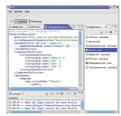


Version 3.2 - November 2006

Programming Reference Booklet

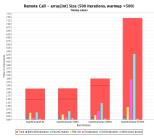


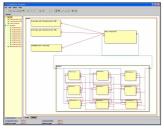




Graphical ant Textual Visualization

Application launcher





Application performance evaluation

Components GUI





ProActive is a Java library for **parallel**, **distributed**, and **concurrent** computing, also featuring **mobility** and **security** in a uniform framework. **ProActive** provides a comprehensive API and a graphical interface. The library is based on an Active Object pattern that is a uniform way to encapsulate:

- a remotely accessible object,
- a thread as an asynchronous activity.
- an actor with its own script,
- a server of incoming requests,
- a mobile and potentially secure entity,
- a component with server and client interfaces.

ProActive is only made of standard Java classes, and requires **no changes to the Java Virtual Machine**. Overall, it simplifies the programming of applications distributed over Local Area Network (LAN), Clusters, Intranet or Internet GRIDs.

ProActive interoperates with the following official or de facto standards:

- Web Service Exportation
- JMX Connector, JavaSpaces
- HTTP Transport, Jini, OSGi
- SSH, RSH, RMI/SSH Tunneling
- LSF, PBS, OAR, Sun Grid Engine
- Globus GT2, GT3, GT4
- sshGSI
- NorduGrid
- Unicore, EGEE gLite

Table of Contents

Main concepts and definitions	3
Main principles:	4
asynchronous method calls and implicit futures	4
Explicit Synchronization	4
Programming AO Activity and Services	
Reactive Active Object	
Service methods	
Active Object Creation	
Groups	
Explicit Group Synchronizations	
OO SPMD	
Migration	
Exceptions	
Components	
Web services	
Deployment	
File Transfer Deployment	
Peer-to-Peer Infrastructure	
Fault-Tolerance	
Security	
Branch and Bound API	
D. G. C. C. G. C.	. 10

2

ProActive 3.2 - Reference Booklet

Main concepts and definitions A remote object, with its own thread, receiving calls on Active Objects its public methods (AO) FIFO activity An AO, by default, executes the request it receives one after the other, in the order they were received Standard Java objects cannot be referenced from 2 AOs, ensured by deep-copy of constructor params, No-sharing method params, and results Method calls towards AOs are asynchronous Asynchrony The result of a non-void asynchronous method call The occurrence of a method call towards an AO Request Service The execution by an AO of a request Reply After a service, the result is sent back to the caller Wait-by-necessity Automatic wait upon the use of a still awaited future Automatic Transmission of futures and replies between AO and An AO moving from one JVM to another, computational Migration weak mobility: the AO decides to migrate; stack is lost A typed group of objects or AOs. Methods are called in Group parallel on all group members

_O	Component	Made of AOs, a component defines server and client interfaces
SING	Primitive Component	Directly made of Java code and AOs
COMPC	Composite Component	Contains other components (primitives or composites)
0	Parallel Component	A composite that is using groups to multicast calls to inner components

	Virtual Node (VN)	An abstraction (a string) representing where to locate AOs at creation
	Deployment descriptor	An XML file where a mapping VN → JVMs → Machine is specified
	Node	The result of a mapping VN → JVMs. After activation, a
9	Nouc	VN contains a set of nodes living in a set of JVMs
ΑI	P2P	A P2P network of self-organized JVMs, on which to
9		deploy applications
DEPLOYING	Fault-Tolerance	Applications can be turned fault-tolerant simply by modifying the deployment descriptor
	Security	X.509 Authentication, Integrity, and Confidentiality
		defined at deployment in an XML
	IC2D	Interactive Control and Debugging of Distribution: a
		Graphical environment for monitoring and steering
		Grid applications

Main principles: Asynchronous Method Calls and Implicit Futures

```
A a = (A) ProActive.newActive("A", params, node);

// Create an active Object of type A in the JVM specified by Node
a.foo (param);

// A one way typed asynchronous communication towards the (remote)

// AO a. A request is sent to a.

v = a.bar (param);

// A typed asynchronous communication with result.

// v is first an awaited Future, to be transparently filled up

// after service of the request, and the reception of a reply

v.gee (param);

// Use of the result of an asynchronous call.

// If v is still an awaited future, it triggers an automatic

// wait: Wait-by-necessity
```

