JAX – A JAVA AGENT X SUBAGENT TOOLKIT

Frank Strauß *Jürgen Schönwälder* *Sven Mertens*
Computer Science Department
Technical University Braunschweig
Bültenweg 74/75, 38106 Braunschweig, Germany
{strauss,schoenw,mertens}@ibr.cs.tu-bs.de

July 2000

ABSTRACT

There is a growing demand for system components and applications to become manageable via SNMP. The Agent Extensibility Protocol [1] allows developers of such components to supply management functionality through an AgentX master agent without explicit knowledge of SNMP. Another observation reveals that more and more components are written in Java to achieve portability and to benefit from Java language features.

This paper presents the object-oriented architecture of a Java AgentX sub-agent toolkit [2]. A class model is presented that encapsulates the AgentX specific functionality in core classes. Only code which is specific to particular managed objects has to be developed individually by the programmer. Even this code can be based on class templates generated automatically by a MIB compiler.

1 INTRODUCTION

Three cases of MIB design and deployment can be distinguished: First, there are standard MIBs for specific protocols or classes of system components. These MIBs are designed to be generally applicable. Second, there are MIBs designed by vendors that model more specific and implementation dependent information. And third, there are users who need site-specific management information that cannot be retrieved from existent implementations of the two aforementioned categories, so that they have to design and implement their own MIBs.

Some MIBs of the first category are widely implemented and deployed. MIBs of the second category are used to fulfill vendor specific needs. However, vendors have to decide whether these MIBs can be defined and implemented at reasonable costs. Similarly, MIBs of the third category are costly from the user's point of view.

Besides the costs, there are also technical problems with the traditional concept of SNMP agents: Different system components and applications use different ways to export their management information. This leads to complicated and error-prone code integration or to multiple SNMP agents running on a single host on different non-standard ports.

Starting back in 1990 there have been different efforts to address these technical problems. Common to all approaches is the general concept of a master/subagent architecture, where only the master agent handles and demultiplexes the incoming SNMP requests and delegates them to those sub-agents that have registered with the master agent being responsible for particular subsets of managed objects. The most recent and standardized proposal that is expected to replace previous proprietary solutions is the Agent Extensibility Protocol (AgentX) [1].

The research efforts presented in this paper are concerned with the design and implementation of an AgentX sub-agent toolkit that fits into arbitrary Java components. A primary goal is to minimize the required knowledge of SNMP and AgentX expected from the component developer. This is wishful, since many companies are asked for SNMP support in their products without having the man power to delve into the details of SNMP. Furthermore, Java seems to be a good choice for the category of site-specific MIBs, since Java is a commonly known programming language that allows rapid prototyping at low costs.

We try to achieve these goals by supplying a high-level AgentX client interface together with a MIB compiler that generates Java code stubs from MIB modules.

The paper is organized as follows. Section 2 describes the AgentX framework for extensible SNMP agents. In Section 3 we introduce a simple MIB that will be used in subsequent examples. Section 4 documents the architecture of the JAX toolkit by means of UML diagrams. Section 5 describes the implementation of a code generator for MIB tables, scalar groups, and notifications. The performance of JAX sub-agents is compared to other MIB implementation approaches in Section 6. Some applications of JAX sub-agents are discussed in Section 7. Section 8 compares JAX with other Java management frameworks and AgentX implementations. We conclude with some remarks on the use of Java in AgentX sub-agents and the goals achieved by the JAX architecture in Section 9.

2 AGENT X

The master/sub-agent concept of extensible agents is not a new idea. The SNMP Multiplexing Protocol (SMUX) [3] and the SNMP Distributed Program Interface (DPI) [4] have been specified in 1991. Both approaches suffer from some performance, fault tolerance and data consistency problems. The DPI has been enhanced in 1994 [5], but both efforts are regarded as historic or experimental, respectively. Some companies developed proprietary architectures for extensible agents, e.g. Digital developed a simplified DPI, and Epilogue Technologies and SNMP Research created their own products Envoy and EMANATE [6].

In parallel to these industry efforts, the IETF SNMP Agent Extensibility Working Group [7] has been chartered in 1995 to work on a standardized agent extensibility framework [1] and an accompanied MIB [8]. The framework describes the processing entities (master agent, sub-agents), the encoding of all data types in the AgentX protocol, 18 administrative and management processing AgentX PDUs, and the elements of procedure by which the extensible agent processes SNMP messages and administrative AgentX messages. Administrative PDUs exist for opening and closing AgentX sessions, and registering and unregistering MIB regions with the master. Other administrative PDUs allow a sub-agent to allocate and deallocate indices of tables, to add and remove agent capabilities entries in the `sysORTable` [9] and to check whether the master agent is still running. Most SNMP PDUs are mapped to similar AgentX PDUs: Get, GetNext, GetBulk, and Notify. SNMP Set-PDUs must be handled by a 3-phase-commit protocol to achieve atomicity. This is realized by four AgentX PDUs: TestSet, CommitSet, UndoSet, and CleanupSet.

