An XML-based Framework for Dynamic SNMP MIB Extension

Ajita John, Keith Vanderveen, and Binay Sugla

Bell Labs Innovations, Lucent Technologies,
101 Crawfords Corner Rd
Holmdel, NJ 07733, USA,
{ajita,vandervn,sugla}@research.bell-labs.com

Abstract. Current SNMP-based management frameworks make it difficult for a network administrator to dynamically choose the variables comprising the MIB at managed elements. This is because most SNMP implementations represent the MIB implicitly as part of the agent code - an approach which impedes the runtime transfer of the MIB as a separate entity to be easily shipped around the network and incorporated into different applications. We propose the use of XML to represent MIBs at managed elements. We describe a network management toolkit which uses XML and the Document Object Model (DOM) to specify a MIB at runtime. This approach allows the MIB structure to be serialized and shipped over the network between managers and agents. This use of the DOM for MIB specification facilitates dynamic MIB modification. The use of XML also allows the MIB to be easily browsed and seamlessly integrated with online documentation for management tasks. XML further allows for the easy interchange of data between network management applications using different information models.

1 Introduction

Network management tools and applications which use the Simple Network Management Protocol (SNMP) tend to have low scalability and are inflexible and difficult to use. This is because, for most SNMP agents, the Management Information Base (MIB) is not a self-contained module but is instead dispersed throughout the agent code [16, 19]. This makes it difficult to modify or replace the MIB without replacing the agent, which causes the MIB to be out of date with respect to the needs of the network administrator.

The network administrator is often faced with a situation where some required information is not available in the MIB. This is illustrated in the following scenario: Hearing of the epidemic spread of an e-mail virus through the Internet, the administrator of an enterprise network wants to add some additional monitoring capability to his company's e-mail servers. Knowing that the virus replicates by e-mailing multiple copies of itself to addresses in the address book of a victim, the administrator decides to monitor the ratio of the number of e-mails received by users in his company to the number of e-mail senders. The

R. Stadler and B. Stiller (Eds.): DSOM'99, LNCS 1700, pp. 107-120, 1999.

decision to choose this ratio is based on the presumption that address books of people within an enterprise contain many addresses within the enterprise and a small number of infected users would send out email messages to a large number of people within the enterprise. If this ratio becomes higher than usual, a virus infection may be occurring. The number of stored e-mails is available as mtaStoredMessages from the Mail Monitoring MIB [12], but the number of e-mail senders is not in this MIB. This leads to the situation depicted in Figure 1(a) where the administrator searches for a variable $\#ESenders$ (number of e-mail senders) in a MIB but finds that $\#ESenders$ is not in the MIB. Current solutions to this problem either involve shutting the agent down and recompiling new instrumentation into the agent code [1] or spawning new sub-agents under a master agent [15]. Recompilation is inelegant and may not be feasible in all environments. The sub-agent approach is not scalable because each subagent runs as a separate process, and because of the overhead of sub-agent-to-master communication.

Another problem caused by the tight coupling of agent code and MIB is that there is no convenient way to transfer the MIB from an agent to another application. Determining an SNMP agent's MIB requires a MIB walk, which can take a number of network messages proportional to the size of the MIB [16]. Even after doing a MIB walk, variables from a part of the agent which are not working (because a sub-agent or the instrumentation for that part is not working) will not be visible [16] to a management application. We believe that transparent, efficient transfer of MIBs will become a necessity as we move towards larger networks, more complex and diverse network management applications, and multiple (possibly hierarchical) managers.

