RZ 3363 (#93409) 08/27/01
Computer Science 9 pages

Research Report

Storage management using CIM and JMX

Christian Hörtnagl

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

hoe@zurich.ibm.com

Storage management using CIM and JMX

Christian Hörtnagl

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

We present a transparent CIM-to-JMX gateway for combining complementary advantages of two partly competing management technologies, the Common Information Model (CIM) and and Java Management Extensions (JMX), namely a rich data model and straightforward language binding. We include overviews of both technologies and discuss how this combination in particular enables quick prototyping of CIM provider code for a storage management application.

Keywords: management instrumentation, data model mapping, Common Information Model (CIM), Java Management Extensions (JMX), storage management.

1 INTRODUCTION

We are working on a research prototype for policy-based storage management that is suitable for service providers [1]. Its aim is to enable intelligent placement decisions for file and digital media resources by taking both current availability and performance of contributing resources into account. As storage systems increase in capacity and complexity, common abstractions such as file systems subsume an increasing number of distributed system components. For instance, in the case of Storage Area Networks (SANs) a substantial network infrastructure must be controlled before storage services can be delivered with predictable levels of service (QoS).

We therefore investigate policies as a means to translate between service-level business requirements and technology-level resource provisioning steps in an automatic and scaleable fashion. We want to enable service providers to bundle storage offerings as part of larger application hosting suites (together with server and network resources), and to grant different levels of services to groups of customers.

In order to enable a flexible design of the policy decision layer, we are interested in building on top of a common resource description framework that allows us to inquire and control actual system components. Different management technologies, such as SNMP [2], CIM [3], and JMX [4] address this need to different extents by offering appropriate data models and/or APIs.

Our preliminary selection of the Common Information Model (CIM) [3] was driven by the Storage Networking Industry Association’s (SNIA) investment in this standard. Combination of CIM with Java Management Extensions (JMX) [4, 5] was also attractive because all of our software is written in Java.

For the storage management application, we simulate new storage networking architectures by suitably configuring commodity equipment, such as Linux workstations (as IP storage routers, say). Very often we need to prototype our own (CIM) adapters, and obviously we cannot spend much effort on this item during development. In this context a combination of CIM and JMX proved very useful and efficient. We will discuss the solution’s potential in more general terms throughout the rest of this paper. The basic rationale for the combination results from the following observations:

- CIM defines a management data model with a fairly deep and wide coverage; it was designed in a relatively short time, and its definitions therefore avoid inconsistencies and duplications (perhaps at the price of some blind spots). Its access protocols from the Web-Based Enterprise Management (WBEM) extension [6] adopt contemporary web standards. Hence we were interested in CIM for its rich and convenient data model and adoption of XML-based protocols.
- JMX deals with management instrumentation of Java software (or of other entities that receive Java-based wrappers). It establishes conventions for a new sub-category of JavaBeans, called MBeans. JavaBeans are the basic building blocks of Java’s component model, and as such are widely understood and deployed.

In section 2 we present a more complete characterization of the two technologies; section 3 explains the transparent CIM-to-JMX gateway, by introducing its run-time (generic CIM provider) and compile-time (MOF compiler) components in turn. Both the Distributed Management Task Force (DMTF), which coordinates the CIM effort, and Sun’s Java Community Process program, whose mandate includes JMX, still sponsor active working groups which may look at mutual gateways from their own perspectives at some time. We present first experiences after designing, implementing and using an actual gateway as possibly guide to further action. The final section 4 presents our conclusions and briefly discusses some of the limitations that relate to the current prototype.

2 COMPARISON

Figure 1 characterizes CIM and JMX in terms of relative scopes (bars) and specializations (labels). Both of them and the Internet's Simple Network Management Protocol (SNMP) [2] share basic architectural elements: agents are in charge of single (or few) managed resources. They report on and influence resource states using appropriate local means (providers), are typically colocated with resources, and in the case of software resources may share the same address space (subagents). The information from several remote agents is collected and accumulated at a central location by a manager, using a suitable intermediary communication protocol. This is also where clients, such as the storage management application, may retrieve consolidated management information.

