to a remote MBeanServer and list available domains and mbeans registered in this server.

To launch the connector server side, execute the connector.[sh|bat] script. To connect this server, execute the simpleJmx.[sh|bat] script and specify the MBean serber name where is hosted the Mbean server:

```
--- JMC Test client connector-----------------------------------------------
Enter the name of the JMX MBean Server :  [default is '//localhost/serverName']
(Type "exit" to quit)
```

Just typing Enter should work since the server location of the mbean server launched with the connector.[sh|bat] script is the default one.

The console shows the domains list:

```
Registered Domains :
 [ 0 ] JMImplementation
 [ 1 ] com.sun.management
 [ 2 ] org.objectweb.proactive
 [ 3 ] org.objectweb.proactive.core.runtimes
 [ 4 ] org.objectweb.proactive.core.node
 [ 5 ] java.lang
 [ 6 ] java.util.logging
```

[3] http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServerConnection.html
[4] http://java.sun.com/developer/technicalArticles/J2SE/jmx.html

```
Type the domain number to see all registered MBeans in this domain :
```

By choosing a specific domain, the console will show the Mbeans registered in this domain:

Choosing the java.lang domain, you should see the following registered MBeans:

```
[ 0 ] java.lang:type=OperatingSystem
[ 1 ] java.lang:type=Compilation
[ 2 ] java.lang:type=MemoryPool,name=PS Old Gen
[ 3 ] java.lang:type=Memory
[ 4 ] java.lang:type=MemoryPool,name=PS Perm Gen
[ 5 ] java.lang:type=Runtime
[ 6 ] java.lang:type=GarbageCollector,name=PS MarkSweep
[ 7 ] java.lang:type=Threading
[ 8 ] java.lang:type=GarbageCollector,name=PS Scavenge
[ 9 ] java.lang:type=ClassLoading
[ 10 ] java.lang:type=MemoryPool,name=PS Survivor Space
[ 11 ] java.lang:type=MemoryManager,name=CodeCacheManager
[ 12 ] java.lang:type=MemoryPool,name=Code Cache
[ 13 ] java.lang:type=MemoryPool,name=PS Eden Space
[ D ]  Domains list

Type the mbean number to see its properties :
```

If you wish to get information about Memory, choose 3, and the console will show the whole information about this MBean.

Chapter 8. Existing MBean and JMX notifications in ProActive

8.1. Principles

In order to be able to monitor and controle ProActive application, each ProActive object has an MBean:

- ProActiveRuntimeImpl has a ProActiveRuntimeWrapperMBean
- NocalNode has a NodeWrapperMBean
- AbstractBody has a BodyWrapperMBean

Thus, through these beans, it is possible to receive the JMX notification emitted by the MBean, or to invoke a method on the MBean.

8.2. How to subscribe/unsubscribe to the notifications of a MBean?

Utility classes are provided in order to make the subscription of JMX notifications easy. If you subscribe to a MBean of a remote MBean server, this one sends you the notifications of the wanted MBean. However, in case of an active object, if the object migrates it is necessary to unsubscribe the listener from the MBean Server and to open a new connection to the new MBean Server and subscribe to this one. If you do not do those steps you will not continue to receive the notifications. In order to resolve this problem, we provide the **JMXNotificationManager** class.

8.2.1. Subscribe to the JMX notifications of a ProActive object

```
JMXNotificationManager.getInstance().subscribe(ObjectName objectName, NotificationListener listener, String
 runtimeUrl);
```

objectName is the identifier of the MBean.

The FactoryName class gives you some methods in order to get the objectName:

- FactoryName.createActiveObjectName(UniqueID id);
- FactoryName.createNodeObjectName(String runtimeUrl,String nodeName);
- FactoryName.createRuntimeObjectName(String url);
- FactoryName.createVirtualNodeObjectName(String vnName,String jobID);

listener is a JMX notification listener (It has to implement the NotificationListener interface).

runtimeUrl The url of the runtime where to find the MBean.

If you want to subscribe a notification listener to a Local JMX MBean, you can do as follows:

```
JMXNotificationManager.getInstance().subscribe(
      ProActiveRuntimeImpl.getProActiveRuntime().getMBean().getObjectName(), listener);
```

8.2.2. Unsubscribe to the JMX notifications

```
JMXNotificationManager.getInstance().unsubscribe(ObjectName objectName, NotificationListener listener);
```

For instance, the following piece of code unsubscribes the listener from the ProActive runtime Mbean:

```
JMXNotificationManager.getInstance().subscribe(
```

```
        ProActiveRuntimeImpl.getProActiveRuntime().getMBean().getObjectName(), listener);
```

8.3. The ProActive JMX Notifications

8.3.1. How to send a JMX notification?

The ProActiveRuntimeImpl, LocalNode and AbstractBody classes contain a MBean and the **getMBean()** method.

On the MBean, you can call two different methods:

- sendNotification(String type);
- sendNotification(String type, Object userData);

type is the type of the notification. The NotificationType class contains a lot of existing notification types.

userData is the object to send to the listener. The package **org.objectweb.proactive.core.jmx.notification** contains some existing notification.

8.3.2. Example of notification listener

```
import javax.management.Notification;
import javax.management.NotificationListener;

import org.objectweb.proactive.examples.documentation.jmx.mbeans.Hello;


public class MyListener implements NotificationListener {

   public void handleNotification(Notification notification, Object handback) {

      String type = notification.getType();

      System.out.println("\nReceiving Notification: ");
      System.out.println("My type is " + type);
      if (type.equals(Hello.NOTIFICATION_NAME)) {
         Hello h = (Hello) notification.getUserData();
         System.out.println("my current message is: " + h.getMessage());
         h.saySomething();
      }
   }
}
```

8.3.3. The JMX notifications sent by the ProActive MBean

This section details the notifications that can be sent by active object or ProActive runtime MBeans.

8.3.3.1. Notifications sent by the active object MBean.

Warning

For performance reasons, the MBean of an active object (BodyWrapperMBean) sends a set of notifications.

Type of the notification: Notification.setOfNotifications

User Data: ConcurrentLinkedQueue<Notification>

Type of notification	UserData
NotificationType.migrationAboutToStart	String (Url of the destination node)
NotificationType.migratedBodyRestarted	null
NotificationType.migrationFinished	String (Url of the destination runtime
Notification.migrationExceptionThrown	MigrationException

Table 8.1. Migration information

Type of notification	UserData
NotificationType.requestSent	RequestNotificationData
NotificationType.requestReceived	RequestNotificationData
NotificationType.replySent	null
Notification.replyReceived	null
NotificationType.servingStarted	Integer (Request queue Size)
NotificationType.voidRequestServed	Integer (Request queue Size)
NotificationType.waitForRequest	null

Table 8.2. Request information

Type of notification	UserData
NotificationType.waitByNecessity	FutureNotificationData
NotificationType.receivedFutureResult	FutureNotificationData

Table 8.3. Future information

8.3.3.2. Notifications sent by the ProActiveRuntime MBean

Type of notification	UserData
NotificationType.bodyCreated	BodyNotificationData
NotificationType.nodeCreated	NodeNotificationData
NotificationType.runtimeRegistered	RuntimeNotificationData
Notification.runtimeAcquired	RuntimeNotificationData

Table 8.4. Creation information

Type of notification	UserData
NotificationType.bodyDestroyed	BodyNotificationData
NotificationType.nodeDestroyed	NodeNotificationData
NotificationType.runtimeDestroyed	null
Notification.runtimeUnregistered	RuntimeNotificationData

Table 8.5. Destruction information

Chapter 9. TimIt API

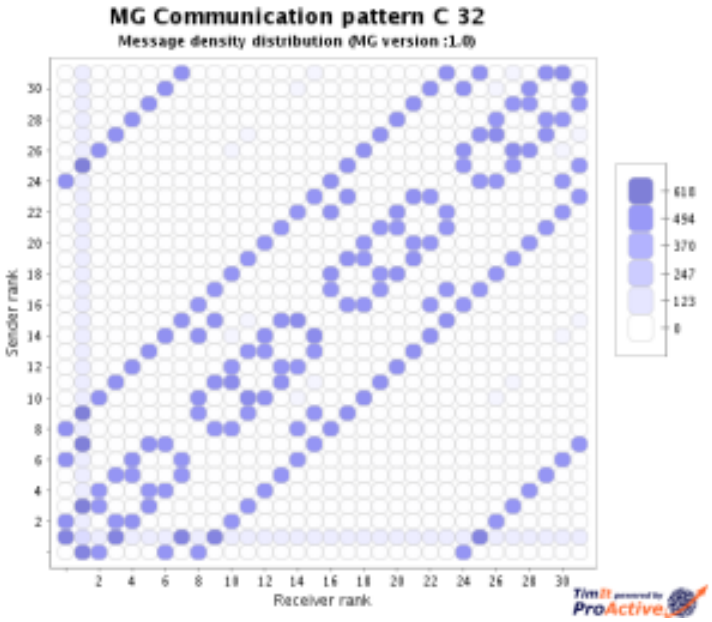
9.1. Overview

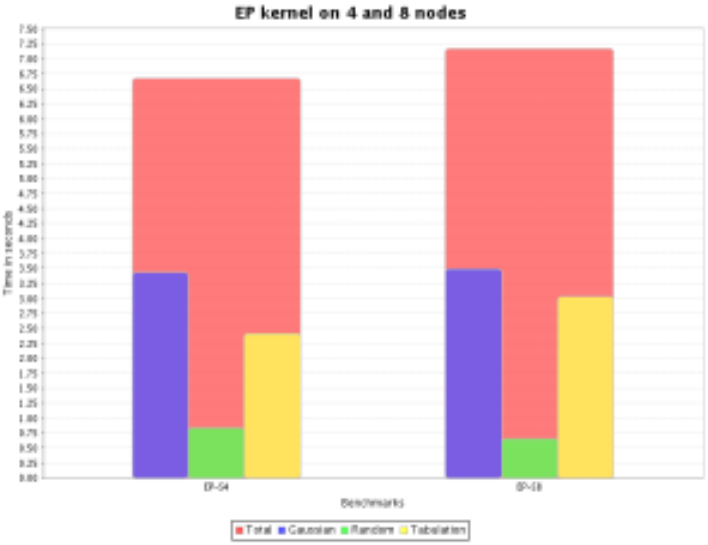
TimIt offers a complete solution to benchmark an application. It is an API which provides some advanced timing and event observing services. Benchmarking your ProActive application will permit you to enhance performance of it. Thanks to **generated statistics charts**, you will be able to determine critical points of your application.

TimIt is best suited for SPMD applications. Using TimIt for other purposes is not recommended but possible. All TimIt SPMD classes are located in **org.objectweb.proactive.extensions.timitspmd** package in the ProActive sources folder.

Different kind of statistics can be collected. You can setup different timers with **hierarchical capabilities** and see them in charts. For instance, event observers can be used to study communication pattern between your application's workers.

TimIt generates charts and results XML file, with exact timing and event observers values. Here are some examples of charts and XML files generated by TimIt:






```
<timit>
 <FinalStatistics name="Example2 4" runs="10" timeoutErrors="0"
                  date="2006-11-05 10:46:56.742">

  <timers>
   <timer name="total"
          min="2095.0" avg="2191.250" max="2357.0" dev="1.603" sum="2187.750">
    <timer name="work"
           min="1453.0" avg="1466.000" max="1473.0" dev="0.951" sum="0.000" />
    <timer name="init"
           min="147.0" avg="175.250" max="205.0" dev="2.932" sum="0.000" />
    <timer name="end"
           min="467.0" avg="546.500" max="679.0" dev="1.439" sum="0.000" />
   </timer>
  </timers>
```

```
  <events>
    <event name="nbComms" min="92.000" avg="92.000" max="92.000" dev="0.000" />
    <event name="commPattern" value="Too complex value, first run shown">.
10  0 13  0
 0 13  0 10
13  0 10  0
 0 10  0 13
    </event>
    <event name="densityPattern" value="Too complex value, first run shown">.
  20    0 2080    0
   0 2080    0   20
2080    0   20    0
   0   20    0 2080
    </event>
  </events>


  <informations>
    <deployer jvm="Java HotSpot(TM) Client VM 1.5.0_06-64 - Version 1.5.0_06"
            os="ppc Mac OS X 10.4.8" processors="1" />
  </informations>


 </FinalStatistics>

</timit>
```

9.2. Quick start

9.2.1. Introduction

In order to use TimIt, you first have to create a configuration file as explained in Section 9.2.2, “Define your TimIt configuration file”. This configuration file is used to describe the application to be launched as well as outputs you wants (reports and charts). This file will then be used as an argument of the TimIt main method:

```
java org.objectweb.proactive.extensions.timitspmd.TimIt -c configuration_file.xml
```

9.2.2. Define your TimIt configuration file

Configuring TimIt is done through an XML configuration file which is axed around four major tags:

9.2.2.1. Global variables definition

This part sets variables which can be used both inside this file as you can see in next parts, but also in **ProActive** descriptor file.

TimIt offers a nice tool to deal with variables and redundancy: the **sequences variables**

These variables are very useful to reduce your configuration file size and its management.

A **sequence** is a list of values for a variable. In our example, **NP** is a sequence variable which has values **4** and **8** and the **benchmark** tag will be **expanded** into two benchmark tags: one with NP value set to 4 and an other with NP value set to 8.

If a sequence variable is used in a **serie** 's attribute, this serie tag will be expanded to a number of serie tags equals to the number of values there are in your sequence.

For example, these two examples are equivalents :

```
<timit>

 <globalVariables>
  <descriptorVariable name="ALGO" value="Algo1,Algo2"/>
  <descriptorVariable name="NP"   value="4,8"/>
  <descriptorVariable name="TEST" value="#1"/>
 </globalVariables>

 <serie (...) result="${ALGO}">
  <benchmarks>
   <benchmark name="Test ${TEST} : algo ${ALGO} on ${NP} nodes" (...)/>
  </benchmarks>
 </serie>

</timit>
```

```
<timit>

 <globalVariables>
  <descriptorVariable name="TEST" value="#1"/>
 </globalVariables>

 <serie (...) result="Algo1">
  <benchmarks>
   <benchmark name="Test #1 : algo Algo1 on 4 nodes" (...)/>
   <benchmark name="Test #1 : algo Algo1 on 8 nodes" (...)/>
  </benchmarks>
 </serie>

 <serie (...) result="Algo2">
  <benchmarks>
   <benchmark name="Test #1 : algo Algo2 on 4 nodes" (...)/>
   <benchmark name="Test #1 : algo Algo2 on 8 nodes" (...)/>
  </benchmarks>
 </serie>

</timit>
```

Important:

Sequences variables are not handled by **ProActive** descriptor files, so do not use same names for ProActive descriptor and sequence variable names to avoid bad overwriting. To do it, you should prefer overwriting in **benchmark** tag like this:

```
<benchmark name="Test ${TEST} : algo ${ALGO} on ${NP} nodes" (...) >
 <descriptorVariable name="NBNODES" value="${NP} />
</benchmark>
```

Note:

You can use **sequences** without using variables, with #{...} pattern:

```
<benchmark name="Test ${TEST} : algo #{Algo1,Algo2} on ${NP} nodes" (...) >
 <descriptorVariable name="NBNODES" value="${NP} />
</benchmark>
```

9.2.2.2. Serie

A serie represents a suite of benchmarks. For example, if you want to benchmark two algorithms with different parameters, you can specify two series (one for each algorithm) and then specify different benchmarks for all parameters.

