Towards Low-Latency Model-Oriented Distributed Systems Management*

Iván Díaz, Juan Touriño, and Ramón Doallo

Computer Architecture Group
Department of Electronics and Systems,
University of A Coruña, Spain
{idiaz,juan,doallo}@udc.es

Abstract. Windows and Unix systems have been traditionally very different with regard to configuration storage and management. In this paper we have adapted our CIM-based model-driven management framework, AdCIM, to extract, integrate and modify management and configuration information from both types of OS in a multiplatform and seamless way. We have achieved very low latencies and client footprints without sacrificing the model-driven approach. To enable this functionality for a wide range of system administration applications, we have implemented both an efficient CIM XML dialect and a distributed object infrastructure, and we have assessed its performance using two different approaches: CORBA and Web Services.

Keywords: Management, Distributed Systems, WMI, CORBA, Web Services, CIM.

1 Introduction

System administrators have to take into account the great diversity of hardware and software existing nowadays in organizations. Homogeneity can be achievable in some instances, but also has risks; i.e., a monoculture is more vulnerable against viruses and trojans. The combination of Windows and Unix-like machines is usual, whether mixed or in either side of the client/server divide. System administration tasks in both systems are different due to a great variety of interfaces, configuration storage, commands and abstractions.

To close this gap, there have been many attempts to emulate or port the time-proven Unix toolset. Windows Services for Unix [1] are Microsoft's solution, enabling the use of NIS and NFS, Perl, and the Korn shell in Windows, but it is not really integrated with Windows as it is a migration-oriented toolset. Cygwin [2] supports more tools, such as Bash and the GNU Autotools, but it is designed to port POSIX compliant code to Windows. Outwit [3] is a very clever

* This work was funded by the Ministry of Education and Science of Spain under Project TIN2004-07797-C02 and by the Galician Government (Xunta de Galicia) under Project PGIDIT06PXIB105228PR.

S. Ata and C.S. Hong (Eds.): APNOMS 2007, LNCS 4773, pp. 41–50, 2007.
© Springer-Verlag Berlin Heidelberg 2007

port of the Unix tools that integrates Unix pipelines in Windows and allows accessing the registry, ODBC drivers, and the clipboard from a Unix shell, but its scripts are not directly usable in Unix.

The subject of performance and low-latency issues in system administration is discussed in several works. Pras et al [4] find Web Services more efficient than SNMP for bulk administration data retrieval, but not for single object retrieval (i.e. monitoring), and conclude that data interfaces are more important for performance than encoding (BER vs XML). Nikolaidis et al [5] show great benefits by compressing messages in a Web Service-based protocol for residential equipment management, but only use Lempel-Ziv compression. Yoo et al [6] also implement compression and other mechanisms to optimize the NETCONF configuration protocol which uses SOAP (Web Services) messaging.

Also, there are several works that use XML to represent machine configurations, like the ones by Strauss and Klie [7], and Yoon et al. [8], both mapping SNMP to XML. The drawback is that SNMP has a very flat structure that does not represent aspects like associations as flexibly as CIM [9].

The framework used in this paper, AdCIM [10] (`http://adcim.des.udc.es`), can extract configuration and management data from both Windows or Unix machines, represent and integrate these data in a custom space-efficient CIM XML dialect, and manipulate them using standard XML tools such as XSLT. One objective of this framework is to support monitoring applications with low-latency gathering of structured data and small footprint. To achieve multiplatform interoperability and low-latency network messaging, two different distributed object technologies will be used: CORBA [11] and Web Services [12].

The paper is organized following the structure depicted in Fig. 1. Thus, Section 2 presents the XML Schema transformation and the advantages of the miniCIM XML format. Section 3 details the different processes used in Windows and Unix to extract miniCIM configuration data. Section 4 discusses the use and implementation details of both CORBA and Web Services middleware. Section 5 presents experimental results to compare the performance of both approaches. Finally, conclusions and future work are discussed in Section 6.