Explicit Synchronization

```
boolean isAwaited(Object);
   // Returns True if the object is still an awaited Future
void waitFor(Object);
   // Blocks until the object is no longer awaited
   // The object is a future
void waitForAll(Vector);
   // Blocks until all the objects in Vector are no longer awaited
int waitForAny(Vector);
   // Blocks until one of the objects in Vector is no longer awaited.
   // Returns the index of the available future.
```

Programming AO Activity and Services

When an AO must implement an activity that is not FIFO, the RunActive interface has to be implemented: it specifies the AO behaviour in the method named runActivity(Body body).

Example:

4

3

Two other interfaces can also be specified:

```
Interface InitActive
void initActivity(Body body)
   // Initializes the activity of the active object.
   // Not called in case of restart after migration.
   // Called before runActivity() method, and only once.

Interface EndActive
void endActivity(Body body)
   // Finalizes the active object after the activity stops by itself.
   // Called after the execution of runActivity() method, and only
   // once. Not called before a migration.
```

Reactive Active Object

Even when an AO is busy doing its own work, it can remain reactive to external events (method calls). One just has to program non-blocking services to take into account external inputs.

It also allows one to specify explicit termination of AOs (there is currently no Distributed Garbage Collector). Of course, the reactivity is up to the length of going around the loop. Similar techniques can be used to start, suspend, restart, and stop AOs.

Service methods

A service Method selects a given request amongst the pending calls, and executes it. Those methods are to be used when a FIFO service is not appropriate, when a RunActivity method is programmed.

Non-blocking services: returns immediately if no matching request is pending

```
void serveOldest();
    // Serves the oldest request in the request queue
```

ProActive 3.2 - Reference Booklet

6

5

```
void serveOldest(String methodName)
// Serves the oldest request aimed at a method of name methodName
void serveOldest(RequestFilter requestFilter)
           Serves the oldest request matching the criteria given by the
 Blocking services: waits until a matching request can be served
void blockingServeOldest();
// Serves the oldest request in the request queue

void blockingServeOldest(String methodName)

// Serves the oldest request aimed at a method of name methodName
// octres the ordest request aimed at a method of name methodNar
void blockingServeOldest(RequestFilter requestFilter)
// Serves the oldest request matching the criteria given by the
// filter
 Blocking timed services: wait a matching request at most a time given in ms
void blockingServeoldest (long timeout)
   // Serves the oldest request in the request queue.
   // Returns after timeout (in ms) if no request is available
void blockingServeoldest(String methodName, long timeout)
   // Serves the oldest request aimed at a method of name methodName
   // Returns after timeout (in ms) if no request is available
void blockingServeoldest(RequestFilter requestFilter)
   // Serves the oldest request matching the criteria given by the
      // Serves the oldest request matching the criteria given by the
      // filter
 Waiting primitives:
void waitForRequest();
           Wait until a request is available or until the body terminates
void waitForRequest(String methodName);
    // Wait until a request is available on the given method name,
    // or until the body terminates
 Others:
void fifoServing();
        '/ Start a FIFO service policy. Call does not return. In case of 
// a migration, a new runActivity() is started on the new site.
      // Invoke a LIFO policy. Call does not return. In case of // a migration, a new runActivity() will be started on the new
// a migration, a new luminitivity() will be started // site.

void serveYoungest()

// Serves the youngest request in the request queue void flushAll()
```