Using AgentX for providing management information has various advantages:

- AgentX is a standardized protocol. This means any programming language can be used to implement a MIB. AgentX master and sub-agent processes can be located and replaced arbitrarily.
- A complete sub-agent can be implemented in the context of the system component or application that populates the MIB. This is significantly easier than developing an agent that has to communicate and synchronize with the application. It looks reasonable for many embedded system components to implement their managed objects locally and communicate them via the AgentX protocol to the master agent. Similarly, it seems advisable to implement a MIB for a certain service within the correspondent service daemon, even if this daemon is written in a programming language different from the SNMP agent. E.g. the WWW-MIB [10] might be implemented within a Java WWW server like the W3C's Web server platform Jigsaw [11].
- The master agent does all security and access control processing. The sub-agents do not have to fiddle with these error-prone details.
- The AgentX protocol uses its own simplified encoding. There is no costly duplicated BER encoding/decoding at the master agent and the sub-agent.
- An AgentX sub-agent is independent of the SNMP version.

Currently, the AgentX specification is at proposed standard status. Several free and commercial implementations are available [12] and gain an increasing deployment.

3 A MIB EXAMPLE

In order to illustrate the Java classes of the JAX architecture, the code generation and the runtime behavior of a JAX sub-agent, we will use a simple example. Figure 1 shows a `TUBS-IBR-PING-MIB` that contains a single table with three columnar objects. The pingIndex column is the table index which uniquely identifies a table row. The pingAddress column contains an IPv4 target address. When the agent receives a Set-request for a pingAddress with an unused pingIndex, a new row will be created and the according ping measurement will be started. When the round trip time is measured it will be available from the pingRtt object of that row.

In sections 4 and 5 we use this MIB to describe the design and implementation of a JAX sub-agent.

```
TUBS-IBR-PING-MIB DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE, Unsigned32
        FROM SNMPv2-SMI
    ibr
        FROM TUBS-SMI;

pingMIB MODULE-IDENTITY
    LAST-UPDATED "200007070000Z"
    ORGANIZATION "TU Braunschweig"
    CONTACT-INFO
        "Frank Strauss
         TU Braunschweig
         Bueltenweg 74/75
         38106 Braunschweig
         Germany

         Tel: +49 531 391 3266
         Fax: +49 531 391 5936
         E-mail: strauss@ibr.cs.tu-bs.de"
    DESCRIPTION
        "Experimental MIB module for JAX code generator tests."
    REVISION     "200007070000Z"
    DESCRIPTION
        "Added scalar group and notification."
    REVISION     "200003300000Z"
    DESCRIPTION
        "The initial revision of this module."
    ::= { ibr 8 }

pingNotifications OBJECT IDENTIFIER ::= { pingMIB 0 }

pingObjects OBJECT IDENTIFIER ::= { pingMIB 1 }

-- Object definitions.

pingTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF PingEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
        "A list of ping destinations and their round trip times."
    ::= { pingObjects 1 }

pingEntry OBJECT-TYPE
    SYNTAX      PingEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
            "An entry containing a ping destinations and its
             round trip time."
    INDEX   { pingIndex }
    ::= { pingTable 1 }

PingEntry ::=
    SEQUENCE {
        pingIndex               Unsigned32,
        pingAddress             IpAddress,
        pingRtt                 Unsigned32
    }

pingIndex OBJECT-TYPE
    SYNTAX      Unsigned32
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
            "A unique value identifying the ping entry."
    ::= { pingEntry 1 }

pingAddress OBJECT-TYPE
    SYNTAX      IpAddress
    MAX-ACCESS  read-create
    STATUS      current
    DESCRIPTION
            "The destination IP address for the ping entry.
             Writing to an instance of this object with an
             unused pingIndex creates a new pingTable entry."
    ::= { pingEntry 2 }

pingRtt OBJECT-TYPE
    SYNTAX      Unsigned32
    UNITS       "milliseconds"
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "The measured round trip time for the ping entry.
             The value zero denotes a ping target that did
             not respond (yet)."
    ::= { pingEntry 3 }

pingStatistics OBJECT IDENTIFIER ::= { pingObjects 2 }

pingTimeout OBJECT-TYPE
    SYNTAX      Unsigned32
    UNITS       "milliseconds"
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
            "Maximum time to wait for a response."
    ::= { pingStatistics 1 }

pingAvgRtt OBJECT-TYPE
    SYNTAX      Unsigned32
    UNITS       "milliseconds"
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "The average round trip time for all ping entries."
    ::= { pingStatistics 2 }

-- Notification definitions.

pingNoResponse NOTIFICATION-TYPE
    OBJECTS     { pingRtt, pingTimeout }
    STATUS      current
    DESCRIPTION
            "This notification is generated whenever a ping
             target did not reply within the current timeout.
             The pingAddress objects contains the target address
             and the pingTimeout objects contains the timeout
             value in milli seconds."
    ::= { pingNotifications 1 }

END
```

Figure 1: TUBS-IBR-PING-MIB

4 JAX ARCHITECTURE

Since this paper focuses on AgentX entities in the sub-agent role, we will now discuss the sub-agent's functional blocks and their mapping to an object-oriented class model, which is shown in Figure 2:

- When the sub-agent starts, it first has to establish a connection to the master agent. The current AgentX specification suggests transport mappings over UNIX domain sockets and over TCP. Since Java does not support UNIX domain sockets, JAX has to use TCP. A sub-agent connection to the master agent is represented by the AgentXConnection class.
- Next, the sub-agent has to open one or more session(s) over the established connection. All subsequent AgentX operations are related to such a session. A session is represented by the AgentXSession class.
- The sub-agent has to register every region of the OID tree with the master agent, which it claims responsibility for. There are mechanisms to

register ranges of sub-identifiers and to use priorities if multiple sessions register for a common OID region. Such a registration is represented by the `AgentXRegistration` class.

