

# An Aspects oriented approach to dynamically manage applications

Bernard Kaddour  
LIRIS – Université Lyon 1  
43 boulevard du 11 novembre 1918  
F-69622 Villeurbanne cedex  
bkaddour@liris.cnrs.fr

Joël Quinqueton  
LIRMM  
161 rue Ada  
F-34392 Montpellier cedex 5  
jq@lirmm.fr

## Abstract

*The emergence of middleware solutions and new services even on small devices will need adapted distributed management solutions which address these specificities, both in terms of software design and in terms of performance. We propose a management system where these high level and low level management concerns are separated.*

*The high level management part relies on messages interception mechanisms which, coupled with Aspects Oriented Programming concepts, provides facilities for management applications to dynamically operate, enhance and manage JAVA based applications. The management is transparent for the application which doesn't need to be modified to support management operations as we take advantage of both the JAVA introspection mechanisms and the facilities some Aspects Frameworks offer.*

## 1. Introduction

Since some years, more and more smart phones or personal digital assistants are widely available and used. These small devices still suffer from some limitations compared with high end fixed terminals such as lower CPU performance or smaller memory size, but even now days they already have enough processing capabilities to host complete high Operating Systems - either Windows or Linux dedicated versions - and they appear more and more as autonomous embedded systems. Meanwhile, industrially accepted middleware solutions appear [8] [15] for new distributed applications. These middleware are currently available for small devices [16] and even Multi-Agents Systems based solutions have been introduced in the research area [5].

Such new distributed application whose components may partially or totally be executed by small devices implies new evolutive and flexible needs for their management to be effective.

Another worth to note point is the large acceptance of the JAVA language, both for J2EE [9] or Corba applications and even for Corba frameworks design themselves.

We concentrate to provide solutions widely applicable which use these particularities and we conceptually separate the high and the low parts of the management. The former takes advantage of the large utilization of JAVA for middleware applications and uses Aspect Oriented Programming concepts [12] to manage applications without invading the underlying middleware. This approach avoids modifications of already existing applications and would ease the translation of introduced mechanisms towards new emerging middleware solutions [14]. The later introduces monitoring tasks tightly related to the code of the managed JAVA entities.

In this article we concentrate on the high part of the management system and the low-level management part [11] is not developed here.

We present in section 2. the principles and main ideas which lead our management system for distributed components. The section 3. discusses how managed objects are obtained and section 4. develops the entities used for management. The section 5. presents how they are layered. We conclude with some future works and perspectives.

## 2. Key features of our architecture

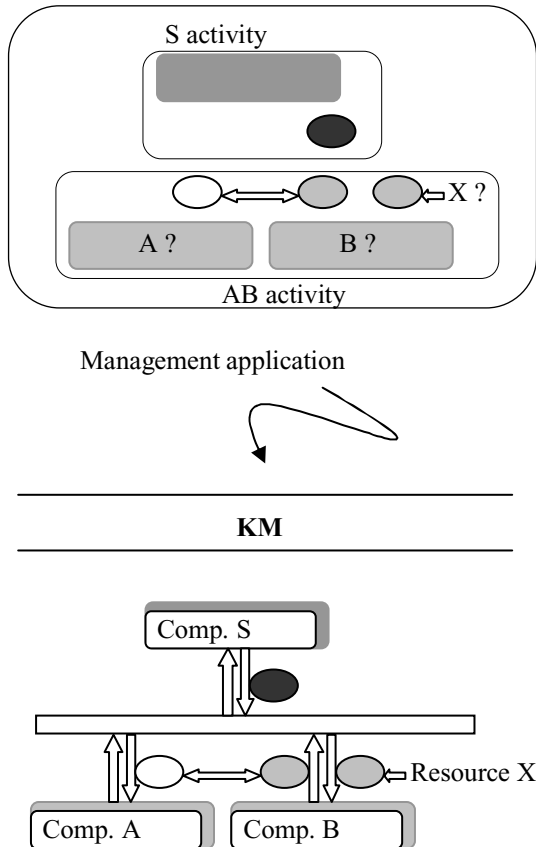
Managing disperse heterogonous entities has already been investigated in the network community and standards defined [17][18], but their intrinsic centralized and frozen aspects are very limitative [13]. Newer and more application centric solutions have appeared [6][7][10], but they may not adequately address the evolutive aspect of management: management functions must be easy to place and re-use, easy to remove or to stop and many management functions can simultaneously manage an entity.

