An extensible Agent Toolkit for Device Management

R. State, O. Festor, B. Zores
LORIA-INRIA Lorraine
615 rue du Jardin Botanique
54602 Villers-lès-Nancy
France

Abstract
The advent of multiple connected and interoperating devices is becoming almost common place in modern lifestyle. The management of these devices is an important asset for providing acceptable services to end users. Novel approaches are being developed that respect the communication, processing and power capacities of these limited devices. One of these approaches is SyncML Device Management. In this paper, we present an open-source agent toolkit built around the SyncML model and evaluate the performance and cost of this approach in the context of limited devices supposed to host management agents.

Keywords
SyncML, Device Management, Agent Toolkit

1. Introduction

The recent proliferation of wireless devices, network enabled personal digital assistants, enhanced mobile phones tightly integrated in the digital environment of nowadays users demands the design of new management approaches and tools particularly suited for it. Wireless transport technology requires the efficient usage of bandwidth and additional reliability for the management plane, while limited processing and storage capacities in these devices put constraints on the processors and footprint dedicated to the management agents. Multiple services and applications provided over the air to end-users also require new solutions for dynamically extending the management agent. The typical scenario is the over the air service provisioning, where a management agent co-located on the device is responsible for the management of this device. A new service is purchased and delivered over the air to the device. If management of this service is required, then somehow the management agent should be able to extend itself in order to cover the management of this new service.
We describe in this paper a developed agent toolkit for device management. Our
approach is in line with the recently proposed SyncML device management framework. To present the realized infrastructure, the paper is organized as follows: we start in section 2 with a brief introduction to the SyncML device management standard. We continue in section 3 with particular motivations for the design of a SyncML DM toolkit. Section 4 gives a high level view of the agent functionality and describes some details of the software implementation. Some experimental benchmarking results are given in section 5. Related work is addressed in section 6, while section 7 concludes the paper with pointers to future work.

2. SyncML Device Management

The SyncML initiative [1] is a broadly supported effort towards universal data synchronization. Many device manufacturers and service providers have jointly proposed a standardized framework for device synchronization. In order to avoid multiple closed data formats and incompatible synchronization formats, a common data model (based on XML) and a synchronization protocol have been proposed by this forum.

A distinct effort in this forum addresses the issue of device management. Based on the synchronization framework developed by the SyncML forum, a new device management framework emerged. The SyncML DM (Device Management) standard comprises several building blocks:

- A Representation protocol [2], [3] specifying the structure and syntax of the management protocol and the management information. This is the core component on which all other SyncML components rely. The representation protocol comprises commands (functionality of the type get/set/add/delete) used by the management protocol as well as language structures to describe status of issued commands.
- A Device management framework [5], [6] including a language to describe management information (a logical equivalent of the SMI used in SNMP), a set of standardized management objects (to be supported by any device) and the specification of the operations to be performed on the management tree.
- Finally, a management protocol [4] is proposed, specifying the different states and protocol transitions among them. The management protocol consists in a series of package exchanges (5 possible packages). A message exchange takes part between a management server and a device. A message corresponds to a SyncML encoded character string (binary encodings might be used for efficiency sake) and contains several commands and/or results. The management protocol is formed by a series of package exchanges, where each package might consists in one or several messages.
- Additional status information is included in every message. This information is related to previously received management requests. They are used to notify the initiator about the execution status of these requests. There is a communication overhead in the management protocol, justified by the target
deployment environment (wireless and/or unreliable links).

Figure 1 provides a global image on the SyncML Device Management Framework.

3. Motivations for a SyncML agent toolkit

The design and development of a SyncML DM agent architecture is motivated by several challenges. Currently, no open-source or freely available frameworks exist. An open-source implementation for the synchronization protocol [11] exists, however, the device management framework is not addressed in that work. Probably, most vendors do have proprietary in-house made toolkits, but these are not available to the academic or public research. We consider that the existence of a freely available toolkit will benefit to the management community for comparative studies and evaluation testbeds, as well as to the SyncML technology itself. From a technical
point of view, the major difficulty in conceiving an agent toolkit lies in the level of required extensibility. This extensibility comes in two flavors. At run-time, service providers can dynamically deploy services on a device and request the extension of the agent in order to manage these new services. In this case, the already running agent must be able to parse the SyncML described managed object, extend its management tree with the correct entries, install the service/device specific management code and dynamically bind the managed object to this code. This extension should be possible at run-time without requiring a new compilation of the agent. This is also one of the reasons why we use both terms agent and toolkit in our paper. A run-time extensible agent can be seen as a toolkit, the addition of a new MIB generates a new extended agent. The second extensibility is of a static nature. We consider that our agent toolkit should be easily adapted to the management of any new device. That is, device specific functionality should be added to the agent in the easiest way. What other features make a SyncML agent conceptually different from a SNMP one? The most important one is the session orientation of SyncML device management protocol. Management agents must implement a protocol engine more complex than the simple get/set/notify operations of SNMP.

