Sync4j Server DM Programming Guide

31/05/2005

Table of Contents

1. Introduction

The OMA DM (former SyncML Device Management) protocol specifies the message sequence and behaviors that will allow device management commands to be executed against management objects on a SyncML DM compliant device. Management objects might include configuration parameters that enable Internet connectivity, e-mail connectivity, WAP connectivity, MMS settings, and basic network configuration options allowing voice access to an operator network. Other management objects may include the Java Runtime Environment on J2ME enabled devices or any other applicable software environment where extensions of features and functionality can be added via an Over The Air upgrade to those environments. SyncML DM protocol is not limited to any particular set of management objects that can be modified via OTA, although the protocol does define a specific methodology and object management tree structure that serves as a profile on how a DM server accesses specific management objects on a particular device.

Sync4j DM Server is a server side implementation of the OMA DM protocol and an extensible framework for the development of device management based applications. The Sync4j DM Server architecture and implementation derives from the Sync4j OMA DS platform (http://www.sync4j.org).

1.1. Purpose

The purpose of this document is to provide to a developer audience the basic concepts and guidance in order to be able to extends the Sync4j DM Server with new functionalities or integrate it with external applications.
With the information in this document, a developer will acquire the following skills:

- a basic understanding of the OMA DM protocol
- a good understanding of the overall Sync4j DM Server architecture
- ability to integrate the Sync4j DM Server with external applications
- ability to pre and post process incoming and outgoing OMA DM messages
- ability to implement new management operations

1.2. Audience

The intended audience includes any development/integration team that needs details on the internal architecture of the server and on how the base product can be extended.

1.3. Definitions, Acronyms, and Abbreviations

OMA	Open Mobile Alliance
API	Application Programming Interface
DM	Device Management

1.4. References

[1] SyncML Device Management Protocol, version 1.1.2, Open Mobile Alliance
[2] SyncML Device Management Tree and Description, version 1.1.2, Open Mobile Alliance
[3] SyncML Device Management Bootstrap, version 1.1.2, Open Mobile Alliance
[4] SyncML Notification Initiated Session, version 1.1.2, Open Mobile Alliance
[5] SyncML Device Management Security, version 1.1.2, Open Mobile Alliance
[6] SyncML Device Management Standardized Objects, version 1.1.2, Open Mobile Alliance
[7] SyncML Device Management Representation Protocol, version 1.1.2, Open Mobile Alliance
[8] Sync4j Architecture, Sync4j (http://sync4j.funambol.com/main.jsp?main=documentation)
[9] BeanShell Web Site, BeanShell, http://www.BeanShell.org
[10] BeanShell User's Guide, BeanShell, http://www.beanshell.org/manual/contents.html
[11] SyncML Data Sync Protocol, version 1.1.2. Open Mobile Alliance
[12] SyncML Representation Protocol, version 1.1, Open Mobile Alliance
[13] Apache Axis, http://ws.apache.org/axis/

2. SyncML DM

2.1. SyncML DM Protocol Message Sequence Overview

The SyncML DM Protocol is relatively simple from a messaging sequence standpoint. The message sequence is essentially broken into three parts:

1. Alert phase – used only for unsolicited management initiation from the server to the client.
2. Set up phase (authentication and device information exchange).
3. Management phase.

2.1.1. Transaction 1 (Not required if client is contacting server): Alert Phase – server to client only

SyncML DM supports the concept of unsolicited alerts via a "notification initiation alert" mechanism. This mechanism allows a management server to initiate a management session with a device, rather than solely relying on a client device to initiate a session. Some devices may be capable of listening on a particular port for alert messages; other devices may not be capable of this paradigm and need an alternate method to trigger a management session. SyncML will rely on two primary methods for delivery of unsolicited alerts:

1. WAP Push – This method will deliver the alert via a Push Initiator through a Push Proxy Gateway as defined by the WAP protocol. The SyncML server will act as a Push Initiator in this example, and will deliver the message via an SMS message. The message will have a unique application ID and the message will be routed to the device management user agent per the WAP Specification.
2. OBEX – The OBEX protocol can be used to deliver unsolicited alerts to a device via the PUT command as defined by the protocol.

2.1.2. Transaction 2 (Always required): Set Up Phase – Client to server

The set up phase consists of a request from the client and the response from the server. The initial client request of the set up phase will contain 3 primary pieces of information.

1. The first piece of information in the set up phase request from the client will be the information contained in the DevInfo (Device Information) object. The information the device info object represents is the following:

   Ext - An optional, internal object, marking up the single branch of the DevInfo sub tree into which extensions can be added, permanently or dynamically.
   Bearer - An optional, internal object, marking up a branch of the DevInfo sub tree into which items related to the bearer (CDMA, etc.) are stored. Use of this sub tree can be mandated by other standards.
   DevId - A unique identifier for the device. SHOULD be globally unique.

Man - The manufacturer identifier.
Mod - A model identifier (manufacturer specified string).
DmV - A SyncML device management client version identifier (manufacturer specified string).
Lang - The current language setting of the device.

2. The second piece of information that is contained in the client request message is client credentials information used for authenticating the client.
3. The third piece of information is a token that informs the server if this is a client initiated session or server initiated session. This information is required so the server can synchronize a server-initiated session with an initial incoming request from the client. From the server perspective, server initiated sessions will look the same as client initiated sessions and a token must be present so the server can distinguish both types of transactions.

2.1.3. Transaction 3 (Always required): Set-up phase server to client

The server will respond to the initial client request with server credentials, so as to identify the server to the client for authentication and identification purposes. The server may also send user interaction commands with the response, as well as initial management data.

2.1.4. Transaction 4 (Only required if management data or user interaction commands were sent in the previous message) - Management Phase – client to server

The client will respond to the server with the results of the management message sent in the previous transaction, as well as any user interaction command results.