ProActive 3.2 - Reference Booklet

// Removes all requests in the pending queue

```
Active Object Creation
```

Groups

A typed collection of active objects on which calls are performed in parallel (hiding latency), and asynchronously (returning a group of futures).

```
A ga = (A) ProActiveGroup.newGroup( "A", params, nodes);

// Creates at once a group of AO of type "A" in the JVMs specified
// by nodes. ga is a Typed Group of type "A".
// The number of AO created matches the number of param arrays.
// Nodes can be a Virtual Node defined in an XML descriptor
ga.foo(...);
// A general group communication without result.
// A request to foo is sent in parallel to AOs in group ga
V gv = ga.bar(...);
// A general group communication with a result.
// gv is a typed group of "V", which is first a group
// of awaited Futures, to be filled up asynchronously
gv.gee (...);
// Use of the result of an asynchronous group call. It is also a
// collective operation: gee method is called in parallel on each
// object in group.
// Wait-by-necessity occurs when results are awaited
Group ag = ProActiveGroup.getGroup(ga);
// Get the group representation of a typed group
ag.add(o);
// Add o in the group ag. o can be a standard Java object or
// an AO, and in any case must be of a compatible type
ag.remove(index)
// Removes the object at the specified index
A ga2 = (A) ag.getGroupByType();
// Returns the typed view of a group
// communication is sent to all as it is (deep copy of the group).
// When set to scatter, upon a group call (ga.foo(g)) such a
// scatter parameter is dispatched in a round robing fashion to
// AOs in the target group, e.g. upon ga.foo(g)
```

```
void unsetScatterGroup(g);
   // Get back to the default: entire group transmission in all group
   // communications, e.g. upon ga.foo(g)
```

Explicit Group Synchronizations

Methods to wait for the availability of all results of a group call, or the first one(s) to be available. Methods are both in interface Group, and static in class ProActiveGroup.

```
boolean ProActiveGroup.allAwaited (Object);
    // Returns true if object is a group and all members are still
    // awaited
boolean ProActiveGroup.allArrived (Object);
    // Returns False only if at least one member is still awaited
void ProActiveGroup.waitAll (Object);
    // Wait for all the members in group to arrive (all no longer
    // awaited)
void ProActiveGroup.waitN (Object, int nb);
    // Wait for at least nb members in group to arrive
int ProActiveGroup.waitOneAndGetIndex (Object);
    // Waits for at least one member to arrived, and returns its index
```

OO SPMD

A group in which each group member has a proxy to the group it belongs to. Typically used in applications with sub-domain decomposition, numerical SPMD (Simple Program, Multiple Data), etc.

```
A spmdGroup = (A) ProSPMD.newSPMDGroup("A", params, nodes);

// Creates an SPMD group and creates all members with params on
// the nodes.

// An SPMD group is a typed group in which every member has a
// reference to the others (the SPMD group itself).

A mySpmdGroup = (A) ProSPMD.getSPMDGroup();

// Returns the SPMD group of the activity.
int rank = ProSPMD.getMyRank();

// Returns the rank of the activity in its SPMD group.

ProSPMD.barrier("barrierID");

// Blocks the activity (after the end of the current service)
// until all other members of the SPMD group invoke the same
// barrier. Three barriers are available: total barrier,
// neighbors based barrier and method based barrier.
```

Migratior

Computational mobility: an active object changes of JVM at execution. Migration is weak: stack is to be lost, and migrateTo primitive is static.

```
void migrateTo(Object o);
   // Migrate the current AO to the same JVM as the AO
void void migrateTo(String nodeURL);
   // Migrate the current AO to JVM given by the node URL
int void migrateTo(Node node);
   // Migrate the current AO to JVM given by the node
```

To initiate the migration of an object from outside, define a public method, that upon service will call migrateTo primitive:

```
public void moveTo(Object o) {
    try{
        ProActive.migrateTo(o);
    } catch (Exception e) {
        e.printStackTrace();
        logger.info("Cannot migrate.");
    }
}

void onDeparture(String MethodName);
    // Specification of a method to execute before migration
void onArrival(String MethodName);
    // Specification of a method to execute after migration, upon the
    // arrival in a new JVM
void setMigrationStrategy(MigrationStrategy ms);
    // Specifies a migration itinerary
void migrationStrategy.add(Destination d);
    // Adds a JVM destination to an itinerary
void migrationStrategy.remove(Destination d);
    // Remove a JVM destination in an itinerary
```

Exceptions

ProActive has two exception mechanisms because there are two kinds of exceptions: functional and non-functional ones. First an example with functional exceptions that permits asynchronous calls with exceptions:

```
ProActive.tryWithCatch(MyException.class);// Just before the try
try { // Some asynchronous calls with potential MyException
ProActive.endTryWithCatch();// At the end of the try
} catch (MyException e) {
    // ...
} finally {
    ProActive.removeTryWithCatch();// At the beginning of the finally
}
```

Non-functional exceptions make use of handlers. They can be added on a JVM and on an AO. It's also possible to specify an exception type to handle.