- In order to handle incoming AgentX requests and responses asynchronously, JAX has an `AgentXReader` class, which represents a thread that is spawned for every opened `AgentXConnection`. This thread reads data from the AgentX master, parses it and decodes it to instances of the appropriate AgentX PDU classes shown in Figure 4. When a valid PDU has been identified it is passed to the appropriate PDU handler method of the `AgentXSession` which is identified by the header's session ID value. This handling of incoming PDUs does not interfere with the main application's thread(s).

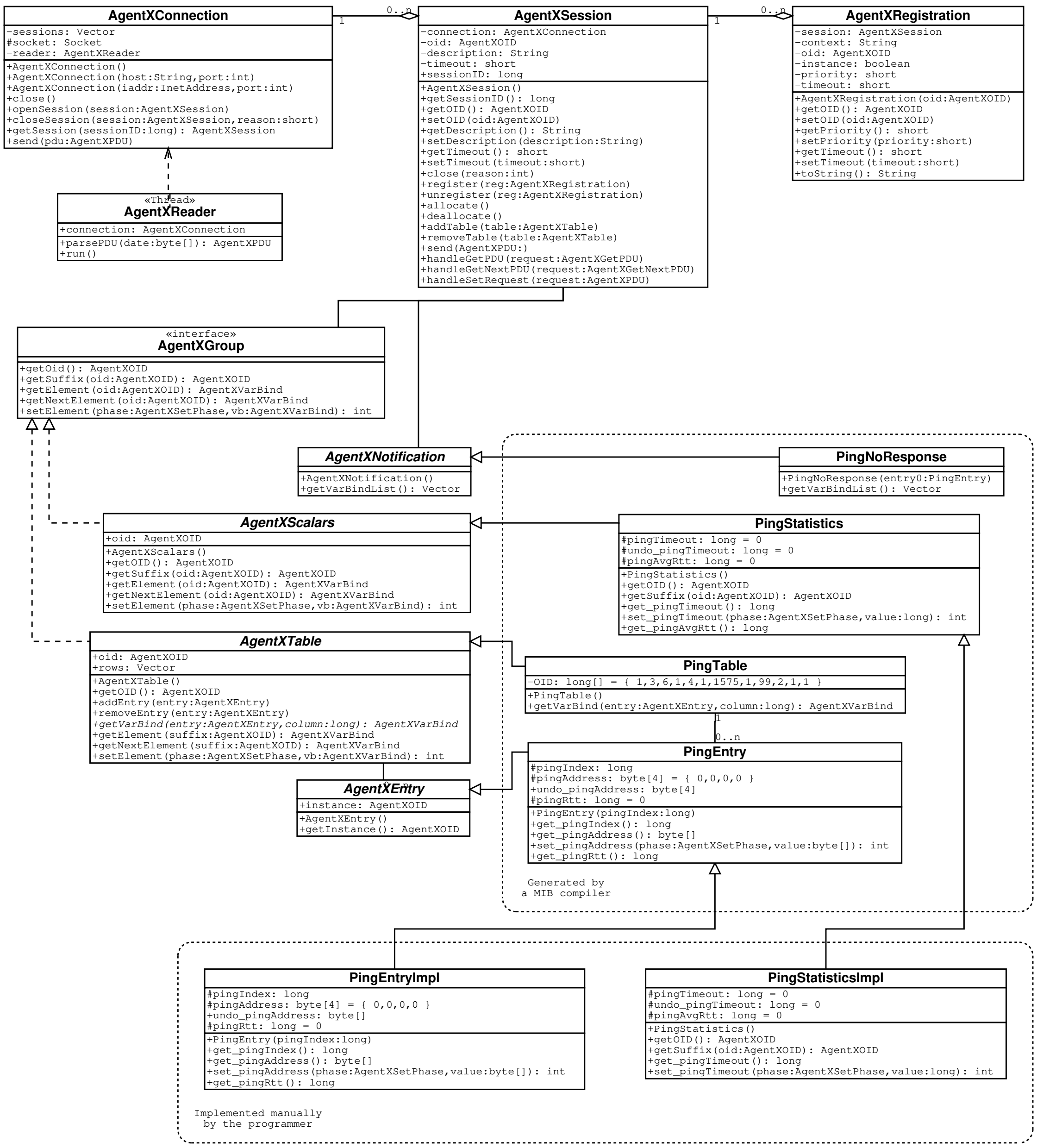

Figure 2: JAX class diagram

- The main task of a sub-agent is to implement managed objects. JAX organizes table rows of columnar objects and groups of scalar objects in classes. These classes are named by the leaf objects' common parent MIB node. This mapping into classes is equivalent to the CORBA JIDM mapping [13]. The `PingEntry` class in the upper dashed box in Figure 2 shows the grouping of the three columns of the `TUBS-IBR-PING-MIB` example. Similarly, the `PingStatistics` class shows the grouping of two scalar objects under the `pingStatistics` MIB node.

  Groups of scalar objects derived from the `AgentXScalars` class exist statically during the agent's runtime once they are created and added to the AgentX session with the `addGroup()` method. In contrast, table rows, e.g. entries like `PingEntry` instances, can be created and removed dynamically during runtime. This can be achieved by passing instances of classes derived from `AgentXEntry` to the appropriate table classes derived from `AgentXTable` using the `addEntry()` and `removeEntry()` methods.

