A JMX Toolkit for Merging Network Management Systems

Feng Lu **Kris Bubendorfer**

School of Mathematical and Computing Sciences
Victoria University of Wellington,
P. O. Box 600 Wellington, New Zealand,
Email: Feng.Lu@mcs.vuw.ac.nz, kris@mcs.vuw.ac.nz

Abstract

The ever increasing size of networks has resulted in a corresponding escalation of administration costs and lengthy deployment cycles. Clearly, more scalable and flexible network management systems are required to replace existing centralised services. The work described in this paper forms part of a new network management system that fuses dynamic extensibility, Java Management Extension (JMX), and mobile agents. The primary focus is on integration with the many widely deployed legacy SNMP-based network management systems. One of the primary contributions is the design of a generic SNMP adaptor to enable JMX compliant agents to be accessed by SNMP-based management applications. A set of SNMP APIs have been developed to support the development of the SNMP adaptor. A number of other tools have been developed to support the SNMP adaptor, these include: a Management Information Base (MIB) compiler that automatically generates MBeans representing a given SNMP MIB; and a SNMP proxy service to allow non-SNMP management applications to access the SNMP agent using a variety of protocols.

Keywords: JMX, Network Management, SNMP

1 Introduction

Traditional network management (NM) approaches, such as SNMP (Simple Network Management Protocol) and CMIP (Common Management Information Protocol), are based on a static centralised management platform: a centralised manager acting as client controls the entire network through agents which reside in each network node acting as server. Agents are responsible for monitoring and controlling managed objects in the network. The manager has the responsibility of collecting data from agents, interpreting that data and directing the agents (Yemini et al. 1991). However, with the rapid growth of networks, such approaches are no longer suitable as the increasing complexity of these systems results in high administration costs and long deployment cycles. The need for more scalable and flexible network management approaches is leading to greater decentralisation (Simoes 1999).

Dynamic extensibility is one of the solutions that has been used to support the distributed network management model. The extensible agent can add or delete managed objects and management services at run time upon requests by other management entities. Recently, Sun Microsystems introduced a set of standards to equip Java with an extensible agent model for distributed management, known as Java Management Extension (JMX).

JMX aims to achieve the goal of scalable, distributed network management (JMX1.2 2002). Its support for mobile code enables the transfer of lightweight applications to management agents at runtime, delegates the management tasks from a centralised manager to management agents distributed around the network, and place the management tasks closer to the management data. This reduces network traffic and increases scalability (Lange et al. 1999). Also, the JMX component based architecture allows each JMX resource or service to be plugged into or removed from the management agent dynamically, depending on the runtime network requirements. This means that a JMX based implementation can scale from small handhold devices to large telecommunications switches.

However, legacy management systems are still widely deployed. Network managers have to rely on the legacy management protocol to access the network resources in the heterogeneous network environment. Therfore, a JMX based solution needs to coexist with and integrate with traditional network management systems, like SNMP, instead of replacing them. It is attractive and cost-efficient to develop and deploy a distributed management system using JMX that can cooperate with deployed legacy system. This interoperability can be achieved by equipping the JMX-based solution with SNMP capability. Without it, the JMX-based solution will not become a general solution for distributed network management.

The research efforts presented in this paper focus on the the integration of JMX with traditional SNMP-based network management systems. One of the primary contributions is the design of a generic SNMP adaptor to enable JMX compliant agents to be accessed by SNMP-based management applications. A set of SNMP APIs have been developed to support the development of the SNMP adaptor. A number of other tools have been developed to support the SNMP adaptor, these include: a Management Information Base (MIB) compiler that automatically generates MBeans representing a given SNMP MIB; and a SNMP proxy service to allow non-SNMP management applications to access the SNMP agent using a variety of protocols.

2 Background

The majority of deployed network management systems utilise a centralised approach, where the management application periodically accesses the data collected by a set of software modules on network devices. The software modules on network devices are mainly concerned with information gathering and simple calculation, while the management application handles decision making and higher level functions. The centralised approach is driven by two assumptions (Goldszmidt 1993):

- Network devices lack resources to execute complex computational tasks.
- Management data and functions are relatively simple.

The Simple Network Management Protocol is the dominant protocol in existing managed systems. The protocol is designed to be an easily implemented, basic network management tool. The SNMP set of standards defines an information management model along with a protocol for the exchange of the information between a managed device with an SNMP agent and an SNMP manager. International Standard Organisation (ISO10165-1 1993) presents another general management information model of OSI systems management information. The ISO model has a similar approach to the SNMP management model, but differs in the way it operates.

The rapid expansion of networks has resulted in real network management problems that can't be adequately addressed (Meyer et al. 1995). Also, the computational capability of network devices has increased. The increase in the capability of managed devices has made it possible to distribute complex computations and significant duties to the managed devices (Puliafito et al. 2000). Research on the decentralised approaches to network management began as early as SNMPv1's RMON (Remote Network Monitoring) MIB. The core of RMON are the remote monitors, that take responsibility for the collection and analysis of statistical information on network traffic and device status on sub-networks. The remote monitors report only significant information to the SNMP managers. The enhanced SNMPv2 provides a manager-to-manager (M2M) MIB to support a hierarchical management architecture. Similar to the RMON, the M2M allows a sub-manager to function as a remote monitor for a sub-network. The latest SNMPv3 management framework makes it possible to develop a set of distributed entities, composed of several interacting modules. However, the SNMP management framework does not specifically address distribute network management. Instead, the IETF DIAMAN working group proposes a distributed management architecture based on the SNMP management framework (DISMAN 1996).