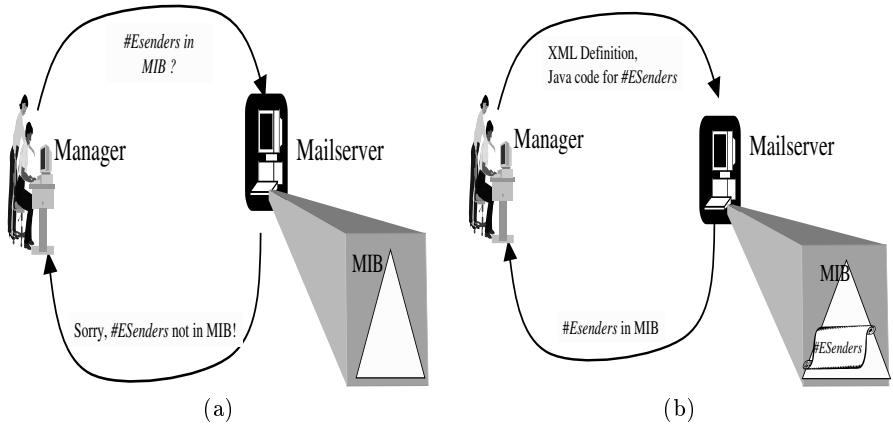

Fig. 1. Administrator (a) unable to get value of variable $\#ESenders$ (b) downloading definition and access code for $\#ESenders$

We introduce XNAMI, an XML-based architecture for SNMP management with a *dynamically reconfigurable* or *eXtensible* MIB, to overcome the above-mentioned limitations of MIBs. In practice, there may be several MIBs per agent, e.g. MIB-II, the RMON MIB, the ATM MIB, etc. For simplicity of discussion, we group all these real MIBs into a single logical MIB that encompasses all the others. An XNAMI agent's MIB is represented as a document in XML [18], which allows the MIB structure to be serialized and shipped over the network between managers, agents, and other applications. The use of XML also allows the MIB representation to be easily browsed and seamlessly integrated with online documentation for management tasks. XML further allows for the easy interchange of data between network management applications using different information models, because of the widespread availability of tools to convert from one XML Document Type Definition (DTD) to another. Finally, XML's Document Object Model (DOM) [20] provides a convenient API which allows variables to be added to or deleted from the XNAMI MIB at runtime. In the preceding e-mail example, the administrator can write Java code which determines the value of *#ESenders* on most systems by scanning the files which hold mail for each user (on UNIX systems, this is usually in /var/spool/mail) and summing the results. If the mail servers are running XNAMI agents, the administrator can add the variable *#ESenders* to the MIB along with the associated Java access code (Figure 1(b)). The administrator can also add a *meta-variable* [7] which represents the ratio of the number of e-mails received by users in his company to the number of e-mail senders. The administrator can then monitor the ratio visually by displaying a graph of its value on a monitor, or set an SNMP trap to be triggered if the ratio rises above a certain threshold, or both.

To be able to add new instrumentation to an XNAMI MIB, a network administrator has to have both the skill to program the method code and access to the particular subsystem of the network element being instrumented. At present, most vendor-supplied network elements are closed boxes. However, by opening up at least a part of the internals of their products, network equipment vendors can help network providers achieve greater service differentiation. Therefore, we believe that network equipment will follow the trend toward open systems now being witnessed in the software industry.

The rest of this paper is organized as follows: Section 2 summarizes related work, Section 3 gives an introduction to XML and partial XML documents for some MIBs, , Section 4 describes the XNAMI agent and manager, and conclusions and directions for future work are given in Section 5.

2 Related Work

In a typical SNMP agent implementation, the MIB is hardwired into the agent code [16, 19]. That is to say, pointers to method routines for each MIB variable are stored in a data structure (usually a table of some sort), indexed by the OBJECT IDENTIFIER prefix that forms the name of the OBJECT-TYPE of the instance whose value is to be retrieved or changed. This data structure is

compiled into the agent code, which necessitates a recompilation each time the MIB is changed. The only way provided by SNMP to see what variables are present in the MIB is to do a 'MIB-walk' by repeatedly invoking the get-next operation or the get-bulk operation on the agent, which requires many network messages and can be a time-consuming process [16].

Kalyanasundaram et al. describe an SNMP agent in which a part of the MIB is decoupled from the agent code [11]. Their agent contains a new kind of MIB module built around the notion of a spreadsheet, with data organized in two-dimensional tables of cells. Each cell contains an executable script which specifies how to compute the value of that cell from the values of other cells, or from data retrieved from other MIB modules or SNMP agents. A manager of the spreadsheet agent can create new cells by sending SNMP SET requests containing the scripts for computing the values in the new cells. A key difference between this spreadsheet agent and the XNAMI agent is that the spreadsheet cell values are derived, while in XNAMI new MIB variables may be defined which access system data directly. A related difference is that XNAMI provides control over the structure of the entire MIB residing at the agent, while the spreadsheet agent allows modifications only to the spreadsheet part of the MIB. The authors of [21] propose building a MIB on a remote agent using a scripting language, which is similar to the idea behind XNAMI. However, no implementation of this idea is described.