2.1.5. Transaction 5 (Always required if transaction 4 was initiated) – Management Phase server to client.

This transaction will occur to either close the management session or to begin a new iteration if more management operations are needed. If additional management operations are needed the response to this message will be the same transaction type as transaction 4. This iteration will continue until the management server sends a message in this transaction to close the session with the client.
The diagram of Figure 1 is a representation of each transaction described in the above sections.

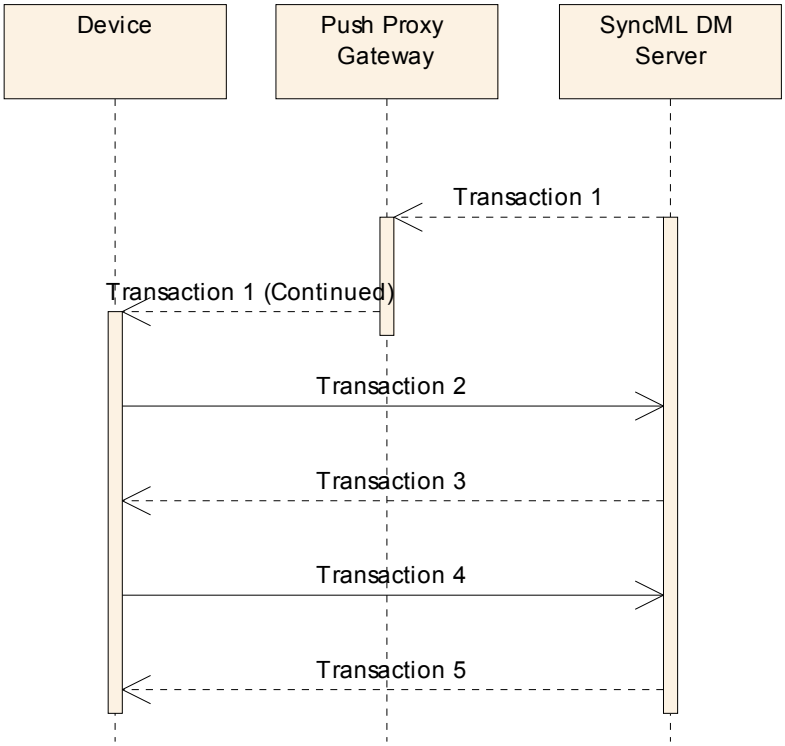

Figure 1 - Transaction model diagram reference

2.2. SyncML Device Management Tree Overview

The SyncML DM protocol identifies various messages and message content, sequence of messages, security framework etc. The device itself must adhere to a specific methodology of managing various functions. Because features and functionality is device specific and often proprietary, a framework defining how a device is to utilize device management messages must be specified and in place to operate properly with a device management server. This framework will allow a device manufacturer to add new devices or functionality to the market, then modify or add a new device description to the device management server's library of device profiles. This framework is defined as the "Device Management Tree". The tree data structure allows URI addressing of SyncML DM messages as well as provides a common framework for device management object addressing.

The device management tree is a data structure of manageable device objects. Device Objects can be anything from a single parameter to a splash screen GIF file to an entire application. The device management tree is essentially mapped to permanent or dynamic objects as an addressing schema to manipulate these objects. Permanent objects can be thought of as objects that are built into the device at the time of manufacture and typically cannot be deleted e.g. the Device Info object that defines the basic information about a device such as manufacturer or model number. Dynamic objects are objects that can be added or deleted e.g. ring tones or wallpaper.

2.2.1. The Device Management Tree ./DevInfo Node

As mentioned previously, the initial request from the client will always contain information retrieved from ./DevInfo (or Device Info) sub tree. The ./DevInfo Node is only part of the overall device management tree structure, and it maps to basic device parameters that will allow initial operations and inspection of the device by the CRM specialist. Figure 2 is an example of this Node and is illustrative on how the device management tree maps to certain objects.

2.2.1.1. Properties of each Management Tree Object

Each management object will have a set of properties associated with the object. These properties define metadata information about an object to allow things such as access control etc.

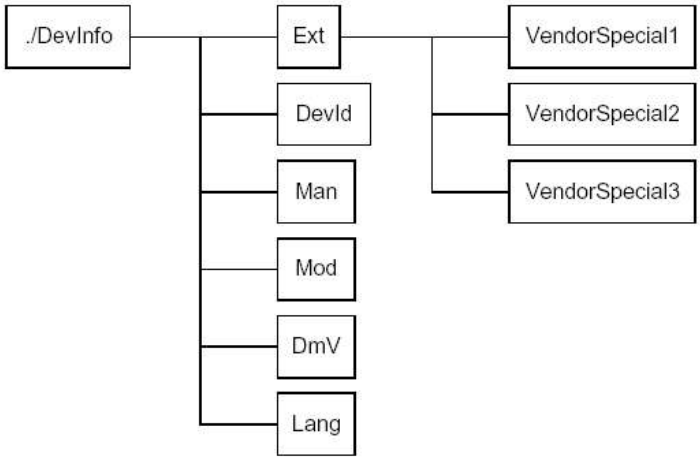

Figure 2 - Example of DM tree object mapping

The properties are:

- **ACL –** Access Control List: This is a REQUIRED property that defines who can manipulate the underlying object.
- **Format –** REQUIRED: how an object should be interpreted i.e. if the underlying object is a URL for a particular management server the Format may be defined as chr (character).
- **Name –** REQUIRED: f the object in the tree.
- **Size –** REQUIRED for Leaf objects, Size is not applicable for interior Nodes. The size of the object in bytes.
- **Title –** OPTIONAL: User-friendly name.
- **Tstamp –** OPTIONAL: The time stamp of the last modification.
- **Type –** REQUIRED for Leaf Objects, OPTIONAL for Interior Nodes. The MIME type of the object.
- **VerNo –** OPTIONAL: The version Number of the object.

2.2.2. Management Objects Manipulation

Management objects can be manipulated via SyncML messages with the following commands through a valid SyncML DM message.