  Specific managed object variables are represented by `get_()`- and `set_()`-methods of their table entry and scalar group classes. These methods can implement simple read/write access to variables in the instrumented code or even more complex operations according to their MIB definitions, e.g. row creation and deletion upon Set-requests on columnar objects of type `RowStatus` [14].

- Finally, a sub-agent can initiate asynchronous notifications: First, an instance of the MIB specific sub-class of `AgentXNotification` has to be created, e.g. `PingNoResponse` in Figure 2. This object is implicitly filled with the notification's mandatory var-binds. Further var-binds can be added optionally. Finally, the notification can be sent to the master agent via the `notify()` method of the AgentX session.

Note that all classes in Figure 2 are contained in the JAX class package, except those MIB specific classes in dashed boxes. In fact, even the classes in the upper dashed box are usually not written manually by the programmer, since they can be generated by a MIB compiler, which is described in the following section. The sub-agent programmer only has to manually override the actual MIB variable retrieval and set methods in a class inherited from the corresponding entry or scalar group class, e.g. the `PingEntryImpl` class in the lower dashed box in Figure 2. The usual process of sub-agent development is shown in Figure 3.

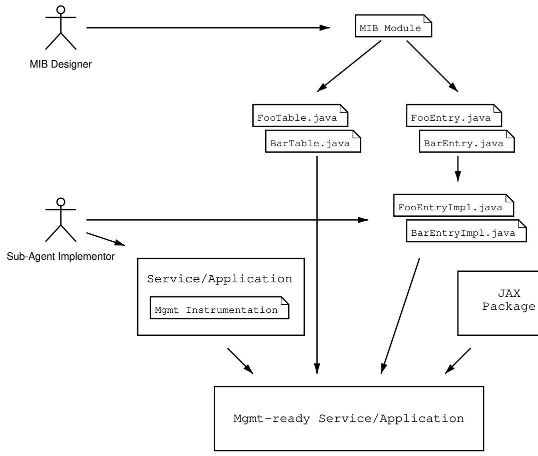

Figure 3: JAX sub-agent development process

Besides the classes described above, there are a few utility classes that are used by several other components. Some of them are shown in Figure 4.

- All of the 18 AgentX PDU classes are inherited from the generic `AgentXPDU` class.
- The `AgentXOID` class represents a numerical object identifier. It contains methods for modifying and comparing OIDs.
- The `AgentXVarBind` class represents a VarBind. Vectors of this class are used to represent VarBind lists.

While the class model describes the principal elements of the JAX architecture and their associations, it is also important to understand the dynamic behavior during runtime. Two of the most common activities in the sub-agent are shown in the sequence diagram of Figure 5.

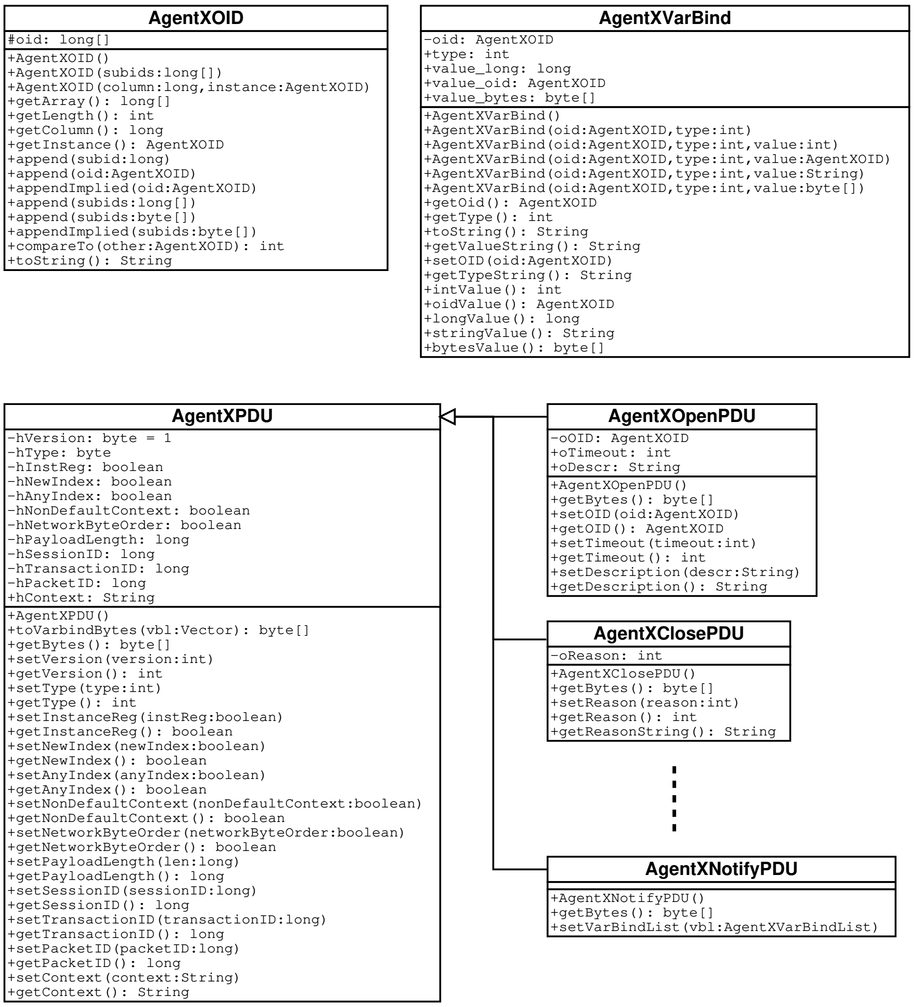

Figure 4: Some JAX utility classes

The top half documents the creation and registration of the relevant sub-agent's classes in the JAX toolkit: First, the application creates a new connection, which in turn spawns a new reader thread. Afterwards, a session object has to be created and opened by calling the connection's `openSession()` method. This method blocks until the reader receives the corresponding response from the master agent. Next, the application creates and registers the implemented MIB tables (`addTable()`) and their entries (`addEntry()`).