On the other hand, Yemini and Goldszmidt (Yemini et al. 1991) proposed the Management by Delegation (MbD) model for distributed network management. The fundamental idea behind this approach is to dynamically distribute management functions amongst management entities. The MbD model is based on the technology "code mobility". It moves the code, describing management functions, closer to the data they process. Moving code is more efficient if the amount of data that needs to be transferred is larger, and reduces the total amount of network management traffic (Schonwalder 1997). The MbD model consist of three parts: a delegation protocol, a delegation language and an agent. The delegation protocol is used to communicate between managers and agents. The delegation protocol enables the manager to transfer the delegation code, to control the behaviour of the delegation code (execute, suspend and stop etc.), and to retrieve the results of the execution. The delegation language is used to write management functions, that can be executed at runtime. Several different languages have been in different research prototypes ranging from high-level interpreted languages to low-level stack-oriented languages (Schonwalder 1997). A MbD agent acts in both an agent role and a manager role. To managers requesting information from the MbD Manager, it is an agent, while to those agents it queries, it is a manager. The MbD agent provides the services to parse and execute received delegation code. It also provides the interfaces that enable the remote manager to control the execution of the delegation and retrieve the results. In addition, the MbD agent can delegate its management functions to other MbD agents.

Dynamic extensibility has been used to support the MbD model for distributed network management. Extensible agents are MbD agent that can dynamically add or delete managed objects upon the requests from other management entities. The early extensible agent model is based on the SNMP framework with a distributed MIB consisting of a static MIB residing in the master agent and several temporary MIB dynamically registered by subagents.

This paper relates our experience in designing and implementing a JMX based network management toolkit.

3 JMX Architecture

The JMX specification provides a framework for a distributed management model based on manageable resources, dynamically extensible agents and distributed management services as shown in Figure 1. The JMX architecture is separated into three layers: the instrumentation level, the agent level and the distributed services level.

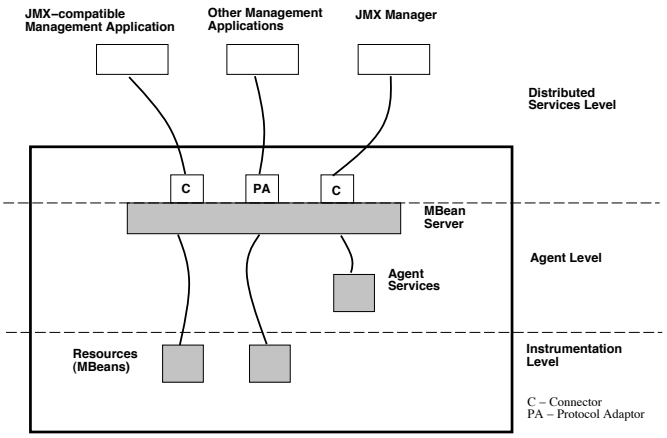

Figure 1: *JMX Architecture*

3.1 Instrumentation Level

The instrumentation level provides a specification for implementing JMX manageable resources. A resource can be an application, a device, or the implementation of a service. A JMX manageable resource must comply with the MBean standard defined in the JMX specification, and may be dynamically added to or removed from the JMX agent. MBeans encapsulate manageable objects as attributes and operations through their public methods, and utilise de-

sign patterns to expose them to management applications (JMX1.2 2002).

There are two kind of MBean. Standard MBeans provide a static management interface, which is fixed at compile time and is invoked by reflection. The standard MBeans' interfaces are made up of the methods for reading and writing attributes and for invoking operations. The design pattern followed by a standard MBean is derived from the JavaBeans component model (JavaBeans 1999). In this design pattern, attributes are exposed through the getter and setter methods in the MBeans' interface. Attributes may be read-only, write-only or read-write. The return value or arguments of methods for attributes must conform to the data type of attributes. Operations are exposed by the methods other than getter and setter in the MBeans' interface. They can be defined with any number of arguments with any data types.

```
public interface DynamicMBean {

    public Object getAttribute(String attribute)
      throws AttributeNotFoundException, MBeanException, ReflectionException;

    public AttributeList getAttributes(String[] attributes);

    public MBeanInfo getMBeanInfo();

    public Object invoke(String actionName, Object[] params, String[] signature)
      throws MBeanException, ReflectionException;

    public void setAttribute(Attribute attribute)
      throws AttributeNotFoundException,InvalidAttributeValueException,
             MBeanException, ReflectionException;

    public AttributeList setAttributes(AttributeList attributes);
}
```

Figure 2: *DynamicMBean Interface*

Dynamic MBeans conform to a specific interface that exposes the management interface at runtime. Unlike standard MBeans, dynamic MBeans do not have getter or setter methods for each attribute and operation. Instead, the *DynamicMBean* interface is defined to provide generic method for getting or setting an attribute and for invoking an operation. As shown in Figure 2, the *getMBeanInfo* method defined in the *DynamicMBean* interface returns an object which contains meta information about the MBean's attributes, operations and notifications that may be emitted by the MBean.

Using this meta information, management applications can access the MBean's attributes and invoke the MBean's operations through generic methods defined by the *DynamicMBean* interface. Compared with standard MBeans, dynamic MBeans provide a more flexible way to instrument resources and make it simple to instrument existing JMX incompatible resources (legacy management resources, etc.).

Dynamic MBeans can be further refined into two useful specialisations:

- An open MBean is a dynamic MBean that relies on a small, predefined set of universal Java Types to describe managed objects. It is useful where a management application and agent do not share application-specific data types.
- The model MBean, is a generic configurable management template for managed resources. Model MBeans can be used to instrument almost any resources rapidly.

3.2 Agent Level

The agent level provides a specification for implementing the JMX agents that control the MBean resources and make them available to management applications. A JMX agent consists of a MBean server, a set of agent services, and at least one communication protocol adaptor or connector, see section 3.3. The MBean server acts as a central registry for MBeans managed by the agent. Only registered MBeans may be accessed from outside of the MBean server. The MBean server provides a set of interfaces to manipulate MBeans. All management requests are handled by the MBean server, which dispatches them to the appropriate MBean. Through the MBean server, management applications may: register or deregister MBeans, browse and query MBeans, discover the management interface of MBeans, read or write the values of MBeans' attributes, invoke the operations exposed by MBeans, and register and deregister notification listeners for MBeans.

JMX agent services are also MBeans that provide services for other MBeans or management applications. There are four standard services defined in the JMX specification: Dynamic Loading Service, Monitoring Service, Timer Service and Relation Service. Dynamic Loading Service allows the agent to instantiate MBeans using Java classes and native libraries dynamically downloaded from the network. Monitoring Service notifies its listeners on certain conditions or events. Timer Service sends notifications at predetermined intervals and acts as a scheduler. Relation Service defines associations between MBeans.

