AGENTAPI: AN API FOR THE DEVELOPMENT OF MANAGED AGENTS

Rui PedroLopes
Escola Superior de Tecnologia e de Gestão, Instituto Politécnico de Bragança, Bragança, Portugal
Email: rlopes@ipb.pt

José Luís Oliveira
Departamento de Electrónica e de Telecomunicações, Universidade de Aveiro, Aveiro, Portugal
Email: jlo@det.ua.pt

Keywords: Network Management, SNMP, Agent development

Abstract: Managed agents, namely SNMP agents, costs too much to develop, test and maintain. Although assuming simplicity since its origins, the SNMP model has several intrinsic aspects that make the development of management applications a complex task. However, there are tools available which intend to simplify this process by generating automatic code based on the management information definition. Unfortunately, these tools are usually complicated to use and require a strong background of programming experience and network management knowledge. This paper describes an API for managed agent development which also provides multiprotocol capabilities. Without changing the code, the resulting agent can be managed by SNMP, web browsers, wap browsers, CORBA or any other access method either simultaneously or individually.

1 INTRODUCTION

Research communities frequently tend to consider the development as the less noble phase of the research timetable. In this scenario, software development in particular is largely seen as a simple engineering task and typically do not capture too much attention. However, the synergy between development and the project modeling frequently set up interesting issues that are underestimate in the analysis and in the architectural phases.

SNMP products have been around for a decade and have conquered the market for network and system management. While the maturity of products and the increasing acceptance of this protocol have spread the idea that the research on this mater is more or less stable, the development of products from scratch shows that this is in fact a misconception.

The early motivations for our study were centred on the topic "distribution management and SNMP". This goal led us to the evaluation of documents and proposals for several IETF charters (like the Distributed Management charter) (DISMAN) and to propose new solutions related to the distribution of management tasks. On this trail, and since most of the proposals did not presented yet any implementation, we decided to develop several MIB modules in order to have a solid knowledge of what was under assessment. Examples of the performed work are the Schedule, Event and Expression MIB (Lopes, 2000) modules of DISMAN and the MAF-MIB defined to deal with mobile agents under SNMP (Lopes, 2001).

While some results from this research have been written and presented, we realize that a main piece of the work was taking little attention. In fact it was mainly a development feature – although crucial within the overall architecture. This paper presents the AgentAPI, that is:

- an Application Programming Interface that simplifies the development of SNMP agents;
- beyond SNMP, a multi-protocol agent builder (HTTP, RMI, CORBA);
- a persistence capable API;
- an embedded SMIv2 parser;
- an open source software, with a reasonable acceptance on this area (AgentAPI).

2 AGENT API

Usually, the development of SNMP agents is ruled by the following procedure:

1. Define or retrieve the MIB definition in SMI.
2. Generate the source code (mostly C, C++ or Java) correspondent to the MIB module through a specific MIB compiler.
3. Update the generated source code with the agent functionality (programming).
4. Compile, test and deploy the SNMP agent.

At the time of the initial development, the tools which generate Java source code where sparse and many required commercial licenses. It was not possible to find a reasonable public domain tool and the available commercial tools were simply too expensive. Moreover, typically, its functionality was not clearly defined.

The available tools generate large and monolithic code (i. e., a single source file for each SMI file, sometimes mixing similar but unrelated management concepts) which results in increasing difficulties for the programmer. On the other hand, the tools are usually dependent on the SNMP stack. The use of other protocols or access methods is not easily achieved without gateways, proxies or explicit programming.

To cope with these difficulties we decided that a specific, full featured API for the development of management agents was required. In substitution of a MIB compiler, the programmer should be able to develop an agent by extending classes and invoking methods on a software layer which provides all the common, low level aspects, such as protocol message processing, agent extension, managed object identification and others. In fact, this strategy is already commonly used in GUI and client/server APIs (HTTP servers, database access, Peer to Peer applications).

2.1 Design decisions

The fundamental design goal was that it should be easy to use. This would allow reducing the learning time and provide an excellent tool to be used by students. In other words, it should be simple and straightforward even to inexperienced programmers. Mapping specific management concepts such as ‘agent’, ‘agent object’, ‘MIB table’ and others as classes in an object oriented language reduces the gap between theoretical concepts and practical development.