Wellens and Auerbach proposed using HTTP and HTML to access an agent's MIB [7]. The rationale behind their proposal was to replace UDP with HTTP as the transport protocol, thereby making retrieval of large amounts of data more efficient. They also proposed replacing the SNMP get, get-next, and get-bulk operators (and their set analogues) with get- and set-subtree operators that retrieve entire subtrees of an agent's MIB at once. Marvel [2] and the Push vs. Pull approach [14] are continuations of the Wellens and Auerbach work. Tsai and Chan implemented an SNMP agent which embodies these ideas but is 'bilingual' in that it can be accessed either through HTTP and HTML pages or SNMP [4]. Because HTML provides no way to define new elements, however, values of MIB variables have to be encoded in HTML documents either as binary data or using non-standard markup tags. Because XML allows the definition of new elements, XNAMI's representation of MIB variables uses just UTF-8 (which includes ASCII) characters and conforms to the XML standard. XNAMI also differs from both the Wellens et al. and Tsai et al. agents in that new variables can be added to the MIB at runtime by sending an XML description of the new variables, as well as code for the set and get methods. XNAMI's representation can also be extended to include new types of MIB variables and remain in conformance to the XML standard. Although XNAMI currently uses UDP as the transport protocol for communication between manager and agent, it could be modified easily to use HTTP.

HTML and HTTP have been used for network management applications which do not have SNMP MIBs at all [3, 17]. Among the advantages cited are reduced cost compared to SNMP/SMI MIB-based management applications and

easy integration of displays of network information with on-line documentation or manuals. The HTML output of many of these systems is designed to be human readable, but not necessarily machine readable [3]. Another disadvantage of breaking with SNMP altogether is that new software must be written to communicate with SNMP-speaking agents and applications. XNAMI allows easy integration of MIBs into online documents without departing from the SNMP standard.

Recently, a new representation for management data called the Common Information Model (CIM) has been proposed [9]. CIM is based on the Unified Modeling Language (UML), and captures information about a network by modeling it as a collection of objects having certain relationships with each other. To date, no API or communication protocol has been defined for exchanging CIM information between applications. Like XNAMI, CIM will also use XML to exchange object definitions [9]. This makes it particularly easy to extend XNAMI's DOM MIB representation to represent CIM models as well. This would allow an XNAMI agent to be a CIM agent in addition to being an SNMP agent, as soon as a communication protocol for CIM is specified.

3 An XML Document Representation of MIBs

This section gives a brief introduction to the Extensible Markup Language (XML) and the Document Object Model (DOM) and presents a partial XML document for describing SMIv2 MIBs.

3.1 XML

XML is a *markup metalanguage*, i.e. a language for defining markup languages [6]. A *markup language* is a set of conventions used together for annotating text. The best known example of a markup language is Hypertext Markup Language (HTML), which is used to create web pages. HTML is defined using the markup metalanguage Standard Generalized Markup Language (SGML), a predecessor of XML [18]. XML is a simplified dialect of SGML which was designed to allow it to be served on the Web in a similar manner to HTML.

A self-contained piece of XML is called a *document*, and consists of storage units called *entities*, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form the character data in the document, and some of which form *markup*, a description of the document's storage layout and logical structure. Unparsed data may consist of characters or binary data, and contains no markup. Markup in XML (and SGML) consists of *elements*, which contain text or other elements. An example of an element is the list element in HTML, which is <li> ...some text... . The element consists of a *start tag*, <li>, the contents ("...some text..." in the above example), and an *end tag*, . Elements may also have *attributes*, which contain additional information about an element instance. An example of an element containing attributes is the following date element, in which YEAR, MONTH, and DAY are all attributes: <

date YEAR=1998 MONTH=07 DAY=28 />. This date element has no content because all of the information is present in the attributes, so the end tag is omitted and a backslash is placed just before the closing angle bracket instead.