3.3 Distributed Services Level

The distributed services level defines management interfaces and components that allow remote management applications to perform operations on agents through different protocol adaptors and connectors. Both protocol adaptors and connectors use the services of the MBean server to apply the management operations they receive to the target MBeans, and to forward notifications, such as an attribute change notification, to management applications. Both protocol adaptors and connectors should preferably be implemented as MBeans. This offers greater flexibility to their operation as they can be activated or deactivated through any of the other available adaptors or connectors.

There are two main differences between protocol adaptors and connectors, though they are similar in terms of functionality:

- Management applications that connect to protocol adaptors access the JMX agent through operations defined by the given protocol, and the operations are then received by protocol adaptors and are mapped to those of the MBean server through protocol adaptors; whereas connectors provide a higher level view for the JMX agent through the local representation of the MBean server. The remote management applications using connectors may access the JMX agent as if it were local.
- Management applications that connect to protocol adaptors are usually tied to a given protocol, whereas management applications which use connectors may use different protocols as long as corresponding connectors are provided.

4 SNMP Adaptor

The SNMP adaptor makes the JMX agent accessible from legacy SNMP managers. The SNMP adaptor emulates the standard SNMP agent, and is configured dynamically to provide mappings between SNMP and MBeans and JMX Notifications via XML mapping files. As shown in Figure 3, the SNMP adaptor consists of a SNMP protocol engine and a MIB registry. The SNMP protocol engine is used to receive and

parse SNMP messages to determine the type of request and the Object Identifier (OID) of the MIB object. The engine queries the MIB registry and gets the proxy for the MBean object identified by the OID. The engine then invokes the appropriate access function on the proxy, which will forward the invocation to the appropriate MBean object registered with the MBean server. The notification listener receives the notifications, which the SNMP adaptor is interested in, and forwards them to the SNMP protocol engine. The SNMP protocol engine generates the corresponding SNMP trap message.

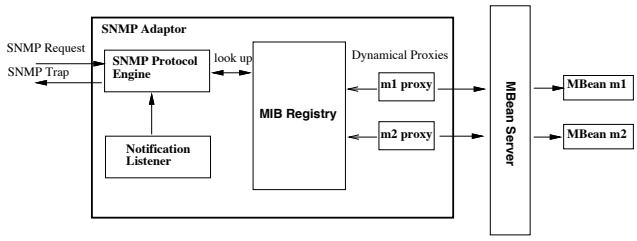

Figure 3: *SNMP Adaptor*

4.1 SNMP Protocol Engine

The SNMP protocol engine is built on top of the JoeSNMP API (OpenNMS 2002) and supports SNMP protocol versions V1, V2c. It consists of several components: transport layer, message dispatcher, message handler and trap generator. These components interact with each other to facilitate communications between the SNMP manager and the SNMP adaptor. Figure 4 describes how the SNMP message is handled by the SNMP engine.

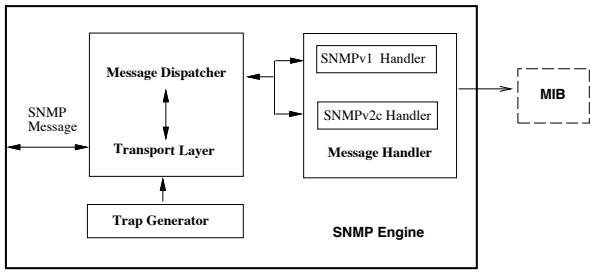

Figure 4: *SNMP Protocol Engine*

1. The SNMP request message is received by the transport layer, and then is forwarded to the message dispatcher. The current transport layer supports UDP.
2. The message dispatcher parses the SNMP message to determine the SNMP version and to extract the Protocol Data Unit (PDU) from the message. Then, it forwards the extracted PDU to the appropriate message handler.
3. There are two message handlers, the SNMPv1 handler and SNMPv2c handlers which are responsible for the corresponding version's SNMP message. The message handler parses the PDU to determine the PDU type and the OIDs of the required MIB objects. The message handler then looks up the MIB Registry to get the required MIB objects.
4. The message handler invokes the appropriate access method on the MIB objects, and then constructs a response message with the new values of the MIB objects.
5. The response message is returned back to the message dispatcher and then is forwarded to the transport layer.
6. The transport layer returns the response message. The transport layer also forwards SNMP trap messages to registered SNMP managers.

4.2 MIB Registry

The MIB registry organises MBean proxies into a SNMP OID tree structure. Figure 5 shows the class hierarchy for the MIB objects in the MIB registry. These objects are organised as several *MIBGroup* objects. A MIBGroup object can not contain other MIBGroup objects. The managed objects in the MIBGroup are represented as *MIBLeafProxy* objects or *MIBTableProxy* objects in terms of the node type.

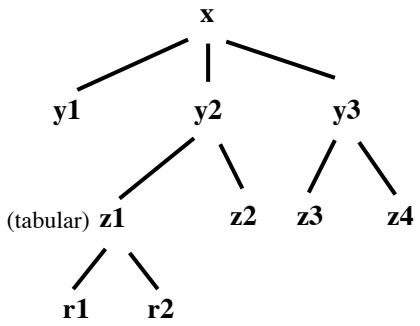

Figure 6: *MIBGroup Example*

For instance in Figure 6, the node `x` has three child nodes: `y1`, `y2` and `y3`. The node `y1` is a leaf node. The node `y2` has two child nodes: `z1` (tabular node) and `z2` (leaf node). The node `y3` also has two child nodes: `z3` (leaf node) and `z4` (leaf node). This OID tree can be represented as follows:

- *MIBGroup* `x` contains *MIBLeafProxy* `y1`
- *MIBGroup* `y2` contains *MIBTableProxy* `z1` and *MIBLeafProxy* `z2`
- *MIBGroup* `y3` contains *MIBLeafProxy* `z3` and *MIBLeafProxy* `z4`

Referring back to Figure 5, the *MIBEntry* abstract class describes the basic structure for a managed object. It contains the attribute *oid* which is used to identify the managed object. It also defines three abstract methods *getRequest*, *getNextRequest* and *setRequest* to handle three primitive SNMP actions: *GET*, *GETNEXT* and *SET*. Both the *MIBLeaf* class and the *MIBTable* class are sub class of the *MIBEntry* abstract class.

