23

Application management by actors for SNMP

Bernard Kaddour and Michel Beigbeder
Ecole des Mines de Saint-Etienne
158, cours Fauriel,
42023 Saint-Etienne cedex 2
France
Tel: (+33) 4 77 42 01 74 Fax: (+33) 4 77 42 66 66
kaddour@emse.fr, mbeig@emse.fr

Abstract

This document focuses on management of application resources. Starting from the *message bus* design, our original approach is based on *active objects*. It provides a dynamic management scheme allowing evolutive and reusable management functions. Despite important differences with standard managenent frameworks, our management design is integrated into existing network management protocols, and so it provides application management capabilities to existing tools.

Keywords

Message bus, actors, interfaces, evolutivity, MIB integration

1 INTRODUCTION

To reduce the time and cost of application development, *Integrated Environments* (Reiss 1990) or brokers (O.M.G. 1991) (Brockschmidt 1995) were designed. They allow to build new applications by aggregation of existing software components communicating with themselves. Likeness with network entities appears at first glance.

The increasing complexity and sophistication of network services required dedicated network management tools and protocols, it will — soon — be the same for the applications.

Network management platforms have been largely available since the standardization of network management. Nevertheless, some of their characteristics are limitative. Centralization is the first one, only *managers* are able to execute complex management operations. Secondly, these platforms are restricted to manage hardware network components and not communicating applications. These limitations compel network management platforms to use specific administration functions that other platforms can't reuse.

Intelligent Networks and Intelligence in Networks D. Gaiti (Ed.)
Published by Chapman & Hall

We will describe an application modelization based on active objects and will use it for application management. We will emphasize evolutivity needs in application management which network management doesn't provide.

2 APPLICATION MANAGEMENT

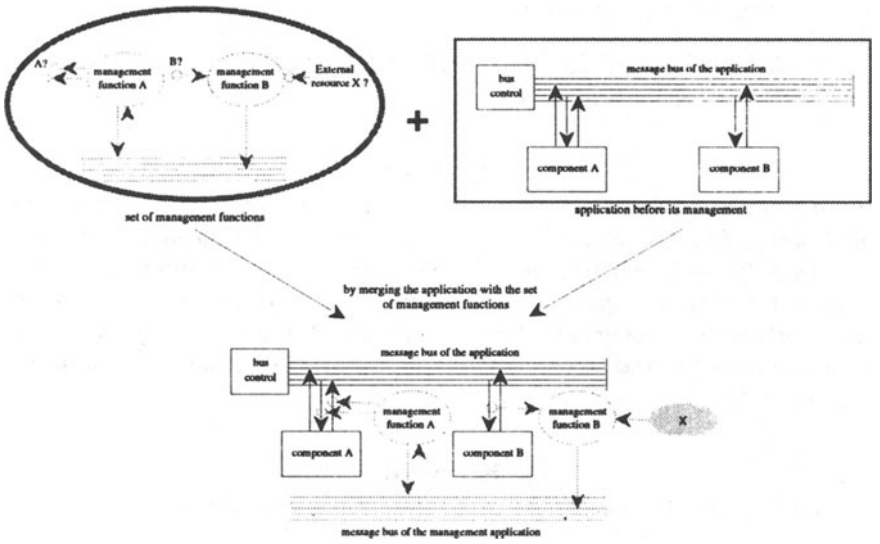

Figure 1 Example of an application management achieved by cooperating management functions

Application management is concerned with the optimization of the *service per cost* ratio, even when the application runs under slightly disturbed working conditions. Main points are:

quality of service: Checking the fulfilment of the declared services. This is achieved by monitoring sensible parts of the application.

adaptability: Customizing current services to fulfill other application requirements. For example an interpersonal mail message exchange service can be used to partially provide file transfer system facilities (Rollin 1986).

optimization of the use of resources: disks, network throughput, etc.

So, application management concerns appear close to usual network management ones.

Network management frameworks (ISO 1989) (Case, Fedor & al. 1990) based upon a passive object representation of the entities to manage is actually mainly concerned with physical components. Meanwhile, these management standards induce some limitations that application management have to bypass, such as centralization in management platforms.

A new approach for management *TINA* (Chapman, Dupy & Nilson 1995) lastly appears. We will differ from it mainly by the fact that we don't want to depict another management platform but rather to allow processing capabilities for management, taking into account basic protocols aspects. From this point of view, our approach draws closer to the *JMAPI* * spirit.