Attribute description:

- **descriptorBase** (Optional): the file containing the base ProActive deployment descriptor
- **class** (Compulsory): the class of your application which is **Startable** (see Section 9.2.3, " Add time counters and event observers in your source files ")
- **result** (Compulsory): the output file for writing final results
- **errorFile** (Optional): if an error occurs (recoverable), logs will be outputted into this file

9.2.2.3. Chart definition

In order to describe charts, you have to use **chart** tags inside a **charts** one. Those charts will be generated thanks to benchmark results.

Attribute description:

Some attributes are independent on the chart type:

- **type** (Compulsory): the type of chart you want to create
- **title** (Compulsory): your chart title
- **subtitle** (Compulsory): your chart subtitle
- **xaxislabel** (Compulsory): the X axis label
- **yaxislabel** (Compulsory): the Y axis label
- **width** (Optional): the width of the output chart
- **height** (Optional): the height of the output chart
- **filename** (Compulsory): the chart output filename (will produce both a *.PNG* and *.SVG* files)

But some others are chart type specific:

- **filter** (Optional): the name of the counter (event) you want to involve in this chart. All activated counters (events) are involved if not specified (available only for HierarchicalBarChart and Line2dChart)
- **tag** (Compulsory): the tag to deal with (timers or events) must be associated with **attribute** (available only for Line2dChart)
- **attribute** (Compulsory): the attribute value (min, average, max or deviation) to use for the chart (available only for Line2dchart)
- **legendFormatMode** (Optional): the format of the legend (Default, None, K1000, K1024) to show value in legend as standard, power of 2 or power of 10 numbers (available only for MatrixChart)
- **scaleMode** (Optional): the scale mode (Default, Linear, Logarithmic) for the chart (available only for **MatrixChart**)

9.2.2.4. Benchmark suite definition

To define the suite of tests with different parameters, you have to use **benchmark** tags inside a **benchmarks** one. Each test will generate a result file and an entry in chart.

Attribute description:

- **name** (Compulsory): the name of the benchmark. Will be set in result file.
- **run** (Compulsory): the number of runs you want to perform. Final result will give the min/average/max/deviation between these runs.
- **warmup** (Optional): the number of "untimed" runs you want to perform before starting the real runs.
- **timeout** (Optional): the time in seconds before restarting a run (with a maximum of 3 restarts per benchmark).
- **descriptorGenerated** (Optional): the ouput file where TimIt will put the ProActive deployment descriptor.
- **removeExtremums** (Optional): if true, max and min values between all runs will be removed.
- **note** (Optional): the text entered here will be copied into the result file. Useful for specifying launch environment.

- **parameters** (Compulsory): the parameters to launch your application.
- **output** (Compulsory): result of all runs will be outputted into this output file.

In addition to these attributes, you can specify descriptorVariable tags which will be copied into generated ProActive deployment descriptor file.

9.2.2.5. Example of a complete configuration file

Here is a complete example of a configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<timit>

  <!-- GLOBAL VARIABLES DEFINITION
     Will replace those in deployment descriptor -->
  <globalVariables>
    <descriptorVariable name="VMARGS" value="-Xmx32M -Xms32M" />
    <descriptorVariable name="CLASS_PREFIX"
      value="org.objectweb.proactive.examples.documentation.timit" />
    <descriptorVariable name="NP" value="4,8" />
    <descriptorVariable name="RUN" value="1" />
    <descriptorVariable name="WARMUP" value="0" />
  </globalVariables>

  <series descriptorBase="descriptors/TimItApplication.xml"
    result="results/serie/example.4-8.xml"
    class="${CLASS_PREFIX}.Example">
    <charts>
      <chart type="MatrixChart"
        eventName="communicationPattern"
        title="Example"
        subtitle="Communications pattern"
        xaxislabel="Receiver rank" yaxislabel="Sender rank"
        scalemode="logarithmic" legendFormatMode="pow2"
        filename="results/serie/charts/example.Pattern" />
      <chart type="HierarchicalBarChart"
        filter="total,init,end" title="Example on 4 and 8 nodes"
        subtitle="Timing values" width="800" height="600"
        xaxislabel="Benchmarks" yaxislabel="Time in seconds"
        filename="results/serie/charts/example.Timing" />
    </charts>
    <benchmarks>
      <benchmark name="Example ${NP}"
        run="${RUN}" warmup="${WARMUP}" timeout="100"
        descriptorGenerated="descriptors/generated.xml"
        removeExtremums="true"
        note="My first test"
        parameters="descriptors/generated.xml ${NP}"
        output="results/serie/Example/example-${NP}.xml">
        <descriptorVariable name="NODES" value="${NP}" />
      </benchmark>
    </benchmarks>
  </series>
</timit>
```

9.2.3. Add time counters and event observers in your source files

This section shows how to add time counters and event observers in your source files. It is really straight forward but you have nevertheless two small conditions to respect:

1. Your main class has to implement the **Startable** interface:

```
public class Example implements Startable {

  private GCMApplication pad;
  private Worker workers;

  /** TimIt needs a noarg constructor (can be implicit) **/
  public Example() {
  }

  /** The main method is not used by TimIt **/
  public static void main(String[] args) {
     new Example().start(args);
  }

  /** Invoked by TimIt to start your application **/
  public void start(String[] args) {

     // Common stuff about ProActive deployment
     // ...
     // Gets an array of node called 'arrayNode'


     // Creation of the Timed object(s)
     // It can be by example :
     //   - a classic java object
     //   - an active object
     //   - a group of objects
     try {
        this.workers = (Worker) PASPMD.newSPMDGroup(Worker.class.getName(), params, nodeArray);
     } catch (Exception e) {
        e.printStackTrace();
     }

     // You have to create an instance of TimItManager and
     // give it to the Timed objects
     TimItManager tManager = TimItManager.getInstance();

     tManager.setTimedObjects(this.workers);

     // Timed objects start their jobs
     this.workers.start();

     // At the end of your application, you must invoke
     // the getBenchmarkStatistics to retrieve the results
     // from the Timed objects
     BenchmarkStatistics bStats = tManager.getBenchmarkStatistics();
```

```
        // Then, you can modify or print out the results
        System.out.println(bStats);
    }

}
```

2. your analyzed class has to extend the **Timed** class:

```
public class Worker extends Timed {

    /* Declaration of all TimerCounters and EventObservers */

    /** Total time */
    public TimerCounter T_TOTAL, T_INIT, T_END;

    /** Communication Observer */
    public EventObserver E_COMM;

    private int rank, groupSize;

    // An empty no args constructor, as needed by ProActive
    public Worker() {
    }

    public void start() {
        this.rank = PASPMD.getMyRank();
        this.groupSize = PASPMD.getMySPMDGroupSize();

        // First you have to get an instance of TimerStore for this
        // active object.
        // You can create counters with this TimerStore
        // for distribute these counters to every class in this active
        // object.
        // IMPORTANT : TimerStore instance is note integrated in the active
        // object so problems may occur with Fault Tolerance and migration.
        // If you have to do that, do not use TimerStore and pass Counters
        // by another way.
        TimItStore ts = TimItStore.getInstance(this);
        // Add timer counters to the TimItStore

        // Register the TimerCounters and EventObservers
        T_TOTAL = ts.addTimerCounter(new TimerCounter("total"));
        T_INIT = ts.addTimerCounter(new TimerCounter("init"));
        T_END = ts.addTimerCounter(new TimerCounter("end"));
        E_COMM = (CommEventObserver) ts.addEventObserver(new
CommEventObserver("communicationPattern",
            this.groupSize, this.rank));

        // You must activate TimItStore before using your counters and observers
        // It is also possible to activate all your counters and observer at once
        // using ts.activation() but you have to set the proactive.timit.activation
        // property. See existing timit examples to learn more about this possibility.
        super.activate(new EventObserver[] { this.E_COMM });
        super.activate(new TimerCounter[] { this.T_TOTAL, this.T_INIT, this.T_END });
```

```
        // Then, you can use your counters and observers
        T_TOTAL.start();

        T_INIT.start();
        this.sleep(251);
        T_INIT.stop();

        int destRank;
        for (int i = 0; i < 10; i++) {
            destRank = (this.rank + 1) % this.groupSize;
            super.getEventObservable().notifyObservers(new CommEvent(this.E_COMM, destRank, 1));
            if (this.rank > 3)
                this.sleep(20);
        }

        T_END.start();
        this.sleep(101);
        T_END.stop();

        T_TOTAL.stop();

        // Finally, you have to say that timing is done by using finalizeTimer()
        // method. You can specify some textual informations about this worker.
        // This information will be shown in final XML result file.
        // Take care when using it with many nodes... :)
        super.finalizeTimed(this.rank, "Worker" + this.rank + " is OK.");
    }

}
```

9.3. Usage

TimIt provides different kinds of services. By combining them, you will be able to measure many parameters of your application. TimIt package contains few examples for using these services in your application.

9.3.1. Timer counters

It will help you to time some pieces of code in your application. For example you can get total, initialization, working and communication times. These counters are hierarchical. It means that time values will be defined by counter dependencies.

Example of hierarchy:

- Total time = 60 seconds
  - Initialization time = 10 seconds
    - Communication time = 4 seconds
  - Working time = 50 seconds
    - Communication time = 17 seconds

Here you can see **communication** part both in **initialization** and **working** time.

The code associated to this example is:

```
T_TOTAL.start();
```

```
T_INIT.start();
  // Initialization part...
  T_COMM.start();
    // Communications...
  T_COMM.stop();
T_INIT.stop();

T_WORK.start();
  // Working part...
  T_COMM.start();
    // Communications...
  T_COMM.stop();
T_WORK.stop();

T_TOTAL.stop();
```

9.3.2. Event observers

It will help you to keep an eye on different events that occur in your application.

There is two types of events:

- **Default event**

  This event manages a **single value** (a **double**). It can be useful to compute mflops or total number of performed communications.

  Example of usage:

  ```
  // Initialization
  int collapseOperation = DefaultEventData.SUM;
  int notifyOperation = DefaultEventData.SUM;
  EventObserver E_NBCOMMS = TimIt.add(
          new DefaultEventObserver("nbComms", collapseOperation, notifyOperation));
  ```

  The value of **notifyOperation** determines what operation to perform between notifications.

  The value of **collapseOperation** determines what operation to perform between timed objects.

  ```
  // Utilization
  for( int i=0; i<10; i++ ) {
    TimIt.notifyObservers( new Event(E_NBCOMMS, 1) );
  }
  ```

  For each timed object, nbComms value will be 10, and final value would be 30 if we had 3 timed objects.

- **Communication event**

  This event were designed for communications. It manages a **square matrix** which can be used by example to determine topology of communications between timed objects.

  Example of usage:

  ```
  // Initialization
  EventObserver E_COMM = TimIt.add(
          new CommEventObserver("mflops", groupSize, timedID);
  ```

  The value of **groupSize** represents the number of timed objects which are involved in these communications.

the value of **timedID** represents an identification number which represents the current timed object (like the rank).

```
// Utilization
int destID = (timedID + 1) % groupSize;
TimIt.notifyObservers( new CommEvent(E_COMM, destID, 1) );
```

Between each notification, an incrementation of the old value will be performed. Then the collapsing operation between the timed objects will be a sum. In this case, you will obtain a matrix showing the topology of your application.

Lines represent the senders and columns the receivers. Here we obtain a ring topology:

```
1 0 0 0
0 0 0 1
0 0 1 0
0 1 0 0
```

9.4. TimIt extension

TimIt package can be found in the **org.objectweb.proactive.extensions.timitspmd** package. We try to make as easy as possible the way to add a new feature to this application. To do so, TimIt is organized in 5 major points which can be extended:

9.4.1. Configuration file

The subpackage **org.objectweb.proactive.extensions.timitspmd.config** contains all classes related to the configuration file management.

- **ConfigReader**

  This class reads the configuration file. It deals with **globalVariable** and **serie** tags.
- **Tag**

  All created tags (except **globalVariable**) have to extend this class. It makes easier the way to read tag's attributes. If you want to create a new tag, extend this class and take **Benchmark** as an example.

Example:

Suppose you want to add an attribute **myOption** to the **Benchmark** tag where the default value is **1**.

```
// Add these lines in the get method of Benchmark class
if (name.equals("myOption")) {
  return "1";
}
```

Then, you will be able to use it like this:

```
String result = bench.get("myOption");
// ... and do whatever you want with it...
```

9.4.2. Timer counters

The subpackage **org.objectweb.proactive.extensions.timitspmd.util** contains all classes related to the timing management.

- **HierarchicalTimer**

  This class will contain values of all timer counters. Here is all the "intelligence" of the timer. For example, if you want to use nanoseconds instead of milliseconds, you should extend this class and overwrite getCurrentTime() method.

9.4.3. Event observers

The subpackage **org.objectweb.proactive.extensions.timitspmd.util.observing** contains all classes related to the event observers management. Existent event observers are default and communication specific. Default (**DefaultEventObserver**) is base on a single value, while the communication specific (**CommEventObserver**) is based on 2D square matrix.

Event observers are based on the **observer-observable** design pattern.

- **EventObserver**

  This interface must be implemented by all kind of event observers. These implementations will have to deal with an **EventData**.
- **Event**

  Each kind of event should have its own **Event** implementation. An instance of this event will be passed to each notification.
- **EventData**

  Like **HierarchicalTimer** for the timing service, **EventData** is the "intelligence" of event observing. It will contain all data values for a particular **Timed** object. It also contain a collapseWith() method which will be used to merge data values from all **Timed** objects.
- **EventObservable**

  For performance purpose, there is to implementations of this interface. A **FakeEventObservable** and a **RealEventObservable**.
- **EventDataBag**

  This class contains data values from all **Timed** objects. You are able to get it through an **EventStatistics**.
- **EventStatistics**

  When a benchmark run is done, you can get a **BenchmarkStatistics** which contains both timer and event statistics.

Example:

Suppose you want to create a new kind of **Event** which works with a 3D matrix instead of 2D matrix like **CommEventObserver**.

You will have to implement 2 or 3 classes:

1. **MyEventObserver** which implements **EventObserver**

   It will receive notifications and transmit them to your **EventData**.
2. **MyEventData** which implements **EventData**

   It will contain your 3D matrix computed from your notifications.
3. **MyEvent** which implements **Event**

   It will be passed at each notification of your observer and will contain necessary data to update your 3D matrix.

   Notice that you can reuse an other **Event** implementation if existing ones are sufficient.

9.4.4. Chart generation

The subpackage **org.objectweb.proactive.extensions.timitspmd.util.charts** contains all classes related to charts generation. This service is based on **JFreeChart** API (http://www.jfree.org/jfreechart/). Three major type of charts are proposed with **TimIt**:

- **HierarchicalBarChart** - used to represent a serie of hierarchical timing statistics.
- **Line2dChart** - used to represent a serie of single values.
- **MatrixChart** - used to represent communications specific event observer.

Remember that in configuration file, choosing your chart type is done through the type attribute of chart tag. Actually, it represents the classname used to handle your chart creation.

Moreover, to create a new kind of chart, you just have to implement the **Chart** interface. So, you will have access to XML results file, full **BenchmarkStatistics** and all chart parameters given in configuration file (see Section 9.4.1, “Configuration file” to know how to add an attribute).

If you need a very complex rendering chart method, you can implement your own renderer like we did for **HierarchicalBarChart**. Take this class as an example, and refer to the **JFreeChart** documentation.

Chapter 10. Proxy Command

10.1. Overview of proxy command : a bouncing connection command mechanism

For dealing with filtered network topology, ProActive provide an implementation of the OpenSSH ProxyCommand mechanism. It permit to do a bouncing SSH connection using a gateway. A SSH server has to be host by the gateway, and an implementation of the netcat command should be available.

The proxyCommand is implemented as an extension of the RMI+SSH protocol, so the protocol RMI+SSH must be selected by using the ProActive Property :