To define a markup language using XML or SGML, one specifies the elements present in the language, the attributes which each element may or must have, and the relationships between elements (such as, that element type A may contain an element of type B, followed by an element of type C). These specifications occur in a *document type definition* (DTD). The SGML DTD for HTML may be found at [10]. Section 3.2 describes the DTD for the XML representation of the XNAMI MIB.

When an XML document is processed, it is typically parsed into a labeled directed graph where each node is an instance of an element, and node A has an arc to node B if element instance A has a relationship with element instance B. Examples of relationships which might be represented in this graph are "contains", "has as an attribute", "is an attribute of", and "is contained by". This graph is analogous to the parse tree produced by a parser of a high-level programming language such as C or Java. So that different XML applications can share this graph structure of a document with each other, a standard interface has been defined. This interface is known as the Document Object Model (DOM). DOM defines how the graph produced by parsing an XML document should look, and it also defines object interfaces to the nodes and arcs of this graph in OMG IDL [20]. A DOM instance of an XML document is realized by creating an object for each element of the document, using Java, DCOM, CORBA, or some other distributed object framework [20].

3.2 A Partial DTD Example

Unlike most SNMP agents, the XNAMI agent maintains an explicit runtime representation of its MIB, which is separate from and independent of the agent code itself. XNAMI maintains its MIB as a DOM representation of an XML document which reflects the structure of the OID tree. This representation is generated at startup by parsing an XML document describing the MIB. The objects in XNAMI's DOM representation are Java objects.

Figure 2 shows a Document Type Definition (DTD) for XML documents describing SMIv2 MIBs. A managed object is represented by an element *SMIV2-OBJECT* composed of five mandatory elements *NAME*, *OID*, *SYNTAX*, *MAX-ACCESS*, *STATUS*, *DESCRIPTION*, and three optional elements *REFERENCE*, *INDEXPART*, and *DEFVAL*. Element *NAME* contains an object name readable by people and is defined to be of type *PCDATA* (parsed text). Elements *OID* and *SYNTAX* also contain parsed text. *OID* provides an identifier of the object. *SYNTAX* refers to an abstract data structure corresponding to this object. Access information is given in element *MAX-ACCESS* consisting of exactly one of five possible subelements *not-accessible*, *accessible-for-notify*, *read-only*, *read-write*, or *read-create*. Each of these subelements are defined to be empty (i.e., they do not contain any text or elements). Element *STATUS* indicates whether this definition is current. *STATUS* includes one of three empty subelements

current, *deprecated*, or *obsolete*. Elements *DESCRIPTION* and *REFERENCE* provide textual description and cross-reference for the object. Element *INDEXPART* is used for instance identification when the managed object represents a conceptual row. *INDEXPART* consists of either subelement *INDEX* or subelement *AUGMENTS*. *INDEX* lists indexing objects described by optional empty subelement *IMPLIED* and subelement *OID*. *AUGMENTS* contains subelement *OID* that names the object corresponding to the augmented base conceptual row. Element *DEFVAL* defines an acceptable default value represented as parsed text.

```
<!ELEMENT SMIV2-OBJECT ( NAME,
                         OID,
                         SYNTAX,
                         MAX-ACCESS,
                         STATUS,
                         DESCRIPTION,
                         REFERENCE?,
                         INDEXPART?,
                         DEFVAL? )>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT OID (#PCDATA)>
    <!ELEMENT SYNTAX (#PCDATA)>
    <!ELEMENT MAX-ACCESS (not-accessible
                        | accessible-for-notify
                        | read-only
                        | read-write
                        | read-create)>
        <!ELEMENT not-accessible EMPTY>
        <!ELEMENT accessible-for-notify EMPTY>
        <!ELEMENT read-only EMPTY>
        <!ELEMENT read-write EMPTY>
        <!ELEMENT read-create EMPTY>
    <!ELEMENT STATUS (current | deprecated | obsolete)>
        <!ELEMENT current EMPTY>
        <!ELEMENT deprecated EMPTY>
        <!ELEMENT obsolete EMPTY>
    <!ELEMENT DESCRIPTION (#PCDATA)>
    <!ELEMENT REFERENCE (#PCDATA)>
    <!ELEMENT INDEXPART (INDEX | AUGMENTS)>
        <!ELEMENT INDEX (IMPLIED?, OID)+>
            <!ELEMENT IMPLIED EMPTY>
        <!ELEMENT AUGMENTS (OID)>
    <!ELEMENT DEFVAL (#PCDATA)>
```