ProActive 3.2 - Reference Booklet

10

q

ProActive 3.2 - Reference Booklet

http://localhost:8080/servlet/wsdl?id=a for a service whose name is "a" and whose id is deployed on a web server which location is http://localhost:8080.

```
A a = (A) ProActive.newActive("A", new Object []{});

// Constructs an active object
String [] methods = new String [] {"foo", "bar"};

// A String array containing the exported methods
ProActive.exposeAsWebService(a, "http://localhost:8080","a",methods);

// Export the active object as a web service
ProActive.unExposeAsWebService("a", "http://localhost:8080");

// Undeploy the service "a" on the web server located at

// http://localhost:8080
```

Deployment

Virtual Nodes (VN) allow one to specify the location where to create AOs. A VN is uniquely identified as a String, is defined in an XML Deployment Descriptor where it is mapped onto JVMs. JVMs are themselves mapped onto physical machines: VN --> JVMs --> Machine. Various protocols can be specified to create JVMs onto machines (ssh, Globus, LSF, PBS, rsh, rlogin, Web Services, etc.). After activation, a VN contains a set of nodes, living in a set of JVMs. Overall, VNs and deployment descriptors allow to abstract away from source code: machines, creation, lookup and registry protocols.

Descriptor example: creates one JVM on the local machine

```
ProActive.addNFEListenerOnAO(myAO, new NFEListener() {
   public boolean handleNFE(NonFunctionalException nfe) {
      // Do something with the exception nfe...
      // Return true if we were able to handle it
      return true;
   }});
```

The behaviour of the default handler (if none could handle the exception) is to throw the exception if it's on the proxy side, or log it if it's on the body side.

Components

Components are formed from AOs, a component is linked and communicates with other remote components. A component can be composite, made of other components, and as such itself distributed over several machines. Component systems are defined in XML files (ADL: Architecture Description Language); these files describe the definition, the assembly, and the bindings of components. Components follow the Fractal hierarchical component model specification and API, see http://fractal.objectweb.org. The following methods are specifically added by the ProActive implementation of Fractal.

In the class org.objectweb.proactive.ProActive:

In the class org.objectweb.proactive.core.component.Fractive:

Web services

ProActive allows active objects exportation as web services. The service is deployed onto a Jakarta Tomcat web server with a given url. It is identified by its urn, a unique id of the service. It is also possible to choose the exported methods of the object. The WSDL file matching the service will be accessible at

Compared to the compared

path,...) --> </processDefinition>

```
ProActiveDescriptor pad = ProActive.getProActiveDescriptor(String File);

// Returns a ProActiveDescriptor object from the xml
// descriptor file name
pad.activateMapping(String VN);

// Activates the given Virtual Node: launches or acquires
// all the JVMs the VN is mapped onto
pad.activateMappings();

// Activates all VNs defined in the ProActiveDescriptor
VirtualNode vn = pad.getVirtualNode(String)

// Creates at once a group of AO of type "A" in the JVMs specified
// by the given vn. The Virtual Node is automatically activated if
// not explicitly done before
Node[] n = vn.getNodes();

// Returns all nodes mapped to the target Virtual Node
Object[] n[0].getActiveObjects();

// Returns a reference to all AOs deployed on the target Node
ProActiveRuntime part = n[0].getProActiveRuntime();

// Returns a reference to the ProActive Runtime (the JVM) where
// the node has been created
pad.killall(boolean softly);

// Kills all the JVMs deployed with the descriptor
// not softly: all JVMs are killed abruptely
// softly: all JVMs that originated the creation of a rmi registry
// waits until registry is empty before dying
```

File Transfer Deployment

File Transfer Deployment is a tool for transfering files at deployment time. This files are specified using the ProActive XML Deployment Descriptor in the following way:

```
<VirtualNode name="exampleVNode" FileTransferDeploy="example"/>
....
</deployment>
</fileTransferDefinitions>
```

12

11

```
<FileTransfer id="example">
    <file src="hello.dat" dest="world.dat"/>
    <dir src="exampledir" dest="exampledir"/</pre>
   </FileTransfer>
</FileTransferDefinitions>
cprocessDefinition id="xyz">
     ssnrrocess>...
<FileTransferDeploy="implicit"> <!-- referenceID or keyword
    "implicit" (inherit)-->
    <copyProtocol>processDefault, scp, rcp</copyProtocol>
        <sourceInfo prefix="/home/user"/>
     </sshProcess>
```

Peer-to-Peer Infrastructure

A P2P infrastructure of ProActive JVMs over desktop machines. It is self-organized and configurable. The infrastructure maintains a dynamic network of JVMs for deploying computational applications.