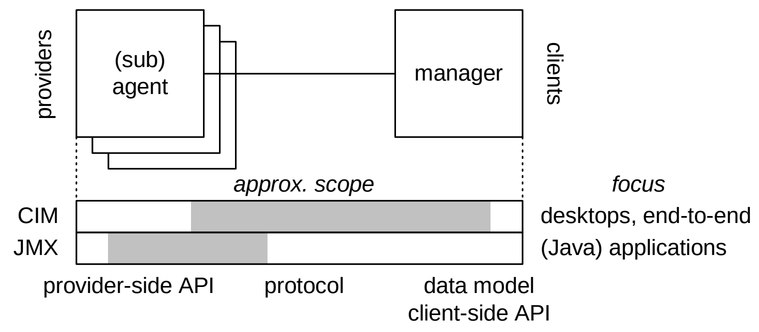

Fig. 1 coarse differentiation of two management technologies.

The Common Information Model (CIM) [3] defines protocols (e.g. XML mapping in WBEM [6]) and a data model capable of representing a range of managed objects from hardware devices and software elements to global policy and end-to-end state information (e.g. DEN [7]). Its data model is broader but still more shallow than SNMP's total, although facilities for extending it and for including SNMP data exist. Existing Java implementations adhere to common versions of both provider-side and client-side APIs (Java WBEM API [8]). However, adding management logic at the provider side requires substantially more coding effort than with JMX (gap on the left).

Java Management Extensions (JMX) [4] is deliberately designed as model-agnostic (at least for the time being [5]), and it imposes no particular protocol nor client-side API (large gap on the right). In addition to the provider-side API that is mostly implicit in the definition of MBeans as first-class Java objects (section 2.2), there are facilities for introducing a range of protocol adapters as needed. We make use of this flexible capacity in the design of the CIM-to-JMX gateway.

Figure 1 shows that the combined scope of CIM and JMX promises broad coverage from language bindings (left aspects) to data models (right aspect), thus supporting storage management solutions that are capable to scale both in terms of implementation cost (software engineering metrics for the provider code) and portability (standard compliance e.g. for client code). We associate the remaining gap on the left side mainly with situations where

Java-based providers may be inappropriate; the gap on the right side refers to imperfections of the CIM model. The next subsections elaborate further on the two target technologies.

A summary of their comparative strengths is given in table 1 (section 4). On their downsides, we have observed that full-scale adoption of the CIM standard is slow, and that implementations tend to be heavyweight. JMX requires a Java execution environment (a heavy burden on many devices that need instrumentation outside a prototyping context) and it has not seen much use except in the niche of web application administration so far.

2.1 Common Information Model (CIM)

At the core of the Common Information Model (CIM) [3] is a broad schema of managed objects specified in Managed Object Format (MOF) notation. Modeled objects range from very coarse and generic (e.g. policies) to fine-grained and specific (e.g. device configuration) classes, hence providing an appropriate basis for uniform end-to-end resource management. Associations and aggregations between objects are made explicit as objects as well. In total, CIM version 2.5 defines approx. 850 classes; the schema is extensible via the usual means of object-oriented data modeling. Figure 2 shows a subset of those CIM classes that are relevant to storage management tasks.

