

Using Aspects and Compilation Techniques to Dynamically Manage Applications

Bernard Kaddour

LIRIS

University of Lyon I
Villeurbanne, France

Joël Quinqueton

LIRMM

Montpellier, France

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 transparently. The low level management part uses an agent tightly tied to the JVM/JVMTI and able to receive and to manipulate programmable monitoring expressions. To avoid conflicts with applications already using JVMTI instrumentation techniques, we dynamically compile these expressions into machine code to reduce overload and minimize performance lost.*

Keywords: software management, JAVA, Aspects, JIT compilation.

1 Introduction

Since some years, 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, 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][18] for new distributed applications. These middleware are currently also available for small devices and even Multi-Agents Systems based solutions have been introduced in the research area [4]. Such new distributed application whose components may partially or totally be executed by small devices implies new flexible and optimized needs for their management to be effective. Network management standards exist [19][20], but their intrinsic centralized and frozen aspects are very limitative [16]. Newer and more application centric solutions have appeared [6][7][10], but

they don't always adequately address the evolutive aspect of management. 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 [13] 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. The later introduces programmable monitoring tasks tightly related to the code of the managed JAVA entities and is concerned by performance optimization aspect. Importance of performance for management has already been depicted [17]. The massive incoming of network services for small devices imposes to take this point into account and we chose to compile our monitoring tasks to minimize management impact from a performance point of view. In this article we briefly 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 and how filters are layered. Section 5. presents the low level management part and the monitoring tasks. We conclude with some future works and perspectives.

2 Key features of our architecture

The approach consists in a core management system – named *kernel of the management system* or *KMS* – where management applications can be deployed. KMS largely relies on the interception of incoming and outgoing messages exchanged by the managed application. KMS can then spy and filter requests sent or results received by the components of the managed application accordingly to management functions requirements.

Management applications supported by the KMS are build up 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.

A management application received by a KMS can be deployed: an environment is created for the management functions while filters are dynamically linked to the components for which they have to intercept messages.

3 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 by byte-code injection [2]. 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. These features give the possibility to consider JAVA applications as manageable entities.

For Corba applications, OMG provides some basic introspection services: Portable Interceptors [18]. 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 management system which are mainly JAVA centric.

Then, from now, we will consider only JAVA applications and we will consider them as fully manageable entities. The following section focuses on the representation of management activities and on the messages filtering system.

4 Major components of the system

The management system *KMS* is implemented as a daemon-process to permit sub-activities exchanges between KMS (currently through sockets). KMS inspects and registers the entities it is in charge of and interacts with AspectWerkz [2] for filtering purposes. It 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

Main parts of an activity are:

- 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,
- Sub activities*: greatest difference with other tools is their expected autonomous abilities. The activity can ask the KMS to send them, as a whole, over the network towards another KMS which will then deploy them remotely. A sub-activity is an activity,
- constraints and preferences*. They can decide to allow, modify, or reject the requests received by the activity itself. The loading of a particular characters font for an application which conforms to an editor interface is such an example,
- monitoring expressions*. These are expressions the activity wants at low-level management for direct code monitoring tasks (Cf. §5.).

An activity is encapsulated in a *.jar* class file. It is the activity responsibility to report to the KMS the interface or set of services it can respond to.

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 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, 0);
    register_inFilter("Service", "sub", ic, 0);
    Positive p =new Positive();
    register_outFilter("Service", "sub", p, MOD_PRIV);
    register_Constr(new UneContrainte(), ANY_IN);
    ...
    register (this, "Count");