```
proactive.communication.protocol=rmissh
```

10.2. Principles

The proxyCommand consists in opening a connection to a specified gateway, and then from this connection, bouncing all request to the desired host. Thus it is possible to rely NATed (Network Address Translation) LANs, Clusters, ...

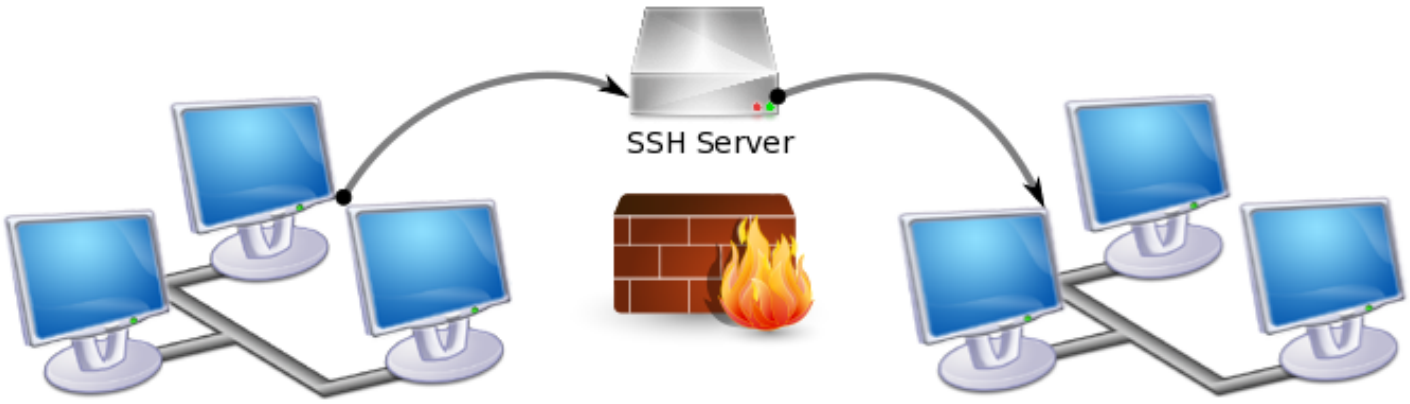


Figure 10.1. An example of proxy command use

10.3. Configuration

The proxyCommand can be configured either by specifying a ProActive property or by using the one found in the file .ssh/config. The property to use is

```
proactive.communication.ssh.proxy.gateway
```

.

- PROPERTY : PROPERTY;RULE | RULE
- RULE : HOSTDEF ':' GATEWAY ':' PORT
- HOSTDEF : HOSTJOKER | SUBNET

- HOSTJOKER : * '.' domain.tld | HOST
- SUBNET : ip '/' cidr
- GATEWAY : HOST
- HOST : hostname | ip
- PORT : integer

Here is an example :

```
*.domain.com:gateway.domain.com:22;192.168.1.0/24:gateway.domain.com:22;
```

Chapter 11. Multi Protocol support

11.1. The support of the Multi Protocol in ProActive

The ProActive framework contains many protocols to fit all the constraints that the network topology will makes.

Therefore, it provides the support of the multi-protocol, this permits to make the most of the underlying network, by using the most suitable protocol on each parts of the network and to maintain reliability between each nodes.

The support of multi-protocol aims to be totally transparent, no user actions are needed, neither for exposition, nor for selection. The only constraint is that the default protocol selected must be able to allow communication between all nodes.

11.2. The configuration properties for multi protocol.

- **proactive.communication.additional_protocols**
  The set of protocol to use separated by commas.
- **proactive.communication.benchmark.parameter**
  This property is used pass parameters to the benchmark.
  This could be a duration time, a size, ...
  This property is expressed as a String.
- **proactive.communication.protocols.order**
  A fixed order could be specified if protocol's performance is known in advance and won't change. This property explain a preferred order for a subset of protocols declared in the property proactive.communication.additional_protocols. If one of the specified protocol isn't exposed, it is ignored. If there are protocols that are not declared in this property but which are exposed, they are used in the order choose by the benchmark mechanism.

  Example :
  Exposed protocols : http,pnp,rmi
  Benchmark Order : rmi > pnp > http
  Order : pnp
  This will give the order of use : pnp > rmi > http
- **proactive.communication.protocols.order**
  A fixed order could be specified if protocol's performance is known in advance and won't change.
  This automatically disabled RemoteObject's Benchmark.
- **proactive.communication.benchmark.class**
  Any kind of benchmark could be implemented, this property permit to specify the class of Benchmark to use.
  See the next section for further explanations.

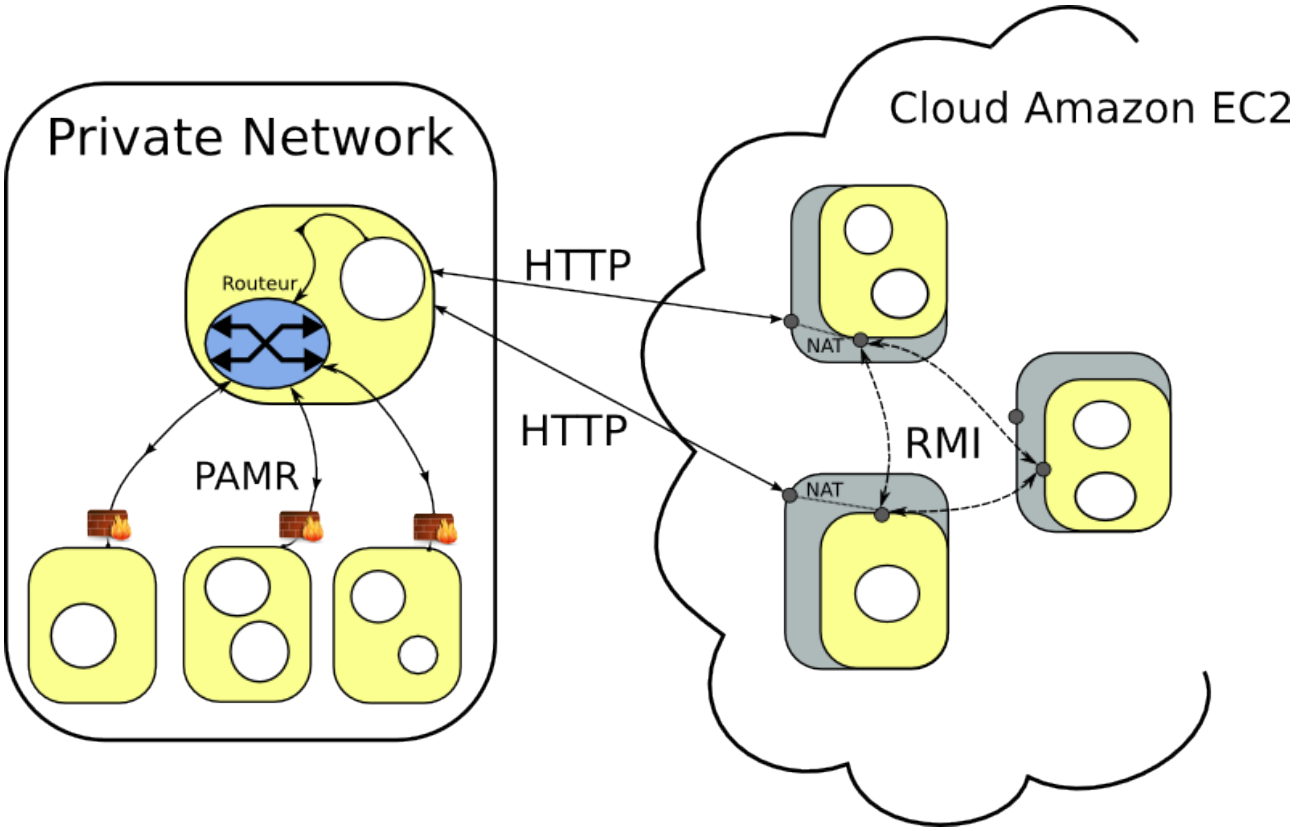


Figure 11.1. An example of multi protocol use

11.3. Benchmarking possibilities

By default, the benchmarking are disabled and only the selection is done. For enabling the benchmark, the property *proactive.communication.benchmark.class* must be set to something else than *org.objectweb.proactive.core.remoteobject.benchmark.SelectionOnly*.

11.3.1. Custom Benchmarks

By implementing the interface *org.objectweb.proactive.core.remoteobject.benchmark.BenchmarkObject* , custom benchmarks could be added. Before every benchmark of a RemoteRemoteObject, a new BenchmarkObject is created. This interface provides these methods :

- **public void init();**
  Which is launch before the benchmark.
- **public int getResult();**
  This method is called at the end and should return a higher value if the protocol is better.
- **public void receiveResponse(Object o);**
  If the benchmark should return a result, this method is called. The Object o is the response return by the RemoteObject.
- **public boolean doTest();**
  This method is the condition for the main while loop : while (BenchmarkObject.doTest) { ... }
- **public RemoteObjectRequest getRequest();**
  This method is the most complicated one, the benchmark is based on a RemoteObjectRequest, which is a request handled by the RemoteObject layer. This method should return an object like that. Basically, a RemoteObjectBenchmark contains a method Object execute() which is called on the server part of the RemoteObject.

11.3.2. Existing benchmarks

- **org.objectweb.proactive.core.remoteobject.benchmark.ThroughputBenchmark**
  The maximum number of short message send in a finite time. The parameter indicate the duration of the benchmark in ms.
- **org.objectweb.proactive.core.remoteobject.benchmark.LargeByteArrayBenchmark**
  The time to send a fixed number of message which contains a large array. The parameter indicate the size of the array in kB.
- **org.objectweb.proactive.core.remoteobject.benchmark.SelectionOnly**
  Only do the selection, do not perform benchmark. The parameter has no impact.

11.4. API for multi-protocol

The API for dealing with multi protocol support is placed in PARemoteObject class.

- **PARemoteObject.forceProtocol(Object o, String protocol)**
  This API call force the object o to use the protocol specified. If this protocol is unavailable (not already exposed, network problem ...) a NotYetExposedException is thrown. WARNING : If the protocol wanted fail, the communication will fail even if there are other reliable protocols.
- **PARemoteObject.forceToDefault(Object o)**
  Same as before but use the default protocol.
- **PARemoteObject.unforceProtocol(Object o)**
  This API call disable the protocol forcing mechanism.
- **PARemoteObject.addProtocol(RemoteObjectExposer roe, String protocol)**
  This API call permit to deploy an object to an additional protocol not specified in the property. Please note that this API call could only be made on server side, i.e. where we can access the RemoteObjectExposer of the target object.

Part II. Extending ProActive

Table of Contents

Chapter 12. How to write ProActive documentation

12.1. Aim of this chapter

This chapter is meant to help you as a reference for writing ProActive-directed documentation. If you have added a new feature and want to help its uptake by documenting it, you should read this chapter.

The examples sections (Section 12.3, “Example use of tags”) describes the use of the main tags you will use (eventually, all the ProActive-allowed docbook tags should be described). The limitations (Section 12.4, “DocBook limitations imposed”) section describes what is allowed in our docbook style, and why we restrict ourselves to a subset of docbook.

12.2. Getting a quick start into writing ProActive doc

First off, all the documentation is written in docbook[1]. You can find all the documentation source files in the ProActive/doc/src/ directory.

Here are the instructions to follow to start well and fast writing documentation for the ProActive middleware:

1. Choose an XML editor: XMLMind XML Editor (**XXE**), Eclipse (an its XML plugin)... You can also use a simple text editor.
2. If you want a new chapter of your own, copy one of the existing files (ProActive/doc/src/WebServices.xml for example).
3. Reference your file in the root element of the doc (it is currently called main.xml)
4. Open your file with the editor you have chosen.
5. If your are using XXE:
   - REMEMBER: YOU ARE EDITING AN XML FILE - you can always edit it with vi if you dare
   - Use generously the icons at the top, they have the essential tags you will use
   - Use the list of tags, just under the icons, to select the item you want to edit
   - Use the column on the right to add tags, when you know their names
   - When you're done, there is a spellchecker integrated, as well as a DocBook validator. Please use these tools!
6. Run the ant target build doc.ProActive.manualHtml and you should have an HTML copy of the doc. If you want to generate all the possible output formats, call build doc.ProActive.manual. Type build to see all the available targets for compiling the documentation. With XXE, You can also see what the results seem to be without compiling!
7. Commit your changes to the SVN repository

12.3. Example use of tags

These are the basic rules to follow to use docbook tags. This document is made up of the files in the docBookTutorial directory, and you may find it with the other manual files in the ProActive/doc/src directory.

12.3.1. Summary of the useful tags

The main tags/structures you should be using are:

- **<figure>** - when you want to insert an image
- **<example>** - when you want an example with a title (should contain a <screen> or a <programlisting>). You can also use <literal> inside paragraphs.
- **<screen>** - when you want to insert a command line, an output or a flat text...
- **<programlisting language="[java|xml|c]">** - when you want to insert a java, an XML or a C snippet of code which will be highlighted.

[1] http://nwalsh.com/docbook/

- **<para>** - to start a paragraph.
- **<section>** - to start a section.
- **<emphasis>** - when you want to have some particular text pointed out.
- **<itemizedlist>** followed by several **<listitem>** - when you want bullets
- **<orderedlist>** followed by several **<listitem>** - when you want numbered bullets
- **<xref>** - when you want to reference another section, chapter, example, etc. using its id.
- **<ulink>** - when you want to reference a web URL.
- **<table>** - when you want to insert a table.

Note

If you are using XXE, you should always use the XXE icons. They have all you need (except for EXAMPLE/SCREEN)! You can also cut'n paste!

12.3.2. Figures

This is the figure example. Please use the **<title>** tag.

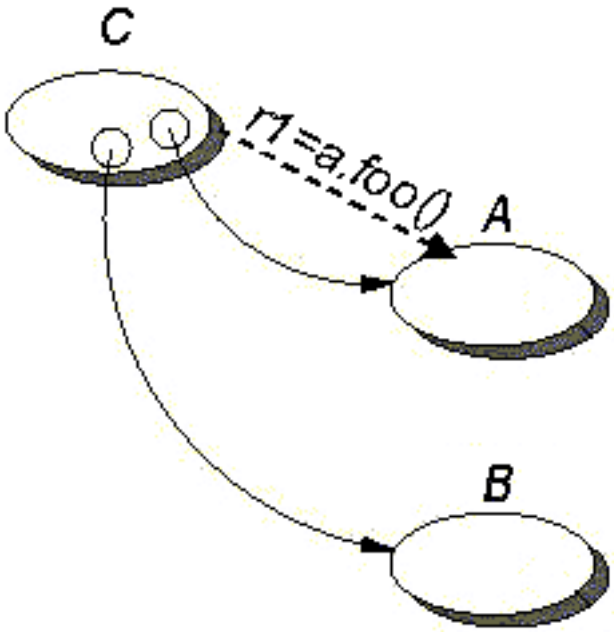


Figure 12.1. A drawing using the <figure> tag

Here is what has been written into the XML file:

```
<figure xml:id="ADrawingusingtheFIGUREtag_42">
   <info>
      <title>A drawing using the &lt;figure&gt; tag</title>
   </info>
   <mediaobject>
      <imageobject>
         <imagedata scalefit="1" width="100%" contentdepth="100%" align="center" fileref="images/png/
e1.png" format="PNG"/>
```

```
      </imageobject>
    </mediaobject>
</figure>
```

12.3.3. Bullets

Use **<itemized>** followed by as many **<listitem>**'s as you want!

- Provide an implementation for the required server-side functionalities
- Provide an empty, no-arg constructor
- Write a method in order to instantiate one server object.

Here is what has been inserted into the XML file:

```
<itemizedlist>
   <listitem>
      <para>Provide an implementation for the required server-side functionalities</para>
   </listitem>
   <listitem>
      <para>Provide an empty, no-arg constructor</para>
   </listitem>
   <listitem>
      <para>Write a method in order to instantiate one server object.</para>
   </listitem>
</itemizedlist>
```

12.3.4. Code

Code sources should be written using the **<programlisting language="java">** tag (possibly language are java, xml or c). You do not have to write valid code, as the highlighting (done by the docbook highlighting) is based on regular expression replacement, and not on language grammars. If you want to show some program output, you can use **<screen>** instead of **<programlisting>**. In any case, watch out, because spaces count (and produce your own indentation)! You can also use the **<example>** tag around your **<programlisting>** or **<screen>** tags, to give a title and be referenced in the table of examples.

There are three different ways to insert code:

- by referencing a file
- by referencing a code snippet
- by directly writing your code into the <programlisting> tag

Warning

We strongly advise to use one of the two first ways. Writing directly codes into the documentation files implies to always update the documentation when source codes change. Referencing a file or a piece of code which is always up-to-date is the best means to keep coherence between the source code and the documentation.

We will now give some examples illustrating each case of writing code into the documentation. After each example, the documentation XML code is exposed.

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
```

```
*    Enterprise Grids & Clouds
*
* Copyright (C) 1997-2012 INRIA/University of
*                 Nice-Sophia Antipolis/ActiveEon
* Contact: proactive@ow2.org or contact@activeeon.com
*
* This library is free software; you can redistribute it and/or
* modify it under the terms of the GNU Affero General Public License
* as published by the Free Software Foundation; version 3 of
* the License.
*
* This library is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
* Affero General Public License for more details.
*
* You should have received a copy of the GNU Affero General Public License
* along with this library; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
* USA
*
* If needed, contact us to obtain a release under GPL Version 2 or 3
* or a different license than the AGPL.
*
*  Initial developer(s):               The ProActive Team
*                        http://proactive.inria.fr/team_members.htm
*  Contributor(s):
*
* ################################################################################
* $$PROACTIVE_INITIAL_DEV$$
*/
package org.objectweb.proactive.examples.hello;

import org.apache.log4j.Logger;
import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.api.PALifeCycle;
import org.objectweb.proactive.core.util.ProActiveInet;
import org.objectweb.proactive.core.util.log.Loggers;
import org.objectweb.proactive.core.util.log.ProActiveLogger;
import org.objectweb.proactive.core.util.wrapper.StringMutableWrapper;
import org.objectweb.proactive.extensions.annotation.ActiveObject;


/** A stripped-down Active Object example.
 * The object has only one public method, sayHello()
 * The object does nothing but reflect the host its on. */
@ActiveObject
public class TinyHello implements java.io.Serializable {
   private final static Logger logger = ProActiveLogger.getLogger(Loggers.EXAMPLES);
   private final String message = "Hello World!";

   /** ProActive compulsory no-args constructor */
   public TinyHello() {
   }
```

```
  /** The Active Object creates and returns information on its location
   * @return a StringWrapper which is a Serialized version, for asynchrony */
  public StringMutableWrapper sayHello() {
     return new StringMutableWrapper(this.message + "\n from " + getHostName() + "\n at " +
        new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new java.util.Date()));
  }

  /** finds the name of the local machine */
  static String getHostName() {
     return ProActiveInet.getInstance().getInetAddress().getHostName();
  }

  /** The call that starts the Acive Objects, and displays results.
   * @param args must contain the name of an xml descriptor */
  public static void main(String[] args) throws Exception {
     // Creates an active instance of class Tiny on the local node
     TinyHello tiny = (TinyHello) PAActiveObject.newActive(TinyHello.class.getName(), // the class to deploy
           null // the arguments to pass to the constructor, here none
           ); // which jvm should be used to hold the Active Object

     // get and display a value
     StringMutableWrapper received = tiny.sayHello(); // possibly remote call
     logger.info("On " + getHostName() + ", a message was received: " + received); // potential wait-by-necessity
     // quitting

     PALifeCycle.exitSuccess();
  }
}
```

Example 12.1. JAVA program listing with file inclusion

```
<example xml:id="JAVAprogramlistingwithfileinclusion_42">
  <info>
    <title>JAVA program listing with file inclusion</title>
  </info>
  <programlisting language="java"><textobject><textdata fileref="../../../src/Examples/org/objectweb/proactive/
examples/hello/TinyHello.java"/></textobject></programlisting>
</example>
```

Example 12.2. Documentation source code of Example 12.1, “JAVA program listing with file inclusion”

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath://org/objectweb/proactive/
core/component/adl/xml/proactive.dtd">

<!-- A user component. It has an interface to the dispatcher. -->

<definition name="org.objectweb.proactive.examples.components.c3d.adl.UserImpl">

<!-- The interfaces the component defines -->

 <interface signature="org.objectweb.proactive.examples.c3d.Dispatcher" role="client" name="user2dispatcher"/
>
```

```
<!-- The implementation of the component -->
 <content class="org.objectweb.proactive.examples.components.c3d.UserImpl"/>
 <controller desc="primitive"/>

<!-- deploy this component only on 'User' VirtualNodes (which must be found in the deploy. descr.) -->
 <virtual-node name="User"/>

</definition>
```

Example 12.3. XML program listing with file inclusion

```
<example xml:id="XMLprogramlistingwithfileinclusion_42">
   <info>
      <title>XML program listing with file inclusion</title>
   </info>
   <programlisting language="xml"><textobject><textdata fileref="../../../src/Examples/org/objectweb/proactive/
examples/components/c3d/adl/UserImpl.fractal"/></textobject></programlisting>
</example>
```

Example 12.4. Documentation source code of Example 12.3, “XML program listing with file inclusion”

Here is a screen example. For instance, this could be some code inside a unix shell:

```
linux > start.sh &
```

Example 12.5. A screen example

```
<example xml:id="screen_42">
      <info>
         <title>A screen example</title>
      </info>
      <screen>linux &gt; start.sh &amp; </screen>
   </example>
</example>
```

Example 12.6. Documentation source code of Example 12.5, “A screen example”

As previously evoked, it is also possible to display a snippet of an existing code. To do this, you should insert the following comments into your source code:

- //**@snippet-start MySnippet** or <!-- **@snippet-start MySnippet** --> - to start your snippet
- <!-- **@snippet-start-with-header MySnippet** --> - to start your snippet adding the XML prologue before (only for XML files)
- //**@snippet-start-with-copyright MySnippet** - to start your snippet adding the copyright before (only for Java files)
- //**@snippet-end MySnippet** or <!-- **@snippet-end MySnippet** --> - to end your snippet
- //**@snippet-break MySnippet** or <!-- **@snippet-break MySnippet** --> - to break your snippet
- //**@snippet-resume MySnippet** or <!-- **@snippet-resume MySnippet** --> - to resume your snippet

Warning

You cannot not insert an annotation before the prologue in case of an XML file and you should not insert an annotation before a copyright in case of a Java file. Adding such an annotation before the prologue of an XML file makes it invalid

and if you add one before the copyright of a Java file, it will be removed the next time we will update copyrights. If you want to have the prologue or the copyright displayed in your snippet, use **@snippet-start-with-header** and **@snippet-start-with-copyright**.

For this moment, only java, xml, c and fractal files are supported for snippet extraction and you must respect some rules:

- The name of your snippet, here "MySnippet", should be unique.
- A started snippet has to be ended.
- A broken snippet has to be resumed before it ends.
- Break blocks of a same snippet cannot be imbricated.
- For a given snippet name, the first annotation is snippet-start.
- For a given snippet name, the last annotation is snippet-end.
- The source file has to be located into the ProActive/src/ or the ProActive/examples/ directory. The other directories are not browsed by the snippet extractor.

When compiling the documentation, we can see errors that occurred during the snippet extraction.

```
/*
 * ################################################################
 *
 * ProActive Parallel Suite(TM): The Java(TM) library for
 *    Parallel, Distributed, Multi-Core Computing for
 *    Enterprise Grids & Clouds
 *
 * Copyright (C) 1997-2012 INRIA/University of
 *                 Nice-Sophia Antipolis/ActiveEon
 * Contact: proactive@ow2.org or contact@activeeon.com
 *
 * This library is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Affero General Public License
 * as published by the Free Software Foundation; version 3 of
 * the License.
 *
 * This library is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this library; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307
 * USA
 *
 * If needed, contact us to obtain a release under GPL Version 2 or 3
 * or a different license than the AGPL.
 *
 *  Initial developer(s):               The ProActive Team
 *                        http://proactive.inria.fr/team_members.htm
 *  Contributor(s):
 *
 * ################################################################
 * $$PROACTIVE_INITIAL_DEV$$
 */
```

```java
package org.objectweb.proactive.examples.documentation.classes;

import java.io.Serializable;

import org.objectweb.proactive.core.util.wrapper.StringWrapper;


/**
 * @author ffonteno
 *
 */
//@snippet-start class_A
public class A implements Serializable {
    protected String str = "Hello";

    /**
     * Empty no-arg constructor
     */
    public A() {
    }

    /**
     * Constructor which initializes str
     *
     * @param str
     */
    public A(String str) {
        this.str = str;
    }

    /**
     * setStrWrapper
     *
     * @param strWrapper StringWrapper to set
     *      StringWrapper is used since this method gives an example
     *      of parameter scatterring. Parameter need to be serializable.
     * @return this
     */
    public A setStrWrapper(StringWrapper strWrapper) {
        this.str = strWrapper.getStringValue();
        return this;
    }

    /**
     * getB
     *
     * @return a B object corresponding to the current Object
     */
    public B getB() {
        if (this instanceof B) {
            return (B) this;
        } else {
            return new B(str);
        }
```

```
    }

    /**
     * display str on the standard output
     */
    public void display() {
        System.out.println("A display =====> " + str);
    }

    //@snippet-break class_A
    //@snippet-start class_A_exception
    /**
     * Example of a method which can throw an exception
     *
     * @param hasToThrow boolean saying whether the method should
     *                   throw an exception
     * @throws Exception
     */
    public void throwsException(boolean hasToThrow) throws Exception {
        Thread.sleep(5000);
        if (hasToThrow)
            throw new Exception("Class A has thrown an exception");
    }
    //@snippet-resume class_A
    //@snippet-end class_A_exception
}
//@snippet-end class_A
```

Example 12.7. File from which the snippet will be extracted

```
public class A implements Serializable {
    protected String str = "Hello";

    /**
     * Empty no-arg constructor
     */
    public A() {
    }

    /**
     * Constructor which initializes str
     *
     * @param str
     */
    public A(String str) {
        this.str = str;
    }

    /**
     * setStrWrapper
     *
     * @param strWrapper StringWrapper to set
     *      StringWrapper is used since this method gives an example
     *      of parameter scatterring. Parameter need to be serializable.
```

```
     * @return this
     */
    public A setStrWrapper(StringWrapper strWrapper) {
        this.str = strWrapper.getStringValue();
        return this;
    }

    /**
     * getB
     *
     * @return a B object corresponding to the current Object
     */
    public B getB() {
        if (this instanceof B) {
            return (B) this;
        } else {
            return new B(str);
        }
    }

    /**
     * display str on the standard output
     */
    public void display() {
        System.out.println("A display =====> " + str);
    }

}
```

Example 12.8. Snippet extraction

```
<example xml:id="snippet_42">
    <info>
        <title>Result of the snippet extraction</title>
    </info>
    <programlisting language="java"><textobject><textdata fileref="automatic_snippets/class_A.snip"/></
textobject></programlisting>
</example>
```

Example 12.9. Documentation source code of Example 12.8, “Snippet extraction”

Finally, here is some java code directly included in the docbook (you can use CDATA to escape & and <):

```
package util;

import java.io.IOException;

/** Just a dummy class. */

public class Dummy {

  /** Just the method description
   * @param fileToConvert the name of the file to convert
   * @return a String created  */
```

```
 String convert(String fileToConvert) throws IOException {
  if (a > b && c < d ) {
    // can use "this" for 'NodeCreationEvent'
    VirtualNode vn = pad.getVirtualNode("vn");
    vn.start();
    }
  return "Hello World";
 }

}
```

Example 12.10. Java program listing with direct inclusion

```
      <example xml:id="programlistingWithDirectInclusion">
        <info>
          <title>Java program listing with direct inclusion</title>
        </info>

        <programlisting language="java">package util;

import java.io.IOException;

/** Just a dummy class. */

public class Dummy {

 /** Just the method description
  * @param fileToConvert the name of the file to convert
  * @return a String created  */
 String convert(String fileToConvert) throws IOException {
  if (a &gt; b &amp;&amp; c &lt; d ) {
    // can use "this" for 'NodeCreationEvent'
    VirtualNode vn = pad.getVirtualNode("vn");
    vn.start();
    }
  return "Hello World";
 }