A *MIBLeaf* class represents a scalar type managed object, but it also can represent a columnar object of a SNMP table. A columnar object defines the behaviour of managed object instances in a particular column of a SNMP table (Agent++ 2000). The *MIBLeaf* object contains an attribute *value* which represents the instance of the managed object. The *MIBLeafProxy* class extends the *MIBleaf* class with two additional attributes: *mbeanName* and *attribute*. The *mbeanName* represents the object name of the target MBean object, and the *attribute* represents one of the attributes of this MBean object. With these two attributes, the *MIBLeafProxy* object acts as a proxy for the MBean object, and maps the SNMP actions to the appropriate methods on the JMX agent. For instance, when the methods *getValue* or *setValue* is invoked, the *MIBLeafProxy* object invoke the method

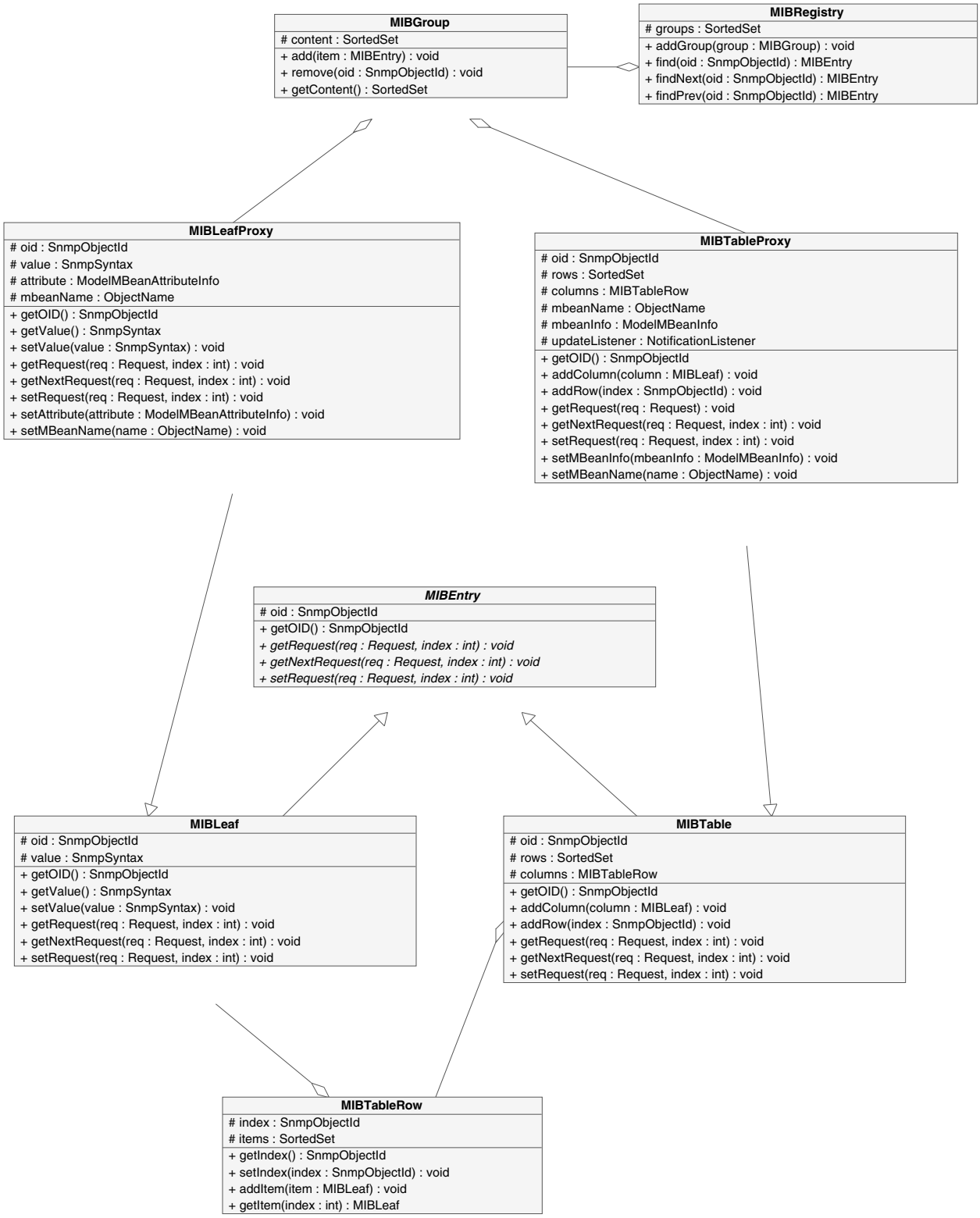

Figure 5: *MIB Registry Class Hierarchy Diagram*

getAttribute or *setAttribute* on the JMX agent with the *mbeanName* and the *attribute* as parameters to access the attribute of the target MBean object.

A *MIBTable* class represents a SNMP table (tabular type managed object). A SNMP table may have multiple rows, and each row consists of multiple columnar objects. The *MIBTableRow* class is defined to represent a row of a SNMP table. It provides the methods to add MIBLeaf objects, which represent columnar objects in the row, and methods to get and remove them. The *MIBTable* object may contain multiple *MIBTableRow* object. The *MIBTable* contains a group of *MIBLeaf* objects named *meta columnar object*, which describe the structure information for the row. This group of MIBLeaf objects is organised as a *MIBTableRow* object named *columns*. When a new row is added, *MIBTable* will clone the *MIBTableRow* object to create a new *MIBTableRow* object. Each columnar object in the new row is the copy of the meta columnar object of its column, but with a different value. The *MIBTable* class provides the methods to manipulate columnar objects.

