ProActive ReferenceCard

ProActive.ObjectWeb.org

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.

Main concepts and definitions:

- Active Objects (AO): a remote object, with its own thread, receiving calls on its public methods
- FIFO activity: an AO, by default, executes the request it receives one after the other, in the order they were received
- No-sharing: standard Java objects cannot be referenced from 2 AOs, ensured by deep-copy of constructor params, method params, and results
- Asynchronous Communications: method calls towards AOs are asynchronous
- Future: the result of a non-void asynchronous method call
- Request: the occurrence of a method call towards an AO
- Service: the execution by an AO of a request
- Reply: after a service, the method result is sent back to the caller
- Wait-by-necessity: automatic wait upon the use of a still awaited future
- Automatic Continuation: transmission of futures and replies between AO and JVMs
- Migration: an AO moving from one JVM to another, computational weak mobility: the AO decides to migrate and stack is lost
- Group: a typed group of objects or AOs. Methods are called in parallel on all group members.
- Component: made of AOs, a component defines server and client interfaces
- Primitive Component: directly made of Java code and AOs
- Composite Component: contains other components (primitives or composites)
- Parallel Component: a composite that is using groups to multicast calls to inner components
- Security: X.509 Authentication, Integrity, and Confidentiality defined at deployment in an XML file on entities such as communications, migration, dynamic code loading.
- 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 mapping a VN to a set of JVMs. After activation, a VN contains a set of nodes, living in a set of JVMs.
- 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 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
   // A request is sent to a,
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 behavior in the method named runActivity():

```
Interface RunActive
void  runActivity(Body body)
   // The activity of the active object instance of the current class
```

Example:

```
public class  A implements  RunActive {
   // Implements RunActive for programming a specific behavior
   // runActivity() is automatically called when such an AO is created
public void  runActivity(Body body) {
Service service = new Service(Body);
while ( Terminate ) {
        ...    // Do some activity on its own
        ...
        ...    // Do some services, e.g. a FIFO service on method named foo
        ...
        service.serveOldest("foo");
        ...
      }
    }
}
```

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.

```
public class  BusyButReactive  implements  RunActive {

public void  runActivity(Body body) {
Service service = new Service(Body);
while ( ! hasToTerminate ) {
        ...    // Do some activity on its own
        ...
        ...    // Non blocking service
        ...
        service.serveOldest("changeParameters", "terminate");   ...
      }
    }
public void  changeParameters () {......    // change computation parameters}
public void   terminate (){ hasToTerminate=true;}
}
```

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:

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

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

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
void blockingServeOldest(RequestFilter requestFilter)
   // Serves the oldest request matching the criteria given be 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 be 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() will be started on the new site
void   lifoServing()
   // Invoke a LIFO policy. Call does not return. In case of
   // a migration, a new runActivity() will be started on the new site
void  serveYoungest()
```

```
   // Serves the youngest request in the request queue
void  flushAll()
   // Removes all requests in the pending queue
```

Active Object Creation:

```
Object newActive(String classname, Object[] constructorParameters,Node node);
   // Creates a new AO of type classname. The AO is located on the given node,
   // or on a default node in the local JVM if the given node is nul
Object newActive(String classname,Object[] constructorParameters,VirtualNode virtualnode);
   // Creates a new set of AO of type classname.
   // The AO are located on each JVMs the Virtual Node is mapped onto
Object turnActive(Object, Node node);
   // Copy an existing Java object and turns it into an AO.
   // The AO is located on the given node, or on a default node in
```

Groups:

```
A ga = (A) ProActiveGroup.newGroup( "A", params, nodes);
   // Created 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 being 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 AO 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 object 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 to the typed view of a group

void   setScatterGroup(g);
   // By default, a group used as a parameter 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 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
```

Migration:

Methods both in Interface Group, and static in class ProActiveGroup

```
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 the static migrateTo primitive:

```
public void moveTo(Object) {
     try{
       ProActive.migrateTo(t);
     } 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);
   // Specifies a migration itinerary
void migrationStrategy.add(Destination);
   // Adds a JVM destination to an itinerary
void migrationStrategy.remove(Destination d) ;
   // Remove a JVM destination in an itinerary
```

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

```
Component newActiveComponent("A", params, VirtualNode, ComponentParameters);
   // Creates a new ProActive component from the specified class A.
   // The component is distributed on JVMs specified by the Virtual Node
   // The ComponentParameters defines the configuration of a component:
   // name of component, interfaces (server and client), etc.
   // Returns a reference to a component, as defined in the Fractal API
```

Security:

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 specificationand 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">
       <Attributes authentication="required"
                   integrity="required"
                   confidentiality="optional"/>
    </Request>
   </Communication>
   <Migration>denied</Migration>
   <AOCreation>denied</AOCreation>
</Rule>
```

This rule specifies that: from Virual 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.

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.

```
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)
   // Created 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
```