The agents built around the AgentAPI should be accessible by SNMP in all the three versions: SNMPv1, SNMPv2c and SNMPv3. Because we did not want to develop the SNMP stack (there are several high quality implementations available either commercially and in public domain such as JoeSNMP, Westhawk, ModularSNMP or AdventNet SNMP stacks), we provide an interface between the AgentAPI core functionality and the SNMP stack, thus achieving version independence.

We also required that we could build sub-agents through the AgentAPI, connected to master agents by an extensibility protocol such as AgentX (Daniele, 2000). This choice would allow integrating new agents into existing SNMP agents.

Continuing the list of requirements, we also wanted direct agent access through regular Web browsers thus allowing the manager to retrieve and modify management information anytime, anywhere. Moreover, some minor changes allowed us to extend the access method to other devices, such as WAP browsers. This approach helps solving, on one hand, the restricted access to network management console by diversifying the access along a multitude of terminals, including handheld devices. On the other hand, we also aim at reducing the dependence of management operations (SNMP requests) on a central management system through the adoption of distributed management mechanisms.

This multitude of access methods requires a design that is able to support several others protocols or access methods, such as Java RMI (Remote Method Invocation) and CORBA.

Regardless of the access methods, the AgentAPI core should be able to manage all the common agent mechanisms, such as object management (ordering, creating, removing and activating managed objects) as well as allowing object persistency.

2.2 Architecture

The model is structured in two main parts: one related to the user interface and the other devoted to the agent side (Figure 1).

A broad set of protocols can be used both on the manager and on the agent side. For instance, if web-oriented interfaces are a must the system can benefit from using the HTTP protocol. To maintain compatibility with traditional network management stations any SNMP version or AgentX can be used. The shaded blocks constitute the parts of the architecture that were developed and that belong to the API.

On the agent side, regardless of the intrinsic operational differences and the access protocol, the implementations share some common features such as persistency, security and command identification and processing. These operations are grouped into a

transversal block used by all MIB modules, thus improving code reusability and organization.

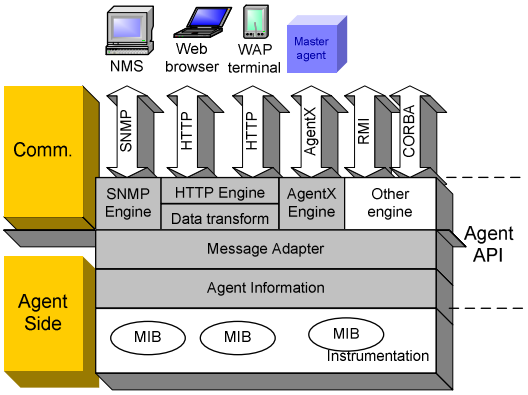

Figure 1: AgentAPI high level architecture.

The proposed model separates the API in two layers (*Message Adapter* and *Agent Information*) in order to allow different protocols and information structures.

The *Agent Information* module contains the instrumentation part of the agent, which follows the SMI description of required MIBs. This information store defines how protocol commands are mapped to platform operations, by modifying or retrieving working parameters. For example, *sysDescr* is defined as read-only and returns a "textual description of the entity". If the agent receives a command to modify this object's value it will not be allowed.

The *Message Adapter* consists of an adaptation layer that allows using different protocol stacks to access the same instrumentation information. To achieve this, the *Message Adapter* registers itself as a "protocol listener" in each communication module (HTTP engine or SNMP stack, for example).

Finally, on the top of AgentAPI, the *Communication Modules* are created according to the user requirements. For example, an agent using exclusively SNMP will only need the SNMP stack – this is the normal SNMP agent. Other agents may require other protocols, such as HTTP, SSL, RMI, CORBA and so on.

The resulting set of classes implement common managed agents' aspects. The agent is then built through the specialization of the base classes (Figure 2).

An agent in the context of the AgentAPI is a specialization of the core class (`AbstractAgent`) which uses a binary tree structure to store references to agent objects (`AgentObject`). It is also maps these references to OIDs (Object Identifiers) which provides the correspondence between managed objects and memory objects. The tree structure has higher efficiency for "walk" operations than arrays or hash tables due to the OID ordering scheme.