}</programlisting>
      </example>
```

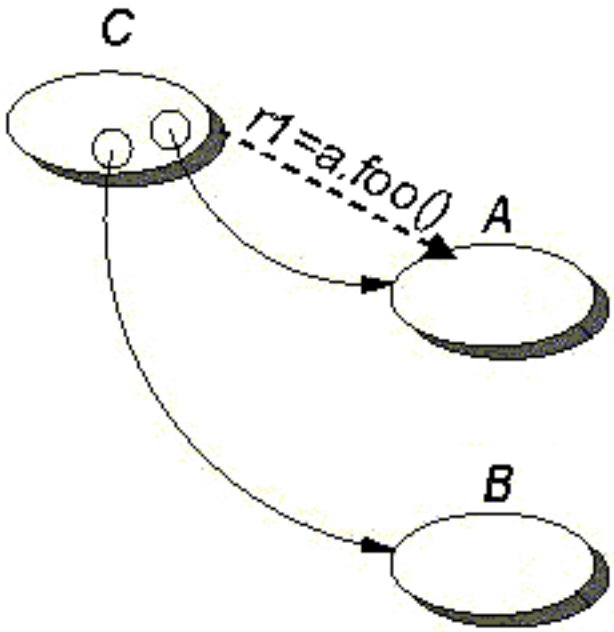
Example 12.11. Documentation source code of Example 12.10, “Java program listing with direct inclusion”

12.3.5. Links

Use **<xref>** tags to point to a element of your documentation. For instance, you can refer to the "Figures" section like that Section 12.3.2, “Figures”. The documentation source code of this link is the following one:

```
<xref linkend="Figures"/>
```

The linkend attribute points to the id which is referenced, for example, in a <section> tag. With the endterm attribute, you can customize the content which will be used to point to the reference. For instance, the following example refers to the same section as previously but using the figure itself:



This has been done using the following code:

```
<xref linkend="Figures" endterm="ADrawingusingtheFIGUREtag_42"/>
```

Use **<link>** tags to point to a web references (ProActive for instance)[2]. Here is the documentation source code for this link:

```
<link xmlns:xlink="http://www.w3.org/1999/xlink" xlink:href="http://proactive.inria.fr/"> a web references
(ProActive for instance)</link>
```

This is the way of writing links as describes in the DocBook documentation. However, the xlink namespace is already defined in the DocBook xsl files. So, it is possible to omit it. You may have already the <ulink> tags for other documents based on DocBook. This is no longer possible with the version 5 of DocBook: this tag has been removed.

Use **<citation>** followed by an **<xref>** for citations. For example, see [BBC02] to learn on groups. All the biblio entries should be put in biblio.xml. This citation has been written using the following piece of code:

```
<citation><xref linkend="BBC02" endterm="BBC02.abbrev"/></citation>
```

12.3.6. Tables

The tag to use is **<table>**.

Name	Hits
Bob	5
Mike	8
Jude	3

Table 12.1. This is an example table

Here is documentation source code for this table:

[2] http://proactive.inria.fr/

```
<table>

  <title xml:id="Thisisanexampletable_42">This is an example table</title>

  <tgroup cols="2">
    <thead>
      <row>
        <entry><para>Name</para></entry>
        <entry><para>Hits</para></entry>
      </row>
    </thead>
    <tbody>
      <row>
        <entry><para>Bob</para></entry>
        <entry><para>5</para></entry>
      </row>

      <row>
        <entry><para>Mike</para></entry>
        <entry><para>8</para></entry>
      </row>

      <row>
        <entry><para>Jude</para></entry>
        <entry><para>3</para></entry>
      </row>
    </tbody>
  </tgroup>
</table>
```

12.4. DocBook limitations imposed

We restrict ourselves to a subset of docbook because we want a maintainable and uniform styled documentation. To achieve this goal, we require minimum learning investment from our ProActive developers, who are meant to be coding, not spending their time writing documentation. So you still want to add a fancy feature? Well, you can, as long as you describe how to use this new tag in this howto, and be extra careful with the pdf output.

There is a schema specifying which are the allowed tags. You can only use the tags that this dtd allows. If you want more freedom, refer to Section 12.5.5, “DocBook subset: the dtd”. You can use the following tags:

- part, appendix, chapter, sect[1-5], title, para, emphasis, xref, link
- table, figure, caption, informalfigure, informaltable
- itemizedlist and orderedlist, listitem
- example, programlisting, screen, and literal
- The others that you might come along, albeit less frequently, are citation, email, indexterm, inlinemediaobject, note, answer, question, subscript, superscript

12.5. Stylesheet Customization

Here are a few notes on how you should go about customizing the output. That means, changing how the pdf and html are written.

12.5.1. File hierarchy

The files for configuration are the following:

- **common.xsl** - This is where all the common specifications are made, i.e. those that go both in pdf and in html.
- **pdf.xsl** - This is where all the pdf specific customizations are made
- **html.xsl** - This is where most html specific customizations are made.
- **onehtml.xsl** and **chunkedhtml.xsl** - Specifics for html, the former on one page, "chunked", one file per chapter, for the latter.
- **main.css** - This is yet another extra layer on top of the html output.

You can find these files in the ProActive/doc/toolchain/xsl/ directory except for the last one which is located on the ProActive/doc/src/ directory.

12.5.2. What you can change

Basically, in the customization layers, you have full control (just do what you want). The only thing is that each block (template, variable...) should be described by a comment. That will help later users. As customization can get cryptic, make a special effort!

12.5.3. The Bible

The book you want to have with you is the following: "DocBook XSL: The Complete Guide", Third Edition, by Bob Stayton, online version at http://www.sagehill.net.

Have a look at the index if you just want to change a little something in the customization. Parse through it at least once if you intend to do some heavy editing. I have found everything I needed in this book, but sometimes in unexpected sections.

12.5.4. The XSL debugging nightmare

If you are editing the xsl stylesheets, and are having a hard time figuring out what's happening, don't panic! Use many messages to find out what the values of the variables are at a given time:

```
<xsl:message>
   <xsl:text> OK, in question.toc, id is </xsl:text> <xsl:value-of select="$id" />
</xsl:message>

<xsl:for-each select="./@*">
   <xsl:message>
      Attribute <xsl:value-of select="name(.)"/> = <xsl:value-of select="."/>
   </xsl:message>
</xsl:for-each>
```

You will very soon find that you still have to dig deeper into the templates, and they certainly are not easy to follow. Here's a little helper:

```
java -cp $CLASSPATH org.apache.xalan.xslt.Process -TT -xsl ...  -in ... -out ...
```

This uses the specified templates with the xsl file specified, but tracing every template called. Useful when you're wondering what's being called.

12.5.5. DocBook subset: the dtd

The dtd is the file detailing which are the allowed tags in our DocBook subset. Some tags have been removed, to make it easier to manage. Please refer to the file called ProActive/doc/src/docbook.dtd to know how much freedom you have been granted.

When you run the manual generation through the ant tasks, the xml is checked for validity. The message you should see is

```
[xml_validate] 1 file(s) have been successfully validated.
```

If you happen to modify the dtd, you should put also copy it on the web, on /proj/oasis/www/proactive/doc/dtd/$version/ or else the previous version one will always be used.

12.6. Ant targets for building the documentation

12.6.1. Javadoc ant targets

Ant targets for building javadoc documentation are the following ones:

```
doc.ProActive.doc.zips              Generate the ProActive Programming javadoc and manual zip archives
doc.ProActive.docs                  Generate the ProActive Programming javadoc, manual, and zip archives
doc.ProActive.javadoc.complete      Generate the ProActive Programming complete javadoc
doc.ProActive.javadoc.completeZip   Generate the ProActive Programming complete javadoc zip
doc.ProActive.javadoc.published     Generate the ProActive Programming published javadoc
doc.ProActive.javadoc.publishedZip  Generate the ProActive Programming published javadoc zip
```

12.6.2. Manual generation ant targets

Ant targets for building the ProActive documentation are the following ones:

```
doc.ProActive.manual                Generate all the ProActive Programming manuals  (html, single html, pdf)
doc.ProActive.manualHtml            Generate the ProActive Programming HTML manual
doc.ProActive.manualHtmlZip         Generate the ProActive Programming HTML manual zip
doc.ProActive.manualPdf             Generate the ProActive Programming PDF manual
doc.ProActive.manualSingleHtml      Generate the ProActive Programming single HTML manual
doc.ProActive.manualSingleHtmlZip   Generate the ProActive Programming single HTML manual zip
doc.ProActive.schemas               Build documentation for GCM schemas
```

The following picture describes the process of documentation generation:

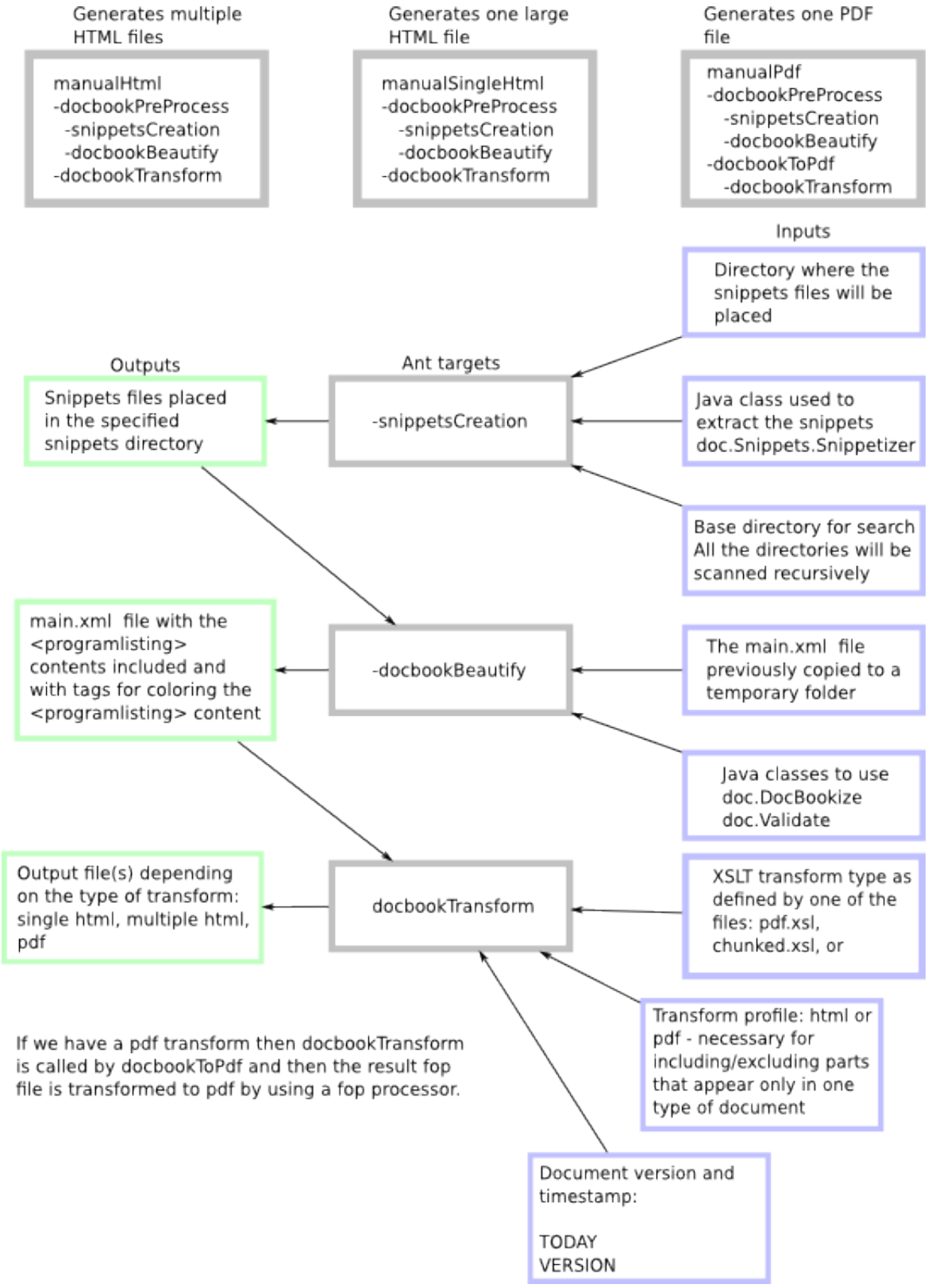


Figure 12.2. Main ant targets used in manual generation

Chapter 13. How to add a new FileTransfer CopyProtocol

FileTransfer protocols can be of two types: **external** or **internal**. Examples of external protocols are **scp** and **rcp** whereas those of internal protocols are **Unicore** and **Globus**.

Usually external FileTransfer happens before the deployment of the process. On the other hand, internal FileTransfer happens at the same time of the process deployment since the specific tools provided by the process are used. This implies that internal FileTransfer protocols cannot be used with other process, but the opposite is valid.

13.1. Adding external FileTransfer CopyProtocol

To add a new external FileTransfer CopyProtocol, follow the steps hereafter:

- **Implement the protocol class**. This is done inside the package: **org.objectweb.proactive.core.process.filetransfer;** by extending the abstract class **AbstractCopyProtocol**.
- **Add the name of the protocol:**. This is done by adding its name in the **ALLOWED_COPY_PROTOCOLS[]** array of the **FileTransferWorkshop** class.
- **Add the instantiation of the protocol class in the copyProtocolFactory(String name) method of FileTransferWorkshop**

Warning

Choosing the correct name for the protocol is simple, but must be done carefully. All names already in the array ALLOWED_COPY_PROTOCOLS are forbidden. This includes the name '**processDefault**', which is also forbidden. Some times, '**processDefault**' will correspond to an external FileTransfer protocol (e.g. ssh with scp), and some other times, it will correspond to an internal protocol.

13.2. Adding internal FileTransfer CopyProtocol

To add a new internal FileTransfer CopyProtocol, follow the steps hereafter:

- **Implement the method protected boolean internalFileTransferDefaultProtocol() inside the process class**. Note that this method will be called if the **processDefault** keyword is specified in the **XML Descriptor Process Section**. Therefore, this method usually must return true, so no other FileTransfer protocols will be tried.
- **Add the name of the protocol:**. This is done by adding its name in the **ALLOWED_COPY_PROTOCOLS[]** array of the **FileTransferWorkshop** class.

Note

When adding an internal FileTransfer protocol, **nothing** must be modified or added to the **copyProtocolFactory(){}** method.

Chapter 14. Adding a Fault-Tolerance Protocol

This documentation is a quick overview of how to add a new fault-tolerance protocol within ProActive.

14.1. Active Object side

Fault-tolerance mechanism in ProActive is mainly based on the org.objectweb.proactive.core.body.ft.protocols.FTManager class. This class contains several hooks that are called before and after the main actions of an active object (e.g. sending or receiving a message, serving a request, etc.)

For example, with the Pessimistic Message Logging protocol (PML), messages are logged just before the delivery of the message to the active object. Main methods for the FTManager of the PML protocol are then:

```
/**
 * Message must be synchronously logged before being delivered.
 * The LatestRcvdIndex table is updated
 * @see
 org.objectweb.proactive.core.body.ft.protocols.FTManager#onDeliverReply(org.objectweb.proactive.core.body.reply.Reply)
 */
@Override
public int onDeliverReply(Reply reply) {
   // if the ao is recovering, message are not logged
   if (!this.isRecovering) {
      try {
         // log the message
         this.storage.storeReply(this.ownerID, reply);
         // update latestIndex table
         this.updateLatestRvdIndexTable(reply);
      } catch (RemoteException e) {
         e.printStackTrace();
      }
   }
   return 0;
}