The approach consists in a core management system - named *kernel of the management system or KMS* - where management applications can be deployed.

The KMS largely relies on the interception of incoming and outgoing messages exchanged by the managed application. It can then spy and filter requests sent or results received by the components of the

managed application accordingly to management functions requirements.

Management applications are composed of *management activities*. Activities are in turn composed of management functions and filters. Management functions are not limited to collect data and can be parts of a complete distributed management application while filters can dynamically be linked with the components they have to interact with when deployed by the KMS.



**Figure 1. Architecture overview.**

Figure 1. depicts the expected scenario.

A management application can be required by the KMS or sent by a peer-KMS towards the application to manage. Management application is a nearly autonomous entity containing its management activities. According to the target application and the activities wishes, the KMS when places the filters of the different management activities around the expected components. The KMS can provide local resources or services to management applications.

The management system permits to several management applications to simultaneously operate upon a whole application or only upon its components. For example, one could manage an application with some logging management functions for profiling purposes even if another activity is already managing some of the application's components for synchronization purposes. This simultaneously-multi-managed feature makes the management activities to appear as enhancements of the managed application.

Some differences with JMX exists. From the JMX point of view, the proposed management system can be viewed as a set of linked and modified *MbeanServer*, differently registering parts of activities, while activities appear as autonomous collections of *Mbeans* inheriting the *KMS Interfaces* where all the necessary behaviors are implemented.

### 3. Requirements and strategies for our management

The introduced architecture deals with entities following the usual client-server model. As we expect to not modify the source code of the managed components, management can be achieved through (i) the external representation of the different elements and the interception of sent and received messages; and (ii) the possibility to directly read and modify some of the variables of the component we manage.

First, the connection with the components we plan to manage as this feature is generally not natively available.

#### 3.1. Connection with the managed entities: Interceptors

We distinguish applications originally written in JAVA from the others, in particular from Corba applications. In the case of JAVA applications, we use the introspection possibilities the language offers and the facilities we have to access compiled byte-code. These facilities can be used to create Interceptors-wrappers or to modify the application with byte-code injection [2][4].

Some tools [1][2] currently go one step further, permitting to operate upon JAVA code (either at source or byte-code level) with high level concepts and high level languages. Performance impact tends to remain acceptable [2].

These features give the possibility to consider JAVA applications as manageable entities.

For Corba applications, OMG provides some basic introspection facilities services: Portable Interceptors

[15]. Interceptors can be used by third parties to spy requests or modify messages exchanged between the middleware and the components. They are often used to enrich Corba with new features, ranging from synchronization to caching [3], but they induce performances penalties. They remain a handy solution as they require only few or not any modification in the original application code. We use them at request and message level (i) for interception purposes and (ii) as a *glue* with the others parts of the system which are mainly JAVA centric.

The following sections focus on the representation of management activities and on the messages filtering system.

## 4. Major components of the system

KMS is implemented as a daemon-process to permit sub-activities exchanges between KMS (currently with sockets). It inspects and registers the entities it is in charge of and interacts with AspectWerkz [2]. It is worth to note that KMS provides JAVA classes and interfaces that management entities mandatory implement or inherit from. That's the way KMS normally use to interact with or force a management entity.

### 4.1. Activities

The original activity concept was first introduced in the Computer Supported Co-operative Work framework. We will continue to use this term although its meaning has deeply been altered.

Main parts of an activity are :

- **Roles**: We distinguish external roles corresponding to resources that other third parties may provide to the activity from internal roles corresponding to sub-activities of the current activity. Both of them are described by means of references towards interfaces,

- **sub activities**: greatest difference with other tools is their expected autonomous abilities as the activity may later ask the KMS to send them, as a whole, over the network, and then deploy them.