Fig. 2. A DTD for SMIv2

sysOREntry is an SMIv2 object defined in [5]. Figure 3 presents *sysOREntry* as a XML document conforming to the DTD given above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SMIV2-OBJECT SYSTEM "smi-object.dtd">
<SMIV2-OBJECT>
    <NAME>
        sysOREntry
    </NAME>
    <OID>
        1.3.6.1.2.1.1.9.1
    </OID>
    <SYNTAX>
        SysOREntry
    </SYNTAX>
    <MAX-ACCESS>
        <not-accessible/>
    </MAX-ACCESS>
    <STATUS>
        <current/>
    </STATUS>
    <DESCRIPTION>
        An entry (conceptual row) in the sysORTable.
    </DESCRIPTION>
    <INDEXPART>
        <INDEX>
            <OID>
                1.3.6.1.2.1.1.9.1.1
            </OID>
        </INDEX>
    </INDEXPART>
</SMIV2-OBJECT>
```

Fig. 3. *sysOREntry* represented as an XML document.

4 XNAMI Architecture

XNAMI provides an elegant and flexible way to manage networks by supporting the following capabilities:

- New variables can be added to the MIB at a managed element. The definition and code required to access the new variable are downloaded from the manager to the agent.

• Unused variables can be deleted from the MIB at a managed element. The definition and code required to access the variable are removed from memory or disk, thereby conserving system resources.

• A view of the MIB for each managed element is available to a manager for browsing.

• Variables in the MIB at a managed element can be selected for monitoring. Each variable can have a customized monitoring patch which determines how the value of that variable will be displayed at the manager. A plotting chart is invoked for each monitored variable which displays the results of periodic polling of the value of the variable.

The entire XNAMI implementation is in Java. It uses the SNMPv3 protocol to send messages between the manager and the Agent. It also provides a web interface for the manager to send commands for adding, deleting, and monitoring variables to the Agent.

The requirements placed by XNAMI on the managed element are as follows:

1. There must be a Java Virtual Machine at the managed element.

2. The code to access an added variable in the MIB has to have the permission to execute at the managed element, i.e. must run with privileges if it needs to.

Figure 4 shows the architecture for the XNAMI system. It consists of three parts: the agent code which will reside on the managed element, the manager code which can reside on any system, and a web browser interface that the manager uses to interact with the agent. The remainder of this section describes the individual components of the manager and the agent, and describes how XNAMI uses XML to achieve MIB extensibility.

4.1 XNAMI MIB

An example of an XNAMI MIB tree is shown in Figure 5. Some of the uppermost nodes in the tree have been omitted for clarity. In the XNAMI MIB tree, leaf nodes represent items of management information which cannot be further subdivided (columnar or scalar objects in SNMP parlance). All other objects are internal nodes. The objects in a MIB actually represent classes of management information which may have one or more instances. In the XNAMI MIB, the code to perform a GET or SET on an instance of an object is stored in the node for that object. Any additional information concerning an instance of an object is stored in a cell of an array indexed by the instance number, and the array is stored at the object node. When XNAMI receives a GET or SET request, it takes the OID argument to the request and traverses the MIB tree until it finds the node having that OID. Because only instances of scalar or columnar objects can be the targets of GETs or SETs, this node must be a leaf node. XNAMI then retrieves the code (using Java's dynamic classloading capabilities) for performing the GET or SET and executes it, passing in the instance number as a parameter. The GET or SET code can use the instance number to access the array to retrieve information associated with the instance; in the extreme case, this could even be more code to execute, specific to that instance. A new