The bottom half of Figure 5 shows the processing of an incoming AgentX GetPDU: When the reader thread has successfully parsed an incoming PDU, it calls its session's `getSession()` method to retrieve the right session object by the PDU's session-ID header element. Now, the reader can call the session's `handleGetPDU()` method. Then, the session walks through the PDU's var-bind list. For every OID it determines the appropriate table which must have been registered before by `addTable()`. For these OIDs it calls their tables' `getElement()` methods which identify the requested column and instance identifier. For these exact var-binds the local `getVarBind()` method is called, which finally retrieves the particular value from the corresponding entry's `get_()`-method. The retrieved value is passed back to the table object, which builds a var-bind and passes it back to the session. When the session has called `getElement()` for every OID of the GetPDU, it builds a ResponsePDU and sends it to the master agent by calling the `send()` method. Finally, it returns back to the reader, which is then ready to process the next incoming PDU.

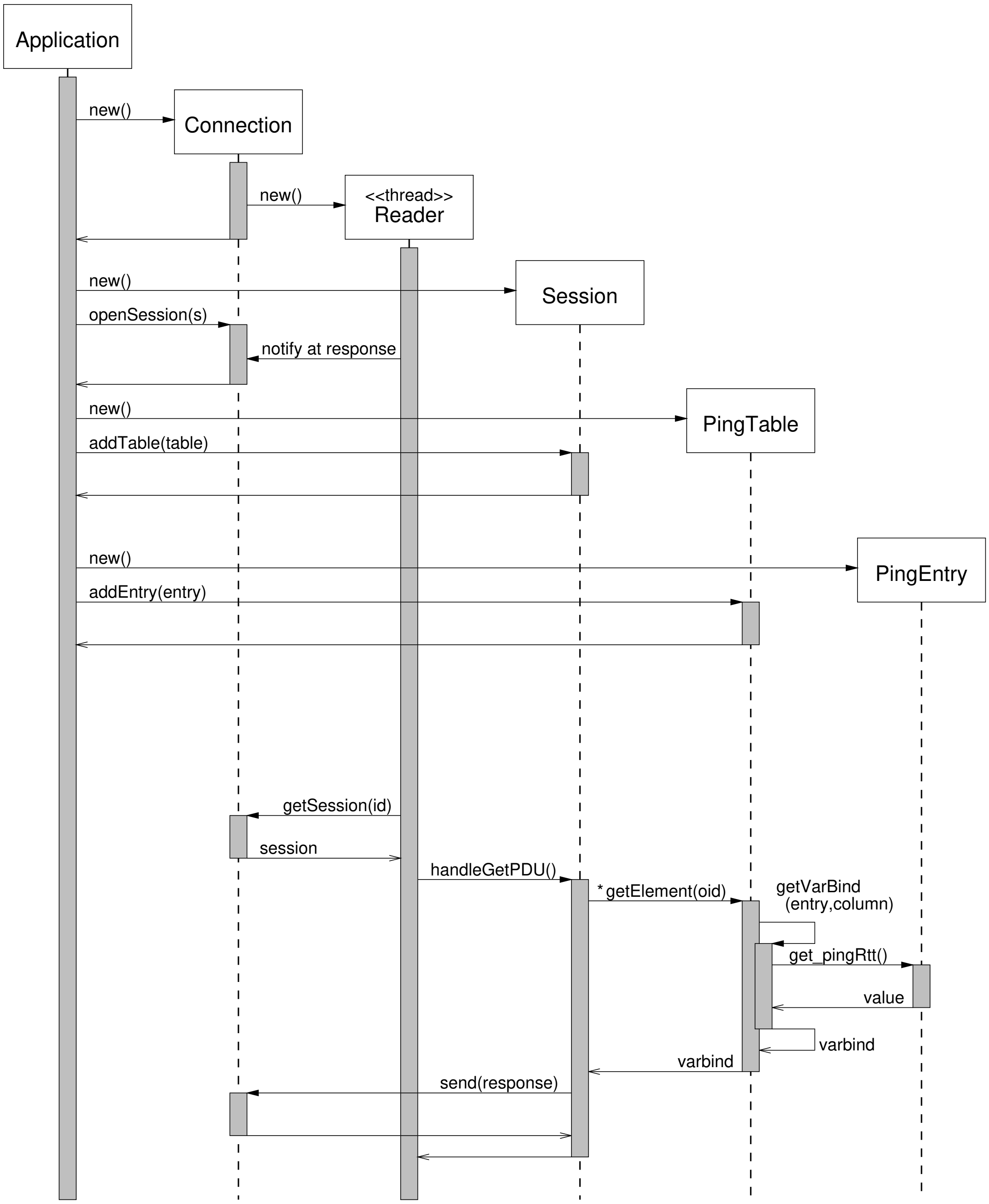

Figure 5: Sequence diagram of JAX initialization and Get-PDU processing

5 CODE GENERATION

SMI MIB specifications are used to describe the structure and semantics of SNMP MIB objects. JAX contains a MIB compiler that creates source code files of Java classes for every table and scalar group of a given MIB module like those shown in the upper dashed box in Figure 2.