    Printing s =getServiceInterface(
        "A_PrintingService", FROM_ANY);
    ... = s.doSomething();
    ...
} } ]

```

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 partially be managed), its filters moved or stopped by the KMS.

4.3 Deployment of an activity

When an activity *A* is deployed by the KMS, the KMS firstly ensures by introspection 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 towards another KMS) or requires 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 the necessary retries before sending back to *A* the resulting service. This C&P mechanism gives to the management system some capability to adapt environment resources with activities wishes.

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.

4.4 Aspect oriented programming

This paradigm mainly due to [13] has for objectives to capture some singularities that aren't properly take in charge by the object oriented model (OO). In practice, Aspect oriented programming (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 then introduced some concepts for the addition or the 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* define both the code of the aspect and, combined with join points, when they will have to be executed. Usual advices are *before advices* if we expect the new aspect's code to be executed before the join point, and *after advices* if we expect the new aspect's code to be executed after the join point.

AOP frameworks are now available [1][2]. AspectWerkz (AW) [2] is preferred as it provides a so-called online-mode to operate upon already compiled pieces of code (.class) and, most interesting, has a useful API providing all of the requirements needed to dynamically manipulate aspects.

4.5 Layout of the filters

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. 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 and 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 for 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 the 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 follows 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.

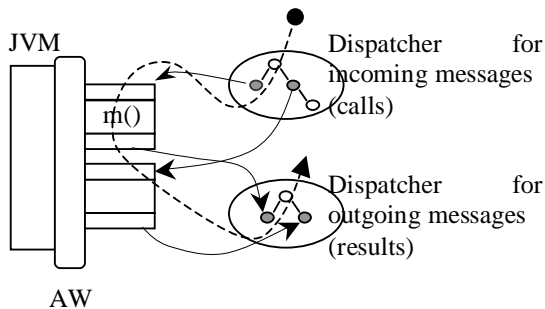


Figure 1. Filters and messages paths.

Left part of Fig. 1 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 has to pass through activity's out-filters and finally the KMS out-filters before being sent back to the requester.

5 Monitoring expressions

Managing software components composed of classes and/or instances typed variables, some low-level management means are necessary to directly deal with them for collecting or monitoring purposes. These tasks are devoted to the low-level part of the management system build up of *monitoring expressions (eMons)* and

their associated agent which is the core of our monitoring system.

A monitoring expression is mainly composed of (i) expressions based on fields and variables present in the different classes and instances found in the managed application code and (ii) an event to emit when the main expression becomes true.

eMons can be considered near to the DISMAN [14] – and its associated MIBs – spirit and as a generalization which doesn't only limits to SNMP variables but also supports JAVA software applications and their JAVA byte-code. Let's consider the class:

```
class C {
    static public int value =0;
    int field;
    public void m() {
        int local_m0;
        int local_m;
        :
    }
}
```

Some management task *T* could ask to be informed with an *ev_value* event when the expression $C.field + C.value - C.m().local_m$ becomes negative. This could be achieved by introducing the monitoring expression:

Watch for: $C.field + C.value - C.m().local_m < 0$
Event: *ev_value*

To cover more general needs, we have introduced a small language to describe eMons, their scopes, relationships and definitions. Here's the core of an eMon entry:

```
eMon_name {
    ...
    dec ::= {
        m is #"C;"m().local_m;
        m0 is #"C;"m().#0; //==local_m0
        C isClass #"C;"
        float temp_1;
        int temp_2;
    }
    exp ::= {
        temp_1 =(float)(m0 -m +2);
        {
            #C.field + #C.value -m <0
        }
        temp_2 =@@last();
    }
    ...
    event callback 12; // ev_value
    ...
}
```

describing an eMon named "eMon_name". Scope, thread or specific tasks to execute when an eMon is removed are omitted here. We will concentrate on expressions embedded within the eMon as they constitute the most important part of an eMon.

We start with some declarations. They can be simple shortcuts for ease, typed or not, in relation with the monitored code, e.g. (m , $m0$, C), or some local temporary variables used for the eMon own needs. Local temporary variables, such as $temp_1$, must be typed but they have not relation with the monitored code.

The expressions body, composed of the main monitoring expression enclosed in brackets. It can be preceded by expressions to evaluate and store *before* the main expression is evaluated whereas following the main monitoring expression are expressions to proceed *after* the main expression evaluation. To gain in flexibility, some add-ons are allowed:

- built-in functions provided by the system. This set of functions includes usual system-call functions ($kms_date()$, $kms_arch()$, $kms_strstr()$, ...).
- Special MIB access functions. As the system is designed to interact with the SNMP protocol [19], a group of functions allowing MIB [20] accesses is included [12].
- built-in functions in relation with the underlying monitoring system. They give to the eMon information directly provided by the JVM. A thread, object or class ID can be obtained this way. For example, $@@last()$ is the last evaluated value of the main expression.

When the expression is removed by the system from its table of expressions to monitor, a notification is sent to the activity owner of the expression (by mean of an *int expressionReached(int expId)* call, a method implemented by the *Activity* class). SNMP events are also supported [12].

As activities can automatically insert large sets of eMons, their processing and evaluation are to be optimized. Both a compiled approach and the introduction of an algorithm avoiding unnecessary re-evaluations of eMon's expressions are used for better performances. This problem of the re-evaluation of number of expressions has similarities with the challenge of filtering high throughput network [9][21], and some of their ideas [21] are used in the evaluation algorithm. On the other hand, compilation will target real CPU native code. We chose the IA32(x86) as it is the most representative and common CPU currently used.

Some differences with usual compilation exist. First, we have over the used variables a much more lighter control

than a compiler usually has. Variables are "lend" by the JVM and can at the best be read or modified through getters and setters functions. An optimization is to avoid these time consumers Java Native Interface /JVM Tool Interface (JVMTI) [11] functions as far as we can and rather use a cache mechanism for the objects, fields and local variables the expressions need. Second, eMons' expressions are loosely coupled with the application code they monitor and are asynchronously evaluated with the Java code executed by the JVM. Hence, we can't statically discover from the monitored code some registers optimizations for the eMon's generated code.

5.1 The overall architecture

The used architecture takes advantage of features provided by the JVMTI introduced by Sun. At first available through the JVM Debugger Interface /JVM Profiler Interface couple, this API offers the possibility to attach an agent to the original java byte-code. This agent can then register customized hooks at run-time to the java byte-code, e.g. to be informed when a field is modified or a class loaded into the JVM.

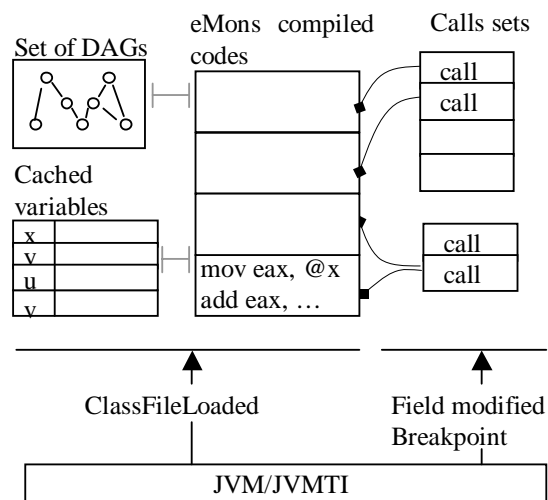


Figure 2. Agent architecture.

A possibility is, for an agent, to modify or inject new java byte-code in the original one. This is known as instrumentation [15]. Despite attractive as the new injected byte-code is next automatically compiled by the underlying JIT, we have not chosen a pure instrumentation approach for at least two reasons.

First, instrumentation freezes our monitoring process. When inserted, an eMon is considered as always active and can't afterwards easily be removed, nor we can

dynamically add or activate new incoming eMons. Second, only one agent can modify the byte-code this way, this is the choice AspectWerkz [2] already made to provide aspects used by the high level management part of the system.

We then chose the more drastic approach to take in charge the eMons compilation process. Our agent is the core of the monitoring system. First, the agent places its hook functions to be executed when a class is loaded, when a thread enters in or exits from a method, when a field is modified, etc. or when a breakpoint event is generated by the JVMTI. The agent creates its own memory zones, mainly the pool of cached variables and storage areas for eMons compiled code.

When an eMon is inserted in the agent, it is parsed and its DAG representation constructed, but the code can't be immediately generated as neither the variables/fields types nor their storage zones are known. The set of these awaited information is build and is collected by the different hooks when they are activated.

Our previously registered *loadClass* hook is called when a new class is loaded. We can then obtain types information for fields or method local variables while cached memory zones are allocated for the agent needs. These cached zones will be later used by the eMons compiled code. A point is that the current JVMTI provides the necessary API to synchronize cached fields by means of a dedicated events/hooks mechanism, but nothing is available for local variables. Then, the agent, without any instrumentation modification intents, walks through the byte-code methods and looks for java op-code offsets where local variables are modified. Breakpoints (JVMTI's breakpoints are simple events sent to a *Breakpoint() event manager*) are placed for each one of the encountered offset while we register the index of the modified variable. Finally, two tables are created for the breakpoint event manager.

The first one is an association *offset-calls sets* where a calls set is a sequence of call cells, each cell containing a call execution code towards the compiled code to execute for the re-evaluation of one expression involving the modified local variable. A call cell can be empty and is modified as the eMon activations/un-activations. Calls sets are modified accordingly to eMons removal or new eMons arrival. This two stage call process – rather than one or two list(s) containing addresses of compiled code to re-evaluate – was preferred as eMons arrival/removal once the java code is in execution remains rare. This scheme avoids unnecessary C(agent)/Asm(eMons) switches and CPU cycles losts.

The second is a simple table (offset, local variable index) used by the breakpoint event hook to later synchronize the

cached variables when updating the value of the modified local variable (*GetLocal<Type>()*).

Finally, eMon's DAGs for which the new incoming class was the last expected one to complete information variables are then compiled. The operations types are evaluated and the associated code generated using addresses in the pool of cached variables. The generated code tries to minimize the expressions re-evaluation process.

5.2 Underlying compilation idea

To describe how the most often used parts of expressions are compiled to reduce the overall re-evaluation process, we'll treat the case of the expression $E: (x + y) + (u * v)$, without object reference for the sake of simplicity. We name *bloc* a part of the DAG with only one operator. For each bloc, we reserve a storage cell where the last evaluated result can be store. The expression E is formed by the blocs $(x + y)$ and $(u * v)$. The generated code of each bloc is immediately obtained from code templates that we instantiate according to the types, the operation and an available free register at compile time, tracking of free registers is done with a usual free registers stack.

The generated code for the bloc $(x + y)$ obtained from the (int, int, add) code template is:

```
mem_xy    dw 0
ev_xy     mov reg, @x; // free reg == reg
          add reg, @y; // op == add
          mov mem_xy, reg
          ret
```

as storage zone of x and y in the cache are known. The same goes for the bloc $(u * v)$, with the next free register.

The whole machine code associated to the expression E is obtained in linear time from the generated codes of $(x + y)$ and $(u * v)$. It is composed of an evaluation code if x or y is modified, an evaluation code if u or v is modified and a complete evaluation for the most general case. Here is the evaluation code of E when x or y is modified:

```
mem       dw 0
_Entree_xy call ev_xy
          add reg, mem_el_uv
          mov mem, reg
          ret
```

A table (modified variable, entry point) is maintained and later used to select the appropriate evaluation code to execute (via a call cell) when a variable is modified. This way minimizes the re-evaluation of common sub-expressions. For example in the case of $0 < (x + y) + (u * v)$

`<5,(x+y >0 && x + y <3);` the bloc `x + y` is effectively evaluated only once if `x` (or `y`) is modified and never for a modification of `u`.

A major drawback of the approach is the extra memory needed. This is an often seen counterpart when using cloning techniques [5] as expression cloning appears, not depending of variables types, but depending of the modified variables. Another disadvantage of this approach is to consider for small expressions with e.g. 3 variables, where a usual compilation consummates less memory and gives a better execution time. But when the number of variables is over 3 or when a common sub-expression is shared by many expressions, using a partial re-evaluation mechanism seems better. It has finally been chosen to use partial re-evaluation only for DAGs used more than 3 times by eMons (the extra-memory used by the generated code is then acceptable) and if no more than one object instance is needed.

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 dynamic manner. We suggest to use Aspects frameworks facilities as a possible solution to allow an evolutive and decentralized high level management scheme without modifying neither the underlying middleware nor the managed objects.

A complementary low-level management part able to deal with managed software components at run-time is proposed. It uses facilities provided by the JVM/JVMTI and compiles monitoring expressions on-the-fly to improve performance of the management system.

Currently both high and low level management parts remain too tightly coupled. In particular in the case of distributed management, complex monitoring expressions involving instances executed on different systems (for example variables in the client classes code and variables in the server classes code) need to communicate through their associated activities. We plan to definitively separate the monitoring part of the management system and enhance agents with communication abilities for variables values exchanges, leading to an autonomous distributed monitoring system.

A second investigating field could be the creation of MIBs from which standardized management tools could interact with the proposed system.

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, pp. 551–579, 2003.

[4] C. Carabelea, O. Boissier, Multi-Agent Platforms on Smart Devices: Dream or Reality ?, *1st Smart Object Conference*, 2003.

[5] J. Dean, G. Chambers, Towards better inlining decisions using inlining trials, *LISP and Functional Programming*. pp. 273-282, 1994

[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] P. Gupta, N. McKeown, Packet Classification on Multiple Fields, Proc. *ACM SIGCOMM.*, 1999.

[10] JMX, Java Management extensions, <http://java.sun.com/products/JavaManagement/>, 2000.

[11] JVMTI, JVM Tool Interface, v1.0, <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>, 2005.

[12] B. Kaddour, J. Quinqueton, Administration d'applications distribuées par Aspects et expressions programmables, *GRES'05*, 2005.

[13] G. Kiczales & al, Aspect oriented Programming, *Proc. of ECOOP'97*, pp. 220-242, 1997.

[14] R.P. Lopes, J.L. Oliviera, Delegation of expressions for distributed SNMP information processing. pp. 395-408, Proc. *IM 2003*. 2003.

[15] J. Maebe, K. De Bosschere, Instrumenting self-modifying code, *AADEBUG2003*, 2003.

[16] J.P. Martin-Flatin, S. Znaty, A Simple Typology of Distributed Network Management Paradigms, *DSOM'97*, pp. 13-24, 1997.

[17] J.P. Martin-Flatin, Web-based management of IP networks and systems, *PhD Thesis*, 2000.

[18] OMG, <http://www.omg.org/>, O.M.G., 2004.

[19] RFC1905, Operations for Version 2 of the Simple Network Management Protocol (SNMPv2), <http://www.faqs.org/rfcs/rfc1905.html>, 1996.

[20] RFC1907, Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2), <http://www.faqs.org/rfcs/rfc1907.html>, 1996.

[21] V. Srinivasan & al., Packet Classification Using Tuple Space Search, Proc. *ACM SIGCOMM.*, 1999.