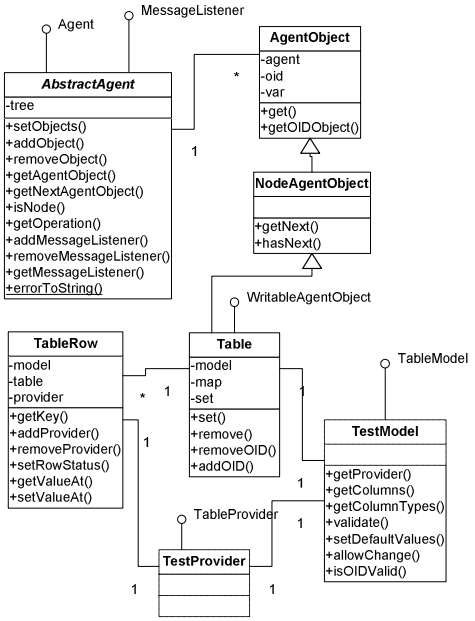

Figure 2: AgentAPI core class diagram.

The managed objects may be specializations of three kinds of classes: simple objects, node objects or SNMP tables. Every object inherits the SNMP related set of operations from `AgentObject` and introduces a new set, therefore providing the specialization required by the MIB module.

Table objects are not intended to be derived (although permitted). To drive the table behavior a `TableModel` may be used. It defines the columns type and number, validates the information and generates a repository object for each row of data (`TableProvider`).

The interface `MessageListener` is the responsible for defining the methods used in the communication between protocol engines and the agent. Any class which implements this interface can be used as a communication engine. The agent also implements the same interface, allowing bi-directional communication.

2.3 Data transformation

It was already mentioned that the AgentAPI provides the mechanisms to allow simultaneous access through several protocols, including HTTP. With a careful data transformation the HTTP client can be a regular Web browser or a WAP browser through a gateway usually residing at the Internet Service Provider (We have an online demo of HTTP

access. Try http://nms.estig.ipb.pt/see/disman to browse the agent through a common Web browser and http://nms.estig.ipb.pt/see/disman?wml=true through a WAP terminal).

The transformation implies that the structure of management information (MIB definition) inherent to the agent must be somehow transformed to HTML, WML or others. Advances in web technologies suggests mechanisms to transform XML code into custom code, including HTML, text or WML by XSL transformation (XSL). To be able to use this mechanism it is necessary to describe the structure of agent information in XML. Moreover, past IETF work focuses on defining a Document Type Definition (DTD) to allow XML parsing applications to read or edit original SMI definitions (Schoenwaelder, 2000).

We have build a communication module where the XML definition is complemented with a XSLT post-processor that dynamically generates management views in a format that is the best suited for the client's interface (). Naturally this strategy implies larger agents. However, the price is acceptable when dealing with complex agents where the overcharge is negligible and when it is important to have ubiquitous access to agents.

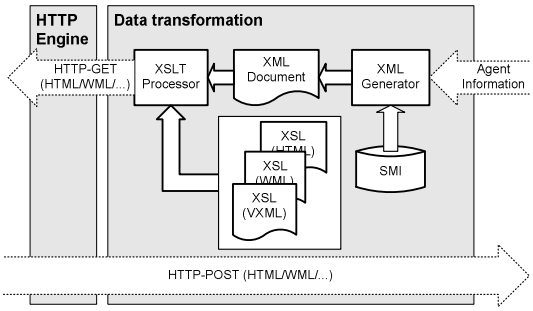

Figure 3: XSLT processor on SMI information architecture.

The communication module has as input two sources: SMI files for the agent MIB structure and the agent information for the values. This joined information is converted to XML by the XML Generator and is then forwarded to the XSLT Processor. The XSL sheets provide the guidelines for transforming the common XML input to different output documents, such as HTML, WML or VXML. The result is then sent to the embedded HTTP server.

The IETF's DTD definition, provided to represent the MIB structure in XML, does not define a tag to specify the value of MIB objects (Schoenwaelder, 2000). For the architecture described in this paper we have included a special tag, `<value>`, so that the user can have access not only to the information structure but also to the agent object value. The XML generator builds a document based on SMI definition and augments the structure with the data from the local SNMP agent.

When the user accesses the HTTP engine, a login page shows up. This provides a minimum security level through user authentication. After being successfully authenticated, the user can monitor and control (through HTML or WML pages) the state of the agent. More security may be achieved by using HTTP over SSL, a common option on regular web servers.