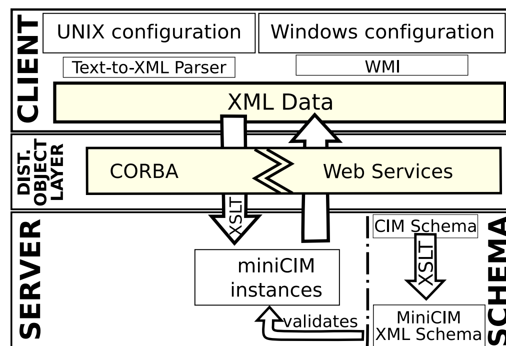

Fig. 1. Overview of the integrated management framework

2 Schema Transformation

This section details how the CIM schema is translated into an XML Schema, and then derived into an abbreviated XML syntax of CIM (which we denoted as miniCIM) that keeps all semantic constraints and helps to reduce latency and transfer times in our framework.

```
<CIM_InetdService namespace="dc=udc">
  <SystemCreationClassName>CIM_ComputerSystem</SystemCreationClassName>
  <SystemName>shalmaneser</SystemName>
  <CreationClassName>CIM_InetdService</CreationClassName>
  <Name>ftp</Name>
  <SocketType>stream</SocketType>
  <Protocol>tcp</Protocol>
  <Wait>nowait</Wait>
  <User>root</User>
  <Command>root/usr/sbin/tcpd</Command>
</CIM_InetdService>
```

Fig. 2. Example of miniCIM inetd service instance

cimXML [13] is the official DMTF representation of CIM in XML, which represents both the CIM schema and CIM instances. Since these two aspects are not separated it is very cumbersome to represent instances. Schema information can be merged from an external file, but there is still overhead; e.g, key properties must be present both in the name of the instance and as properties. Thus, a declaration of services of a machine in cimXML is 305Kb long, which is reduced to 3Kb in our approach by removing schema information from instances. This greatly reduces message size, which will be shown later as a main factor in the speed of the framework.

In order to simplify matters, we decided to translate the schema information to XML Schema [14], which supports type inheritance, abstract classes and keys, with XSLT. The resulting XML Schema defines a much terser instance syntax, miniCIM. Such an instance for inetd services can be seen in Fig. 2. Its format is property-value pair based, but semantic information is not lost, only moved to the XML Schema. Invalid instances, or dangling references, are reported by the XML validator.

An XSLT stylesheet processes the cimXML schema and its CIM classes depending on their abstractness status, superclass, and association type if applicable. These will be mapped to attributes of a new type in the final schema. For example, association-related properties are represented as application-defined attributes, and others like abstractness and superclass, supported using XML Schema inheritance constructs.

Association references are also mapped as properties. Constraints are mapped to XML Schema constructs, such as cardinality, addressed with `minOccurs` and `maxOccurs`. The reference properties `Antecedent` and `Dependent` in each association can have their names changed by child classes. To account for that, the `Override` qualifier is also supported.

3 Configuration Data Extraction

This section describes the methods used both in Unix and Windows systems to extract configuration data from system files. Configuration information is collected from two sources: flat text sources, such as files and internal commands, and the WMI (Windows Management Instrumentation) subsystem present in all post-2000 Windows systems.

Usually, Unix-based OS codify almost all configuration data in text files and directory structures that are seldom available in XML format, so our framework parses and transforms them to XML to facilitate further processing. To do so, data are serialized to plain text and processed with grammar rules.

```
  import Martel; from xml.sax import saxutils
  def Item(name): return Martel.Group(name,Martel.Re("\S+\s+"))
  fields =Item("name")+Item("socktype")+Item("proto")+Item("flags")+Item("user")+Martel.ToEol("args")
4 offline =Martel.Re("#<off>#\s*")+Martel.Group("off",fields)
  commentary=Martel.Re("#")+Martel.Group("com",Martel.ToEol())
  serviceline =Martel.Group("service",fields)
  blank=Martel.Str("\n")
  format=Martel.Group("inetd",Martel.Rep(Martel.Alt(offline, blank, commentary, serviceline)))
9 parser = format.make_parser()
  parser.setContentHandler(saxutils.XMLGenerator())
  parser.parseFile(open("inetd.conf"))
```