/**
 * Message must be synchronously logged before being delivered.
 * The LatestRcvdIndex table is updated
 * @see
 org.objectweb.proactive.core.body.ft.protocols.FTManager#onReceiveRequest(org.objectweb.proactive.core.body.request.Reque
 */
@Override
public int onDeliverRequest(Request request) {
   // if the ao is recovering, message are not logged
   if (!this.isRecovering) {
      try {
         // log the message
         this.storage.storeRequest(this.ownerID, request);
         // update latestIndex table
         this.updateLatestRvdIndexTable(request);
```

```
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
    return 0;
}
```

The local variable this.storage is a remote reference to the checkpoint server. The FTManager class contains a reference to each fault-tolerance server: fault-detector, checkpoint storage and localization server. Those reference are initialized during the creation of the active object.

A FTManager has to define also a beforeRestartAfterRecovery() method, which is called when an active object is recovered. This method usually restore the state of the active object so as to be consistent with the others active objects of the application.

For example, with the PML protocol, all the messages logged before the failure has to be delivered to the active object. The method beforeRestartAfterRecovery() thus looks like:

```
/**
 * Message logs are contained in the checkpoint info structure.
 */
@Override
public int beforeRestartAfterRecovery(CheckpointInfo ci, int inc) {
    // recovery mode: received message no longer logged
    this.isRecovering = true;

    //first must register incoming futures deserialized by the recovery thread
    this.owner.registerIncomingFutures();

    //get messages
    List<Reply> replies = ((CheckpointInfoPMLRB) ci).getReplyLog();
    List<Request> request = ((CheckpointInfoPMLRB) ci).getRequestLog();

    // deal with potential duplicata of request
    // duplicata of replies are not treated since they are automaticaly ignored.
    Request potentialDuplicata = request.get(request.size() - 1);
    this.potentialDuplicataSender = potentialDuplicata.getSourceBodyID();
    this.potentialDuplicataSequence = potentialDuplicata.getSequenceNumber();

    // add messages in the body context
    Iterator<Request> itRequest = request.iterator();
    BlockingRequestQueue queue = owner.getRequestQueue();

    while (itRequest.hasNext()) {
        queue.add((itRequest.next()));
    }

    // replies
    Iterator<Reply> itReplies = replies.iterator();
    FuturePool fp = owner.getFuturePool();
    try {
        while (itReplies.hasNext()) {
            Reply current = itReplies.next();
            fp.receiveFutureValue(current.getSequenceNumber(), current.getSourceBodyID(), current
                    .getResult(), current);
```

```
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    //    add pending request to reuqestQueue
    Request pendingRequest = ((CheckpointInfoPMLRB) ci).getPendingRequest();

    // pending request could be null
    if (pendingRequest != null) {
        queue.addToFront(pendingRequest);
    }

    // normal mode
    this.isRecovering = false;

    // enable communication
    this.owner.acceptCommunication();

    try {
        // update servers
        this.location.updateLocation(ownerID, owner.getRemoteAdapter());
        this.recovery.updateState(ownerID, RecoveryProcess.RUNNING);
    } catch (RemoteException e) {
        logger.error("Unable to connect with location server");
        e.printStackTrace();
    }

    this.checkpointTimer = System.currentTimeMillis();

    return 0;
}
```

The parameter ci is a org.objectweb.proactive.core.body.ft.checkpointing.CheckpointInfo . This object contains all the information linked to the checkpoint used for recovering the active object, and is used to restore its state. The programmer might define his own class implementing CheckpointInfo, to add needed information, depending on the protocol.

14.2. Server side

ProActive includes a global server that provides fault detection, active object localization, resource service and checkpoint storage. For developing a new fault-tolerance protocol, the programmer might specify the behavior of the checkpoint storage by extending the org.objectweb.proactive.core.body.ft.servers.storage.CheckpointServerImpl class. For example, only for the PML protocol and not for the CIC protocol, the checkpoint server must be able to synchronously log messages. The other parts of the server can be used directly.

To specify the recovery algorithm, the programmer has to extend the org.objectweb.proactive.core.body.ft.servers.recovery.RecoveryProcessImpl class. In the case of the CIC protocol, all the active object of the application must recover after one failure, while only the faulty process must restart with the PML protocol. This specific behavior is coded in the recovery process.

Chapter 15. MOP: Metaobject Protocol

15.1. Implementation: a Meta-Object Protocol

ProActive is built on top of a metaobject protocol (MOP) that permits reification of method invocation and constructor call. As this MOP is not limited to the implementation of our transparent remote objects library, it also provides an open framework for implementing powerful libraries for the Java language.

As for any other element of ProActive, this MOP is entirely written in Java and does not require any modification or extension to the Java Virtual Machine, as opposed to other metaobject protocols for Java {Kleinoeder96}. It makes extensive use of the Java Reflection API, thus requiring JDK 1.1 or higher. JDK 1.2 is required in order to suppress default Java language access control checks when executing reified non-public method or constructor calls.

15.2. Principles

If the programmer wants to implement a new metabehavior using our metaobject protocol, he (or she) has to write both a concrete (as opposed to abstract) class and an interface. The concrete class provides an implementation for the metabehavior he wants to achieve while the interface contains its declarative part.

The concrete class implements the **Proxy** interface and provides an implementation for the given behavior through the **reify** method:

```
public Object reify (MethodCall c) throws Throwable;
```

This method takes a reified call as a parameter and returns the value returned by the execution of this reified call. Automatic wrapping and unwrapping of primitive types is provided. If the execution of the call completes abruptly by throwing an exception, it is propagated to the calling method, just as if the call had not been reified.

The interface that holds the declarative part of the metabehavior has to be a subinterface of Reflect (the root interface for all metabehaviors implemented using ProActive). The purpose of this interface is to declare the name of the proxy class that implements the given behavior. Then, any instance of a class implementing this interface will be automatically created with a proxy that implements this behavior, provided that this instance is not created using the standard new keyword but rather through a special static method: MOP.newInstance. This is the only required modification to the application code. Another static method, MOP.newWrapper, adds a proxy to an already-existing object. The turnActive function of ProActive, for example, is implemented through this feature.

15.3. Example of a different metabehavior: EchoProxy

Here's the implementation of a very simple yet useful metabehavior: for each reified call, the name of the invoked method is printed out on the standard output stream and the call is then executed. This may be a starting point for building debugging or profiling environments.

```
class EchoProxy extends Object implements Proxy {
  // here are constructor and variables declaration
  // [...]
  public Object reify (MethodCall c) throws Throwable {
     System.out.println (c.getName());
     return c.execute (targetObject);
  }
}
```

```
interface Echo extends Reflect {
  public String PROXY_CLASS= 'EchoProxy';
}
```

15.3.1. Instantiating with the metabehavior

Instantiating an object of any class with this metabehavior can be done in three different ways: instantiation-based, class-based or object-based. Let's say we want to instantiate a Vector object with an Echo behavior.

- Standard Java code would be:

```
Vector v = new Vector(3);
```

- ProActive code, with instantiation-based declaration of the metabehavior (the last parameter is null because we do not have any additional parameter to pass to the proxy):

```
Object[] params = {new Integer (3)};
Vector v = (Vector) MOP.newInstance('Vector', params, 'EchoProxy', null);
```

- with class-based declaration:

```
public class MyVector extends Vector implements Echo {};

Object[] params = {new Integer (3)} ;
Vector v = (Vector) MOP.newInstance('Vector', params, null);
```

- with object-based declaration:

```
Vector v = new Vector (3);
v = (Vector) MOP.newWrapper('EchoProxy',v);
```

This is the only way to give a metabehavior to an object that is created in a place where we cannot edit source code. A typical example could be an object returned by a method that is part of an API distributed as a JAR file, without source code. Please note that, when using newWrapper, the invocation of the constructor of the class Vector is not reified.

15.4. The Reflect interface

All the interfaces used for declaring **metabehaviors** inherit directly or indirectly from Reflect. This leads to a hierarchy of metabehaviors such as shown in the figure below:

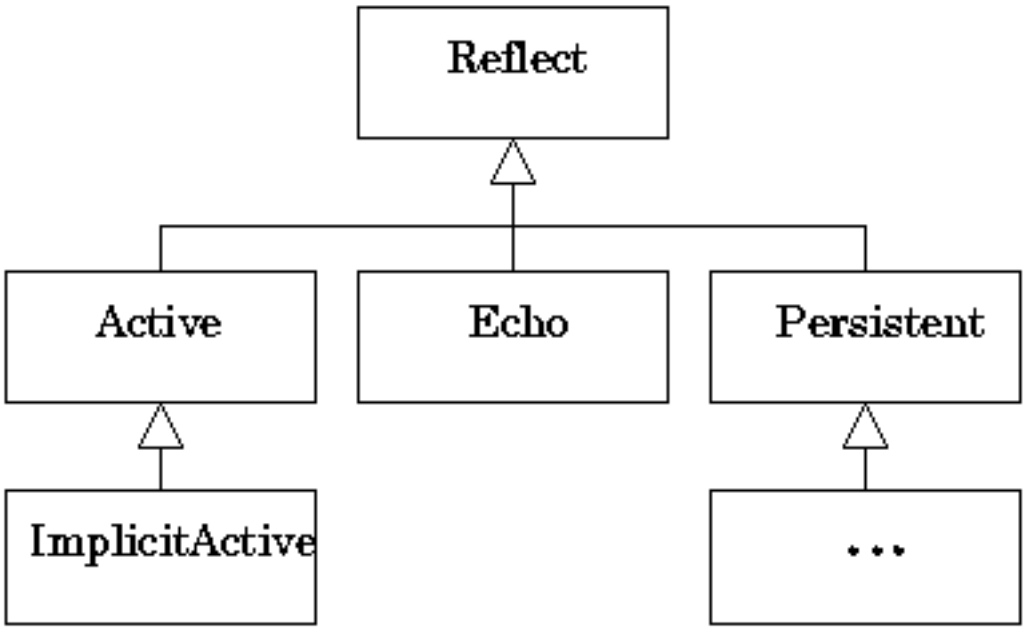


Figure 15.1. Metabehavior hierarchy

Note that ImplicitActive inherits from Active to highlight the fact that implicit synchronization somewhere always relies on some hidden explicit mechanism. Interfaces inheriting from Reflect can thus be logically grouped and assembled using multiple inheritance in order to build new metabehaviors out of existing ones.

15.5. Limitations

Due to its commitment to be a 100% Java library, the MOP has a few limitations:

- Calls sent to instances of final classes (which includes all arrays) cannot be reified.
- Primitive types cannot be reified because they are not instance of a standard class.
- Final classes (which includes all arrays) cannot be reified because they cannot be subclassed.

Part III. ProActive Extra Packages

Table of Contents

Chapter 16. Branch and Bound API

16.1. Overview

The Branch and Bound (BnB) consists in an algorithmic technique for exploring a solution tree from which returns the optimal solution.

The main goal of this BnB API is to provide a set of tools for helping the developers to parallelize his BnB problem implementation.

The main features are:

- Hiding computation distribution.
- Dynamic task splitting.
- Automatic solution gathering.
- Task communications for broadcasting the best current solution.
- Different behaviors for task allocation, provided by the API or on your own.
- Open API for extensions.

16.2. The Model Architecture

The following figure shows the API architecture:

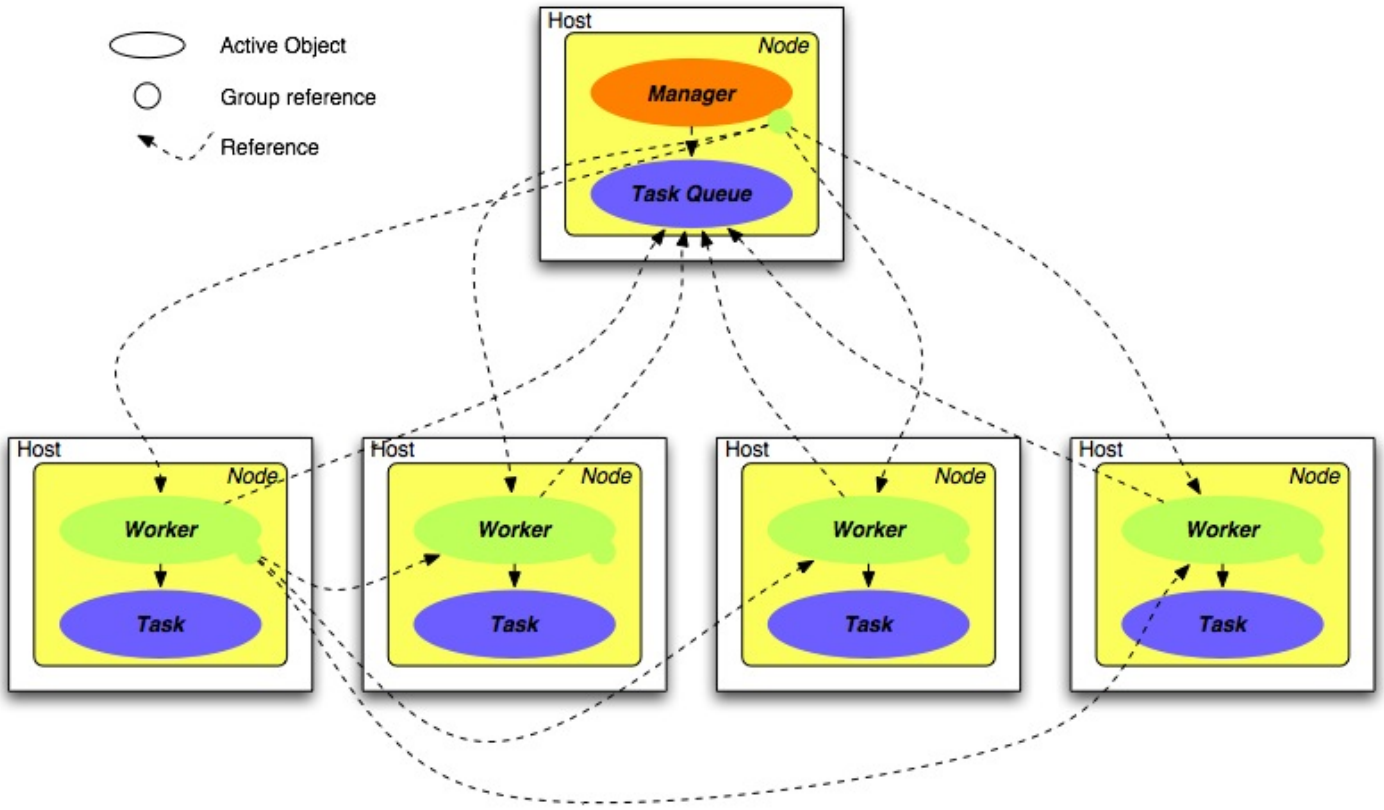


Figure 16.1. The API architecture

The API active objects are:

- **Manager**: the main point of the API. It is the master for deploying and managing Workers. Also, it attributes Tasks to free workers. The Tasks are provided by the Task Queue.
- **Task Queue**: provides Task in a specific order to the Manager.
- **Worker**: broadcasts the solution to all Tasks, and provides the API environment to the Tasks.
- **Task**: the user code to compute.

All Workers have a group reference on all the others. The figure hereafter shows step by step how a Task can share a new better solution with all others:

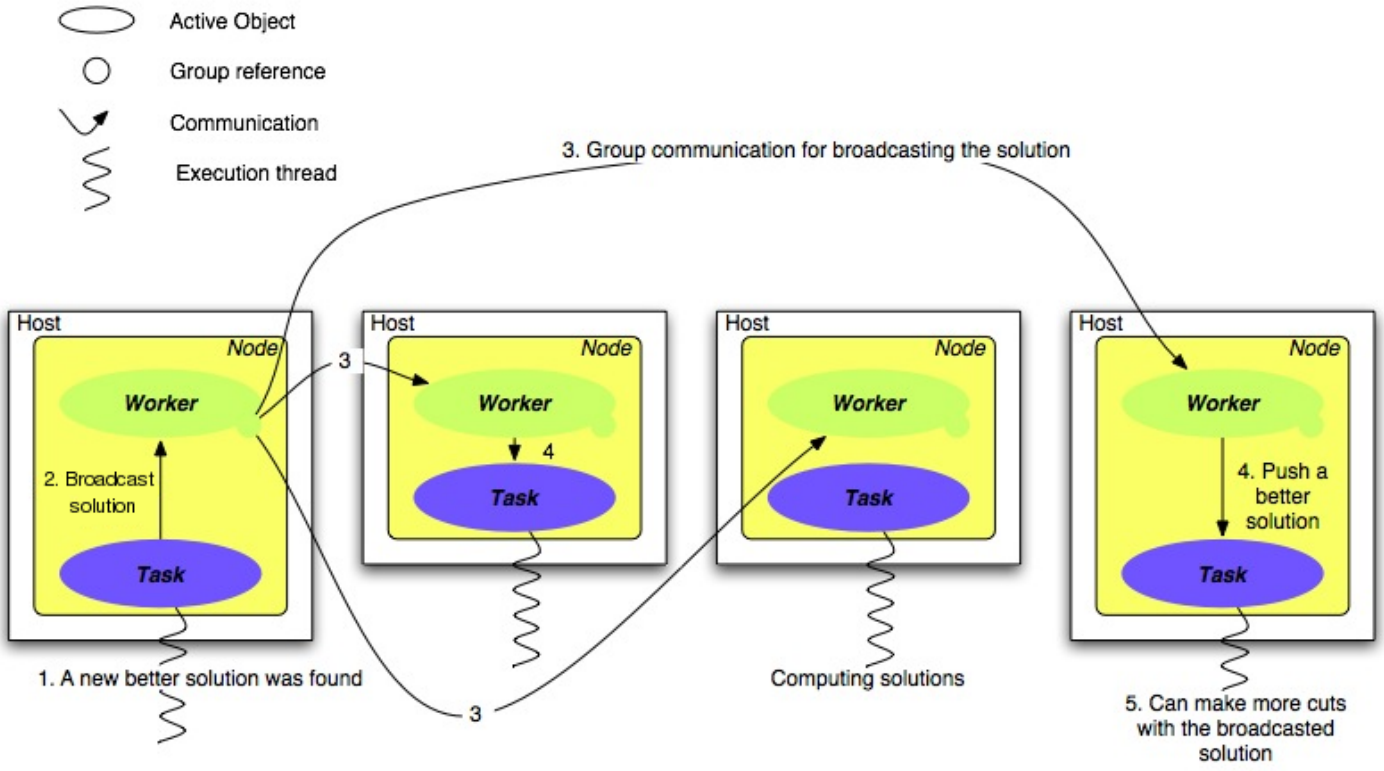


Figure 16.2. Broadcasting a new solution

Finally, here is the order execution of methods:

1. rootTask.initLowerBound(); // compute a first lower bound
2. rootTask.initUpperBound(); // compute a first upper bound
3. Vector splitted = rootTask.split(); // generate a set of tasks
4. for i in splitted do in parallel

   splitted[i].initLowerBound();

   splitted[i].initUpperBound();

   Result ri = splitted.execute()
5. Result final = rootTask.gather(Result[] ri); // gather all result

Keep in mind that is only 'initLower/UpperBound' and 'split' methods are called on the root task. The 'execute' method is called on the root task's splitted task.

16.3. The API Details

16.3.1. The Task Description

The **Task** object is located in this followed package:

```
org.objectweb.proactive.extra.branchnbound.core
```

All methods are described bellow:

16.3.1.1. public Result execute()

It is the place where the user has to put his code for solving a part and/or the totality of his BnB problem. There are 2 main usages of it. The first one consists in dividing the task and returning no result. The second is to try to improve the best solution.

16.3.1.2. public Vector<Task> split()

This is for helping the user when he wants to divide a task. In a future work we have planned to use this method in an automatic way.

16.3.1.3. public void initLowerBound()

Initialize a lower bound local to the task.

16.3.1.4. public void initUpperBound()

Initialize a upper bound local to the task.

16.3.1.5. public Result gather(Result[] results)

The default behavior is to return the smallest Result gave by the compareTo method. That's why it is also recommended to override the **compareTo(Object)** method.

Some instance variables are provided by the API to help the user for keeping a code clear. See next their descriptions:

```
protected Result initLowerBound; // to store the lower bound
protected Result initUpperBound; // to store the upper bound
protected Object bestKnownSolution; // setted automaticaly by the API with the best current solution
protected Worker worker; // to interact with the API (see after)
```

From the Task, specially within the execute() method, the user has to interact with the API for sending to the task queue sub-tasks resulting from a split call, or for broadcasting to other tasks a new better solution, etc.

The way to do that is to use the **worker** instance variable:

- Broadcasting a new better solution to all the other workers:

  ```
  this.worker.setBestCurrentResult(newBetterSolution);
  ```
- Sending a set of sub-tasks for computing:

  ```
  this.worker.sendSubTasksToTheManager(subTaskVector);
  ```
- For a smarter split, checking that the task queue needs more tasks:

  ```
  BooleanWrapper workersAvailable = this.worker.isHungry();
  ```

16.3.2. The Task Queue Description

The Task Queue manages the task allocation. The main functions are: providing tasks in a special order and keeping results back.

For the moment, there are 2 different queue types provided by the API:

- **BasicQueueImpl:** provides tasks in FIFO order.
- **LargerQueueImpl**: provides tasks in a larger order, as Breadth First Search algorithm.

By extending the **TaskQueue** abstract class, you can use a specialized task allocator for your need.

16.3.3. The ProActiveBranchNBound Description

Finally, it is the main entry point for starting and controlling your computation.

```
Task task = new YourTask(someArguments);
Manager manager =  ProActiveBranchNBound.newBnB(task,
                   nodes,
                   LargerQueueImpl.class.getName());
Result futureResult = manager.start(); // this call is asynchronous
```

Use the constructor ProActiveBranchNBound.newBnB(Task, VirtualNode[], String) and do not activate virtual nodes

This method provides a faster deployment and active objects creation way. Communications between workers are also optimized by a hierarchic group based on the array of virtual nodes. That means when it is possible, define a virtual node by clusters.

16.4. An Example: FlowShop

This example solves the permutation flowshop scheduling problem, with the monoobjective case. The main objective is to minimized the overall completion time for all the jobs, i.e. makespan. A flowshop problem can be represented as a set of n jobs; this jobs have to be scheduled on a set of m machines. Each jobs is defined by a set of m distinct operations. The goal consists to determine the sequence used for all machines to execute operations.

The algorithm used to find the best solution, tests all permutations and try to cut bad branches is the following one:

First, the **Flowshop Task**:

```
import org.objectweb.proactive.api.PAActiveObject;
import org.objectweb.proactive.extra.branchnbound.core.Result;
import org.objectweb.proactive.extra.branchnbound.core.Task;
import org.objectweb.proactive.extra.branchnbound.core.exception.NoResultsException;


/**
 * A Task for the FlowShop problem.
 *
 * A FlowShopTask can split the problem in subproblem and compute its best
 * makespan. FlowShopTask extends Task, this allow to take advantage of the
 * branch and bound api from ProActive, which manage task comunication and
 * repartition on the available nodes.
 *
 * @author The ProActive Team
 */
public class FlowShopTask extends Task {
   //in ms
   private static final long MAX_TIME_TO_SPLIT = 120000; // 2'
   private FlowShop fs;
```

```
private FlowShopResult fsr;

/**
 * The permutation with we work
 */
private int[] currentPerm;

/**
 * The best know permutation and its makespan
 */

// private int[] bestPerm;
// private long bestMakespan;
/**
 * The last permutation we must explore.
 */
private int[] lastPerm;

/**
 * The depth tree where we tests all permutations
 */
private int depth;

/**
 * for benchmark
 */
private boolean com;
private boolean randomInit;

/**
 *
 */
private long lowerBound;
private long upperBound;
private Result r;

public FlowShopTask() {
   // the empty no args constructor for ProActive
}

/**
 * Contruct a Task which search solution for all permutations to the
 * Flowshop problem. Use it to create the root Task.
 *
 * @param fs the description of the Flowshop problem
 * @param com for bench
 * @param randomInit for bench
 */
public FlowShopTask(FlowShop fs, long lowerBound, long upperBound, boolean com, boolean randomInit) {
   this(fs, lowerBound, upperBound, null, null, 0, com, randomInit);
   currentPerm = new int[fs.jobs.length];
   for (int i = 0; i < currentPerm.length; i++) {
      currentPerm[i] = i;
   }
```

```
    }

    /**
     * @param fs
     * @param lowerBound
     * @param upperBound
     * @param currentPerm
     * @param lastPerm
     * @param depth
     * @param com
     * @param randomInit
     */
    public FlowShopTask(FlowShop fs, long lowerBound, long upperBound, int[] currentPerm, int[] lastPerm,
            int depth, boolean com, boolean randomInit) {
        this.fs = fs;
        this.fsr = new FlowShopResult();
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.currentPerm = (currentPerm == null) ? null : (int[]) currentPerm.clone();
        this.lastPerm = (lastPerm == null) ? null : (int[]) lastPerm.clone();
        this.depth = depth;
        this.com = com;
        this.randomInit = randomInit;
        this.r = new Result();
    }

}
```

Now, implement all Task abstract methods.

Computation **bound** methods:

```
@Override
public void initLowerBound() {
    if (lowerBound == -1) {
        lowerBound = FlowShop.computeLowerBound(this.fs);
        Main.logger.info("We compute a lower bound: " + lowerBound);
    }
}

@Override
public void initUpperBound() {
    int[] randomPerm = currentPerm.clone();
    for (int i = depth + 1; i < randomPerm.length; i++) {
        int randomI = (int) (i + (Math.random() * (randomPerm.length - (i + 1))));
        int tmp = randomPerm[i];
        randomPerm[i] = randomPerm[randomI];
        randomPerm[randomI] = tmp;
    }
    Main.logger.info("initUpperBound => " +
        (randomInit ? ("random Perm : " + Permutation.string(randomPerm) + " her makespan " + FlowShop
                .computeMakespan(fs, randomPerm)) : (" non random Perm " +
            Permutation.string(currentPerm) + FlowShop.computeMakespan(fs, currentPerm))));
    fsr.makespan = randomInit ? FlowShop.computeMakespan(fs, randomPerm) : FlowShop.computeMakespan(fs,
```

```
            currentPerm);
    fsr.permutation = randomInit ? randomPerm : (int[]) currentPerm.clone();
}
```

The **split** method:

```
/**
 * Split the root Task in subtask. Can be called by the method execute() if
 * we want to split again.
 *
 * @see org.objectweb.proactive.extra.branchnbound.core.Task#split()
 */
@Override
public Vector split() {
    int nbTasks = fs.jobs.length - depth;

    Vector tasks = new Vector(nbTasks);

    int[] perm = currentPerm.clone();
    int[] beginPerm = new int[perm.length];

    do {
        if ((lastPerm != null) && (Permutation.compareTo(perm, lastPerm) > 0)) {
            break;
        }
        System.arraycopy(perm, 0, beginPerm, 0, perm.length);

        Permutation.jumpPerm(perm, perm.length - (depth + 1));

        tasks.add(new FlowShopTask(fs, this.lowerBound, this.upperBound, beginPerm, perm, depth + 1, com,
            randomInit));
    } while (Permutation.nextPerm(perm) != null);
    if (tasks.size() != 0) {
        ((FlowShopTask) tasks.lastElement()).lastPerm = lastPerm;
    }
    Main.logger.info("We split in " + tasks.size() + " subtask at depth " + depth + " : " +
        Permutation.string(currentPerm) + ", " + Permutation.string(lastPerm));

    /*
     * Iterator i = tasks.iterator(); while (i.hasNext()) { Task t = (Task) i.next();
     * Main.logger.info(t); Main.logger.info(""); }
     */
    return tasks;
}
```

Then, the **execute** method:

```
/**
 * Explore all permutation between currentPerm and lastPerm. May decide
 * also to split in sub Task.
 *
 * @see org.objectweb.proactive.extra.branchnbound.core.Task#execute()
 */
```

```
@Override
public Result execute() {
   int[] timeMachine = new int[fs.nbMachine];
   long time = System.currentTimeMillis();
   long nbPerm = 1;

   //       int[] cutbacks = new int[fs.jobs.length];
   int nbLoop = 0;
   int theLastJobFixed = currentPerm[depth - 1];

   //       Main.logger.info("depth  " +Permutation.string(currentPerm));
   //CHANGE HERE THE DEPTH OF SPLIT
   boolean mustSplit = ((depth < 2) && ((currentPerm.length - depth) > 2)); //Why not ?

   if (com) {
      this.bestKnownSolution = fsr;
      r.setSolution(fsr);
      this.worker.setBestCurrentResult(r);
   } else {
      this.bestKnownSolution = fsr;
   }
   if (!mustSplit) {
      int[][] tmpPerm = new int[currentPerm.length][];
      for (int i = 0; i < tmpPerm.length; i++) {
         tmpPerm[i] = new int[i];
      }
      while ((FlowShopTask.nextPerm(currentPerm)) != null) {
         nbLoop++;
         if ((lastPerm != null) && (currentPerm[depth - 1] != theLastJobFixed)) {
            //                (Permutation.compareTo(currentPerm, lastPerm) >= 0)) {
            //                Main.logger.info("depth " + depth + "  cmp  " + Permutation.string(currentPerm) + " >= " +
Permutation.string(lastPerm));
            break;
         }
         int currentMakespan;

         if (com) {
            fsr.makespan = ((FlowShopResult) this.bestKnownSolution).makespan;
            fsr.permutation = ((FlowShopResult) this.bestKnownSolution).permutation;
         }

         if ((currentMakespan = FlowShopTask.computeConditionalMakespan(fs, currentPerm,
               ((FlowShopResult) this.bestKnownSolution).makespan, timeMachine)) < 0) {
            //bad branch
            int n = currentPerm.length + currentMakespan;
            FlowShopTask.jumpPerm(currentPerm, n, tmpPerm[n]);
            //                cutbacks[-currentMakespan - 1]++;
            if (nbLoop > 100000000) { // TODO verify
               if (((System.currentTimeMillis() - time) > MAX_TIME_TO_SPLIT) &&
                  worker.isHungry().getBooleanValue()) { // avoid too tasks
                  mustSplit = true;
                  nbPerm++;
                  break;
               } else {
```