Minor additional features are:
- access control. In SyncML each managed object has an access control list (ACL) property. This property indicates the allowed operations for one or several management applications.
- Process flow control. Management requests in SyncML can be executed in a predefined sequence with additional atomicity and transactional support.

4. The Toolkit/Agent architecture

4.1 Functional Architecture

This section describes the functional architecture of the SyncML Agent Toolkit. The major requirement was to develop a tool automating the design and implementation of a device specific SyncML agent. The target usage was to input the device management tree specification and to output a generic agent skeleton requiring minimal modification for a specific device customization. We soon realized that a two-faced tool can be developed by conceiving an extendible management agent. The extension potential consists in allowing dynamical run-time agent extension. Thus, a device specific agent can be obtained by starting with an empty agent skeleton and extending it with a complete device management tree specification. The obtained agent can be next deployed on a device. The second face of the tool consists in providing a SyncML agent that can be dynamically extended. This feature is interesting when new services are deployed on a device and their management has to be integrated in the device management plane.
The high level architecture of the toolkit is shown in figure 2 and is composed of an agent toolkit and a generic extendible management agent. The latter is composed of the following entities:

Transport level protocol adaptors. Currently, a TCP and respectively a HTTP adaptor are implemented. They are responsible to encode and respectively to decode SyncML requests to a particular transport technology.

The DM Protocol Engine implements the SyncML Management Protocol. Its major functionality consists in:

- processing the received SyncML messages,
- generating the outgoing SyncML messages,
- keeping track of the current management package (0 to 4),
- authenticating the management server (this is done by interacting with the Security Manager, described next).

The Security Manager performs two major tasks. Firstly, it is used to store server authentication information. Secondly it implements the access control mechanism. Each request to be performed on a managed object is verified with respect to the ACL property of the managed object.

The DM Tree Manager is the most complex component. It performs several functions. The main function is to manage the management tree. Management requests (add/replace/get) are performed by this entity. It has several interfaces. The first one is used to interact with the DM Protocol Engine. Management requests and their associated results are exchanged on this interface. The second interface goes to the Security Manager. The third interface is used to dynamically add management objects to an existing agent. This is performed through the interface going to the Extension Manager. This manager is also the link between the agent toolkit and a generic agent. It handles two types of extensions.

1. **Agent Customization Extension.** This first extension concerns the relationship between the managed object and its physical counter-part. This is done by dynamically binding operations performed on the logical managed object to native (device specific) code operations. These dynamic bindings are represented by entities called DeviceLets. Such a DeviceLet is nothing more than a mixture of native code invocation and a common API used by a service provider or device manufacturer. A device manufacturer will register a specific factory (DeviceLet Factory) with the agent. This factory implements an object creation API. Whenever a managed logical object is created, the agent requires the creation of an associated DeviceLet from the factory. Management operations performed on a logical MO are delegated to the associated DeviceLet.

2. **Agent MIB Extension.** A SyncML based representation of a managed object can be dynamically added to the agent. In the figure, this corresponds to the SyncML DM MO Box. The representation consists in the specification of the runtime properties and framework properties of a tree structure of managed objects [5]. This SyncML document is parsed and the necessary internal structure of the management tree is updated. Obviously, this
extension must be followed by an agent customization extension (previously described) in order to associate the newly created managed objects with device specific/native code.

Finally, the DM Tree browser allows to visually browse the management tree.

**Figure 2: Functional Architecture of the SyncML agent toolkit**

The DM Protocol engine is specific to SyncML device management. SyncML device management is session oriented, thus a particular protocol engine is required. A simplified functioning (for clarity sake we did not show logical flows when a session is aborted and the server-alert notification) of the protocol engine is illustrated in figure 3.

1. The start of a session corresponds to a first activity. In this activity, the agent creates a SyncML message containing: credentials (MD5 and B64 encoded username password) needed to authenticate itself. This message is delivered to the management application (server).
2. A second activity is initiated when the server replies. In this activity, the server must be authenticated and management operations requested by the server are performed.
3. In a third phase the agent will deliver the third package to the server. Results requested by the server are returned. One or several SyncML messages constitute a package. Splitting the result in several messages can be done if either the reply is larger than the maximal allowed size, or in order to reduce the waiting time at the server side.
4. In the fourth activity, additional replies from the server are treated. If no results are required the management session is done. Otherwise, the session resumes with the activity 3.
4.2. Implementation

A first release version of the Java based toolkit (GPL license – www.gnu.org/licenses/gpl.htm) can be downloaded from our web site: www.madynes.org/software.html. We will describe some of the components in the following, focusing more on the extensibility property of the agent/toolkit.