2.1 Message bus and management

Integrated Environments used for new application development are classified in families (Boyer 1994) depending on their underlying design. Nevertheless, one of these design is actually largely available from manufacturers, and it provides flexibility and genericity: the *message bus.*

A message bus acts as a server where every client registers with the Software Communication Group it belongs to. At any time, a client may indicate the message types it expects to receive and to process, then the message bus handles messages and routes them to clients (Cf. upper right part of Figure 1).

The message bus model enables management to operate on the application as a whole, or on its components only. So, the application management is no longer centralized: management functions dynamically link with the components they want to interact with. Moreover, management functions are not limited to collect data, and can be parts of a complete distributed management application (Cf. Figure 1).

3 ACTIVE OBJECT APPROACH

With the *passive* object oriented model, network entities are hardwired *templates* and their management is operated by platform dependant supervision services. When new managed entities are inserted, new templates with new specific management entities are to be (re)written. This scheme can't gracefully take into account the dynamic and autonomous aspects of the applications we want to manage. The *actor* (or *active object*) model (Hewitt 1977, Agha 1986) seems to be the most suitable for our purpose. Moreover, it allows unification of managed and management applications, providing an homogeneous framework. Lastly, the *description* of these entities enters in this framework.

Two points need some explanations to make our point of view accept-

*http://java.sun.com/products/JavaManagement/

able: *i)* We have to bypass some burdens of the actor calculus model, for example the unification in the extreme which doesn't facilitate the model use (Venkatasubramanian & Talcott 1993), *ii)* We have to take into account existing software and existing management frameworks. In particular, arrangements are necessary to draw closer to, and finally merge with, the current network management architecture.

We will introduce three kinds of entities to address our main points of § 2: *interfaces*, *activities* and *contexts*.

3.1 Interfaces

Interfaces play a descriptive role part in our application management system. They describe the operations any interface conformant actor must support, as*ASN-1 templates* do in *SNMP* or *CMIP*. However, as the expected management entities have to operate between themselves and with managed entities, an interface integrates the methods that a conformant actor renders to the community and the services it requires. Interface finally draws closer to *typed active objects* (Nierstrasz 1993).

Interfaces differ from typed active objects by the set of standardized operations an interface responds to. These operations give interfaces capabilities to manage version numbers or to combine between themselves to describe new *families* of actors.

Roughly speaking, an interface is composed of the set of the typed variables — according to SNMP allowed types. and the set of the method signatures for each one of the states that a conformant actor may enter.

An interface example will be the action table an entity has to respond to to be conformant with the *editor* notion of company *X*. For example *Apple* defines an *editor* as a program responding to a set of *Apple Events*.

Interfaces of a special kind, called *translation* interfaces, ensure that any actor conformant to a translation interface may adapt some elements — under some restrictive conditions — to be conformant to another interface (Kaddour & Beigbeder 1996).

Our design thus differs from the usual network management scheme which needs a tree of classes (or types) and a tree of instanciations, the former containing abstract representations of managed objects, the latter containing managed objects. Our approach needs a single tree which contains management entity descriptions, management entities, managed entities, and managed entity descriptions where entities descriptions are represented by interfaces.

3.2 Activities

Most of our management entities are *activities.*

The original *activity* concept has first been introduced in the *Computer Supported Co-operative Work* framework (Danielsen, Pankoke & al. 1986) (Brun 1987). 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 are described by means of references to interfaces.
- *constraints* and *preferences.* They are the combination of logical conditions based upon the events the activity can receive. They associate an *internal tool* to be executed when they are verified. The loading of a particular character set font as a new incomer conforming to an editor interface is such an example.
- *internal tools*: It is a set of functions embedded in the activity for it's own needs. These tools, usually inactive, can be triggered at any time by the arrival of a new element (via a preference e.g.) or by needs of the activity. Particular tools are the *incoming* or *outgoing filters* acting upon messages received or emitted by the activity and *variables filters* (Cf § a).
- *the main body of the activity* made 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.

This structure of the activity enables both the description of applications build as communication components that we want to manage, and the description of management applications.

It is the activity responsibility to report to the system management the interface or set of services it can respond to.

So, an activity can be seen as an actor solely composed of a set of sub-activities, variables and methods that can be called for particular processings, but also has a set of elements mainly composed of filtering functions appliable to messages. The former vision is suited to the applications to manage, the latter to the management applications. Both of them are unified under the activity concepts.