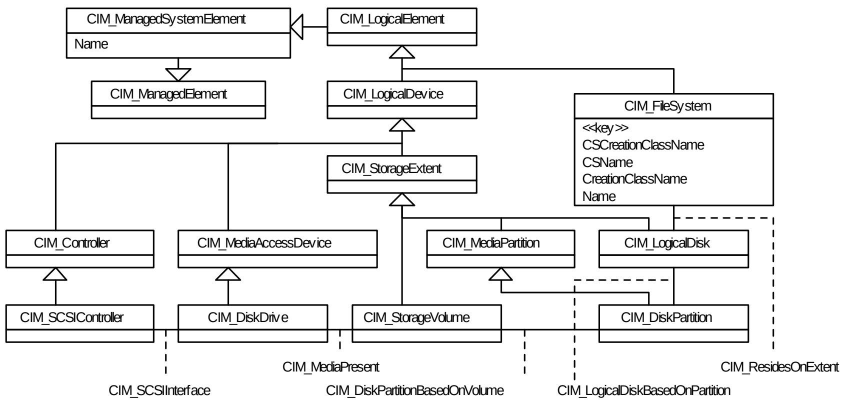

Fig. 2 hierarchy of CIM classes for storage management (samples).

The data model at the core is complemented by protocol facilities and specialized schemas, with different vendors now committing to different subsets for example as follows: Microsoft WMI (Windows Management Instrumentation) complies with an early version of the core CIM schema and concentrates on desktop management. The Web-Based Enterprise Management (WBEM) [6] initiative adds HTTP/XML protocol mappings; an implementation is available from Sun (Solaris WBEM Services). The Directory Enabled Networks (DEN) initiative investigates directory-based control and extensions of the core schema towards network elements and services; Cisco has made an early commitment there [7]. Although SNMP competes with CIM in practice, the IETF policy working group uses CIM terminology for its work.

Our prototype is based on SNIA's open-source Java implementation of CIM [8]. As a portable CIM framework, it does not include any specific providers, and the CIM-to-JMX gateway is our methodology for quickly adding the necessary instrumentation to the framework. The SNIA implementation adheres to the provider-side and client-side portions of the Java WBEM API. The same is true for the Solaris implementation, hence this API forms a de-facto standard for CIM access in Java implementations, and its formal ratification is under way [9].

The basic entities in figure 1 translate into the following CIM terminology (variations exist): agents correspond to CIM object managers (CIMOMs), subagents correspond to CIM providers, and from the CIM perspective, the CIM-to-JMX gateway occurs as a CIM provider for JMX MBeans.

We observe that CIM makes a single shared data model its primary concern; we call this a top-down approach where the model's conventions (shown at the top in figure 3) are uniformly imposed across a range of underlying resources in an attempt to facilitate global management decisions by hiding differences in an abstraction layer. The top-down characterization will allow us to contrast JMX and place the CIM-to-JMX gateway as a solution that combines both ways in synthesis but keeps an emphasis on top-down to help adoption of a consistent data model.

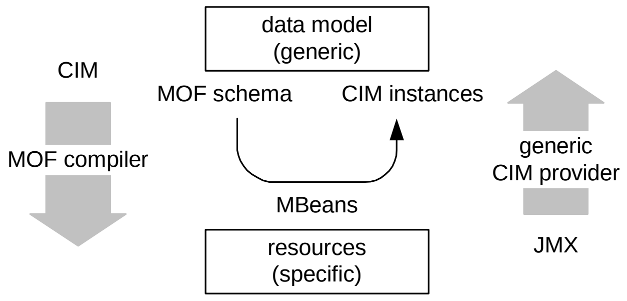

Fig. 3 top-down vs. bottom-up relationships.

2.2 Java Management Extensions (JMX)

JMX [4, 5] proposes itself as an insulation layer between resources that need to expose their management information in a simple and generic way (by manipulating first-class Java objects) and management tools that make technology-specific assumptions about management protocols and APIs (e.g. SNMP, CIM). Because interpreted Java byte code contains redundant type information that can be inquired at run-time using reflection, a degree of freedom is added by introducing Java-based interfaces at the boundary between agents and managers.

In effect, two advantages result when Java interfaces replace (compile-time) interface definition languages such as MOF: first, they have a well-defined run-time (plus their compile-time) representation; mapping decisions may therefore be delayed until run-time. Second, if managed resources consist of code that is (at least partly) written in Java, the relevant interfaces are readily available and management instrumentation code has the form of normal first-class objects implementing those interfaces. Because of the second, JMX is particularly suitable for (Java or mixed) application management.