Fig. 3. Martel program used for parsing `inetd.conf` to XML

```
#echo   stream  tcp nowait  root     internal
ftp     stream  tcp nowait  root     /usr/sbin/tcpd  /usr/sbin/proftpd
#<off>sgi_fam/1-2 stream rpc/tcp wait     root     /usr/sbin/famd fam
```

(a) Sample lines with original `inetd.conf` format

```
<doc>
  <commentary> <com>#</com>echo stream tcp nowait root internal </commentary>
  <line> <id>ftp</id> <ws> </ws> <id>stream</id> <ws> </ws> <id>tcp</id> <ws> </ws>
        <id>nowait</id> <ws> </ws> <id>root</id> <ws> </ws> <id>/usr/sbin/tcpd</id>
  <ws> </ws> <id>/usr/sbin/proftpd</id> </line>
  ........
  <off> <com>#</com>&lt;off&gt;#<ws> </ws>
    <line> <id>sgi_fam/1-2</id> <ws> </ws> <id>stream</id> <ws> </ws> <id>rpc/tcp</id>
    <ws> </ws> <id>wait</id> <ws> </ws> <id>root</id> <ws> </ws>
    <id>/usr/sbin/famd</id> <ws> </ws> <id>fam</id> </line>
  </off>
  </doc>
```

(b) Result of parsing `inetd.conf` to XML

Fig. 4. Input and output of parsing `inetd.conf` to XML

Grammar rules are described using Martel [15], a Python module to parse text files into SAX events, then directly transcribable as XML data. Fig. 3 shows an example of a Martel program that produces a structured XML file from the inetd services configuration file `/etc/inetd.conf` (shown in Fig. 4(a)). In this code, Martel operators `Re` and `Alt` represent the "`*`" and "`|`" regular expression operators, respectively, and operator `Group` aggregates its second argument into a single XML element. Finally, `ToEol` matches any text before the next end of line.

This Martel code defines the `inetd.conf` file as composed of three types of lines: off lines (line 4), commentaries (line 5), and normal lines (line 6). Every normal line maps to an enabled service, and off lines to temporarily disabled services. The program also has to discriminate between commentaries and the #<off># sequence that begins an off line. Each line is partitioned as a list of items

```
import Martel; from xml.sax import saxutils
def Group(x,y): return Martel.Group(x,y)
def Re(x): return Martel.Re(x)
def Item(name): return Martel.Group(name,Martel.Re("\S+"))
def Date(name): return Martel.Group(name,Martel.Re("\S+\s+\d+\s+[0-9:]*"))
def Space(): return Martel.Re("\s*")
def Colon(): return Martel.Re(":\s*")
def Origin(): return G("origin",Re("\w+"))+Col()
def OriginPid(): return (Group("origin", Group("name", Re("[\w()_-]+")) +Re("\[")+
    Group("pid", Re("[0-9]+")) +Re("\]")+ Colon()))
fields =(Date("date") +Space()+ Item("host") +Space()+ Martel.Alt(OriginPid(),Origin(),Space()) +
    Martel.UntilEol("message") + Martel.ToEol())
format=Group("file",Martel.Rep(fields))
parser = format.make_parser()
parser.setContentHandler(saxutils.XMLGenerator())
parser.parseFile(open("m"))
```