Section 2 introduced a simple MIB which contains a single table with three columns, a scalar group with two scalar objects, and a notification. Thus, the MIB compiler generates six files for this module:

- `PingTable`: an `AgentXTable` derived class that represents the table itself,

- PingEntry: an AgentXEntry derived class that represents a single table row,
- PingEntryImpl: a class derived from PingEntry that is intended to be modified manually to implement the tabular objects,
- PingStatistics: an AgentXScalars derived class that represents the scalar group,
- PingStatisticsImpl: a class derived from PingStatistics that is intended to be modified manually to implement the scalar objects, and
- PingNoResponse: an AgentXNotification derived class that represents the notification.

Two of these classes are shown in Figures 6 and 7.

The PingTable class in Figure 6 is the only piece of code that contains numerical OID information of the MIB table. On creation it registers all implemented and retrievable columns. This is required for subsequent GetNext-processing. The getVarBind() method dispatches Get-requests by the column argument and calls the appropriate get_() methods of the PingEntry class. Similarly, setEntry() dispatches Set-requests and calls the PingEntry's set_() methods for the given Set phase, after some type checking on the incoming var-bind has been done.

The PingEntry class in Figure 7 represents the implementation of a PingTable row. Each instance is created by calling the constructor with an argument that specifies the table index value. Subsequently, each instance's get_() and set_() methods are called upon Get- and Set-requests: Every columnar object that has at least a read-only MAX-ACCESS value is represented by a get_() method. Similarly, there is a set_() method for every columnar object that has at least a read-write MAX-ACCESS value. While get_() methods return a value, the set_() methods get two arguments that specify the phase of the Set-request processing (Test, Commit, Undo, Cleanup) and the new value.

The JAX MIB compiler is built on top of the freely available SMI library libsmi [15, 16]. The libsmi software distribution contains the command line tool smidump that uses the SMI library as a front-end to parse MIB modules into internal data structures. Afterwards, it calls a specific builtin output driver to dump MIB contents in one of several formats. The JAX MIB compiler has been realized as a new output driver which is distributed with recent libsmi releases.

6 PERFORMANCE

One major drawback of using Java in network management environments is its limited performance in contrast to a platform's native code. In order to get the performance of JAX sub-agents compared to traditional monolithic native SNMP agent code and to AgentX sub-agents written as the agent host's native code, a simple MIB object that serves an octet string from the underlying Linux system's /proc/version file has been implemented as (a) a UCD-SNMP 4.2pre master agent MIB object, (b) a UCD-SNMP 4.2pre AgentX sub-agent talking to the UCD-SNMP master agent, and (c) a JAX AgentX sub-agent talking to the UCD-SNMP master agent.

The processing time has been measured and averaged in all three cases in the master agent's snmp_agent.c:handle_var_list() function that dispatches the var-bind list of incoming SNMP requests. Hence, the times are comparable in all three cases and are not influenced by the SNMP interaction between the manager and the master agent.

As expected, the native monolithic agent reveals the best performance with an average of 79 $\mu secs$ per request, followed by the native AgentX sub-agent with an average of 99 $\mu secs$. After all, the JAX Java implementation reaches an average of 120 $\mu secs$, which is not too bad. All measurements were performed on a 400 MHz Intel Pentium II PC under a Linux 2.2.14 kernel.

7 APPLICATIONS

This section describes two scenarios in which JAX sub-agents could be useful.

Work on JAX has been motivated by the Jasmin project [17], a joint project by NEC C&C Research Laboratories Europe and the Technical University Braunschweig. Jasmin is concerned with a DISMAN Script-MIB [18] implementation, that allows to delegate management scripts to mid-level managers. These "scripts" can be Java class files that fulfill management tasks.

The Script-MIB itself contains just one octet string object to return script results to managers. In some cases, this is not sufficient. It turned out to be useful to allow management scripts to implement arbitrary MIB objects itself. This can be achieved by making a management script act as an AgentX sub-agent entity [19].

```
/*
 * This Java file has been generated by smidump 0.2.4. Do not edit!
 * It is intended to be used within a Java AgentX sub-agent environment.
 */

/**
    This class represents a Java AgentX (JAX) implementation of
    the table pingTable defined in TUBS-IBR-PING-MIB.

    @version 1
    @author  smidump 0.2.4
    @see     AgentXTable
 */

import java.util.Vector;

import jax.AgentXOID;
import jax.AgentXVarBind;
import jax.AgentXResponsePDU;
import jax.AgentXSetPhase;
import jax.AgentXTable;
import jax.AgentXEntry;

public class PingTable extends AgentXTable
{
    // entry OID
    private final static long[] OID = {1, 3, 6, 1, 4, 1, 1575, 1, 8, 1, 1, 1};

    // constructor
    public PingTable()
    {
        oid = new AgentXOID(OID);

        // register implemented columns
        columns.addElement(new Long(2));
        columns.addElement(new Long(3));
    }

    public AgentXVarBind getVarBind(AgentXEntry entry, long column)
    {
        AgentXOID oid = new AgentXOID(getOID(), column, entry.getInstance());

        switch ((int)column) {
        case 2: // pingAddress
        {
            byte[] value = ((PingEntry)entry).get_pingAddress();
            return new AgentXVarBind(oid, AgentXVarBind.IPADDRESS, value);
        }
        case 3: // pingRtt
        {
            long value = ((PingEntry)entry).get_pingRtt();
            return new AgentXVarBind(oid, AgentXVarBind.GAUGE32, value);
        }
        }

        return null;
    }

    public int setEntry(AgentXSetPhase phase,
                        AgentXEntry entry,
                        long column,
                        AgentXVarBind vb)
    {

        switch ((int)column) {
        case 2: // pingAddress
        {
            if (vb.getType() != AgentXVarBind.IPADDRESS)
                return AgentXResponsePDU.WRONG_TYPE;
            else
                return ((PingEntry)entry).set_pingAddress(phase, vb.bytesValue());
        }
        }

        return AgentXResponsePDU.NOT_WRITABLE;
    }
}
```