JMX management interfaces comprise normal Java interfaces with the name suffix MBean. Their conventions for providing read/write access to management attributes, for allowing invocation of management operations, and for triggering/receiving management events are all inherited from JavaBeans. JMX is therefore an extension of the normal Java component model, and its components are called MBeans (JavaBeans for management). Standard MBeans implement the management interfaces directly. Dynamic MBeans report their interfaces in an equivalent data structure, thereby allowing variations at run-time and implementations in other languages. Open MBeans restrict themselves to using only few Java classes (this helps with packaging and class loading), and model MBeans capsule advanced functionality, such as persistence control, in a ready-to-use format.

A JMX implementation essentially provides the following: a set of code naming conventions borrowed from JavaBeans, new object naming conventions for MBeans, an MBean server that registers and brokers MBeans, services that help to keep the MBean server's content in a consistent state (e.g. trigger alarms when values exceed thresholds), and an open set of protocol converters. Such converters are themselves MBeans, and converters to SNMP, RMI, HTTP, and other choices are available. For instance, an HTTP adapter renders the current states of MBeans into web page markups.

Each JMX name consist of a domain (type) name plus key-value pairs; their combination is unique per MBean, thus allowing the MBean server to hide Java object references within and only advertise object names across APIs. Besides the Sun reference implementation, IBM Tivoli [10] also makes one available, and we were using it during development.

The basic entities in figure 1 translate into the following JMX terminology: agents correspond to MBean servers, and subagents correspond to MBeans. From the JMX perspective, the CIM-to-JMX gateway would appear as a CIM protocol converter. However, this choice of terminology underemphasizes that CIM (unlike JMX) places primary importance on the model at the top (figure 3).

The JMX insulation layer works like the CIM abstraction layer mentioned above; its shape is not determined by the requirement of a uniform data model but by resource instrumentation code that is effectively in place. For the purpose of local comparison we therefore characterize JMX as bottom-up approach towards management instrumentation. It endorses no particular data model a-priori, but percolates available management code into interface descriptions, thereby facilitating singular models on demand. This observation corresponds to the fact that JMX does not define its own management data model, which is part of the reason why a combination with CIM is attractive in the first place.

3 TRANSPARENT CIM-TO-JMX GATEWAY

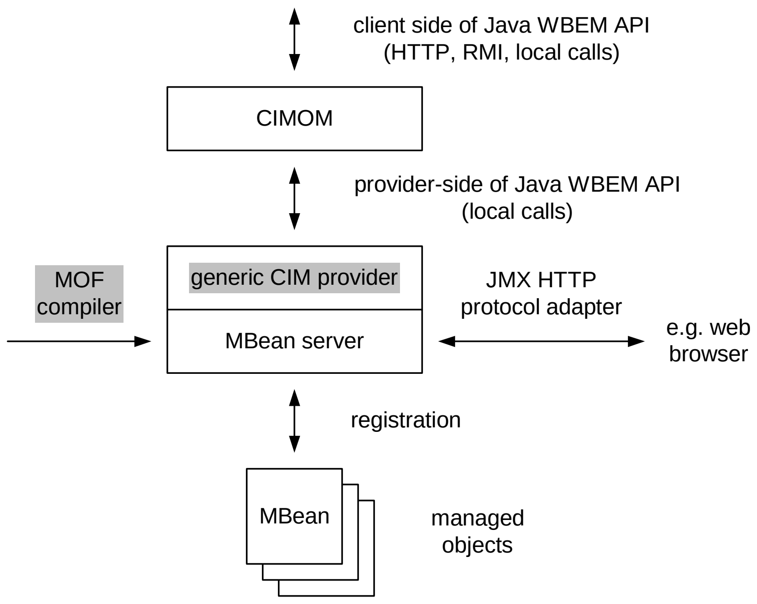

Fig. 4 basic architecture of transparent CIM-to-JMX gateway.