The *MIBTableProxy* class is a sub class of the *MIBTable* class. It acts as the proxy for a special kind of MBean object, which has a *TabularData* type attribute. *TabularData* is defined in the JMX specification and describes a table structure with an arbitrary number of rows that can be indexed by any number of columns (JMX1.2 2002). Each row is a *CompositeData* object, which is a hash map with multiple data items. The *CompositeType* object is used to describe the *CompositeData* object. All rows in a *TabularData* object must be associated with the same *CompositeType* object. This special kind of MBean object is automatically generated from the SNMP table by the MIB compiler (see Section 4.3).

The *MIBTableProxy* class supports a cache mechanism for efficiency. When a *MIBTableProxy* object initialised, *MIBTableProxy* queries the MBean (identified by the *MIBTableProxy*'s two attributes: *mbeanName* and *MBeanInfo*), and stores the MBean object's *TabularData* type attribute in the cache. The *MIBTableProxy* object also registers a notification listener for the MBean object. When the MBean's *TabularData* object is changed, the *MIBTableProxy* object will receive a notification and will query the MBean to update the cache. The cache mechanism is more efficient because the *MIBTableProxy* object does not need to contact the MBean when a SNMP management application performs *GET* or *GETBULK* actions on it. Only *SET* actions cause the *MIBTableProxy* object to update the MBean's *TabularData* object.

4.2.1 Generating Dynamic MBean Proxies

Both the *MIBLeafProxy* and *MIBTableProxy* objects act as proxies for a MBean. The SNMP operations performed on them are mapped to the accessor methods on the appropriate MBean objects, and then are forwarded to the MBean server. The MBean server finds the target MBean object, invokes the method on it and then returns the result or raises the exception. Proxies are dynamically generated from XML mapping files and are added into the MIB Registry. The mapping files define the mapping relationship between the MIB and MBean objects. Figure 7 provides an example of the mapping of a MBean object into the MIB.

The *RMIConnectorServer* is implemented as a MBean so that it can also be managed through protocol adaptors or connectors. There must be a relationship between the *RMIConnectorServer* and the nodes in the MIB; otherwise the SNMP adaptor has no idea how to map a SNMP request to the operations on the *RMIConnectorServer*. The *RMIConnectorServer* exposes four methods defined in the interface *JMXConnectorServerBean*, see Figure 7. Two of them, *isActive* and *getAddress*, are the get methods of the attributes *active* and *address*, and other two are operations according to design pattern described in the JMX 1.2 specification. Our prototype only supports the mapping of MBean attributes as SNMP does not support objects. The file *MBeansToMIB.xml* is used to describe how to map MBeans into the MIB. The mapping file assigns the OID "1.3.6.1.4.9876.1.1" to the MBean *RMIConnectorServer*, and describes the MBean's ObjectName so that the SNMP adaptor can locate the *RMIConnectorServer* instance through the MBean server. It also assigns the OID respectively to the attributes *active* and *address*.

The mapping file also maps the Java data type of the MBean attributes to the MIB data type. In this case, the SNMP adaptor loads the mapping file, and generates a *MIBGroup* object with two *MIBLeafProxy* objects which respectively represent the attributes *address* and *active*.

4.3 MIB Compiler

As described in Section 4.2.1, existing MBeans can be mapped into the MIB using the *MBeansToMIB.xml* file. However, JMX manageable resources must follow the design patterns and interfaces defined in the JMX 1.2 specification. Any incompatible resources must be instrumented as MBeans so that they can be managed by a JMX agent.

Our MIB compiler automatically generates MBeans representing a given SNMP MIB. The MIB compiler consist of two components: a MIB parser and a code generator. The MIB parser imports a MIB file and generates an intermediate representation. The code generator generates the Java source code and the XML file. The generated code is based on the JMX's model MBean specification (JMX1.2 2002) and can be used to create a model MBean on the fly. The generated XML is used to dynamically configure the model MBean. The model MBean provides management interfaces for non JMX compatible resources. This significantly reduces the programming burden and means that a developer can instrument existing resources according to the JMX specification as little as three or five lines of code.

4.3.1 MIB Parser

The MIB file is a normal text file written in Abstract Syntax Notation One (ASN.1) (ISO8824 1990) language, a formal language used to define abstract syntaxes of application data.

Rather than having single ASN.1 compiler with a lexer, a parser and a code generator, we utilise delegating compiler objects (DCO) (Bosch 1996), a novel approach to compiler construction that provides modular and extensible implementation of compilers. In DCO compilation is achieved through the cooperation of a group of compiler objects. A compiler object is only responsible for a particular part of the syntax, and has its own lexer and parser. The programming language is decomposed into a set of structure. Each structure is compiled by its associated compiler object. As shown in Figure 8, a MIB file can be decomposed into ten modules. The *TypeAssignment* module is used to define a new data type. The new data type can be Simple Type, Structured Type or Subtype. The *ValueAssignment* module is used to assign a value to a variable. In the MIB file, the *ValueAssignment* module is mostly used to assign a value to the Object Identifier variable. The *Import* module is used to import the types and variable