- **constraints** and **preferences**. They are activate by the requests the activity can receive. They can decide to allow, modify, or reject a request. The loading of a particular characters font as a new application conforming to an editor interface is such an example,

- **internal tools**: It's a set of functions embedded in the activity for it's own needs. These tools, usually inactive, can be trigger at any time by the arrival of a new element (via a preference e.g.) or for the activity needs. Particular tools are the *incoming* or *outgoing filters* acting upon messages received or emitted by the managed component,

- **monitoring expressions**. These are expressions the activity wants at low-level management for direct code monitoring tasks,

- the main body of the activity composed of methods the activity responds to and the set of its private variables. Security policy to apply for each received message or termination of the activity are known methods that every activity must implement or delegate.

- **set of attributes** from which the activity can be designated. E.g. an edition activity can specify an octet string attribute file which is the name of the file it proceeds with.

An activity is encapsulated in a .jar class file. It is the activity responsibility to report to the management system the interface or set of services it can respond to whereas the KMS will soon verify that the different parts of the activity conform to the expected JAVA interfaces.

Interceptors are particular activities as they are special parts of the KMS with high privileges.

### 4.2. An example of activity

To depict how an activity is made up, let us consider an application registered as «Service» and providing the «sub» and «add» operations. This application can be managed by a simple «PositiveAccount» activity (i) to count the number of operation requests received and (ii) to force the «sub» operation to return a positive or nil result.

```
PositiveAccount.jar activity ::= [
class Count {
static int count =0;
public static count() { return count; }
}
class Inc_count implements InFilter {
public Object exec(JoinPoint joinPoint) {
Count.count ++;
}}
class Positive implements OutFilter {
public Object exec(JoinPoint joinPoint) {
Object ret= joinPoint.proceedMod();
if (ret.intValue() <0)
ret =new Integer(0);
return ret;
}}
class UneContrainte implements Constraints {
Public Object exec(Activity a) {
If (a.haveCompatibleInterface("edit"))
...
}}
}
class PositiveAccount implements Activity {
```

```

public int expressionReached(int e) { ... }

public int init() {
    Inc_count ic =new Inc_count();
    register_inFilter(« Service », « add », ic);
    register_inFilter(« Service », « sub », ic);
    Positive p =new Positive();
    register_outFilter(« Service », « sub »,
p,MOD_PRIV);
    register_Constr(new      UneContrainte(),
ANY_IN);
    ...
    register (this, «Count»);
    return 1;
} } ]

```

First the activity registers its filters which have to operate around the «Service» managed application. Inc\_Count will count the number of «add» or «sub» requests the application will receive while a Positive instance will check and maybe modify the value returned by «sub» to be positive or nil. Finally the management activity registers itself as a «Count» activity. It can in turn be suspend or remove (and even itself be partially managed), its filters moved or stopped by the KMS.

### 4.3. Contexts

When an activity **A** is deployed by the KMS, the KMS firstly ensures by introspections that the different parts of the activity conform to the expected ones: *outFilter* class inherits from the *KMS\_outFilter* class, *Activity* from *KMS\_Activity*, etc. Lastly, KMS starts the *init()* method in a dedicated thread.

The activity then creates its own objects and asks the KMS to register objects which interact with other parts of the system. Such objects are (i) filters that the KMS will connect to managed objects of the managed application and (ii) constraints & preferences which may help the activity to customize the environment.

Finally, the activity can use KMS services (such as communication services to send its sub-activities over the network towards another KMS) or require extra services provided by other activities.

Major differences between filters and constraints & preferences is the kind of tasks they are concerned with. Constraints and Preferences (C&P) are used by the activity to express its desires and restrictions. Once registered by the KMS on behalf of the activity, the KMS ensures that before providing a service (or an external activity) to A, it will first check it against the constraints and then against the preferences registered by the activity A. It is up to the KMS to make as many necessary retries before it sends to A the resulting

service. This C&P mechanism gives to the management system some capability to adapt environment resources with activities wishes.

The set of tools, constraints, preferences, services provided to others represent the environment (named *context*) created by the deployed activity.

In turn, an activity must be deployed in an already existing context.