In section 2 we referred to CIM and JMX in terms of top-down vs. bottom-up relationships. It is now straightforward to understand the two parts that make up the transparent CIM-to-JMX gateway in equivalent terms; several symmetries emerge. Figure 4 shows the basic architecture with CIM clients, such as the storage management application, at the top (optional JMX clients at the right) and providers at the bottom.

The MOF compiler (optional compile-time component) constitutes the top-down component (figure 3): it generates Java skeleton source code for MBeans and thereby pushes generic MOF schema definitions “down” into specific Java source code. This allows us to impose a standard data model on JMX. The transparent CIM provider (mandatory run-time component) works as bottom-up component: it pulls information from instantiated MBeans “up” after exploiting a one-to-one relationship between CIM instances and MBeans that is enabled by corresponding sets of names in both realms. The following steps still require manual coding (darker shades in figure 5):

- refine generic CIM provider and register in MOF file (one global step);
- generate MBean skeletons by running MOF compiler and refine into specific versions that implement management logic (one local step per desired MBean);
- implement management application (client) in terms of client-side Java WBEM API.

The rest of the transition occurs transparently inside tools (MOF compiler in section 3.2) or at run-time (generic CIM provider in section 3.1). Figure 5 outlines all steps in sequence; the following two sections add more technical information.

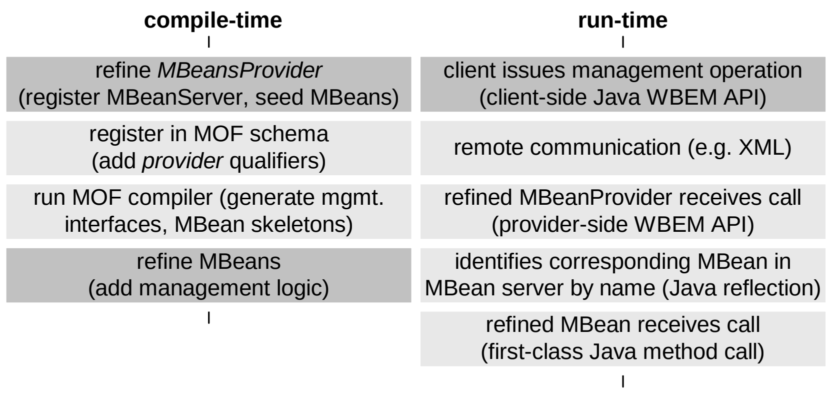
compile-time
refine *MBeansProvider* (register MBeanServer, seed MBeans)
register in MOF schema (add *provider* qualifiers)
run MOF compiler (generate mgmt. interfaces, MBean skeletons)
refine MBeans (add management logic)
run-time
client issues management operation (client-side Java WBEM API)
remote communication (e.g. XML)
refined MBeanProvider receives call (provider-side WBEM API)
identifies corresponding MBean in MBean server by name (Java reflection)
refined MBean receives call (first-class Java method call)

Fig. 5 individual steps taken (darker shades mark coding steps).

3.1 Generic CIM provider

CIM’s MOF syntax includes qualifiers as a generic means to refine the definition of managed objects. With the SNIA CIM implementation, the qualifier *provider* is used with CIM classes (including associations), properties, and methods to name Java classes that are instantiated and called via the provider-side of the Java WBEM API whenever the CIMOM identifies a request to the corresponding entity.

The prototype includes a generic provider *MBeansProvider* that implements the required core functionality. The generic version creates its own JMX MBean server and registers a JMX HTTP protocol adapter as single MBean. This allows management information to be exposed via an alternative path in addition to CIM (figure 4). Any web browser can serve as management console in this case.

The user is in charge of refining the generic version (e.g. through inheritance) to accomplish the following: alternatively link to an existing MBean server, if applicable, and seed the MBean server with MBeans corresponding to initially known managed resources (e.g.

LinuxMBeansProvider). It may also start up threads that observe the environment and add/remove managed objects/MBeans according to circumstances. In practice only a part of the full schema definition needs to be covered, so the effort required during this manual coding step is kept at bay (skeleton methods continue to return *null* as per default).