Figure 4 illustrates some of the major classes as well as the relationships among them. For the sake of clarity we do not present the operations and the attributes for these classes.

The class SyncMLAgent represents a SyncML agent. Management operations are performed by this agent using the SyncMLToolkit class. For instance, an instantiated toolkit contains the necessary information about the managed objects (represented by the class DMTreeNode) and the root of the tree (represented by the DMTree class). A managed object is represented by the class DMTreeNode. This class contains all runtime properties (like for instance the ACL – represented by the ACL class) and the framework properties (see the class AccessType representing the possible access types) and additional attributes of the class DMTreeNode. We decided to use our own internal format to represent a Managed Object and not to rely on the XML-JDOM [14] standard tree for efficiency reasons: small memory footprint forces us to use SAX parsing by the agent when new SyncML device management information is
One should note that our agent uses an internal tree structure to represent the managed objects. The nodes of this internal tree are logically equivalent to the managed objects. All management request performed by the agent are done on the logical object. Each object delegates a management request to a delegate object (represented by an object of the ImplAction class). The mapping between a logical object and its delegate can be created at run-time. The effective and device specific management is done by an object of this (or a subclass of) ImplAction. Run-time mapping is done at managed object creation time.

Whenever a new node is added to the management tree, this node is associated with an object of the type ImplAction. Since these implementation objects are device specific, each device manufacturer or service provider must develop a DMFactory object. An object of this type is responsible to create the necessary ImplAction objects. The toolkit provides a generic DMFactory class (see the class GenericDMFactory) as well as a generic ImplAction class (see the class GenericImplAction).

The above mentioned DeviceLets are in fact implemented by objects having as a superclass the ImplAction class.

Our agent can be seen both at an extendible SyncML agent and as a toolkit. The device management protocol engine is common for every agent. Device specific agents can be obtained by dynamically adding managed objects (using SyncML management device framework compliant specifications) and the device specific code (ImplAction objects) This extendible agent can be run as it is, or serialized and deployed on a device (EEPROM burned or otherwise deployed).

Figure 4: Software architecture
An example of using our toolkit is illustrated in the next figure (see figure 5). Assuming that a device manufacturer wishes to extend our toolkit in order to build an agent for his new device, the following steps are performed:

- A device management tree is specified compliant with the SyncML device management framework methodology.
- For each managed object a subclass of `ImplAction` is responsible to interact with the device. A particular subclass will implement `ImplAction` API (get/set) with device native code (using for example the Java Native Interface – JNI)
- A subclass of the `DMFactory` is developed in order to create `ImplAction` objects on request.
- An object of this subclass is registered within the agent. In figure 5 we use the `GenericDMFactory` for this purpose.
- This registration is performed by the `SyncMLFactoryManager` class, and can be done at runtime.
- On the creation of a managed object, the `GenericDMFactory` is requested to provide an object of the class `ImplAction`. In our example, the generic object (of the class `GenericImplAction`) is created and returned.
- Whenever a Get request is performed by the agent on the managed object, this request is delegated to the `GenericImplAction` object.

![Figure 5: Extension example](image)

A screenshot from the DM Tree browser is shown in figure 6. This browser allows browsing the Device management tree of a device and analyzing both SyncML framework properties and runtime properties associated with the managed object. For instance, on the left-hand pane the management tree can be browsed, while properties of a managed object are given on the right side. In this example we are detailing the address type of an address.
Runtime properties (associated to each instantiated object) are:

1. ACL-only 127.0.0.1 can get this property. For illustration sake we use the IP address as the server identifier of the management server.
2. The format (chr) shows that we expect a string of characters as a value for this object.
3. The TStamp entry holds the last update time of this object.
4. The Size entry holds the length of the value.

The framework properties (statically associated this object) are:

1. AccessType – both get and set type of requests are possible
2. The default value is IPV4
3. The Occurrence entry indicates how many objects of this type can be instantiated. In this case, only one can.

The last entry (Value) holds the current value of this managed object (equal to IPV4). This shows that the currently in use connectivity is IPV4.

![Figure 6: DM browser screenshot]

This screenshot also shows one possible run-time extension of the agent. A new node in the management tree can be added. Runtime properties, framework properties and device specific code bindings can be attached to a node, according to the SyncML DTD.