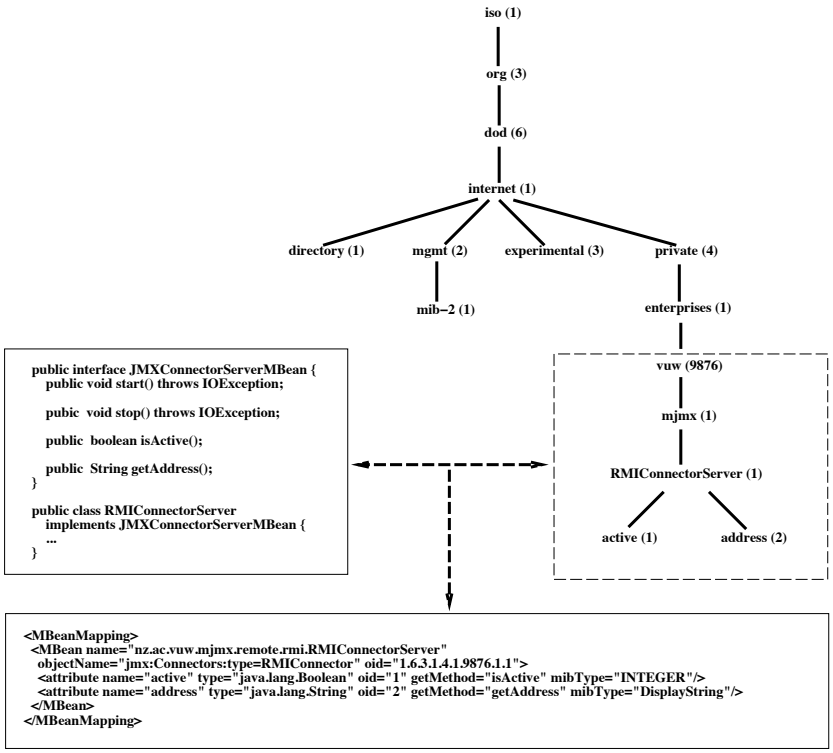

Figure 7: *Mapping a MBean into the MIB*

declared by other MIB files. The other seven modules, including *ModularIdentity*, *ObjectType*, *TextualConvention*, *ObjectGroup*, *NotificationType*, *NotificationGroup* and *ModuleComppliance*, represent seven macros defined in the MIB specifications. Our MIB compiler utilises a separate compiler object for each of the ten MIB modules. Each compiler object has a its own lexer and parser.

MIB compilation firstly eliminates all unneeded information in the MIB file (such as comments) and then passes the stream through the ten DCO compiler objects. Each DCO compiler performs its lexical analysis and then its parser provides the syntactic analysis. The entire syntactic analysis of the MIB file is the result of the collaboration of the different DCO parsers. Since only a very small subset of ASN.1 syntax is used in each DCO, the complexity of the implementation of DCO parser is significantly reduced. The output of DCO parsers are module objects representing the different modules. Module objects are divided into two groups: type groups and variable groups. Objects in the type group represent a data type, and objects in variable group represent an instance with type and value. Then, the module objects go through semantic analysis to check if objects are legal and meaningful (for instance, the values are valid, the types are defined, the compulsory attributes are set, and so on). The final step is to organise the objects as a MIB tree in terms of the OID value of each object. An optional XML file is also generated to represent the MIB file in the XML format.

4.3.2 Code Generator

The code generator walks through the MIB tree exported by the MIB parser and generates the instrumentation code and configuration files for the variables in the MIB tree. The code generator generates a Java class for each MIB group node. Every leaf node in the MIB group is represented by an attribute of the Java class. The corresponding accessor methods, such as *getX* or *setX* are defined in the Java class. In this case, *X* denotes the attribute name. Also, the code generator generates the Java class for each SNMP table. The methods to access the rows in the SNMP table are defined in this Java class. In addition, each generated Java class is associated with a XML configuration file that describes the mapping relationship between the Java class and the MIB. However, the generated instrumentation code only define the interfaces and provide skeleton code to describe how JMX incompatible resources can be accessed. The remaining manual tasks are to complete the skeleton code and implement the defined interfaces.

The generated Java classes can not be accessed directly by the JMX agent and must be wrapped into the model MBeans. The model MBean provides a set of interfaces which allow the JMX agent to perform the management operations on the resources wrapped in the model MBean object. The wrapping process starts by extracting the information for attributes, operations and notifications from the XML file associated with each Java class and then added this information to the model MBean. The whole process is done in one line code as follow. The method convertXmlToMBeanInfo converts the xml file into the MBeanInfo object which describes custom attributes, operations and notifications information, and then the RequiredModelMBean constructor use these information to construct a customised model MBean instance. The wrapping process is done automatically. The users can edit the file *JMXModelMBeanInfo.xml* to add the configuration file location and name for resources they want to wrap. When a JMX agent is initialzied, it checks out the file *JMXModelMBeanInfo.xml* and then generates the model MBean for the resources.

```
new RequiredModelMBean(convertXmlToMBeanInfo(xml));
```

4.3.3 A Code Generation Example

A code generation example for a MIB group is shown in Figure 9. The *system* group describes a set of objects common to all managed systems. It consists

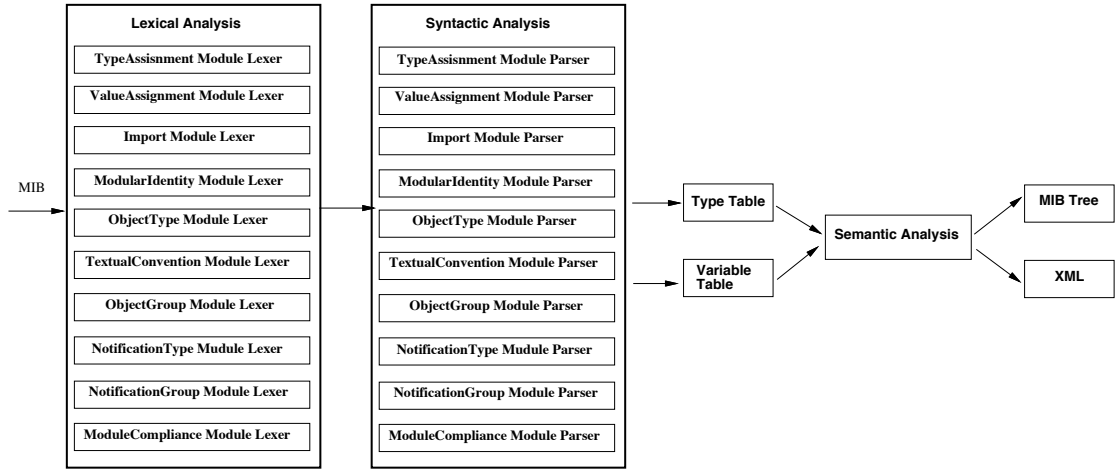

Figure 8: *MIB Parser*

of eight scalar objects and a table object with four columns. The MIB compiler compiles the *system* group MIB file and generates five files:

- System.java: a Java class representing the whole *system* group except the table object *sysORTable*
- System.xml: a configuration file describing the mapping relationship between the *System* class and the *system* group in the MIB file
- SystemORTable.java: a Java class representing the table object *sysORTable*in the MIB file
- SystemOREntry.java: a Java class representing four column objects in the table object *sysORTable*
- SystemORTable.xml: a configuration file describing the mapping relationship between the *sysORTable* class and the *sysORTable* object in the MIB file

The *System* class in Figure 9 contains eight attributes that represent the eight scalar objects in the *system* group. The accessor methods for these attributes are also included. The *System* class does not include any OID information, but the configuration file *System.xml* describes the mapping relation between the attributes of the *System* class and the scalar objects of the *system* group. When the *System* class is wrapped into a model MBean and is registered within a JMX agent, a proxy object is dynamically generated from the *System.xml* file and is registered within the *MIBRegistry* in the SNMP adaptor (see Section 4.2.1). This proxy object will call the *System* class's *get* and *set* method upon *Get* and *SET* SNMP request.