Two abstract contexts are introduced:

- *site-context* corresponds to the operating system abstract activity. It is the main owner of the local resources (hardware, communication links, etc) and it have higher privileges. This context can offer services to its sub-contexts or acts as delegate for them. For example, a sub-context can delegate its security to the site-context security policies.

- *user-context* corresponds to a user environment in which will be (by default) deployed the activities that the user will later execute.

Deployment of management activities creates contexts that are then organized as a tree. Each context has a father-context and may have child-contexts. Child-contexts can be either contexts tied to sub-activities or contexts created by another activity explicitly deployed in the current context.

*Constraints* and *Preferences* (C&P) deal with contexts messages (or actions) received by the context such as requests for the creation or the insertion of new contexts.

In particular, before being deployed in a context **C**, an activity will have to conform to the constraints imposed by any context surrounding **C**. Starting from constraints of the site-context to constraints of **C**.

This mechanism gives to the management system the capability to adapt activities with the environment wishes explained by means of C&P.

On the other hand, *Filters* deal with sent or received messages (methods calls and return values) by the managed application. This is obtained transparently (from the managed object point of view) by using Aspect oriented programming concepts and tools.

## 5. The management of the methods & the filtering

### 5.1. Aspect oriented programming

This paradigm mainly due to [12] has for objectives to capture some singularities that aren't actually properly take in charge by the object oriented model

(OO). Worth to note that Aspect oriented programming (AOP) doesn't tend to replace OO model but rather tends to improve and to ease the software development work.

In practice, AOP complements OO programming by allowing to dynamically modify and improve the static OO model with the inclusion of some new needed code required to fulfill some new expected requirements.

AOP has introduced some concepts for addition or modification of existent code, mostly:

- **Join Points.** These are well-defined points in the flow of a program. Methods call or return, exception handler entry point and even field set or get operations are Join points examples.

- **Points cut.** They are mainly used to identify Join points from different classes.

- **Advices** which define both the code of the aspect and, combined with join points, when it will have to be executed. Usual advices are *before* advices as we expect the new aspect's code to be executed before the join point, and *after* advices as we expect the new aspect's code to be executed after the join point.

AOP frameworks are now available [1][2]. We prefer AspectWerkz (AW) [2] as it provides a so-called *online-mode* to operate upon already compiled pieces of code (.class) and, most interesting, have a useful API providing all of the requirements needed to dynamically manipulate aspects. Finally, AW (from WebLogic/BEA) intends in a mid-future to fully interact with Jboss and a bringing together of AW and AspectJ is on the way.

We note that classes are AOP main concerns, objects being at glance out of concern. That's to bypass this limitation and to have the possibility to select and operate upon classes instances that naming attributes have been introduced as part of our activities.

## 5.2. Layout of the filters

When launched under AW using:

```
aspectwerkz -Dkms.xml -cp kms app.jar
```

any method `m()` of `app.jar` will be executed after the before aspects known by `kms` have been executed and the result of `m()` will be delivered after the after aspects known by `kms` have been executed.

Filters registered by the KMS on behalf of an activity are inserted in the in-filters or in the out-filters chain according to the activity's request and to the interface they inherited from. From the managed object point of view, each filter then appears as a *before aspect* if placed in the in-filters chain or as an *after aspect* if placed in the out-filters chain. Many filters can simultaneously be present in a chain, these filters may have been required by one or several different

activities. Filters placed by the KMS have highest priority, follow activities filters, then sub-activities filters then sub-sub-activities filters etc.

Connected to AW, KMS is informed of the methods calls and returns of each object. For a method call, depending on the caller and on the call parameters, KMS selects in the in-filters the list of filters to apply. These filters are then sequentially executed using AW API (mainly through the method `Object joinPoint.proceedMod()`, a modify version of the original `proceed()` one). If desired, parameters can be altered by using the AW API. Registered filters to select and apply for a method return value follow the same approach.

To avoid incoherence in these chains, the possibility of one or several different activities managing an object -e.g. a logging activity initiated by a user A and a synchronization activity required by a second user B - depends of the application and of the KMS's choices. Moreover, only filters registered with enhanced privileges or filters registered by the KMS for its own needs can modify or stop a message.