(a) Wrappers

Particular activities are *wrappers*. They acts as *gateways* between the management system and the real world we have to interact with. From the system point of view, they represent real world applications when these one don't naturally support the mechanisms required to be managed.

The manner a wrapper interacts with the software it represents is highly software dependent and cannot be exactly specified. This is the same kind of problem found in usual network management between a managed object and its management agent.

Finally, we emphasize that management functions are obtained by means of input, output or variables filters tied to wrappers as depicted by Figure 2.

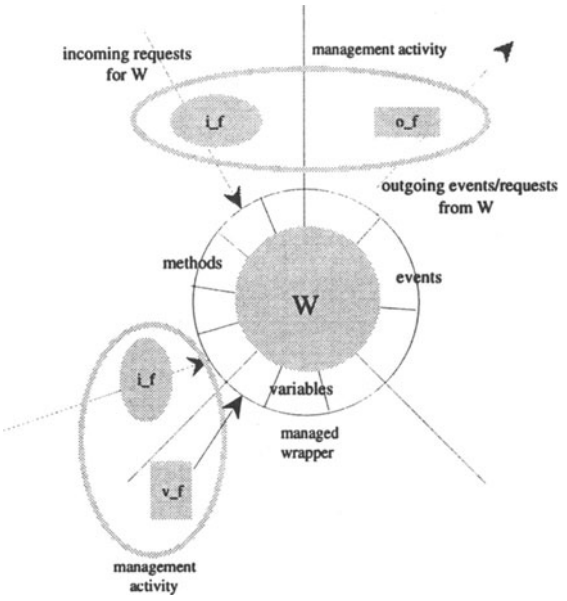

Figure 2 A wrapper with activities of management by means of filters

3.3 Contexts

The execution of any activity takes place in a dedicated context. This context represents the *environment* in which the activity operates. Taking into account the different pieces an activity is compound of, its associated context conceptually contains:

- *attributes* from which this particular instance of the activity can be designated. Some of these attributes directly come from the activity description, others, such as the set of provided services are dynamically obtained.
- a *reception port* that all the messages sent to the activity have to pass through. The port acts as a security manager (or a forwarder to a security delegate).
- the *sons-context* of the context attached to the considered activity. These are execution instances of internal sub-activities or of external resources required by the activity, provided by the management system.
- *constraints* and *preferences* which have tasks close to that of filters. The main difference is that the former proceed on incoming and outgoing activities, whereas the latter proceed on messages.
- *filters*, — incoming, outgoing and variables — registered or in execution.

(a) Filters

Incoming (outgoing) filters are lightweight processes that can receive the messages sent to (emitted by) the activity. They can notify the arrival (departure) of a message to a delegate, read, write and modify the message.

To be considered as active, a filter has to register itself to the management system.

The format of registration and the main structure of filters are standardized. At registration time, the filter provides the required privilege (notification, read or write access) and the t-uple set (*address-identification of the originator*, *method's name*, *arguments*) characterizing the incoming messages the filter wants the system to notify it.

Each part of this set may describe one or several elements. For example, (*"user=bk".cambur.emse.fr*, *m0* OR *m1*, *ANY*) is interested by messages coming from the context identified as *"user=bk".cambur.emse.fr*, calling the method *m0* or *m1*, regardless of the arguments.

From theses pieces of information, the management system either provides message manipulation functions or refuses the filter's registration.

The fact that the signature of a method has been split between its name and its arguments enables us to construct the tree of the potentially activable filters for the context.

The walk through the tree of filters is then accomplished for each received message, according to the originator's address-identification in the first stage, according to the method's name called in the second stage and finally according to the arguments in the last stage.

For each of theses stages, we search a node of the tree matching the processed message in the corresponding level. If such a node is found, firstly the filter registered with a writing/modifying privilege is spawn, thus, if after the processing of the message by this filter the node found still matchs the message, the reading and notifying registered filters are simultaneously spawn whereas the next stage begins.