The *SysORTable* class contains the attribute *sysORTable* which represents the table object of the *system* group. The row of the *sysORTable* is represented by the *SysOREntry* class. The *SysOREntry* class contains four attributes that represent four column objects. The access methods for these attributes are also defined. The *SysORTable* class defines the methods to manipulate the table object. The *getSysORTable* method is used to retrieve all rows in the table. The *updateEntry* method is used to update an existing row or add a new row in the table, and the *deleteEntry* is used to delete a row from the table. Similar to the *System* class, a proxy object is also generated from the *SysORTable.xml* and is associated with the *SysORTable* class.

4.4 SNMP Proxy

The SNMP adaptor makes JMX resources accessible to legacy SNMP managers. However, non-SNMP management applications can not access SNMP resources directly since they do not support the SNMP protocol. We have designed and developed a SNMP proxy to address this interoperability issue. As shown in Figure 10, the MIB supported by a remote SNMP agent is represented by multiple model MBeans. These model MBeans are registered within the MBean server and can be accessed by multiple protocols, such as Java RMI. These model MBeans act as proxies and the operations on them are forwarded to the appropriate remote SNMP agents through the SNMP proxy.

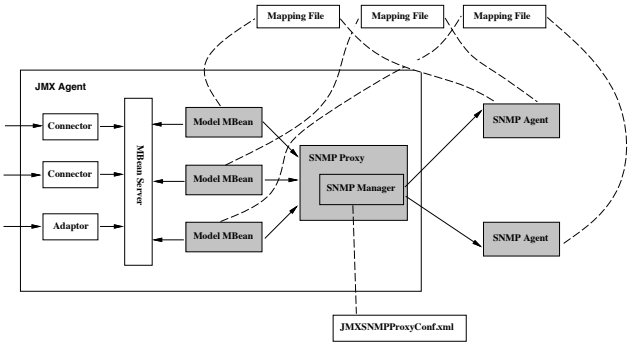

Figure 10: *SNMP Proxy*

To create proxy objects representing the remote SNMP agent's MIB, the MIB compiler is used to generated instrumentation code and xml configuration files from the remote SNMP agent's MIB file. However, only xml configuration files are used to create proxy objects. The instrumentation code is simply discarded. The JMX implementation used in this project provides two basic model MBeans: *RequiredModelMBean* and *JMXSNMPProxyModelMBean*. The *RequiredModelMBean* is used to instrument MBean incompatible managed resources. The operations on the *RequiredModelMBean* are forwarded to the managed resource. The *JMXSNMPProxyModelMBean* does not instrument any managed resources, but forwards the operations on it to the SNMP proxy. Generating a *JMXSNMPProxyModelMBean* instance using the toolkit is a single line of code (as shown below).