- **Add –** Add an object (Node) to a tree.
- **Get –** Returns a Node name based on the URI passed with the GET request
- **Replace –** Replaces an Object on the Tree.
- **Delete –** Deletes an Object on the tree.
- **Copy –** Copies an Object (Node) on the tree.

2.2.3. Management Objects Security

As mentioned earlier, the ACL property defines the security framework for objects within a tree. This framework will allow only certain server's access to objects for manipulation. This will allow tight

control on how objects are added, changed, deleted or replaced, as well as how object properties are manipulated, and more importantly *who* is allowed to manipulate objects.

2.3. SyncML Security and Initial Provisioning (Bootstrap)

2.3.1. Security

Security is a primary concern when modifying any attributes on a device. SyncML DM protocol specifies that authentication take place in either the transport level or the SyncML DM protocol level. If the transport level authentication is considered too weak, then authentication must occur at the protocol level.

Example 1 – Transport Level Authentication: A device may authenticate itself to a WAP server using basic HTTP authentication. Authentication credentials accompany each request after the initial transaction is sent to the WAP gateway. The WAP gateway in this case would be considered “trusted” since it serves as a Proxy to the SyncML DM server, and additional authentication may not be required at the SyncML protocol level if the requests come via the trusted proxy.

Example 2 – Session Level Authentication: Similar to example 1, this example assumes that a GPRS device authenticates to the operator’s portal via TLS or HTTPS. The underlying session is established and considered authenticated therefore any messages that are a part of this secured session can be considered authenticated.

Example 3 – If session level or transport level authentication is not available or considered weak, then the SyncML protocol level authentication must occur. SyncML requires that regardless what the underlying security mechanism that is in place, if the server or client requests credentials one or both must comply.
The 4 basic credentials are:
1: Server ID
2: Username
3: Password
4: Nonce

SyncML DM requires that Basic, MD-5 (server side) and HMAC authentication must be supported.

2.3.2. Bootstrap Provisioning

SyncML DM defines 2 different use cases of bootstrapping a device and two methods for initial Bootstrapping.

2.3.2.1. Bootstrap Use cases

- Factory Bootstrap: Devices are loaded with SyncML DM bootstrap information at the time of manufacture or initial distribution.
- Server Initiated Bootstrap: Server initiated bootstrap is intended for devices that do not have the necessary configuration parameters set to establish a SyncML DM session.

2.3.3. Bootstrap Methods

2.3.3.1. WAP Profile Provisioning

Other aspects of *server initiated bootstrap* are very similar if not identical to *WAP bootstrap provisioning*. If the device supports WAP Provisioning, extensions to the WAP profile that define how SyncML parameters are mapped into the SyncML DM management object are defined in the SyncML specifications, and can be used to configure the device for SyncML DM via WAP Bootstrap provisioning.

2.3.3.2. Plain Profile Bootstrap Provisioning

Plain profile is currently defined for devices that do not support WAP Bootstrap provisioning. This method utilizes the SyncML DM format for the bootstrap message, and uses the same bootstrap method for security as WAP bootstrap provisioning.

WAP defines several methods for authenticating a bootstrap session and these methods are utilized by the SyncML DM protocol:

- NETWPIN: A shared secret is known by the device and server i.e. an IMSI or ESN. No user intervention is required, and is the simplest yet least secure method of authenticating a bootstrap message.
- USERPIN: where the user enters a PIN code delivered out of band i.e. through customer care who will initiates the bootstrap after confirming the identity of the user. A Plain Profile can use any method capable of sending unprompted requests to a device, i.e. OBEX, SMS, and WAP Push.
- USERNETWPIN: A combination of the NETWPIN and USERNETWPIN methods, requiring the use of a shared secret and a user PIN.
- USERPINMAC: The PIN is delivered out of band to the user. This method calculates the PIN based on the actual bootstrap method using a hashing function. When the bootstrap message arrives, the user is prompted to enter the PIN. If the PIN matches the re-hash of the bootstrap message on the device then the message is accepted.

3. Programming Overview

This section provides an overview of the Sync4j DM Server architecture from a point of view of a developer willing to extend the server or integrate it with other applications (i.e. A customer care front end).

3.1. System Architecture

The system architecture of the Sync4j DM Server is shown in Figure 3: the transport and the business logic (protocol handling) are separated in two distinct blocks and handled respectively by a web application running in a J2EE web container and by an Enterprise Java Bean running in a J2EE EJB container.
The web module implements the transport protocol (being OMA DM messages transported over HTTP). The EJB layer contains the real device management server implementation, which is built of many components. This represents the *management engine* of the system. Both the web layer and engine components are described in further details in the following sections.

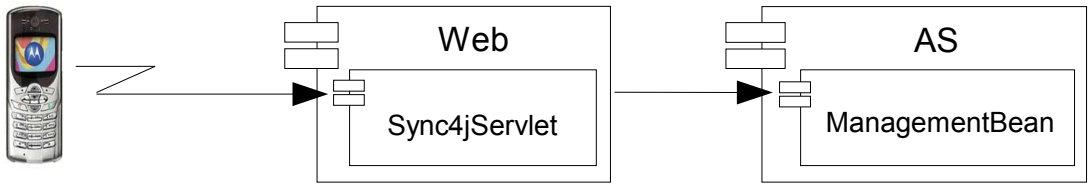

Figure 3 - Sync4j DB Server system architecture

3.2. Sync4j DM Server Architecture Overview

The Sync4j DM Server architecture is layered and modular (Figure 4).