Fig. 5. Martel program used for parsing `/var/log/messages` to XML

```
   import sys, win32com.client, pythoncom, time; from cStringIO import StringIO
   locator = win32com.client.Dispatch("WbemScripting.SWbemLocator")
   wmiService = locator.ConnectServer(".","root\cimv2")
 4 refresher = win32com.client.Dispatch("WbemScripting.SWbemRefresher")
   services = refresher.AddEnum(wmiService, "Win32_Service").objectSet
   refresher.refresh()
   pythoncom.CoInitialize()
   string = StringIO()
 9 for i in services:
     (string.write((
     "<SystemCreationClassName>"+unicode(i.SystemCreationClassName)+"</SystemCreationClassName>"+
     "<CreationClassName>"+unicode(i.CreationClassName)+"</CreationClassName>"+
     "<Name>"+unicode(i.Name)+"</Name>"+"<State>"+unicode(i.State)+"</State>"+
14   "<StartMode>"+unicode(i.StartMode)+"</StartMode>"+"</CIM_Service>").encode("utf8")))
   print string.getvalue()
```

Fig. 6. Python script to extract service information from WMI

that are mapped to fixed properties in the resulting instances. The abridged output in Fig. 4(b) is still a direct representation of the original data in Fig. 4(a), now structured.

Grammar rules can document the configuration format formally, following the original file format very closely, or simply describe the high-level format of the document (e.g. line-oriented with space separators). The latter approach makes it easier to process many formats without doing much work specifying rules, but the former has the benefit of early-on error checking and validation of configuration formats. Since Martel supports backtracking, multiple versions of the same file can be identified using different trees aggregated by an alternation (or) operator at their top.

The output of Fig. 4(b) is processed by an XSLT stylesheet, which can be executed server-side or in the client, but the former is preferred because servers usually have more processing power, and also to reduce footprint and latency in the client. Nevertheless, there are very efficient C XSLT processors [16] that can be used in some client nodes to reduce load on the server.

Figure 5 shows a more complex example of grammar rules that parses the `/var/log/messages` log file, composed of messages, warnings and errors from various system processes and the kernel. The format is line based, each line consisting of date, host name, optional originator and pid (process id), and a free-form message.

Windows discarded files in favor of the registry as system configuration repository as of the Windows 95 release. Thus, to extract configuration data it would seem necessary to manipulate registry data. Instead, we have used the Windows

WMI subsystem, which provides comprehensive data of hardware devices and software abstractions in CIM format, exposed using COM (Component Object Model), the native Windows component framework. WMI is built-in since Windows 2000, but it is also available for previous versions. Queries can also be remote using DCOM (Distributed COM). Its coverage varies with Windows version, but it can be extended by users.

WMI data can be uniformly retrieved using simple code, such as the one shown in Fig. 6, which uses the COM API and directly writes XML data of miniCIM instances. The code uses a locator to create a WMI COM interface named `SWbemRefresher` (see line 4) which makes possible to update WMI instance data without creating additional objects. In the next lines, instances contained in the refresher interface are queried and their data written as miniCIM instances. A `StringIO` Python object (line 8) is used to avoid string object creation overheads.

4 Distributed Object Technologies

This section describes the use of both CORBA and Web Services distributed object middleware in our framework. As can be seen in Fig. 1 the purpose of this middleware is to transfer efficiently miniCIM instances or raw XML data between clients and servers.

CORBA achieves interoperability between different platforms and languages by using abstract interface definitions written in IDL, from which glue code for both clients and servers is generated. This interface is a “contract” to be strictly honored. This enforces strict type checking, but clients become “brittle”: any change or addition in the interface breaks them and needs their recompilation and/or readaptation.

Using an XML Schema validated dialect has two benefits: first, it promotes flexibility, since changes in format can be safely ignored by older clients and, second, preserves strict validation. Both aspects are important due to the extensibility of the CIM model, but in very time-critical instances a direct mapping of a CIM class to IDL is still possible.

To pass XML data via CORBA they are flattened to a string. This solution is not optimal, since time is lost in serialization and de-serialization. A more efficient solution would be to pass the data as a CORBA DOM Valuetype [17], which is passed by value with local methods. Then, the parsed XML structure would not be flattened, so clients would manipulate the XML data without remote invocations. Unfortunately, this is a feature not yet well supported in most production-grade ORBs. Our implementation of choice is omniORB [18], a high-speed CORBA 2.1 compliant ORB with bindings for both C++ and Python.