Figure 6: MIB compiler output: PingTable.java

```
/*
 * This Java file has been generated by smidump 0.2.4. Do not edit!
 * It is intended to be used within a Java AgentX sub-agent environment.
 *
 */

/**
    This class represents a Java AgentX (JAX) implementation of
    the table row pingEntry defined in TUBS-IBR-PING-MIB.

    @version 1
    @author  smidump 0.2.4
    @see     AgentXTable, AgentXEntry
 */

import jax.AgentXOID;
import jax.AgentXSetPhase;
import jax.AgentXResponsePDU;
import jax.AgentXEntry;

public class PingEntry extends AgentXEntry
{
    protected long pingIndex = 0;
    protected byte[] pingAddress = new byte[4];
    protected byte[] undo_pingAddress = new byte[4];
    protected long pingRtt = 0;

    public PingEntry(long pingIndex)
    {
        this.pingIndex = pingIndex;

        instance.append(pingIndex);
    }

    public long get_pingIndex()
    {
        return pingIndex;
    }

    public byte[] get_pingAddress()
    {
        return pingAddress;
    }

    public int set_pingAddress(AgentXSetPhase phase, byte[] value)
    {
        switch (phase.getPhase()) {
        case AgentXSetPhase.TEST_SET:
            break;
        case AgentXSetPhase.COMMIT:
            undo_pingAddress = pingAddress;
            pingAddress = new byte[value.length];
            for(int i = 0; i < value.length; i++)
                pingAddress[i] = value[i];
            break;
        case AgentXSetPhase.UNDO:
            pingAddress = undo_pingAddress;
            break;
        case AgentXSetPhase.CLEANUP:
            undo_pingAddress = null;
            break;
        default:
            return AgentXResponsePDU.PROCESSING_ERROR;
        }
        return AgentXResponsePDU.NO_ERROR;
    }

    public long get_pingRtt()
    {
        return pingRtt;
    }
}
```

Figure 7: MIB compiler output: PingEntry.java

Another scenario is concerned with service daemons implemented in Java. Jigsaw [11] is the W3C's Web server platform. It is an extensible, efficient, and well-documented Web server that is purely written in Java. If there is a demand for a WWW-MIB agent for the Jigsaw Web server there are basically three possible approaches: (a) A separate SNMP agent that fetches all information from logfiles written by Jigsaw. This might still require to patch Jigsaw in order to make otherwise unavailable information accessible by the agent. Furthermore, this might suffer from bad performance and synchronization problems. (b) Jigsaw might be enhanced so that it implements a full SNMP agent. This would blow up the Web server significantly, and the potential problem of multiple concurrent SNMP agents on the same host comes up. (c) Jigsaw might be enhanced so that it implements a WWW-MIB AgentX sub-agent.

The last approach causes the least overhead on the Jigsaw server and it has the advantages mentioned in section 2. Moreover, it can be easily implemented by using the JAX toolkit. This approach has been pursued and is described in a JAX application tutorial [20]. The impact on the original Jigsaw source code was very minimal: Besides the files generated by the MIB compiler and the `Impl`-classes that needed just a few additions, eleven lines of code have been added to one existent Jigsaw Java source file and one new manually written file of less than 200 lines of code had to be adjoined.

8 RELATED WORK

AgentX is a relatively new agent extensibility protocol. The AgentX implementations available today are all based on existing SNMP toolkits. Hence, there is a strong motivation to keep the existing SNMP APIs unchanged so that existing instrumentations can be turned into AgentX sub-agents without changes to the source code. A good example is the approach taken by the UCD-SNMP project [21] to support AgentX.

The Agent++ toolkit [22] uses a C++ class structure in conjunction with a MIB compiler to simplify the implementation of SNMP MIBs. The work being done to support AgentX within Agent++ also tries to keep the existing C++ class interface unchanged.

The well known AdventNet SNMP toolkit [23], which is widely used to implement SNMP software in Java, contains a MIB compiler which generates Java beans that interact with the AdventNet SNMP engine via an event mechanism. AdventNet currently does not support the AgentX protocol. Therefore, instrumented applications are bound to the AdventNet SNMP agent.

The Java Management Extensions (JMX) specification [24] also uses Java beans to separate the instrumentation from the agent. JMX is more ambitious than the AdventNet approach since it tries to support instrumentations without regards as to how the communication with the manager is realized (e.g. SNMP, HTTP or IIOP). JMX uses so called Java MBeans to separate the instrumentation from the agent. The agent itself consists of an MBean server and one or more protocol adaptors. The JMX architecture is thus very extensible. However, there is currently no specification of an AgentX adaptor for JMX.