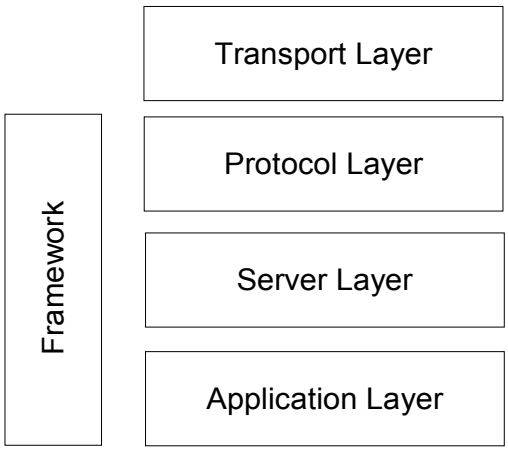

Figure 4 - Sync4j DM Server architecture overview

Layers represent groups of functionality with well defined boundaries and communication interfaces. They are:

- Transport layer (i.e. HTTP)
- Protocol layer (i.e. SyncML)
- Server layer (i.e. Sync4j DM Server)
- Application layer (i.e. customer care front end)

The *transport layer* is the door through which client messages reach the system. The current implementation of the Sync4j DM Server implements the HTTP transport protocol and binding as defined by the HTTP binding OMA DM specification. The system is designed so that other transport protocols may be added in the future.

The *protocol layer* is responsible for the interpretation and handling of the SyncML protocol. It works at both representation and protocol levels. This layer is designed so that other device management protocols may be added in the future.

The *server layer* is the DM Server implementation. It is a J2EE based application that can be deployed on any J2EE compliant application server.

The *application layer* implements the way the Sync4j DM Server interacts with end user DM applications such as the CRM applications used by the customer care staff. It is not a full implemented layer, but more a framework used to extend the server in order to meet any application specific needs.

The *framework* implements and provides services and abstractions used by the different layers to implement the component they are built of. The most important services provided by the framework are:

- Core SyncML representation and protocol
- Configuration framework
- Logging framework
- SyncML DM engine framework
- Security framework
- Commonly used utilities

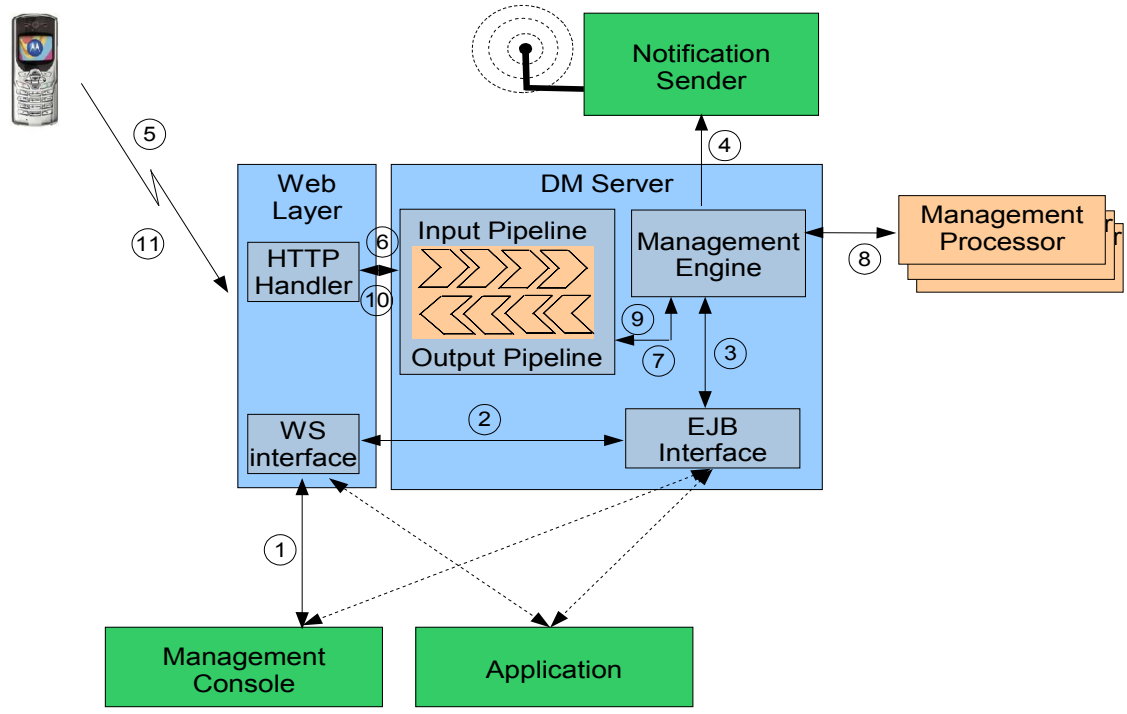

Figure 5 - Execution flow of an OMA DM request

3.3. The Execution Flow

The execution flow of an OMA DM request is shown in Figure 5.
In the picture, the blocks colored in green are external systems or applications. They interact with the Sync4j DM Server directly through its EJB interface or indirectly through a web services interface (not available yet). The Notification Sender is used by the Sync4j DM Server to send PKG 0 notifications to the devices (for server initiated management sessions).
The light blue and violet boxes represent the main Sync4j DM Server building blocks and are part of the core implementation.

Finally, the orange blocks are components added and customized by developers to meet the end-user management application needs.

As know, a OMA DM session can be started by the device by its own, or solicited by the server. In the former case the execution flow of the message is just a bit shorter than in the latter case. When the DM session is server initiated, the execution flow is the following.

1. The management application (for example the Customer Care management console) starts a new “management operation” for a specific device; this results in an interaction between the external application and the Sync4j DM Server for example by the means of a call to a web service deployed into the server:
2. the web service invokes the corresponding service of the DM Sever EJB interface;
3. the EJB wrapper forwards the call to the Management Engine, which is the core of the Sync4j DM Server;
4. the management engine builds the notification message (OMA DM PKG 0) and tells the Notification Sender to push it to the phone; note that in the picture, the Management Engine is a logical component, which in real, is built of many other blocks;
5. the device got the notification message, thus it starts a new OMA DM session sending the OMA DM PKG 1 to the server; the DM message is first received by the HTTP listener and processed by the Sync4j DM Server's HTTP handler;
6. the HTTP handler is now ready to open the DM session on the Sync4j DM Server and start the real message processing; as show in the figure, the incoming message passes through the input pipeline before getting to the Management Engine;
7. the Management Engine processes the request; this includes performing authentication and session management;
8. in order to build the management commands to send to the client, the Management Engine selects and calls the appropriate Management Processor for the management operation requested at step 1;
9. the management actions to perform are now ready to be sent to the client; the outgoing message passes through the output pipeline for post-processing;
10. the return message is then translated into an OMA DM message;
11. the return message is returned to the device.