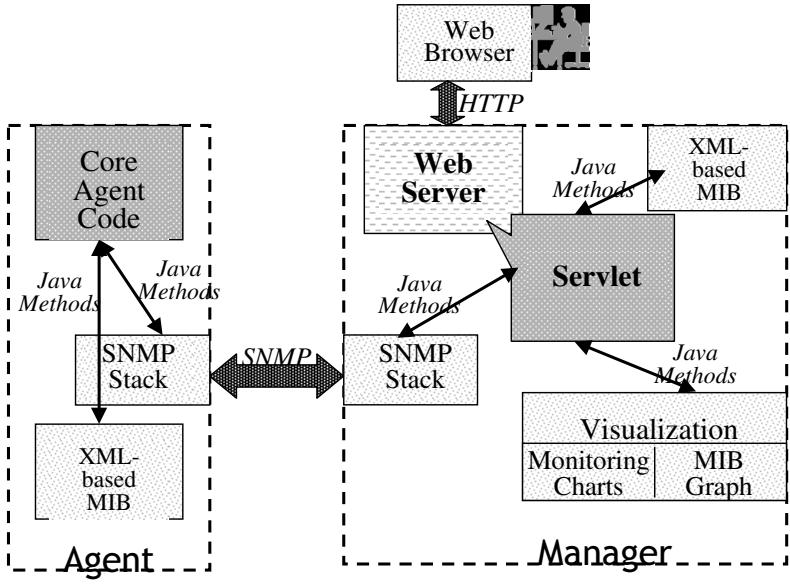

Fig. 4. Architecture for XNAMI

instance of an object can be created by doing a SET on the object with the instance number of the new instance.

To illustrate how objects are stored in an XNAMI MIB, consider tcpConnEntry, which is a composite object representing a TCP connection. It is composed of the objects tcpConnLocalAddress, tcpConnLocalPort, tcpConnRemAddress, and tcpConnRemPort. In the XNAMI MIB, tcpConnLocalAddresss is a leaf node, as are the other components of tcpConnEntry. tcpConnEntry itself is an internal node, with tcpConnLocalAddress and its other components as children. To retrieve the value of instance n of tcpConnEntry, a manager would do GETs on instance n for each of tcpConnEntry's components.

New MIB objects are added to the tree through the two leaf nodes shown in Figure 5, mib_proxy and methods_proxy. To add a subtree onto an agent's MIB, an XNAMI manager performs SET operations on the mib_proxy and methods_proxy objects. The structure of an SNMP SET PDU which performs this SET operation is shown in Figure 6. The PDU contains the OID for each proxy followed by a string containing the value to which it is set. Please note that there is nothing special about the structure of the PDU shown in Figure 6; strings are a common SNMP data type, and any SNMP SET PDU would look similar.

The value to which the manager SETs mib_proxy is a string containing a description in XML of the new objects in the subtree. This string is parsed in the same manner as the XML document describing the MIB at startup, and a DOM representation produced. This DOM representation of the new subtree is then grafted on to the MIB tree.

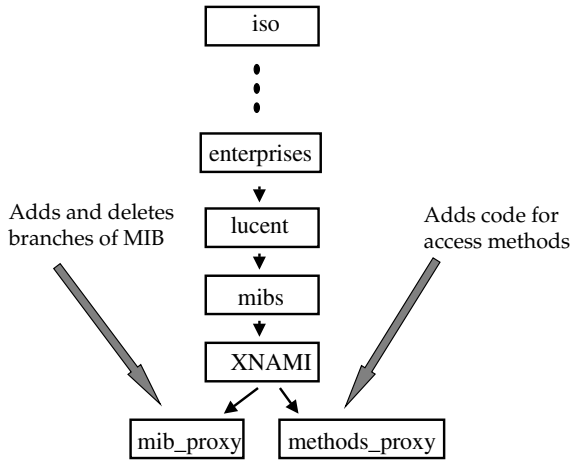

Fig. 5. Two new variables added to MIB

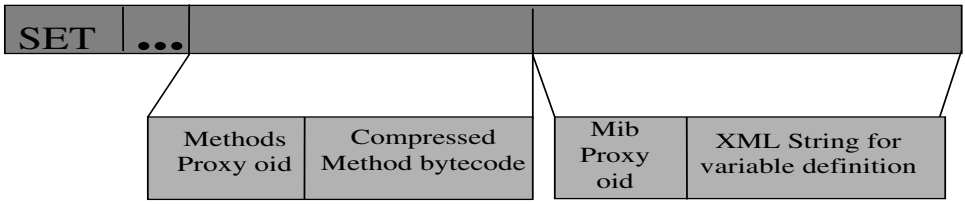

Fig. 6. SNMP PDU

The values which the manager passes to the methods_proxy object are strings containing compressed Java bytecode, one for each leaf node being added to the tree. The bytecode strings contain the code for doing GET and SET operations on the newly-added MIB objects. As the new MIB objects are created, the bytecode for each node is decompressed and loaded as a Java class containing two methods, one for a GET on the MIB object and the other for a SET. An instance of this class is then created and stored in the node created for the new MIB object. The use of compression necessitates an agreement between the manager and agent on the compression algorithm being used. If the bytecode sent to the agent or the data returned by the agent (see Section 4.2) does not fit into a single PDU, then it can be sent over multiple PDUs using a method similar to that described in [8].

Deletion of MIB objects is accomplished simply by performing a SET operation on mib_proxy. The XML string passed as a value contains the simple command "delete" and the OID of the node to be deleted. Deleting a node has the effect of removing the entire subtree rooted at that node.

The code for the GET and SET operations on the mib_proxy and methods_proxy objects is itself stored and accessed in precisely the same way as for any other object in the MIB.

4.2 Web Server and Servlet