In contrast with CORBA, Web Services (WS) solutions provide an interoperation layer than can be both tightly coupled (using XML-RPC messaging) or loosely coupled (XML document-centric). Gradually, WS are being more oriented to support web-based service queries than to offer distributed object middleware, but there is a significant overlapping between the two approaches.

WS use XML dialects for both interface definition (WSDL) and transport (SOAP). It may seem that using XML dialects would promote synergy, but the use of XML as "envelope" of the message and representation of it is orthogonal at best. It is worse, in practice, since the message must be either sent as an attachment (which implies Base64 transformation), or with its XML special characters encoded as character entities to avoid being parsed along with the XML elements of the envelope. Additionally, two XML parsings (and the corresponding encoding) must be done, being added to message-passing latency. In the foreseeable future, there is not any support in view for platform independent parsed XML representation in WS.

As implementation we have chosen the Zolera SOAP Infrastructure [19], the most active and advanced WS library for Python.

5 Experimental Results

We have proceeded to evaluate the performance of our framework for various representative tasks and the impact of the integration technology (Web Services vs CORBA). The tests have been performed using Athlon64 3200+ machines connected by Gigabit Ethernet cards.

We have tested three different cases of use of our framework, each both in Windows and Unix (Linux). The parameters measured for these cases have been total time, latency and message size, with and without compression. Total time is defined as the round-trip time elapsed between a request is sent and the response is completely received. Latency is the round-trip time when the response is a 0-byte message. Two different algorithms have been used for compression: zlib and bzip2. The three cases tested are:

- CPU load retrieval, shown in Fig. 7. This case is representative of monitoring applications, usually invoked several times per second, which need fast response times and low load on the client.
- Service information discovery, shown in Fig. 8. This case represents discovery applications, invoked with a frequency ranging from minutes to hours.
- Log file information retrieval and parsing (data analysis), shown in Fig. 9. This case represents bulk data requests invoked manually or as part of higher-level diagnostic processes. These requests have unspecified total time and data size, so they are invoked ad-hoc, with little or no regularity.

The code examples of Section 3 have been used for the second and third cases. The code of the first case was omitted for brevity.

The first case in Fig. 7 shows lower latency and total time for CORBA vs WS in both platforms. Base latency for CORBA is roughly 0.2 ms, whereas it is 20-30 ms using WS, in great part due to the overhead of parsing the envelope and codifying the message. Compression benefits WS but slows down CORBA performance. Figure 7(b) shows the cause: WS messages are large enough to be slightly benefitted from compression, but CORBA messages (less than 50 bytes long) are actually doubled in size. In the second case (Fig. 8), web service times