Both CIM and JMX define object names as consisting of a generic part plus a set of key-value pairs each (section 2.2). CIM uses another MOF qualifier to mark some properties as keys, whereas JMX does not enforce any correlation between key values and values stored inside specific fields of MBeans.

The generic CIM provider operates by requiring adherence to a stricter naming discipline that applies to all MBeans at this point: they must be registered with the MBean server under "CIM-friendly" names, i.e. field values that correspond to CIM key properties must be included in the name; furthermore, JMX requires that key-value pairs are sorted alphabetically, and interference of HTTP and JMX naming rules (e.g. nested JMX names for CIM associations) requires appropriate escape mechanisms. The following examples refer to JMX names for an ordinary CIM instance (key properties also marked in figure 2) and a CIM association each; the association name contains two other names recursively (underscore as escape character and *base64* [11] encoding).

```
CIM_LocalFileSystem:CreationClassName_3DLinuxMBeansProvider_
2CCSName_3Dvergeletto_2CCSCreationClassName_3DLinuxMBeansPro
vider_2CName_3D/

CIM_HostedFileSystem:GroupComponent=CIM_5FUnitaryComputerSys
tem_3ACreationClassName_3DLinuxMBeansProvider_2CName_3Dverge
letto,PartComponent=CIM_5FLocalFileSystem_3ACreationClassNam
e_3DLinuxMBeansProvider_2CCSName_3Dvergeletto_2CCSCreationCl
assName_3DLinuxMBeansProvider_2CName_3D/
```

The user needs not be concerned with the details of this process, because *MBeanProvider* offers a method that assembles appropriate names using Java reflection. However, she must make sure that this naming style can be adopted in an application. If this is not the case (e.g. because existing MBeans must be accommodated), additional mapping information could be included via extra MOF qualifiers. With the naming rules obeyed, the generic CIM provider is able to work in a completely transparent mode.

3.2 MOF compiler

The MOF compiler is the second part of the current CIM-to-JMX gateway implementation. Unlike the CIM provider its use is optional and occurs at compile-time.

The MOF compiler produces source code for two Java classes per CIM class definition and ensures proper inheritance relationships (input e.g. corresponds to what is shown in figure 2). One resulting class is an MBean and the other is its management interface (section 2.2). In addition to mapping between MOF and Java types, the compiler achieves the following: it makes sure that Java constructors include all fields that are marked as CIM key properties, it generates get/set methods according to read/write access rights, and it overrides the equals method to match CIM key properties. Furthermore, if a CIM class definition includes any method prototype, then the corresponding Java class is marked as abstract.

The generated MBeans serve as skeletons that need to be refined, because in general it is impossible to predict/generate the required management logic. The skeleton defines the required data structures and access functions. The user is responsible for adding code that fills in corresponding values and reacts to their changes accordingly. This is typically achieved by inheritance: each MBean skeleton of interest is refined at a further level of inheritance (without changing the unqualified class name), where it selectively overrides a combination of constructors or get/set methods in order to add specific management logic. This is based on similar conventions for distributed programming using communication skeletons [12].

Overall, the decision how clients choose to extend the functionality of generated MBean skeletons, and whether generated versions are used in the first place can be taken independently of using the transparent CIM-to-JMX gateway (and its mandatory CIM provider part) as such. The CIM-to-JMX gateway works with all MBeans that obey its naming rules. We expect to be able to exploit more sophisticated design patterns as experience with a range of realistic situations grows.

4 CONCLUSIONS

We have reported on a transparent gateway that translates management information between CIM and JMX formats and APIs. Our analysis has shown that the two management technologies have some complementary strengths (summarized in table 1) whose practical combination we have sought.

CIM/WBEM	JMX
uniform data model (CIM schema)	Java mapping for instances (MBeans)
transport formats (XML and others)	provider-side API for introspection (MBean server)
client-side and provider-side API (Java WBEM API)	

Table 1 comparative strengths of CIM/WBEM and JMX (summary).