The obtained agent can be either serialized or deployed on a device (this is the agent toolkit side of our work), or it can continue to execute, if the agent is already deployed on a device.
5. Experimental benchmarking

This section presents some experimental results about the performance of the SyncML agent. We performed this analysis with a second objective to experimentally evaluate some of the SyncML protocol design choices.

The first experiment addressed the necessary agent footprint. We fed the SyncML specifications of all standardized management objects into the agent toolkit. The resulted memory requirements were 22984 bytes.

A second experiment addressed the issue of management bandwidth required for SyncML management. The experiments conditions were as follows:

- The management application and the SyncML agent were collocated on the same machine.
- Management requests were generated randomly in batches of 10, 50, 100, 150 up to 700 operations.

We analyzed two radically different management request encapsulation. In the first one, a SyncML message contained all management requests, that is, we transported the global batch in one SyncML message. In the second series of experiments, we built a SyncML message for every management request.

The results of this series of experiments are shown in the figure 7.

![Figure 7: Experimental Analysis of the management bandwidth](image)

The nature of the results is not a surprise. It is obvious that one message containing several management requests require less bandwidth then a message per request. We were interested in the average bandwidth gain of this approach. The average bandwidth ratio is shown in the next figure (figure 8). In this figure, we plotted the average ratio of the bandwidth used in one message per request, versus one message containing all requests.
We conclude that on the average, using one message per request saves almost 60% in terms of bandwidth. SyncML is very flexible allowing both types of usage. This experiment validates (experimentally) that one of SyncML design choices (allowing several management requests per message) is a good choice.

In a second type of experiment (see figure 9) we analyzed the response time of the agent. We did not consider delays due to network latency. The management application and the SyncML agent were executed on the same machine (a multi-user relatively loaded PC server). The conditions of these experiments were the same as the ones presented in the previous example.

**Figure 8 : Overhead in different encoding strategies**

**Figure 9 : Experimental comparisons of response times**

We can observe that important response time gains are obtained when multiple
management operations are included in one SyncML message. We did these experiments based on our common sense and from conclusions about SNMP shortcomings presented in the paper [23].

6. Related work

As far as we know no other work openly described a SyncML device management agent toolkit. An open source Java based implementation for the synchronization protocol can be downloaded from [11] and a comprehensive description of this work is found [13]. The synchronization protocol and the device information model used in the synchronization framework are different from the ones used in device management. We started our work from the scratch without relying on the synchronization framework implementation. SyncML synchronization in a peer to peer network is proposed in [12] without however addressing the device management issues.

The first dynamically extensible management agent is proposed in [9], where SNMP MIBS are dynamically extended within a XML based framework. Previous work covering compile-time extendable agents is addressed in [9] with the SNMP framework as target deployment.

Over the air device provisioning is described in [8], where a framework for the management of mobile stations is provided. Most closely related to SyncML based management is the work concerning XML centered management. Among the pioneering works in this research direction are the ones described in [7] focusing in the integration of multiple management data models.

Other recent initiatives and principles for XML based network element (router) configuration can be found with the IETF [15], [16] and [17].

Another research direction addressed in [20], [19] and [18] consisted in the integration of SNMP devices in a XML based management framework. These approaches range from exporting SNMP information to XML [21] to more developed SNMP to XML gateways as described in [19], [20].

Our work is different from the above mentioned approaches in several ways. Firstly, we were not concerned with the integration of SNMP devices and the conception of gateways. We did not need them, the SyncML framework proposed both a management protocol and a data model to be used. Secondly, we focused our work towards an extendible management agent. This agent is capable to receive a MIB and the device specific native code for the management of the device. Based on these, the agent extends/shrinks dynamically at run-time. From a conceptual point of view, this is similar to the JMX M-Let [22] functionality.

7. Conclusions and future work

We presented in this paper an agent toolkit/agent architecture for SyncML based device management. Our agent architecture can be seen as a toolkit since it allows dynamical run-time extensions. This extension is performed by adding managed object
descriptions as well as device specific code (Java Native Interface based) to the agent. Device management has been the focus of research and work of the wireless community (operators, device manufacturers) and we consider that the integrated network management community should take part and provide its experience and methodology in this field. The lack of openly available code and toolkit for device management was another motivation for our work. We developed our toolkit which can be freely downloaded from our web site www.madynes.org/software.html. It is still in its early development phase, but most of its functionality is implemented. We analysed in this paper some performance issues regarding the agent architecture as well as some design choices of the SyncML protocol. These evaluations still need some additional work which we will perform in the near future.

Acknowledgement

We would like to thank Mr. Laurent Berroyer for the implementation of the Representation Protocol elements in our toolkit.

References