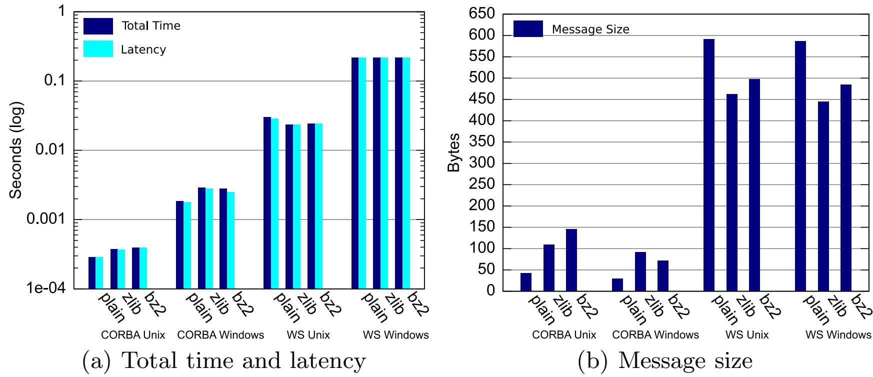

Fig. 7. Performance measurements for test one: CPU Load

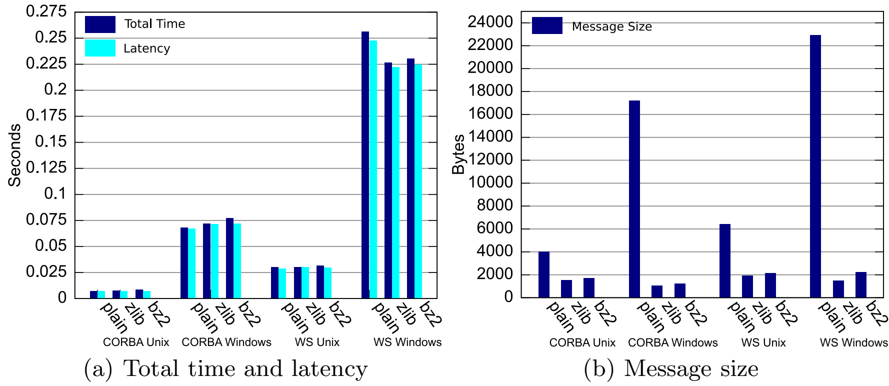

Fig. 8. Performance measurements for test two: Service Discovery

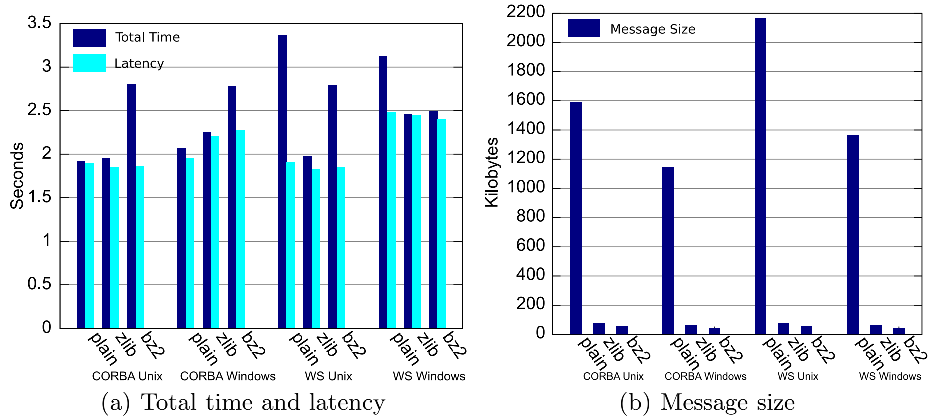

Fig. 9. Performance measurements for test three: Log Parsing

are very similar to those obtained in the previous case, since parsing overhead dominates total time. CORBA times are longer than in the first case, but shorter than those of WS.

In these two test cases, Windows times are higher than those of Unix, because of the overhead of operating with COM objects. Nevertheless, these overheads are smoothed over in the third test case. From figures 7 and 8, it is clear that message size determines total time, affecting WS much more, both due to their XML envelope and codification. The envelope size (almost fixed) clearly affects only the first case, but codification introduces a 20% message size overhead on average.

The third case (Fig. 9) shows a much narrower spread of values due to message size (1Mb+); thus, total time is dominated by transfer time, instead of by protocol overheads. In this test compression in WS achieves times comparable to those of uncompressed CORBA. This would be more noticeable with less bandwidth, as WS compression ratios of 40:1 are reported in Fig. 9(b).

In general, all times are very acceptable, although CORBA has a clear advantage. The benefits of compression are dubious, except in the third test for WS. bzip2 is slower and is oriented to larger data sets than zlib, which is a better choice for the tested cases. Although both WS and CORBA are acceptable solutions for information exchange in our framework, monitoring and low-latency applications strongly favor CORBA over WS due to its message compactness and better processing time.

6 Conclusions

We have explored the adaptation of our AdCIM framework to various system administration applications, focusing on the following relevant features:

- Definition of an XML Schema and a new XML mapping of CIM, named miniCIM, that simplifies the representation and validation of CIM data and allows the use of standard XML Schema tools to manage CIM seamlessly.
- Extraction of monitoring, service and log data from Windows and Unix into miniCIM instances. This is achieved using different techniques (text-to-XML parsing grammars, WMI scripts) due to the different management approaches supported by each OS.