The processing now starts again at step 5 with a new message from the client; in this case, of course, all commands and results exchanged between client and server belong to the same session until the server sends no more commands.

Note that in the case of not solicited new DM session, the client starts at step 5 without receiving any notification message.

4. Sync4j DM Server Database Schema

The database schema needed by the DM Server for internal use is shown in Figure 6. The table below describes the role of each table.

Table name	*Description*
sync4j_user	This table contains basic user information such as username, first and last name and email. An internal user represents an applicative user, which means not a real person, but an application. This field is currently not used, but it was added for future use.
sync4j_role	This table contains the list of the availables roles (for future use)
sync4j_user_role	This table contains the associations between the users and the roles (for future use)
sync4j_device	This table contains basic information about a device. In particular: • The device id • A description • The device type (i.e. Nokia 7650) • Digest (the MD5(user:password)) of the user this device will be associated • client_nonce (the nonce that the client will use to calculate the next session's digest) • server_nonce (the nonce that the server will use to calculate the next session's digest) • server_password (the server password)
sync4j_principal	A principal is an association between a user and a device. This allows more generic scenarios where a user can use many devices and/or a device can be used by many users. The current DM implementation allows an 1:1 association between a user and a device.

Table name	*Description*

sync4j_dm_state	This is the table that contains the pending operations to be performed. In particular:
	• id: record id
	• device: device id
	• mssid: session id
	• state: operation status. One of:
	  • 'N' -> notified
	  • 'P' -> management session in progress
	  • 'E' -> error
	• state_ts: timestamp of the beginning of the session
	• end_ts: timestamp of the end of the session
	• operation: operation to be performed in this session
	• info: application specific details

sync4j_id	This is a table used to create unique ids in a database independent manner. There can be many counters, each identified by its own namespace.

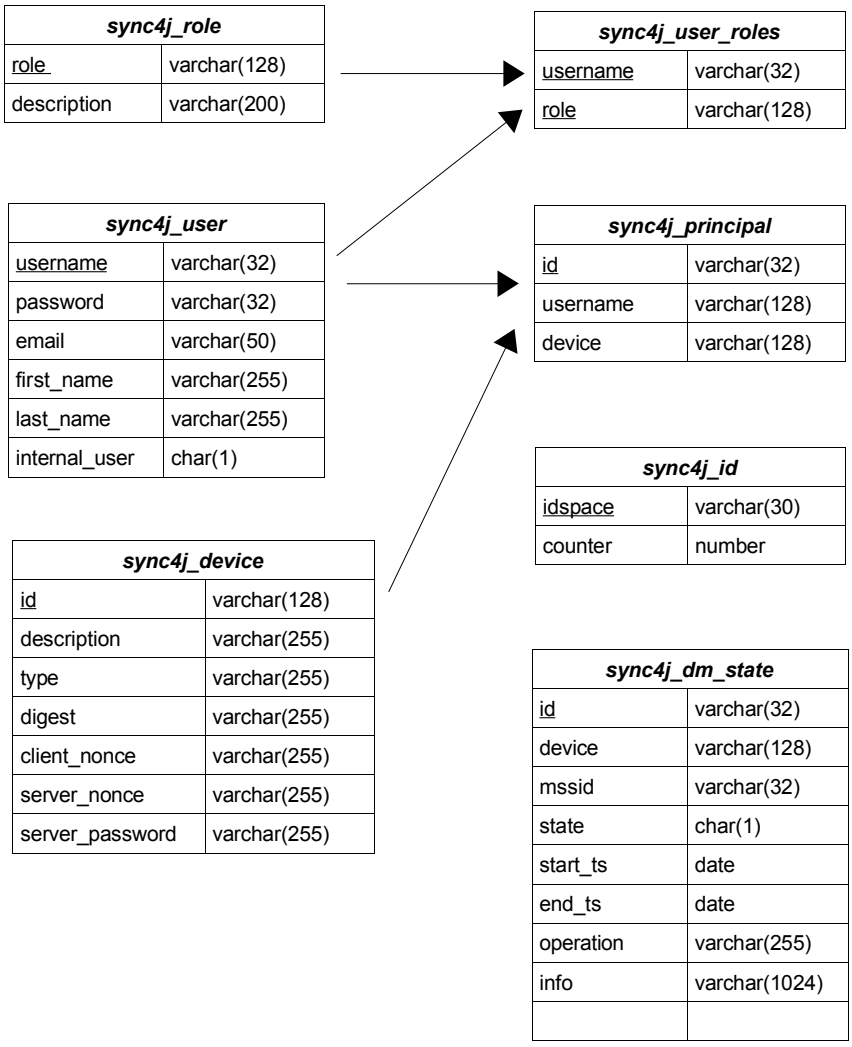

Figure 6 - Sync4j DM Server database schema

5. Sync4j DM Server Configuration Architecture

This section describes how configuration information is stored and used in the Sync4j DM Server and its extensions/customizations.

5.1. Overview

One of the goals of the Sync4j DM Server is to provide a framework that can be used to implement any kind of device management services just extending existing modules or plugging in new modules. This flexibility has a price in terms of configuration complexity and management. All this configuration information should be easily accessible and editable, avoiding complex and huge configuration files.