The XNAMI manager consists of a servlet running on on a web server. Servlets are server-side analogs to Java applets, and allow new server functionality to be added seamlessly (without interrupting web server operation) at run time. Servlets receive http requests which are addressed to them from the web server, and then do some servlet-specific processing on them. The servlet interface facilitates portability in exactly the same way the applet interface does, because a servlet should run on any server which implements the interface [13]. Because the XNAMI manager is a servlet, it can be run on any web server that supports servlets. We have chosen to use the Nexus web server [13] for our work because it is free.

To manage an XNAMI agent, the user enters the agent's address in an HTML form on a web browser and submits it to a web server running the XNAMI manager servlet. The web server passes the form to the XNAMI servlet, which then contacts the agent by sending an SNMP GET request on the agent's mib_proxy variable. If the manager doesn't receive a reply within a timeout period, it sends a web page back to the user's browser explaining that it is unable to reach the agent. If it does receive a reply, the manager extracts the reply's payload which contains an XML string describing the agent's MIB. The manager parses this string to generate a DOM representation of the agent's MIB, which it then displays for the user as a tree.

The user requests the addition or deletion of variables in an agent's MIB through HTML forms generated by the XNAMI manager and displayed on the user's browser. The manager then sends to the agent SNMP SET requests on the mib_proxy and methods_proxy variables to carry out these requests. In response to each SET on mib_proxy, the agent returns an XML string describing the state of its MIB after the SET is performed. This XML string is, in a sense, the 'value' of the mib_proxy variable, and it is consistent with SNMP semantics to return the new value of a variable whenever a SET is performed on that variable. The XNAMI manager uses these XML strings to update the user's graphical display of the agent's MIB, and also to generate new HTML forms for the user's browser which provide choices to the user as to which variables can be uploaded to the agent, deleted from the agent, or monitored. 'Monitored' variables have their values polled periodically by the manager and to the user.

4.3 Distributed Management Issues

In a distributed management scenario, there could be multiple managers for a managed element and the potential for confusion if one manager modifies variables being monitored by another manager. Such confusion can be avoided by using SNMP traps to notify managers if another manager has altered a part

of an agent's MIB. The agent's MIB could also include a log of all the changes made to the MIB. Potential conflicts in OIDs for variables added to an agent's MIB by different managers in an organization can be avoided by having a central repository for OIDs or by giving each manager a separate subtree of the MIB.

5 Conclusions and Future Work

This paper describes the ideas and implementation behind an XML-based architecture for SNMP management of networks and applications. This architecture, called XNAMI, allows runtime MIB extension within the SNMP framework. The use of XML allows the MIB to be shipped over the network, browsed easily, and seamlessly integrated with online documentation for management tasks, especially configuration ones. XML also facilitates interoperability by allowing for the easy interchange of data between management applications using different information models.