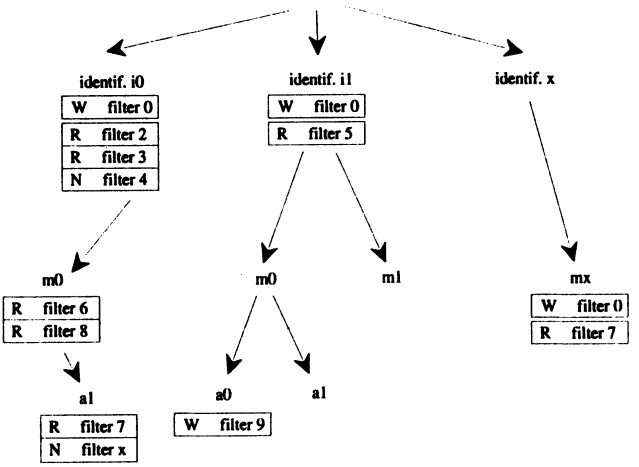

Figure 3 The tree of filters

Based upon the structure of filters, the tree obtained allows an efficient forwarding of the messages across the filters.

This is only after being processed by the activated filters that the message is finally delivered to the activity of the context — assuming that no spawned filters has stopped the forwarding process.

We note that each node of the tree contains at most one filter registered with write privilege for a given context, because allowing the registration of several filters with write privilege for the same node will lead to consider the set of activable filters as a graph, where loops may appear. Preventing any troubles from such loops would overload the system and slow down the walk-trough process having regard to the small system improvement. This "one writer constraint" applies only in a given context.

It is important to note that the particular forwarding mechanism induced by the way filters are registered and act does not conflict with existing communication protocols.

The way outgoing filters act is sensibly the same.

Variables filters are not interested in the way messages are exchanged, but they have to monitor some inner aspects of an activity. They achieve this goal by registering *expressions* made of variables with which the activity is described, for which they want to be notified as soon as they appear to be true.

Monitoring a variable value (that may be modified without any method call) needs control on the run-time or the interpreter of the language used by the application.

The variable filters act upon variables of simple or complex types. Special

keys have been introduced for variables of complex types such as *SEQUENCE* and *SEQUENCE OF* to distinguish if an expression is interested in the modification of the variable (e.g. destruction of an element in a *SEQUENCE OF* variable) or in the modification of one of its component.

3.4 The layout of the contexts

A *context* represents the *environment* in which the activity is executed.

So, the activity behavior is customized to the environment it is diped in according to its needs and to the available tools the *system management* can find or build from the interfaces information.

Without more information, the context of a new activity is placed in the context of the activity which requires the activation. This can be overriden by placement specification.

This mechanism is used to manage applications. For example, the management of an application `b` is achieved by spawning a management activity `a` — mainly made of filters and functions — and the system is told to place the context of `a` as an over-context of the context attached to `b`.

As a consequence, contexts appear as stacked upon each others, almost in the same manner protocols do (Venkatasubramanian & Talcott 1993).

Precisely, the set of contexts is built as a tree. From the object model point of view, a major difference appears between activities and contexts. Activities act as autonomous entities without class or inheritance mechanism. They draw closer to *prototypes* (Lieberman 1986): note that each activity initiator is the kernel of the management system which acts as the default delegate. So, by the layout of the delegates (Stein 1987), at run time the contexts use a mechanism similar to the usual inheritance. Data or services are successively searched in the upper layers of the caller context.

(a) The session

We have to take care in using incoming (outgoing) filters.

The interface of the couple (*filter*, *main activity body*) may present to the outer community is not the same has the one of the activity (this is a way adaptability or evolutivity is achieved). This may lead to some troubles or incoherences. For example with the following scenario:

— A context `c` asks the system for an entity compatible with the interface `i0`. The system provides the entity `i`.

— A writer filter modifies the interface of `i` to interface `i1`.

`c`'s knowledge of `i` is then wrong. This kind of mistakes comes from the stateless nature of the management system. Rather than systematically freezing the attendees (which will disable us to address one of the major point stated in § 2), better seems to preserve the nature of the system but to add the *session* notion. When a session is required by a context, the system either