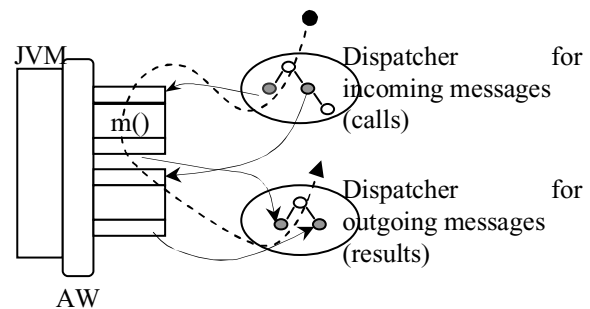


Figure 2. Filters and messages paths.

Left part of Fig. 2 depicts a managed object wrapped in a JVM/AW execution environment. A dash line represents the execution path taken by an `m()` request. First KMS in-filters are applied, then the activity's in-filters before `m()` is effectively called. The returned result have to pass through activity's out-filters and finally the KMS out-filters before being sent back to the requestor.

## 6. Future work and conclusion

The purpose of this work is to investigate the problem of management for distributed applications and provide means to operate in a distributive and dynamic manner. We suggest to use Aspects concepts as a possible solution to allow an evolutive and decentralized high level management scheme without

modifying neither the underlying middleware nor the managed objects.

We plan to define *virtual MIBs* for the KMS which will allow usual SNMP tools to interact with it and provide new facilities to manage software or even hardware component, but some problems remain as our interfaces directly compete with the SNMP OBJECT-TYPE macro.

## 7. References

- [1] AspectJ, The ASpectJ Project, <http://eclipse.org/aspectj/>, 2005.
- [2] AspectWerkz, <http://aspectwerkz.codehaus.org/>, BEA Systems., 2004.
- [3] R. Baldoni, C. Marchetti, and L. Verde, CORBA request portable interceptors: analysis and applications, *Concurrency and Computation: Practice and Experience, Vol 15*, p 551–579, 2003.
- [4] BCEL, The Jakarta Project. The Byte Code Engineering Library, <http://jakarta.apache.org/bcel/>, 2004.
- [5] C. Carabelea, O. Boissier, Multi-Agent Platforms on Smart Devices: Dream or Reality ?, *1st Smart Object Conference*, 2003.
- [6] DMTF-CIM, Common Information Model, <http://www.dmtf.org/standards/cim/>, Distributed Management Task Force, 2004.
- [7] M. Debusmann, K. Geihs, Efficient and Transparent Instrumentation of Application Components using an Aspect-oriented Approach, *DSOM 2003*, 2003.
- [8] EJB, Enterprise JavaBeans Specifications version 2.1, <http://java.sun.com/products/ejb/docs.html>, 2002.
- [9] J2EE, J2EE Specification v1.3, <http://java.sun.com/j2ee/>, 2001.
- [10] JMX, Java Management extensions, <http://java.sun.com/products/JavaManagement/>, 2000.
- [11] B. Kaddour, J. Quinqueton, Administration d'applications distribuées par Aspects et expressions programmables, *GRES'05*, 2005.
- [12] G. Kiczales & al, Aspect oriented Programming, *Proc. of ECOOP'97*, p 220-242, 1997.
- [13] J.P. Martin-Flatin, S. Znaty, A Simple Typology of Distributed Network Management Paradigms, *DSOM'97*, p 13-24, 1997.
- [14] NET, NET Middleware Services: Introduction, <http://technet.microsoft.com/>, 2005.
- [15] OMG, <http://www.omg.org/>, O.M.G., 2004.
- [16] OpenCCM, <http://openccm.objectweb.org/>, ObjectWeb, 2005.
- [17] RFC1905, Operations for Version 2 of the Simple Network Management Protocol (SNMPv2), <http://www.faqs.org/rfcs/rfc1905.html>, 1996.
- [18] RFC1907, Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2), <http://www.faqs.org/rfcs/rfc1907.html>, 1996.