2.4 Extra features

The AgentAPI is complemented with a very complete and full featured SMIv2 parser which allows converting MIB description files into Java objects by including very simple code:

```
MibModule module = MibOps.load("SNMPv2-MIB");
```

To extract the information, the procedure is also straightforward:

```
MibIdentity identity = module.getIdentity();
System.out.println(identity.getOrganization());
```

This tool is independent of the other modules and may be removed if the agent does not need to interpret SMI files. It may also be used in the manager side, if it is required for some management operation. Further programming examples are available at the project web page (Agent API).

2.5 Resources

The presented architecture was implemented using the Java language with several public domain tools and utilities. For the HTTP engine we used the embeddable web server Jetty (Jetty), which includes also an HTTPS engine for higher security. The Apache Foundation contributes with two modules: the XML parser Xerces (Xerces) and the XSLT processor Xalan (Xalan), although these may be removed if the 1.4 version of the Java2 platform is used. The SNMP stack is JoeSNMP (JoeSNMP) and the AgentX implementation is JAX (JAX).

3 USAGE SCENARIOS

The following sections illustrate some programming examples, namely, a read-only agent object and a read-write agent object. After, the agent is put to practice.

The following examples include the managed objects shown in Table 1.

Table 1: Example of some managed objects.

Name	Access	Description
sysUpTime	read-only	The number of hundredths of a second since the agent was started
sysContact	read-write	A String with a contact name

3.1 Programming examples

Any managed object must derive `AgentObject` in the API. So the `SysUpTime java` file should be:

```
public class SysUpTime extends AgentObject {
  long initial;
  public SysUpTime(String oid) {
    super(oid);
    initial = System.currentTimeMillis();
  }
  public VarBind get(String o) throws MessageException {
    long now = System.currentTimeMillis();
    long sysUpTime = (now-initial)/10;
    return new VarBind(getOID(),
      new Counter(new Long(sysUpTime).toString()));
  }
}
```

`SysUpTime` has a member variable to store the time when the object is created. This object will be used to calculate the number of hundredths of a second since the object creation. `AgentObject` has an abstract method "get". This method must be overridden to return the appropriate result. In this case, (now-initial)/10.

`SysContact` is a read-write object, so, in addition to extending `AgentObject`, it must implement the `WritableAgentObject` interface:

```
public class SysContact extends AgentObject
  implements WritableAgentObject {
  Var value = Var.createVar("", Var.OCTETSTRING);
  public SysContact(String oid) {
    super(oid);
  }
  public VarBind get(String o) throws MessageException {
    return new VarBind(new String(getOID()), value);
  }
  public VarBind set(VarBind varBind)
    throws MessageException {
    Var val = varBind.getValue();
    if(val.getType()!=Var.OCTETSTRING) throw
      new MessageException(AbstractAgent.WRONG_TYPE);
    value = Var.createVar(val.toString(),
      Var.OCTETSTRING);
    return new VarBind(new String(getOID()), value);
  }
}
```

Both the methods "get", from the `AgentObject` class, and "set", from `WritableAgentObject` must be overridden. Simple type checking is done on the "set" method, throwing a `MessageException` if the value is not a `STRING`.

The agent is defined with the following code:

```
public class TestAgent extends AbstractAgent {
  public void setObjects() {
    SysUpTime a=new SysUpTime(".1.3.6.1.2.1.1.3.0");
    addObject(a);
    SysContact b=new SysContact(".1.3.6.1.2.1.1.4.0");
    addObject(b);
  }
  public static void main(String[] a) throws Exception {
    Agent agent = new TestAgent();
    EngineFactory.start(agent);
  }
}
```

Finally, to launch the agent it is necessary to compile and execute the application:

```
$> javac *.java
$> java TestAgent
```

4 TEST BED

We have validated and used the AgentAPI in some research projects related to the DISMAN work and management distribution by developing some rather complex and demanding MIBs, namely the Schedule, Expression and initial work with the Event MIB modules. Moreover, we have used it to implement a custom made MIB to manage mobile agents throw SNMP and MAF (Lopes, 2001).