There are two configuration techniques used by Sync4j DM Server:

- Sync4j.properties
- *Server JavaBeans*

5.1.1. The Configuration Path

Configuration files are all stored under the so called *configuration path* (or *configpath*), which contains a tree of subdirectories and is handled in the same way of the JVM classpath. The configuration path is specified by the *sync4j.dm.home* system property, to which *config/sync4j* is appended. For example, if the system property is set to /opt/sync4j, the base directory of configuration files is /opt/sync4j/config/sync4j.

5.1.2. Sync4j.properties

This is the primary Sync4j DM Server configuration file. It is located directly under the configpath and defines the following properties:

Property	***Description***	***Default***
server.uri	The server URI that identifies the server. Sync4j will refuse all management messages addressed to a server URI different from the one indicated by this property.	http://localhost:8080
server.id	Server identifier.	sync4j
syncml.dtdversion	The supported SyncML dtd version	1.1
engine.manifacturer	The manufacturer used in server capabilities.	SyncServer

Property	*Description*	*Default*
engine.modelname	The model name used in server capabilities.	-
engine.oem	The oem used in server capabilities.	-
engine.firmwareversion	The firmware version used in server capabilities.	-
engine.softwareversion	The software version used in server capabilities.	1.4
engine.hardwareversion	The hardware version used in server capabilities.	-
engine.deviceid	The device identifier used in server capabilities.	sync4j
engine.devicetype	The device type used in server capabilities.	-
engine.strategy	The server bean (see the next section) representing a *sync4j.framework.engine.SyncStrategy* object. The given value is searched for in the configuration path first as the name of a serialized object. If no serialized object is found, the value is considered the name of a class and a new instance is created.	sync4j.server.engine.Sync4jStrategy
engine.store	The server bean representing the persistent store manager (see section on the persistent store architecture).	sync4j/server/store/PersistentStoreManager.xml
engine.handler	The server bean representing the session handler.	sync4j.server.session.ManagementSessionHandler
engine.pipeline	The server bean representing the pipeline manager (see section on the Message processing pipeline).	sync4j/framework/engine/pipeline/PipelineManager.xml
security.officier	The server bean representing the security officer.	sync4j.framework.security.DBOfficer
user.manager	The server bean representing the User Manager.	sync4j/server/admin/DBUserManager.xml
server.dm.selector	The server bean representing the processor selector (see section on the Processor Selector).	sync4j/server/dm/OperationProcessorSelector.xml
minXMLMaxMsgSize	The minimum MaxMsgSize allowed for XML messages	3000
minWBXMLMaxMsgSize	The minimum MaxMsgSize allowed for WBXML messages	2000

5.1.3. Server JavaBeans

Apart from the Sync4j.properties, all other Sync4j DM Server components are configured as *server JavaBeans*. Server JavaBeans are JavaBeans used server-side. The idea is to store a bean configuration as the serialized form of a bean instance. In this way, a bean can be instantiated,

configured and serialized to persist its configuration. Later on, the bean can be deserialized in a properly configured instance.

However, it would not be very friendly if a bean had to be instantiated, configured and serialized any time its configuration changes. To solve this problem, Sync4j DM Server makes use of the standard java facility to serialize objects into XML (and to deserialize them from XML). This is achieved by the means of the classes *java.beans.XMLEncoder* and *java.beans.XMLDecoder*. Since configuration files created with such encoder/decoder look quite friendly, they can even be created and modified by hand, without the need of a dedicated GUI, simply with a text editor. An additional advantage of this approach is that server JavaBeans are not requested to implement *java.io.Serializable* because *XMLEncoder* does not require it.
This is an example of a server JavaBean:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_01" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.server.store.PersistentStoreManager">
  <void property="jndiDataSourceName">
    <string>java:/jdbc/sync4j</string>
  </void>
  <void property="stores">
   <array class="java.lang.String" length="2">
     <void index="0">
      <string>sync4j.server.store.SyncPersistentStore</string>
     </void>
     <void index="1">
      <string>sync4j.server.store.EnginePersistentStore</string>
     </void>
   </array>
  </void>
 </object>
</java>
```

In order to help server JavaBeans handling, Sync4j provides the factory class *sync4j.framework.tools.beans.BeanFactory,* which in turn makes use of a customized class loader developed to handle configuration files in the *configpath* as usual class loaders handle classes in the class path.

The XML syntax uses the following conventions:

- Each element represents a method call.
- The "object" tag denotes an expression whose value is to be used as the argument to the enclosing element.
- The "void" tag denotes a statement which will be executed, but whose result will not be used as an argument to the enclosing method.
- Elements which contain elements use those elements as arguments, unless they have the tag: "void".
- The name of the method is denoted by the "method" attribute.
- XML's standard "id" and "idref" attributes are used to make references to previous expressions - so as to deal with circularities in the object graph.
- The "class" attribute is used to specify the target of a static method or constructor explicitly; its value being the fully qualified name of the class.
- Elements with the "void" tag are executed using the outer context as the target if no target is defined by a "class" attribute.
- Java's String class is treated specially and is written <string>Hello, world</string> where the characters of the string are converted to bytes using the UTF-8 character encoding.

Although all object graphs may be written using just these three tags, the following definitions are included so that common data structures can be expressed more concisely:

- The default method name is "new".
- A reference to a java class is written in the form <class>javax.swing.JButton</class>.
- Instances of the wrapper classes for Java's primitive types are written using the name of the primitive type as the tag. For example, an instance of the Integer class could be written: <int>123</int>. Java's reflection is internally used for the conversion between Java's primitive types and their associated "wrapper classes".
- In an element representing a nullary method whose name starts with "get", the "method" attribute is replaced with a "property" attribute whose value is given by removing the "get" prefix and decapitalizing the result.
- In an element representing a monadic method whose name starts with "set", the "method" attribute is replaced with a "property" attribute whose value is given by removing the "set" prefix and decapitalizing the result.
- In an element representing a method named "get" taking one integer argument, the "method" attribute is replaced with an "index" attribute whose value the value of the first argument.
- In an element representing a method named "set" taking two arguments, the first of which is an integer, the "method" attribute is replaced with an "index" attribute whose value the value of the first argument.
- A reference to an array is written using the "array" tag. The "class" and "length" attributes specify the sub-type of the array and its length respectively.

5.1.4. Lazy Initialization

When a bean is deserialized from its XML form, the classloader that loads the serialization file calls, first of all, the bean class's empty constructor and than it sets the bean properties values using the setXXX() methods. However, some classes need additional work to properly initialize; that work has to be done with meaningful properties values (in other words, after the setXXX() methods are called). To support this *lazy initialization* approach, those classes can implement *sync4j.framework.tools.beans.LayInitBean*, which defines a separate *init()* method. When Sync4j DM Server loads a *LazyInitBean*, after the bean instantiation (or deserialization), it calls its *init()* method, giving the bean the opportunity to complete its initialization.

5.2. How to Configure a Standard Component

Making a change to a configuration bean is as easy as editing a text file. Let's take as example the configuration file for the NotificationSender component. The configuration bean full path is sync4j/server/engine/dm/NotificationSender.xml (remember: this path is relative to the configpath) and its content is below:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_01" class="java.beans.XMLDecoder">
 <object class="sync4j.dm.engine.MySender">
  <void property="sendingUrl">
   <string>http://theserver.com/sms/send</string>
  </void>
 </object>
</java>
```