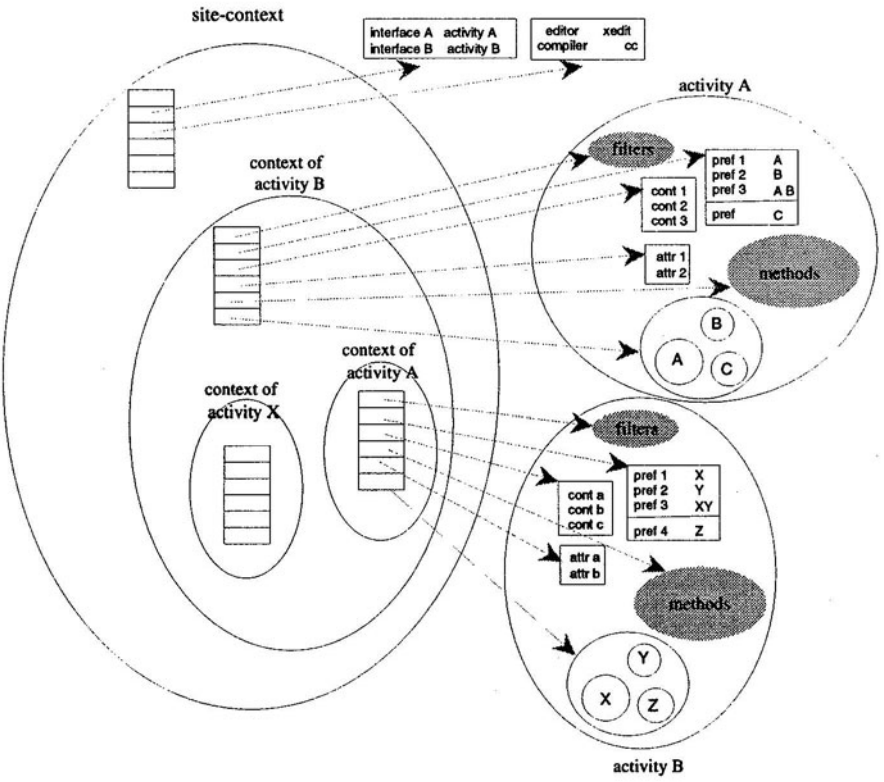

Figure 4 The layout of contexts and links with activities

ensures that the vision the context has will be stable as it will temporally freeze branchs of the tree of contexts, or at least notify the context about modifications that could not be avoided, as for example an exit of `i`.

A session may sometime disables — as long it has not been held down — the execution of attendees (the registration of filter with write privilege e.g.).

(b) Some particular contexts

Finally, to take into account existing computing, two kinds of contexts has been added: *user-contexts* and *site-contexts*.

A user-context includes all the contexts which belong to a given user. Its interest resides in the contribution offered to the security management.

The site-context enables to group the contexts present on a site/computer. While conceptually articifial, it takes into account the real world constraints.

- A context is generally tied to a computer, even if sometimes some of it sub-contexts may execute remotely.

- basic resources are provided by the operating system of the computer (network, access rights, files, ...). Operating systems are generally centralized, in spite of the emergence of distributed operating systems.
- as previously described, management induces particularities in the forwarding of messages. The site-context enables the use of existing network protocols.
  The specific part of our management approach is implemented in this site-context.

It is in the site-context that *translation* capabilities or that the description of services provided by the actors are registered.

4 THE KERNEL OF MANAGEMENT SYSTEM

The kernel of the management system (*kMS*) is used to implement and represent the management system on a computer. It corresponds to the services that the site-context must provide and to the basic services of the management system. *kMS* implements for a site the set of methods that the execution of an activity may require, such as communications, printing capabilities, etc.

Moreover, as representing the site-context, the kMS provides a way for activities that doesn't implement mandatory methods (security methods e.g.) to delegate them to it.

It's worth noting that the kMS has a global vision over the contexts of the site. This makes the kMS the manager of all the site available resources (with their interfaces) that activities may ask for.

More generally, kMS ensures coherence of the contexts in the site and it is in charge to interact with major external components such as languages interpreters or SNMP.

5 A SIMPLE EXAMPLE

In this example we assume that user `U` asks the kMS to provide a tool with editing capabilities, i.e. that is compatible with the editor interface:

```
<implement os:system>
filename: octet string
open
save
printerToUse(octet string)
print(octet string)
...
```