We have also discussed methods and alternatives to implement multiplatform and low-latency transport methods using two different approaches: CORBA and Web Services technologies. Lastly, we have validated and tested the implementations by defining a testing framework for measuring total time, latency and message size.

The chosen domain of processor statistics, network services and log data analysis has illustrated the use of these methods for administration domains particularly dissimilar between operating systems. But the scope of our approach is not limited to such domains, as the model and implementation technologies are truly general and extensible.

As future work, we plan to implement CORBA interfaces based on Valuetypes, design real-time support agents that diagnose and aggregate global network issues, and extend WMI coverage (i.e. for text-based Windows configurations).

References

1. Microsoft Windows Services for Unix [Visited July 2007], http://www.microsoft.com/windows/sfu/
2. Noer, G.J.: Cygwin32 - A Free Win32 Porting Layer for UNIX Applications. In: 2nd USENIX Windows NT Symposium, pp. 31–38 (1998)
3. Spinellis, D.: Outwit - Unix Tool-Based Programming Meets the Windows World. USENIX 2000. In: Technical Conference pp. 149–158 (2000)
4. Pras, A., Drevers, T., van de Meent, R., Quartel, D.: Comparing the Performance of SNMP and Web Services-Based Management. IEEE Electronic Transactions on Network and Service Management 1(2), 1–11 (2004)
5. Nikolaidis, A.E., Doumenis, G.A., Stassinopoulos, G.I., Drakos, M., Anastasopoulos, M.P.: Management Traffic in Emerging Remote Configuration Mechanisms for Residential Gateways and Home Devices. IEEE Communications Magazine 43(5), 154–162 (2005)
6. Yoo, S.M., Ju, H.T., Hong, J.W.: Performance Improvement Methods for NETCONF-Based Configuration Management. In: Kim, Y.-T., Takano, M. (eds.) APNOMS 2006. LNCS, vol. 4238, pp. 242–252. Springer, Heidelberg (2006)
7. Strauss, F., Klie, T.: Towards XML Oriented Internet Management. In: 8th IFIP/IEEE Int'l Symposium on Integrated Network Management, IM 2003, pp. 505–518 (2003)
8. Yoon, J.H., Ju, H.T., Hong, J.W.: Development of SNMP-XML Translator and Gateway for XML-based Integrated Network Management. International Journal of Network Management 13(4), 259–276 (2003)
9. DMTF. Common Information Model (CIM) Standards [Visited July 2007], http://www.dmtf.org/standards/cim
10. Diaz, I., Touriño, J., Salceda, J., Doallo, R.: A Framework Focus on Configuration Modeling and Integration with Transparent Persistence. In: 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005). Workshop on System Management Tools for Large-Scale Parallel Systems, p. 297a (2005)
11. CORBA/IIOP Specifications [Visited July 2007], http://www.omg.org/technology/documents/corba_spec_catalog.htm
12. Web Services Architecture [Visited July 2007], http://www.w3.org/TR/ws-arch/
13. DMTF. Specification for the Representation of CIM in XML [Visited July 2007], http://www.dmtf.org/standards/documents/WBEM/DSP201.html
14. Lee, D., Chu, W.W.: Comparative Analysis of Six XML Schema Languages. ACM SIGMOD Record 29(3), 76–87 (2000)
15. Dalke, A.: Martel [Visited July 2007], http://www.dalkescientific.com/Martel/
16. XSLTC [Visited July 2007] http://xml.apache.org/-xalan-j/-xsltc/-index.html
17. Object Management Group. XMLDOM - DOM/Value Mapping Specification [Visited July 2007], http://www.omg.org/cgi-bin/doc?ptc/2001-04-04
18. Grisby, D.: omniORB - Free High Performance ORB [Visited July 2007], http://omniorb.sourceforge.net/
19. Salz, R.: ZSI - The Zolera SOAP Infrastructure Developer's Guide [Visited July 2007], http://pywebsvcs.sourceforge.net/zsi.html