The object element specifies which Java class will be instantiated and the property element sets the corresponding instance property. Therefore, to change the sending URL (let's suppose the sender has a HTTP based interface) of the service used to send notifications, it is sufficient to edit the file, change the url and save. The next time this bean will be used, the new configuration value will be picked up.

5.3. How to Create a Custom Configurable Object

Any Java object can be configured with this technique, from a simple Java class to a very complex Java object tree. For example, this configures a String object:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <string>Hello world</string>
</java>
```

A more interesting example is the following. Let's suppose we have a “device inventory” component that is able to store some properties of a device and that can be queried to retrieve the device capabilities. The class would look like:

```
public class DeviceInventory {
    private String s1 = "s1";
    public String  s2 = "s2";
    private HashMap capabilities = new HashMap();

    public DeviceInventory() {}

    public void setCapabilities(String model, Capabilities caps) {
        capabilities.put(model, caps);
    }

    public int getMaxMsgSize(String model) {
        return ((Capabilities)capabilities.get(model)).getMaxMsgSize();
    }

    public int getMaxObjSize(String model) {
        return ((Capabilities)capabilities.get(model)).getMaxObjSize();
    }

    public boolean supportNumberOfChanges(String model) {
        return ((Capabilities)capabilities.get(model)).getSupportNumberOfChanges();
    }

    public boolean supportLargeObjects(String model) {
        return ((Capabilities)capabilities.get(model)).getSupportLargeObjects();
    }

    public void setCapabilities(HashMap capabilities) {
        this.capabilities = capabilities;
    }

    public HashMap getCapabilities() {
        return capabilities;
    }
}
```

A possible configuration file for such a class could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2_04" class="java.beans.XMLDecoder">
 <object class="sync4j.dm.examples.DeviceInventory">
  <void property="capabilities">
   <void method="put">
    <string>siemens-s55</string>
    <object class="sync4j.dm.examples.Capabilities">
     <void property="maxMsgSize">
      <int>2700</int>
     </void>
     <void property="supportLargeObjects">
      <boolean>true</boolean>
     </void>
     <void property="supportNumberOfChanges">
      <boolean>true</boolean>
     </void>
    </object>
   </void>
   <void method="put">
    <string>nokia-7650</string>
```

```
    <object class="sync4j.dm.examples.Capabilities">
     <void property="maxMsgSize">
      <int>5000</int>
     </void>
     <void property="maxObjSize">
      <int>10000</int>
     </void>
     <void property="supportLargeObjects">
      <boolean>true</boolean>
     </void>
     <void property="supportNumberOfChanges">
      <boolean>true</boolean>
     </void>
    </object>
   </void>
  </void>
 </object>
</java>
```

5.4. How to Get a Configured Instance

Configuration beans are accessed through the singleton *sync4j.framework.config.Configuration* object. For example, to instantiate a configured DeviceInventory instance, use the code below.

```
Configuration c = Configuration.getConfiguration();
DeviceInventory inventory = c.getBeanInstanceByName("sync4j/server/dm/DeviceInventory.xml");
```

5.4.1. Tips and Tricks

It is not necessary to write a configuration file by hands from scratch. To write a bean instance for the first time, use the *sync4j.framework.tools.beans.BeanFactory*'s *saveBeanInstance()* method to save a configured instance into a file. For example:

```
import sync4j.dm.examples.DeviceInventory;
import sync4j.dm.examples.Capabilities;
import sync4j.framework.tools.beans.BeanFactory;

DeviceInventory inventory = new DeviceInventory();

Capabilities nokia   = new Capabilities();
Capabilities siemens = new Capabilities();

nokia.setMaxMsgSize(5000);
nokia.setMaxObjSize(10000);
nokia.setSupportNumberOfChanges(true);
nokia.setSupportLargeObjects(true);

siemens.setMaxMsgSize(2700);
siemens.setMaxObjSize(0);
siemens.setSupportNumberOfChanges(true);
siemens.setSupportLargeObjects(false);

inventory.setCapabilities("nokia-7650", nokia);
inventory.setCapabilities("siemens-s55", siemens);

BeanFactory.saveBeanInstance(inventory, new java.io.File("inventory.xml"));
```

6. Customizing the Message Processing

This section explains how to extend Sync4j DM Server customizing the processing of incoming and outgoing messages.