Deploying the Infrastructure

```
$ cd ProActive/scripts/unix/p2p
$ ./startP2PService [-acq acquisitionMethod] [-port portNumber] [-s Peer ...]
```

A simple example

first.peer.host\$./startP2PService.sh second.peer.host\$./startP2PService.sh -s //first.peer.host third.peer.host\$./startP2PService.sh -s //second.peer.host

Acquiring Nodes

```
<services>
   <<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre><<pre></p
       <peerSet>
          <peer>//second.peer.host</peer>
  </per>//second.
</perSet>
</P2PService>
</serviceDefinition>
</services>
```

ProActive 3.2 - Reference Booklet

Fault-Tolerance

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 fault-tolerant using Java properties that can be set in the deployment descriptor. The programmer can select at deployment time the most adapted protocol regarding the application and the execution environment.

A Fault-tolerant deployment descriptor

```
<virtualNodesDefinition>
<virtualNode name="NonFT-Workers" property="multiple"/>
<virtualNode name="FT-Workers" property="multiple"
ftServiceId="appli"/>
</virtualNodesDefinition>
  <serviceDefinition id="appli">
   <!-- URL of the fault-tolerance server -->
     <globalServer
      globalserver
url="mmi://localhost:1100/FTServer"></globalServer>
:!-- URL of the resource server; all the nodes mapped on the
FT-Workers virtual node will be registered in as resource
nodes for recovery -->
    resourceServer
url="rmi://localhost:1100/FTServer"></resourceServer>
    <!-- Average time between two consecutive checkpoints for each
       object -->
    <ttc value="5"></ttc><!-- in seconds -->
  </serviceDefinition>
</services>
```

Starting the fault-tolerance server

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

the protocol: -proto [cic|pml]. Default value is cic. the server name: -name [serverName]. Default name is FTServer.
the port number: -port [portNumber]. Default port number is 1100.
the fault detection period: -fdperiod [periodInSec], the time between two consecutive fault detection scanning. Default value is 10 sec. the URL of a p2p service that can be used by the resource server: -p2p

[serviceURL]. No default value.

ProActive 3.2 - Reference Booklet

An X.509 Public Key Infrastructure (PKI) allowing communication Authentication, Integrity, and Confidentiality (AIC) to be configured in an XML security file, at deployment, outside any source code. Security is compatible with mobility, allows for hierarchical domain specification and dynamically negotiated policies.

Example of specification

```
<Rule>
    <From><Entity type="VN" name="VN1"/> </From>
<To> <Entity type="VN" name="VN2"/> </To>
<Communication>
     <Request value="authorized">
        </Request>
  </communication>
</igration>denied</Migration>
<AOCreation>denied</AOCreation>
</Rule>
```

This rule specifies that: from Virtual Node "VN1" to the VN "VN2", the communications (requests) are authorized, provided authentication and integrity are being used, while confidentiality is optional. Migration and AO creation are not authorized.

Branch and Bound API

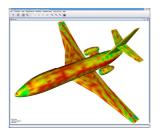
This is a simple API for solving parallel problems using a Branch-And-Bound

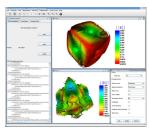
```
public class YourTask extends Task {
    public Result execute() // For computing a solution
public Vector split() // For generating sub-tasks
public Result gather(Result[] results) // Gathering all results
public void initLowerBound() // For computing a lower bound
public void initUpperBound() // For computing a upper bound
 Start the computation:
```

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

JEM3D: Java Electro-Magnetism with

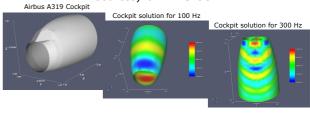






Vibro-Acoustic Code Coupling of **MPI Legacy with Components**

Courtesy of EADS CCR



16

15

13

14