The kMS then locally searches for a registered activity having an interface

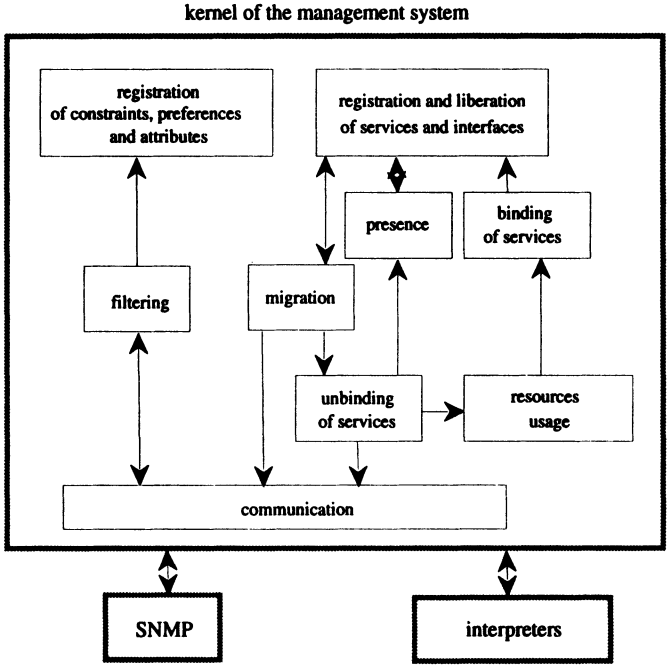

Figure 5 Major components of the kMS

conformant with the editor interface. An activity locally known by the kMS to conform to an editor/compiler/linker interface, thus including the editor interface, can be selected.

The arrival of the editing context triggers a set of management functions. One of them will set the printer to use in this context.

This is done by the triplet: `(from: ANY, method: enter, arg=editor interface)` tied to the function:

```
send("printerToUse", "my-favorite-printer")
```

We will present three management functions for the printer. The first one is initiated by the user of the editor, the other two by the owner of the printer, comparable to the only authorized manager in traditional management schemes: we will assume that the printer belongs to `root` in the rest of this section.

Firstly, we notice that the printer `my-favorite-printer` can be managed according to its known interface:

```
paper: boolean
state: integer // -1:error, 0:ready, 1:busy, 2:not responding
force
```

We assume that the user management function is in fact pain-relieving and will translate its data into a standard two columns Postscript form.

This will be achieved by the introduction of a management function acting by the mean of an output filter which translates the text data to be printed in the required form:

```
filter: (Editor of U, "print", ANY)
code: process(msg) {
        data = getArguments(1, msg);
        ps_data = enscript2rGh(data);
        out(msgHeader, msgMethod, ps_data);
        }
```

This management function is created in the context of **root**, the owner of the printer, as depicted in Figure 6.

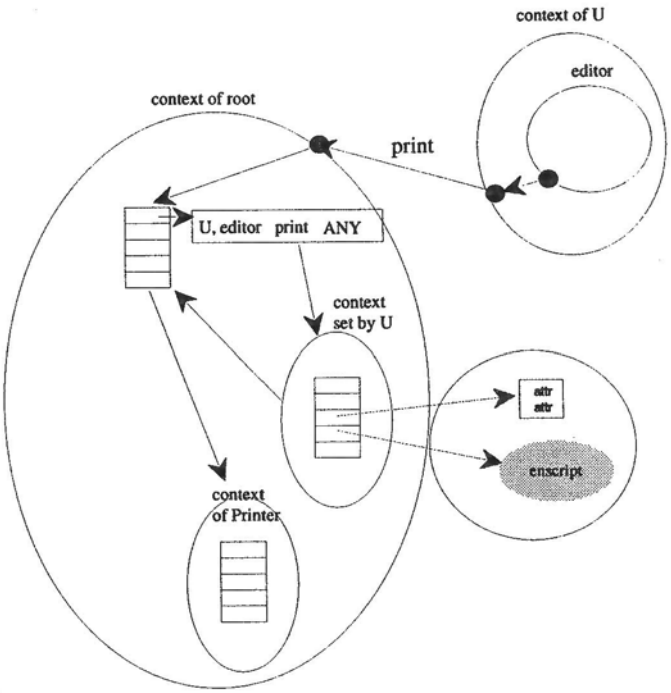

Figure 6 Contexts state after insertion of the user **U** management function

So, anyone can use the management system for its own needs according to its privileges. But, simultaneously, other management functions can coexist around the printer context: for example, management functions introduced by the owner of the printer.

Let's consider two of them. The first one monitors the requests sent to the printer and logs them; the second one automatically solves the paper size trouble (A4 paper *versus* US legal paper), forcing the printing with the actual paper.

The monitoring function is a filter put by **root** around the printer, defined by the triplet **(from=ANY, method="print", arg=ANY)**. The function attached to this filter will in turn locate the sender of the message and displays its name on the administrator's console or logs it in a file.

This function, initiated by the owner of the printer is put in a low level context around the context of the managed printer.

It can be noticed that, because the context of the printer is diped in the printer owner context, a kind of priority between management functions is implicitly induced: highest priorities for the system and the owner of the application, lowest priorities for other users.