The AgentAPI resulted in a valuable tool which provides common agent mechanisms such as object ordering, command processing and multi-protocol access. Each of the modules can be used independently or associated with others by using a `MultiAgent` facility:

```
MultiAgent ma = new MultiAgent();
SchedAgent sched = new SchedAgent();
MAFAgent maf = new MAFAgent();
ExpressionAgent expr = new ExpressionAgent();
ma.addAgent(sched);
ma.addAgent(maf);
ma.addAgent(expr);
EngineFactory.start(ma);
```

Each module has its own purpose and plays a different role specifically tailored for a particular situation. The Schedule MIB, for example, needs a time clock to maintain track of the scheduling operations. This feature is appended to the classes derived from the AgentAPI `AgentObject` thus specializing the general AgentAPI classes to build a specific scheduling behaviour.

The Expression MIB main characteristic is the ability to generate values according to expressions and managed objects values. It thus depends on an expression parser to calculate the functions and operations values. These values are then published in a special purpose table (Lopes, 2000).

The MAF-MIB maps MAF interfaces (MAF) to SMI thus allowing defining a gateway between SNMP and any compliant mobile agents' platforms. The agent instrumentation is performed through CORBA method invocations which interact with the agents' and platforms' life cycle. It also provides searching capabilities to locate resources in given regions.

All these agents provided a successful test bed to the AgentAPI and are available for download as well as its source code at the project web page.

4.1 Availability

Test bed results are further refined by making the API available to the Internet community. The AgentAPI (AgentAPI) is freely available under the GNU Public License (GNU) including all the tools and source code. In the year of 2002 we registered over 1000 downloads which revels a good community acceptance and forecasts reasonable usage experience. We already received strong encouragement notes by users all over the world which reported positive usage experiences.

We would like to express our gratitude to several users and colleagues that have contributed with suggestions and bug corrections which helped to increase the quality of the AgentAPI.

5 CONCLUSIONS

The development of management agents is a complex and tedious task. The community has already proposed and deployed several tools to help the developer by generating code based on the definition of the agent's managed objects. These tools are usually expensive and generate large and redundant code and are also very tied to the SNMP model.

To cope with these difficulties we have developed an open source, extensible API gathering all the common agent procedures. This API is extended to define specific agent behaviour as defined in the MIB module. Moreover, it allows using several simultaneous communication mechanisms thus allowing direct access to the agent through SNMP, RMI, CORBA, HTTP, WAP, AgentX or other existing and future protocols.

The API has already shown it usefulness in the development of SNMP agents but it still has aspects that can be improved. For future work, we are considering adding a MIB compiler to generate the basic code and thus decreasing the development time.

REFERENCES

DISMAN, Distributed Management Charter, (http://www.ietf.org/html.charters/disman-charter.html).

Lopes, R., Oliveira, J., 2000. Distributed Management: Implementation issues. In *proc. of the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet – SSGRR'2000*, l'Aquila, Rome, Italy, August 2000.

Lopes, R., Oliveira, J., 2001. SNMP Management of MASIF Platforms. In *proc. IFIP/IEEE International Symposium on Integrated Management 2001 – IM2001*, May 2001, Seattle, USA.

AgentAPI (http://nms.estig.ipb.pt/).

JoeSNMP (http://freshmeat.net/projects/joesnmp/).

Westhawk's Java SNMP stack (http://www.westhawk.co.uk/resources/snmp/index.html).

ModularSNMP (http://www.teleinfo.uqam.ca/snmp/).

AdventNet (http://www.adventnet.com/).

Daniele, M., Wijnen, B., Ellison, M., Francisco, D., 2000. Agent Extensibility (AgentX) Protocol Version 1. *Internet Request for Comments 2741*, January 2000.

XSL Transformations (XSLT), W3C Recommendation 16 November 1999 (http://www.w3.org/TR/xslt).

Schoenwaelder, J., Strauss, F., 2000. Using XML to Exchange SMI Definitions. *Internet Draft draft-irtf-nmrg-smi-xml-00.txt*, June 2000.

Jetty Java HTTP Servlet Server (http://jetty.mortbay.com/).

The XML parser Xerces (http://xml.apache.org/xerces-j/index.html).

The XSLT processor (http://xml.apache.org/xalan-j/index.html).

JAX - Java AgentX Client Toolkit (http://www.ibr.cs.tu-bs.de/projects/jasmin/jax.html).

MAF. Mobile Agent Facility Specification, Object Management Group, 00-01-02.pdf (ftp://ftp.omg.org/pub/docs/formal/00-01-02.pdf).

GNU Public License (http://www.gnu.org).