We plan to extend the XML description of MIB objects in XNAMI to include suggestions to the XNAMI manager on how data for that object should be presented visually to the user. For example, an object such as the number of dropped packets at an interface might most logically be presented as a line chart and plotted with respect to time, while the up or down status of an interface could be presented using an icon which is either green (up) or red (down). The advantage to storing visual presentation cues such as these with the XNAMI agent is that managers will be able to more appropriately display data from MIB objects with which they have no familiarity. We also plan to explore the use of the XNAMI framework in various application management areas such as distributed program debugging where program variables in memory have to be monitored.

Acknowledgements
Sergey Gorinsky from The University of Texas at Austin helped in the design of the DTD presented in Section 3.2. Jean-Philippe Martin-Flatin from The Swiss Fed. Inst. of Technology, Lausanne (EPFL) helped with comments which improved the final version of the paper.

References

1. AdventNet. http://www.adventnet.com.
2. N. Anerousis. A distributed computing environment for building scalable management services. In *Proceedings of the 6th IFIP/IEEE Integrated Management, Boston MA, May 1999.*, May 1999.
3. B.Bruins. Some Experiences with Emerging Management Technologies. *The Simple Times*, 4(3):9–12, July 1996. http://www.simple-times.org.
4. C.-W.Tsai and R.-S.Chan. SNMP through WWW. *International Journal of Network Management*, 8(2):104–119, March-April 1998.

5. J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. *Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)*, January 1996. IETF RFC 1907.
6. C.M.Sperberg-McQueen and L.Burnard. Guidelines for electronic text encoding and interchange (tei p3). http://www-tei.uic.edu/orgs/tei/sgml/teip3sg/.
7. C.Wellens and K.Auerbach. Towards Useful Management. *ConneXions*, 10(9):2–9, September 1996.
8. J. Schoenwaelder D. Levi. *Definitions of Managed Objects for the Delegation of Management Scripts*, May 1999. IETF RFC 1902.
9. DMTF. Common Information Model Web page. http://www.dmtf.org/spec/cims.html.
10. I.Jacobs D.Raggett, A.Le Hors. Html 4.0 strict dtd. http://www.w3.org/TR/REC-html40/sgml/dtd.html#dtd.
11. P. Kalyanasundaram, A. Sethi, C. Sherwin, and D. Zhu. A Spreadsheet-Based Scripting Environment for SNMP. In *Integrated Network Management, V*, pages 752–765. Chapman and Hall, 1997.
12. S. Kille and N. Freed. *Mail Monitoring MIB*, January 1994. IETF RFC 1566.
13. Anders Kristensen. Nexus web server page. http://www.hplbwww.hpl.hp.com/people/ak/java/nexus.
14. J.P. Martin-Flatin. Push vs. Pull in web-based network management. In S. Mazumdar M. Sloman and E. Lupu (Eds.), editors, *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*. IEEE Press, May 1999.
15. M.White and S.Gudur. An Overview of the AgentX Protocol. *The Simple Times*, 6(1), March 1998. http://www.simple-times.org.
16. D. Perkins and E. McGinnis. *Understanding SNMP MIBs*. Prentice Hall, 1997.
17. P.Mullaney. Overview of a Web-based Agent. *The Simple Times*, 4(3):12–18, July 1996. http://www.simple-times.org.
18. R.Cover. The SGML/XML Web Page. http://www.oasis-open.org/cover/sgml-xml.html.
19. S.Roberts. An Introduction to SNMP MIB Compilers. *The Simple Times*, 2(1), January 1993. http://www.simple-times.org.
20. W3C. The DOM Web page. http://www.w3c.org/DOM.
21. D. Zhu, A. Sethi, and P. Kalyanasundaram. Towards Integrated Network Management Scripting Frameworks. In *Proceedings of the Ninth Annual IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, pages 233–246, Newark, DE, October 1998.