This way, the monitoring function is placed closer to the context of the printer than the user management function (Cf. Figure 7).

The management function in charge of recovering paper size troubles acts in a different way. Let's assume that this dysfunction is not notified by an event (which would then be catched by a management output filter function), and that we are faced to an interface in which such a trouble is only notified by the setting of the variable **printerError** to **PAPERSIZE_ERROR**.

In such a realistic scheme, the paper size management function is achieved with a *variable filter*, this filter checks the condition:

```
printerError == PAPERSIZE_ERROR
```

and triggers the **force** operation when it becomes true.

In this example, we showed that the management allows to automatize the recover of the cumbersome paper size trouble; whereas incoming requests are monitored and any user may put a processing function for it's own convenience.

6 INTEGRATION INTO SNMP

As we want our management system to operate upon existing tools, it is important to incorporate a prototype into the popular SNMP framework. Integration under CMIP would be very similar and SNMP is preferred to SNMPv2, far less popular. Integration under SNMP is achieved by *Smux* (Rose 1991) protocol capabilities and is based on a complete ASN-1 description of the kMS, and as a consequence of the different entities presented in § 3.

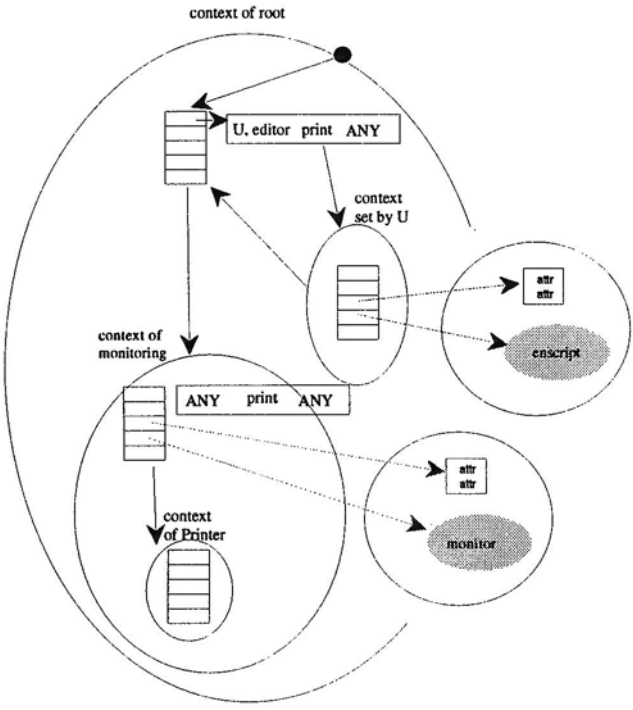

Figure 7 Contexts state after insertion of the monitoring management function

The obtention of the kMS mib isn't a straightforward process. It needs to use both complex indirections (Waldbusser 1991) and multi-types entries (Case & Levi 1993).

Though this description deals with kMS entities, it doesn't solve the interface aspects. Interfaces describe variables and method signatures that entities support. As long as this kind of description is usually achieved using the TYPE-OBJECT macro, interfaces appear to compete with this macro: we could call them *virtual MIBs*.

To be coherent with SMI and SNMP, object identifiers (*OBJ-IDs*) tied to variables and methods described by an interface must be MIB conformant and accessible for clients:

— kMS translate* variables to their equivalent ASN-1 representations, and methods to SNMP groups, according to the SNMP naming policy. From the

*This scheme can be altered as we can explicitely associate OBJ-IDs to variables or method's signatures.

mib point of view, the resulting object identifiers are always set to constant locations, relatively to the context.

— Clients have to know the virtual mib built from the interface, as this information is necessary to properly encode or decode SNMP operation arguments. This process is dynamic and can only be determinated at run time. For each interface OBJ-ID of the virtual mib, the Mib-manager sends a client-specific SNMP-trap from which the client updates its mib knowledge. Figure 8 shows the interaction between the Mib manager and kMS.