The JAX approach described in this paper differs from all these approaches. First of all, JAX is one of the first Java packages for building pure AgentX sub-agents. Second, JAX was designed to be small and simple. The motivation here is to keep the overhead for management instrumentation as small as possible. This is important for Java programs that are dynamically delegated to remote execution environments. Finally, JAX is derived from the AgentX specification and does not carry any SNMP baggage with it. This makes it easier for Java programmers who are not SNMP experts to learn and use JAX.

Besides JAX, there is the J.AgentX package being developed at the University of Coimbra, Portugal [25]. J.AgentX consists of two parts, an extensible SNMP agent and an AgentX API for development of AgentX sub-agents, both written in Java. While JAX contains a MIB compiler that allows to hide MIB details like object OIDs and underlying AgentX types from the sub-agent developer, the user of the J.AgentX API has to fiddle with this information manually, e.g. in AgentX registration methods and VarBind handling. Furthermore, higher level handling of table row creation and deletion and support for sending notifications are not (yet) present in J.AgentX.

9 CONCLUSIONS

The growing number of applications and services implemented in Java and the rising acceptance of Java management solutions in general increases the demand for Java based sub-agent solutions.

JAX is a Java toolkit that allows the rapid development of AgentX sub-agents based on a core AgentX class package and classes generated by a MIB compiler. A simple example illustrated the sub-agent development process and the MIB specific Java code. Two use cases demonstrated the application of JAX in practical scenarios. A comparison with other AgentX and Java agent toolkits revealed that there is high potential in the application of the approach presented in this paper.

ACKNOWLEDGEMENTS

JAX has its roots in the joint Jasmin project by NEC C&C Research Laboratories Europe and the Technical University Braunschweig. The authors of this paper thank NEC CCRLE, Berlin for the effective cooperation.

REFERENCES

[1] M. Daniele, B. Wijnen, M. Ellison, and D. Francisco. Agent Extensibility (AgentX) Protocol Version 1, RFC 2741, January 2000.

[2] JAX - Java AgentX Client Toolkit. WWW Page. http://www.ibr.cs.tu-bs.de/projects/jasmin/jax.html.

[3] M. Rose. SNMP MUX Protocol and MIB, RFC 1227, May 1991.

[4] G. Carpenter and B. Wijnen. SNMP-DPI Simple Network Management Protocol Distributed Program Interface, RFC 1228, May 1991.

[5] B. Wijnen, G. Carpenter, K. Curran, A. Sehgal, and G. Waters. Simple Network Management Protocol Distributed Program Interface Version 2.0, RFC 1592, March 1994.

[6] The Simple Times, Volume 4, Number 2 on Agent Extensibility, April 1996. http://www.simple-times.org/pub/simple-times/issues/4-2.html.

[7] IETF SNMP Agent Extensibility Working Group (agentx). WWW Page. http://www.ietf.org/html.charters/agentx-charter.html.

[8] L. Heintz, S. Gudur, and M. Ellison. Definitions of Managed Objects for Extensible SNMP Agents, RFC 2742, January 2000.

[9] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2), RFC 1907, January 1996.

[10] H. Hazewinkel, C. Kalbfleisch, and J. Schönwälder. Definitions of Managed Objects for WWW Services, RFC 2594, May 1999.

[11] W3C. Jigsaw - The W3C's Web Server. WWW Page. http://www.w3.org/Jigsaw/.

[12] IETF AgentX Working Group (additional information). WWW Page. http://www.scguild.com/agentx/.

[13] The Open Group. Inter-Domain Management: Specification Translation & Interaction Translation. Open group technical standard, January 2000.

[14] K. McCloghrie, D. Perkins, and J. Schönwälder. Textual Conventions for SMIv2, RFC 2579, April 1999.

[15] F. Strauß. A library to access SMI MIB information. WWW Page. http://www.ibr.cs.tu-bs.de/projects/libsmi/.

[16] J. Schönwälder and F. Strauß. Next Generation Structure of Management Information for the Internet. In *Proc. 10th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, pages 93–106. Springer Verlag, October 1999.

[17] Jasmin - A Script-MIB Implementation. WWW Page. http://www.ibr.cs.tu-bs.de/projects/jasmin/.

[18] D. Levi and J. Schönwälder. Definitions of Managed Objects for the Delegation of Management Scripts, RFC 2592, May 1999.

[19] J. Schönwälder, J. Quittek, and C. Kappler. Building Distributed Management Applications with the IETF Script MIB. *IEEE Journal on Selected Areas in Communications*, 18(5), May 2000.

[20] S. Mertens. Extending Jigsaw to an AgentX subagent for the WWW-MIB. WWW Page. http://www.ibr.cs.tu-bs.de/projects/jasmin/jigsaw.html.

[21] The UCD-SNMP Home Page. WWW Page. http://ucd-snmp.ucdavis.edu/.

[22] F. Fock. AGENT++ — An Object Oriented Application Programmers Interface for Development of SNMP Agents Using C++ and SNMP++, Version 2.1. Technical report, 1998. http://www.fock.de/agent++/.

[23] AdventNet, Inc. AdventNet Agent Toolkit, Release 3.2. Technical report, 1999. http://www.adventnet.com/.

[24] Sun Microsystems, Inc. Java Management Extensions White Paper, June 1999. http://java.sun.com/products/JavaManagement/.

[25] J.AgentX. WWW Page. http://www.dei.uc.pt/agentx/.