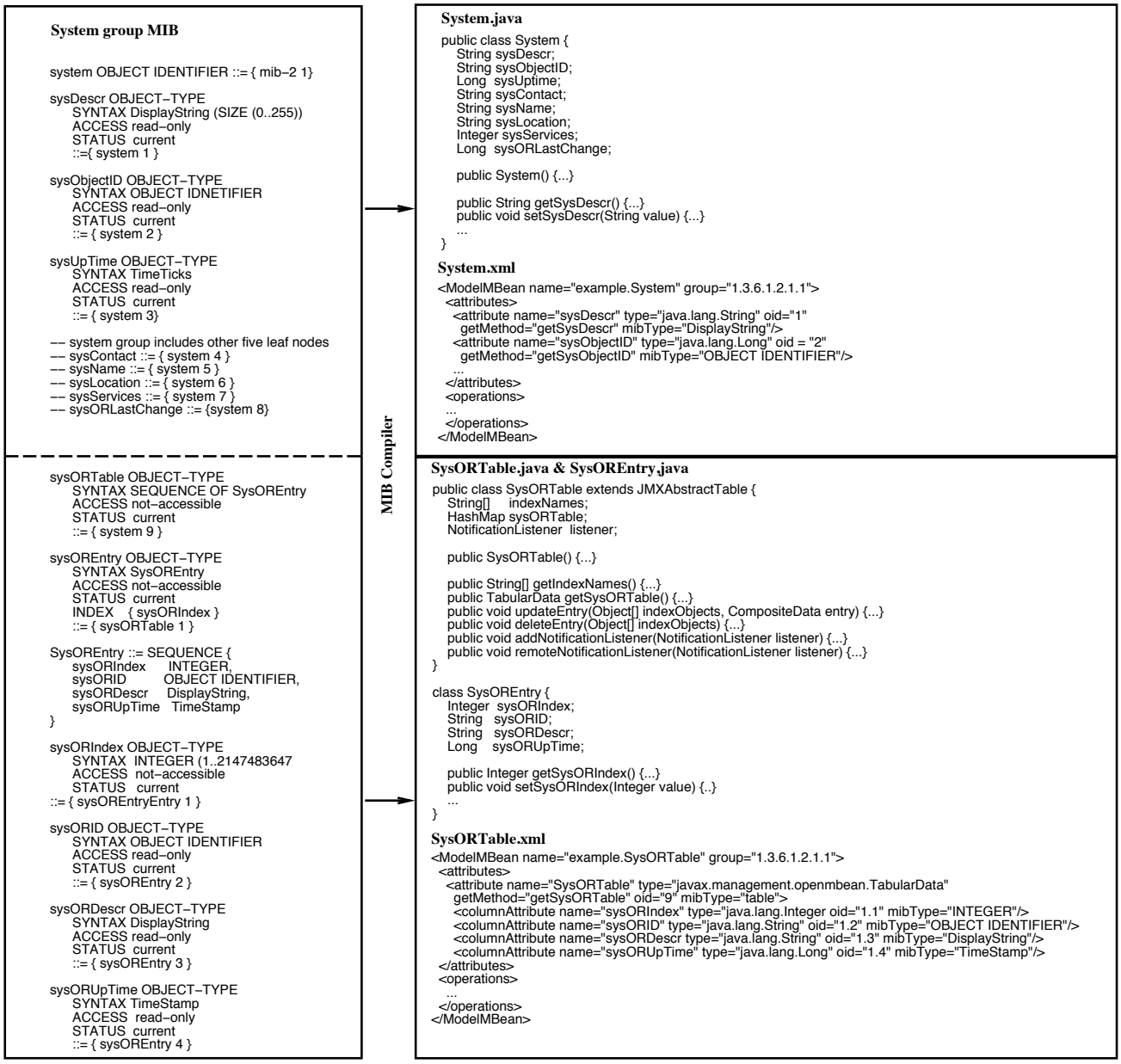

Figure 9: *A Code Generation Example*

```
new JMXSNMPProxyModelMBean(objectName, SNMPProxyRef,
                           convertXmlToMBeanInfo(xml));
```

The *objectName* represents the name of the model MBean object and the *SNMPProxyRef* is a reference to the SNMP Proxy. The convertXmlToMBeanInfo method converts the xml file into the MBeanInfo object that describes custom attributes, operations and notification information.

When a method of the *JMXSNMPProxyModelMBean* object is invoked, the *JMXSNMPProxyModelMBean* object forwards the invocation and object name to the SNMP proxy. The SNMP proxy checks the file *JMXSNMPProxyConf.xml* that describes the relationship between *JMXSNMPProxyModelMBean* objects and remote SNMP agents. For instance, an *JMXSNMPProxyModelMBean* object with object name "jmx:snmpagent:type=system" represents the System group of the MIB supported by a SNMP agent. This SNMP agent resides in the host "130.195.106.3" and listens on the port "161". After locating the SNMP agent, the SNMP proxy converts the invocation to a SNMP PDU, and sends it to the target SNMP agent.

5 Related Work

There are some commercial toolkits that provide broadly similar functionality to the work presented in this paper, such as Sun's JDMK toolkit (JDMK 1999). However, these are proprietary designs and their details are not available in the public domain. None-the-less, there are a number of important differences that we have been able to identify. For example, ordinary MBeans (those not generated by the JDMK's MIB compiler) can't be accessed by SNMP managers, whereas our toolkit enables ordinary MBeans to be accessible to SNMP managers via the "MBean-To-MIB" configuration file. Another difference is that JDMK generated MBeans are directly bound to the SNMP adaptor, whereas our MBean proxies are generated and bound at run time via the MBean Server. This is a cleaner more flexible solution, and conforms to the hourglass protocol model (Shanmugam et al. 2002).

6 Conclusions

The growing number of applications and services implemented in Java has increased the demand for Java based network management solutions. JMX provides a standard way to enable manageability for any Java based application, service or device (JMX1.2 2002). However, most existing management systems can not be managed directly via JMX compliant implementations. In this paper we present a toolkit that allows the rapid development of JMX agents, and that can interoperate with legacy SNMP-based network management systems. The core of this toolkit is a generic SNMP adaptor to enable JMX compliant agents to be accessed by SNMP-based management applications. A set of SNMP APIs have been developed to support the development of the SNMP adaptor. Several other tools have been developed to support the SNMP adaptor, these include: a MIB compiler that automatically generates MBeans representing a given SNMP MIB; and a SNMP proxy service to allow non-SNMP management applications to access the SNMP agent using a variety of protocols. A simple example is also given to illustrate the MBean generation process for a given SNMP MIB.

References

A. Puliafito & O. Tomarchio(2000), Using Mobile Agents to implement flexible Network Management strategies, Computer Communication Journal, 23(8):708–719.

Danny B. Lange & Mitsuru Oshima (1999), Seven Good Reasons for Mobile Agents, Communication of ACM, volume 42.

Frank Fock (2000), Agent++, An Object Oriented Application Programmers Interface for Development of SNMP Agents Using C++ and SNMP++, http://www.agentpp.com.

German Goldszmidt(1993), On Distributed System Management, In Processings of the Third IBM/CAS Conference.

IETF DIMAN Working Group (1999), Distributed Management (DISMAN) Charter, http://www.ietf.org/html.charters/disman-charter.html.

International Organization for Standardization (1990), Information Technology- Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1).

International Organization for Standardization (1993), ISO 10165-1: Information Processing System - Open Systems Interconnection - Structure of Management Information - Part1:Management Inforamation Model.

Jan Bosch (1996), Delegating Compiler Objects: Modularity and Reusability in Language Engineering, Nordic Journal of Computing.

J. Schonwalder(1997), Network Management by Delegation - From Research Prototypes Towards Standards, In Processings of 8th Joint European Networking Conference.

K. Meyer, M. Erlinger, J. Betser, C. Sunshine, G. Goldszmidt & Y. Yemini(1995), Decentralising Control and Intelligence in Network Management, In Processings of International Symposium on Integrated Network Management.

OpenNMS (2002), joeSNMP API, http://sourceforge.net/projects/joesnmp/.

Paulo Simoes (1999), Enable Mobile Agent Technology For Legacy Network Management Frameworks, Technical Report, University of Coimbra.

Sun Microsystem Inc. (2002), Java Management Extensions Instrumentation and Agent Specification, v1.2

Sun Microsystem Inc. (1999), Java Dynamic Management Kit, http://java.sun.com/products/jdmk/.

Sun Microsystem Inc. (1999), JavaBeans Specification 1.0.1, http://java.sun.com/products/jdmk/.

R. Shanmugam, R. Padmini, S. Nivedita. (2002), Special Edition: Using TCP/IP, 2nd Edition, Que.

Y. Yemini, G. Goldszmidt & Mitsuru Oshima (1991), Network Management by Delegation, International Symposium on Integrated Network Management.