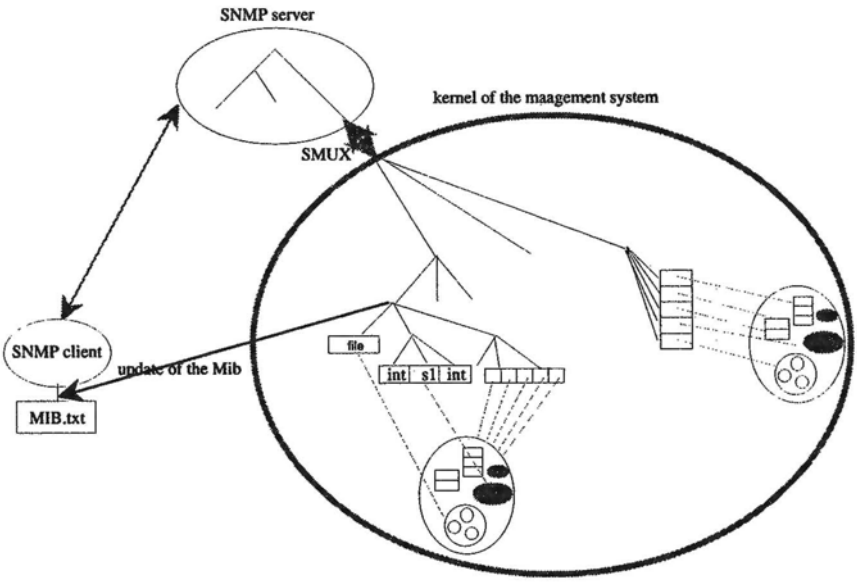

Figure 8 Integration with SNMP

From the Mib-manager point of view, kMS appears as a proprietary mib interacting with Smux, whereas kMS creates for any SNMP-client a context by which the client can interact over the whole management system.

Integration in SNMP gives the human administrator several possibilities to manage applications:

- by building new management functions or activating already existing ones (under both SNMP and kMS control security access),
- by calling over the network a complete management package: the packed management functions will first obey the administrator context environment constraints and then start the application management.

7 CONCLUSION

Although network management is nowadays world widely available, software application management is still under development.

From the basically *Integrated environments* architecture scheme, we demonstrated that applications modelized from active objects provide an homogeneous framework containing both management entities and their description. The system introduced appears to be evolutive and customizable while the frozen *MIB* notion has been reconsidered.

In spite of differences between applications and network management, opportunities offered by standardized management protocols has enabled the application management system to be integrated within existing network management tools.

REFERENCES

Agha, G. (1986), *Actors: a model of concurrent computation in distributed systems*, MIT Press, Cambridge Mass.

Boyer, F. (1994), Coordination entre outils dans un environnement intégré de développement de logiciels, PhD thesis, Bull-Imag et Université J.Fourier.

Brockschmidt, K. (1995), *Inside OLE, 2nd edition*, Microsoft Press.

Brun, P. (1987), Conférence répartie en mode messagerie, PhD thesis, Ecole des Mines et Université de St-Etienne.

Case, J., Fedor, M. & al. (1990), Simple network management protocol (SNMP), Technical report, IAB.

Case, J. & Levi, D. (1993), SNMP mid-level-manager MIB, Technical report, SNMP Research, Inc.

Chapman, M., Dupy, F. & Nilson, G. (1995), 'Overview of the telecommunications information networking architecture', *Tina'95* .

Danielsen, T., Pankoke, U. & al. (1986), 'The amigo project: advanced group communication model for computer-based communication environment', *CSCW 86 proceedings* .

Hewitt, C. (1977), 'Viewing control as patterns of passing messages', *Artificial Intelligence* .

ISO (1989), Information processing systems, open systems interconnection, Technical Report 7498-4, ISO management framework.

Kaddour, B. & Beigbeder, M. (1996), 'Application management by active objects', *Ecoop, workshop on network management* .

Lieberman, H. (1986), 'Using prototypical objects to implement shared behavior in object oriented systems', *OOPSLA* .

Nierstrasz, O. (1993), 'Regular types for active objects', *OOPSLA 93* .

O.M.G. (1991), The common object request broker: Architecture and specification, Technical report, Object Management Group.

Reiss, S. (1990), 'Connecting tools using message passing in the field environment', *IEEE software* .

Rollin, F. (1986), Transfert de fichiers en mode messagerie, PhD thesis, Ecole des Mines et Université de St-Etienne.

Rose, M. (1991), SNMP MUX protocol and MIB, Technical report, IAB.

Stein, L. (1987), 'Delegation is inheritance', *OOPSLA* .

Venkatasubramanian, N. & Talcott, C. (1993), A meta-architecture for distributed ressource management, Technical report, Univerty of Illinois - Stanford University.

Waldbusser, S. (1991), Remote network monitoring management information base, Technical report, IAB.