```
                    nbLoop = 0;
                }
            }
        } else {
            // better branch than previous best
            if (com) {
                fsr.makespan = currentMakespan;
                System.arraycopy(currentPerm, 0, fsr.permutation, 0, currentPerm.length);
                r.setSolution(fsr);
                this.worker.setBestCurrentResult(r);
            } else {
                ((FlowShopResult) this.bestKnownSolution).makespan = currentMakespan;
                System.arraycopy(currentPerm, 0,
                        ((FlowShopResult) this.bestKnownSolution).permutation, 0, currentPerm.length);
            }
        }

        nbPerm++;
    }
  }
  time = System.currentTimeMillis() - time;

  if (mustSplit) {
    this.worker.sendSubTasksToTheManager(((FlowShopTask) PAActiveObject.getStubOnThis()).split());
  }

  Main.logger.info(" -- Explore " + nbPerm + " permutations in " + time + " ms\nBest makespan :" +
    ((FlowShopResult) this.bestKnownSolution).makespan + " with this permutation " +
    Permutation.string(((FlowShopResult) this.bestKnownSolution).permutation));
  //          + ". We have cut " + Permutation.string(cutbacks));
  ((FlowShopResult) this.bestKnownSolution).nbPermutationTested = nbPerm;
  ((FlowShopResult) this.bestKnownSolution).time = time;
  //       ((FlowShopResult) this.bestKnownResult).makespanCut = cutbacks;
  r.setSolution(bestKnownSolution);
  return r;
}
```

This example is available in a complete version in the Proactive/src/Example/org/objectweb/proactive/examples/flowshop/ directory.

16.5. Future Work

- An auto-dynamic task splitting mechanism.
- Providing more queues for task allocation.
- A new task interface for wrapping native code.

Chapter 17. Monte-Carlo API

17.1. Overview

Monte Carlo[1] methods belong to a class of computational algorithms that relies on repeated random sampling to compute their results. Monte Carlo methods are often used when simulating physical and mathematical systems. Because of their reliance on repeated computation and random or pseudo-random numbers, Monte Carlo methods are most suited for computer calculation. They tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm.

The goal of the ProActive Monte-Carlo API is to provide and easy to use API for running Monte-Carlo simulations in a distributed environment.

The main features are:

- Monte-Carlo simulations or other tasks can be run on remote workers. Tasks are defined by implementing an interface.
- It integrates SSJ[2], a distributed random generator written by Pierre l'Ecuyer. Each worker has its own independent stream of random numbers and there is a guaranty that two different workers, will generate sequences that won't overlap until a certain number of experiences (2^{127}).
- It is based on ProActive Master-Worker API as the underlying framework (see Chapter 13. Master-Worker API[3] for more information).
- Deployment of the worker infrastructure is done through GCM descriptors (see Chapter 21. ProActive Grid Component Model Deployment[4] for more information).
- A small set of processes, taken from the financial field, are included as examples.

17.2. API

The Monte-Carlo API is located in the org.objectweb.proactive.extra.montecarlo.example package.

17.2.1. Main Class

The entry point of the Monte-Carlo API is the PAMonteCarlo class. Its main constructor allows you to deploy the monte-carlo framework, using ProActive GCM deployment framework. The first argument is the URL of the GCMApplication file which will be used. The next argument is the virtual node name corresponding to the Workers infrastructure of machines. The third argument (optional) is a virtual node name that will correspond to a remote deployment of the master. And the final argument is to allow changing the default Random Number Generator from the SSJ package.

```
/**
 * Initialize the Monte-Carlo toolkit by giving a descriptor and two virtual node names.
 * Workers will be instanciated on resources created by the first one.
 * The second one should create one single resource. The master will be deployed on that resource
 * The last parameter is a Class object holding the definition of the RandomStream to use.
 * By default, the generator in use is the MRG32k3a one.
 *
 * @param descriptorURL url of a descriptor
 * @param masterVNName virtual node name corresponding to the master
 * @param workersVNName virtual node name corresponding to workers
```

[1] http://en.wikipedia.org/wiki/Monte_Carlo_method
[2] http://www.iro.umontreal.ca/~simardr/ssj/
[3] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/AdvancedFeatures/pdf/../../ReferenceManual/multiple_html/MasterWorker.html
[4] file:///home/hudson/workdir/workspace/ProActive_Documentation/compile/../doc/built/Programming/AdvancedFeatures/pdf/../../multiple_html/ReferenceManual/GCMDeployment.html

```
 * @param randomStreamClass Random Number Generator class that workers will be using
 * @throws ProActiveException
 */
public PAMonteCarlo(URL descriptorURL, String workersVNName, String masterVNName,
      Class<?> randomStreamClass) throws ProActiveException
```

The run method of the PAMonteCarlo class is the starting point of the computation. It runs a single task, called a top-level engine task. This task, while running, will most likely generate new subtasks. The different tasks existing in the framework are described inSection 17.2.2, “Tasks”.

```
/**
 * Runs the simulation by giving the definition of a top-level task.
 * This task is not directly a Monte Carlo simulation, but should rather contain the code of your general algorithm.
 * The top-level task can submit nested general tasks or nested Monte-Carlo simulations (Experience Set).
 * @param toplevelTask top-level task
 * @return the result of the top-level task
 * @throws TaskException if an exception occured during the execution of the top-level task.
 */
public T run(EngineTask<T> toplevelTask) throws TaskException
```

Finally, the terminate method will shut down the framework and every remote JVM created during the initialization phase.

```
/**
 * Terminates the Monte-Carlo toolkit and free created resources.
 */
public void terminate()
```

17.2.2. Tasks

This section describes the different types of tasks and theirs purposes

17.2.2.1. Engine Task

An Engine Task is a general-purpose task, used when any computation which doesn't include running Monte-Carlo simulation needs to be done. The top-level task, submitted to the PAMonteCarlo class is an engine task. An engine task can, through the access to two interfaces (Executor and Simulator), spawn children engine tasks or simulation sets.

```
public interface EngineTask<T extends Serializable> extends Serializable {

   /**
    * Defines a general purpose task which can be run by the Monte-Carlo framework
    *
    * @param simulator gives the possibility to schedule children simulation sets
    * @param executor  gives the possibility to schedule children engine tasks
    * @return the result of this task
    */
   public T run(Simulator simulator, Executor executor);

}
```

17.2.2.2. Simulation Set

A Simulation Set is a specific task for running Monte-Carlo simulations. It is given a random number generator and can use it to generate random double numbers which are by default in the uniform distribution. Classes from the SSJ package can be used to convert this distribution to any wanted one. Unlike the Engine Task, the Simulation Set cannot spawn other tasks and is therefore a "terminal task".

```
public interface SimulationSet<T extends Serializable> extends Serializable {

    /**
     * Defines a Monte-Carlo set of successive experiences, a Random generator is given as a parameter
     *
     * A list of double values is expected as output, result of the successive experiences.
     * These experiences can be independant or correlated, this choice is left to the user inside the implementation of this
 method.
     *
     * @param rng random number generator
     * @return a list of double values
     */
    T simulate(final RandomStream rng);
}
```

17.2.2.3. Simuation Set Post Process

A simulation set task will return as output a huge number of values (e.g. arrays of size 10000 or more). In general, these results are not the result of the general algorithm which uses the Monte-Carlo method. Therefore, a post-treatment needs to be done on this big array. Of course, this post-treatment could be done by an engine task that spawned the simulation set task, but in that case, the big array would be transferred by the network from the Worker which handles the simulation set to the Worker which handles the parent engine task. The Simulation Set Post Process avoids this transfer, and allows to do some computations on the same machine which generated the Monte-Carlo simulations.

```
public interface SimulationSetPostProcess<T extends Serializable, R extends Serializable> {

    /**
     * Defines a post-processing of results received from a simulation set
     * @param experiencesResults results receive from a Simulation Set task
     * @return the result of the post processing
     */
    R postprocess(T experiencesResults);
}
```

17.2.3. Examples

17.2.3.1. Basic Simulations Processes

Some basic Simulation Sets are provided as an example. These basic processes are widely used in the financial risk management theory. An example of such a process is the Geometric Brownian Motion:

```
/**
 * Simulating geometric Brownian motion. This equation is the exact solution of the geometrix brownian motion SDE.
 * @param s0 Initial value at t=0 of geometric Brownian
 * @param mu Drift term
 * @param sigma Volatility
 * @param t time
 * @param N number of experiences
 */
public GeometricBrownianMotion(double s0, double mu, double sigma, double t, int N)
```

17.2.3.2. Basic Example Applications

Two basic example applications are provided. The first one computes PI using the Monte-Carlo method (it's only a theoretic approach as the method is very inefficient to compute PI at a good precision). The second example is an European Option Pricing from the financial

risk analysis world. Both examples can be found in the subpackage example of the monte-carlo package. A script exists to launch the European Option and is located at ProActive/examples/montecarlo

Bibliography

[ACC05] Isabelle Attali, Denis Caromel, and Arnaud Contes. *Deployment-based security for grid applications*. The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25. . LNCS. 2005. Springer Verlag.

[BBC02] Laurent Baduel, Francoise Baude, and Denis Caromel. *Efficient, Flexible, and Typed Group Communications in Java*. 28--36. Joint ACM Java Grande - ISCOPE 2002 Conference. Seattle. . 2002. ACM Press. ISBN 1-58113-559-8.

[BBC05] Laurent Baduel, Francoise Baude, and Denis Caromel. *Object-Oriented SPMD*. Proceedings of Cluster Computing and Grid. Cardiff, United Kingdom. . May 2005.

[BCDH05] Francoise Baude, Denis Caromel, Christian Delbe, and Ludovic Henrio. *A hybrid message logging-cic protocol for constrained checkpointability*. 644--653. Proceedings of EuroPar2005. Lisbon, Portugal. . LNCS. August-September 2005. Springer Verlag.

[BCHV00] Francoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssiere. *Communicating mobile active objects in java*. 633--643. http://www-sop.inria.fr/oasis/Julien.Vayssiere/publications/18230633.pdf. Proceedings of HPCN Europe 2000. . LNCS 1823. May 2000. Springer Verlag.

[BCM+02] Francoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssiere. *Interactive and descriptor-based deployment of object-oriented grid applications*. 93--102. http://www-sop.inria.fr/oasis/personnel/Julien.Vayssiere/publications/hpdc2002_vayssiere.pdf. Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing. Edinburgh, Scotland. . July 2002. IEEE Computer Society.

[BCM03] Francoise Baude, Denis Caromel, and Matthieu Morel. *From distributed objects to hierarchical grid components*. http://proactive.activeeon.com//userfiles/file/papers/HierarchicalGridComponents.pdf. International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November. Springer Verlag. . 2003. Lecture Notes in Computer Science, LNCS. ISBN ??.

[Car93] Denis Caromel. *Toward a method of object-oriented concurrent programming*. 90--102. http://citeseer.ist.psu.edu/300829.html. *Communications of the ACM*. 36. 9. 1993.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Object*. Springer Verlag. 2005.

[CHS04] Denis Caromel, Ludovic Henrio, and Bernard Serpette. *Asynchronous and deterministic objects*. 123--134. http://doi.acm.org/10.1145/964001.964012. Proceedings of the 31st ACM Symposium on Principles of Programming Languages. . 2004. ACM Press.

[CKV98a] Denis Caromel, W. Klauser, and Julien Vayssiere. *Towards seamless computing and metacomputing in java*. 1043--1061. http://proactive.inria.fr/doc/javallCPE.ps. Concurrency Practice and Experience. . Geoffrey C. Fox. 10, (11--13). September-November 1998. Wiley and Sons, Ltd..

[HCB04] Fabrice Huet, Denis Caromel, and Henri E. Bal. *A High Performance Java Middleware with a Real Application*. http://proactive.inria.fr/doc/sc2004.pdf. Proceedings of the Supercomputing conference. Pittsburgh, Pensylvania, USA. . November 2004.

[BCDH04] F. Baude, D. Caromel, C. Delbe, and L. Henrio. *A fault tolerance protocol for asp calculus : Design and proof*. http://www-sop.inria.fr/oasis/personnel/Christian.Delbe/publis/rr5246.pdf. Technical ReportRR-5246. INRIA. 2004.

[FKTT98] Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. *A security architecture for computational grids*. 83--92. http://citeseer.ist.psu.edu/foster98security.html. ACM Conference on Computer and Communications Security. . 1998.

[CDD06c] Denis Caromel, Christian Delbe, and Alexandre di Costanzo. *Peer-to-Peer and Fault-Tolerance: Towards Deployment Based Technical Services* . Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture . Paris, France. . January 2006.

[CCDMCompFrame06] Denis Caromel, Alexandre di Costanzo, Christian Delbe, and Matthieu Morel. *Dynamically-Fulfilled Application Constraints through Technical Services - Towards Flexible Component Deployments* . Proceedings of HPC-GECO/CompFrame 2006, HPC Grid programming Environments and COmponents - Component and Framework Technology in High-Performance and Scientific Computing . Paris, France. . June 2006. IEEE.

[CCMPARCO07] Denis Caromel, Alexandre di Costanzo, and Clement Mathieu. *Peer-to-Peer for Computational Grids: Mixing Clusters and Desktop Machines*. *Parallel Computing Journal on Large Scale Grid*. 2007.

[PhD-Morel] Matthieu Morel. *Components for Grid Computing* . http://www-sop.inria.fr/oasis/personnel/Matthieu.Morel/publis/phd_thesis_matthieu_morel.pdf. PhD thesis. University of Nice Sophia-Antipolis. 2006.

[FACS-06] I. \vCern\'a, P. Va\vrekov\'a, and B. Zimmerova. "Component Substitutability via Equivalencies of Component-Interaction Automata". To appear in ENTCS. 2006.

[JavaA05] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. "A Component Model for Architectural Programming". 2005.

[Fractal04] E. Bruneton, T. Coupaye, M. Leclercp, V. Quema, and J. Stefani. "An Open Component Model and Its Support in Java.". 2004.

[ifip05] Alessandro Coglio and Cordell Green. "A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Appplets". 2005.

[STSLib07] Fabricio Fernandes and Jean-Claude Royer. "The STSLIB Project: Towards a Formal Component Model Based on STS". To appear in ENTCS. 2007.

[InterfaceAutomata2001] Luca de Alfaro, Tom Henzinger. "Interface automata". 2001.

[CCM] OMG. "CORBA components, version 3". 2002.

[Plasil02] F. Plasil and S. Visnovsky. "Behavior Protocols for Software Components". *IEEE Transactions on Software Engineering*. 28. 2002.

[Reussnerm] Reussner, Ralf H.. "Enhanced Component Interfaces to Support Dynamic Adaption and Extension". IEEE. 2001.

[JKP05] P. Jezek, J. Kofron, F. Plasil. "Model Checking of Component Behavior Specification: A Real Life Experience". Electronic Notes in Theoretical Computer Science (ENTCS). 2005.

[MB01] V. Mencl and T. Bures. "Microcomponent-based component controllers: A foundation for component aspects". APSEC. . Dec. 2005. IEEE Computer Society.

[SPC01] L. Seinturier, N. Pessemier, and T. Coupaye. "AOKell: an Aspect-Oriented Implementation of the Fractal Specifications". 2005.

Index