CIM provides a standard data model with appropiate expressive power for storage management tasks, and JMX enables a straightforward Java language binding for ease-of-use. Using this combination, we arrive at a standard-compliant and flexible management solution that allows us e.g. to do quick prototyping of CIM providers, because state changes that occur to normal first-class Java objects (registered as JMX MBeans; section 2) will be exposed to management applications by the transparent CIM-to-JMX gateway. Adding CIM management instrumentation may therefore require as little effort as coding a single Java assignment. Since existing CIM APIs, such as the Java WBEM API [8, 9], are complicated (requiring a steep learning curve) and error-prone (with lots of dynamic type casts, etc.) we benefit a lot in terms of saved effort and software engineering metrics, and are therefore able to lower the barrier for equipment and application vendors to ship their products with a CIM/JMX instrumentation in cases where Java-based agents are permitted.

So far, we have relied on JMX to the extent that it enables desired CIM functionality. In particular, we utilized its code and MBean naming conventions to avoid creating yet another such facility. If a more complete mapping of JMX features were desired (bottom-up focus), JMX relations (from its relationship service) and CIM associations, as well as JMX notifications and CIM events could serve as mapping candidates. (The second pair was not yet considered because of lack of support in the current SNIA software version [8].)

We have streamlined the provider side API by introducing JMX MBeans as appropriate Java representations. Our approach depends on Java insofar as its byte code offers the necessary redundancy for reflection (code inspection) at run-time. In principle, other interpreted languages or component middleware could serve as well. We are also investigating use of WSDL [13] to introduce first-class object semantics on client sides in overall similar terms.

The transparent CIM-to-JMX gateway currently exploits a direct mapping between CIM instances and JMX MBeans. Alternatively, certain CIM associations could alter between explicit and implicit forms inside the gateway. For instance, it may suffice to represent simple one-to-one associations as pointers between the two instances concerned (without another instance representing the association as such). This could reduce CIM overhead and facilitate data consistency (matching lifetimes of instances and their associations).

The implementation is now in use with a research prototype for policy-based storage management, and runs on a mix of Linux and Windows platforms. We plan to try some of the extensions as this work progresses. Meanwhile, our experience indicates that the design and code base are feasible and portable, and that the combined solution is appropriate for a range of management tasks.

REFERENCES

[1] HTRC Group, *The Emerging Internet Storage Infrastructure Market*, White Paper, 2000.

[2] J. Case et al., A Simple Network Management Protocol (SNMP), Internet Engineering Task Force (IETF), RFC 1157, May 1990.

[3] Distributed Management Task Force, *Common Information Model (CIM) Specification*, June 1999.

[4] Sun Microsystems. *Java Management Extensions - Instrumentation and Agent Specification*, v1.0, July 2000.

[5] H. Kreger, Java Management Extensions for application management, *IBM Systems Journal*, 40(1), 2001.

[6] Distributed Management Task Force, *Specification for CIM Operations over HTTP*, Version 1.0, August 1999.

[7] J. Strasser, *Directory Enabled Networks*, Macmillan Technical Publ., 1999.

[8] openCIMOM software distribution, version 0.61.
http://www.snia.org/English/Products/Products_FS.html

[9] Sun Microsystems, *WBEM Services Specification*, Java Specfication Request, JSR 48, March 2000.

[10]TMX4J software distribution, version 1.3.
http://www.alphaworks.ibm.com/

[11]N. Freed, N. Borenstein, *Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies*, Internet Engineering Task Force (IETF), RFC 2045, November 1996.

[12]M. Schaaf, F. Maurer, Integrating Java and CORBA: A Programmer's Perspective, *IEEE Internet Computing*, January/February 2001.

[13]E. Christensen et al., *Web Services Description Language (WSDL) 1.1*, World Wide Web Consortium, W3C Note, March 2001.