6.1. Overview

The OMA DM protocol is an XML-based protocol. This means that each OMA DM message is an XML document.
When a OMA DM message reach the Sync4j DM Server, it passes through some transformations. These are divided into *XML transformations* and *message transformations*. The former works on the message in its XML representation, the latter on a Java representation of the message.

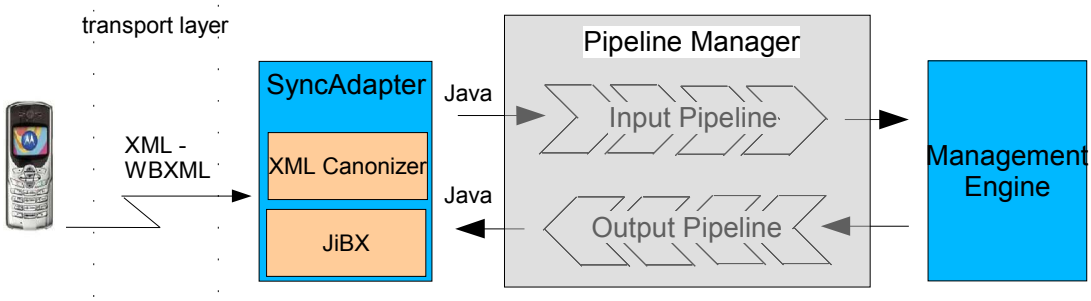

Figure 7 - Message processing architecture

In order to save bandwith and processing power, OMA DM messages can be also WBXML encoded. No matter how the message is coded, its content is first delivered to a SyncAdapter component by the transport layer (). The SyncAdapter first translates the message in XML if it was WBXML encoded and then the XML message is reduced to a “canonical” form in order to get rid of device specific singularities. XML canonization is the XML level transformation.

Even when in the canonical XML form, the message is still hard to manipulate, since XML needs to be parsed. Plus, each component that needs to access any of the OMA DM message elements would have to parse the XML again, with a big impact on performance. For these reasons, the canonic XML message is translated into an object tree that represents exactly the message.
For example, the DM PKG #1 message below will be translated in the object hierarchy of Figure 8.

```
<SyncML xmlns='SYNCML:SYNCML1.1'>
   <SyncHdr>
       <VerDTD>1.1</VerDTD>
       <VerProto>DM/1.1</VerProto>
       <SessionID>5b</SessionID>
       <MsgID>1</MsgID>
```

```
        <Target>
            <LocURI>http://localhost:8080/sync4j/dm</LocURI>
        </Target>
        <Source>
            <LocURI>Sync4jTest</LocURI>
        </Source>
        <Cred>
            <Meta>
                <Format xmlns='syncml:metinf'>b64</Format>
                <Type xmlns='syncml:metinf'>syncml:auth-basic</Type>
            </Meta>
            <Data>c3luYzRqOnN5bmM0ag==</Data>
        </Cred>
        <Meta>
            <MaxMsgSize xmlns='syncml:metinf'>20000</MaxMsgSize>
        </Meta>
    </SyncHdr>
    <SyncBody>
        <Alert>
            <CmdID>1</CmdID>
            <Data>1201</Data>
        </Alert>
        <Replace>
            <CmdID>2</CmdID>
            <Item>
                <Source>
                    <LocURI>./DevInfo/Lang</LocURI>
                </Source>
                <Data>en-us</Data>
            </Item>

            [...]

            <Item>
                <Source>
                    <LocURI>./DevInfo/DevId</LocURI>
                </Source>
                <Data>Sync4jTest</Data>
            </Item>
        </Replace>
        <Final/>
    </SyncBody>
</SyncML>
```

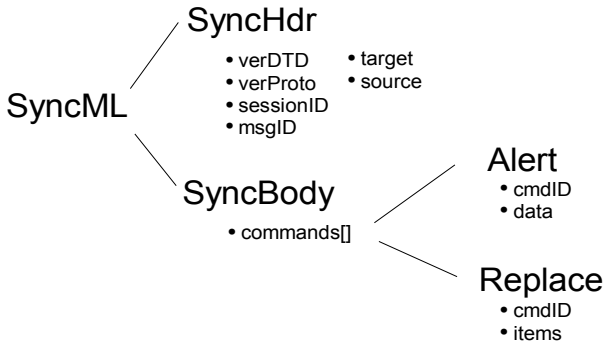

Figure 8 - sync4j.framework.core object tree example

After being translated into an object tree, an incoming message passes through the *input message processing pipeline,* before getting to the ManagementEngine. This gives the opportunity to further processing the message when it is in a manageable representation. In a similar way, a response message going out from the ManagementEngine, passes through the *output message processing pipeline* before getting translated to its XML (and then WBXML) representation.
The input and the output pipelines are completely customizable, so that custom message pre and post processing can be easily added to the system.

Input and output message processing components are also called "*synclets*".

6.2. Preprocessing an Incoming Message

To preprocess an incoming message we have to create and input processor component and to configure the pipeline manger accordingly. This is described below.

6.2.1. Creating an Input Synclet

An input synclet is a class that implements the *sync4j.framework.engine.pipeline.InputMessageProcessor* interface. This interface defines just one method: *void preProcessMessage(MessageProcessingContext context, SyncML msg). context* is a request scoped parameter that is shared amongst all the synclets (both input and output) involved in the message processing. *msg* is the object tree representing the DM message.

An example of an input synclet is the following.

```
public class MotorolaV500
implements InputMessageProcessor {
    // ------------------------------------------------- InputMessageProcessor

    public void preProcessMessage(MessageProcessingContext processingContext,
                                  SyncML                   message          )
    throws Sync4jException {
        List items, validItems;
        List results = message.getSyncBody().getCommands();
        Item item;

        AbstractCommand c;
        Results r;

        Iterator i = results.iterator();

        while (i.hasNext()) {
            c = (AbstractCommand)i.next();
            if (c instanceof Results) {
                r = (Results)c;
                validItems = new ArrayList();
                items = r.getItems();

                Iterator j = items.iterator();
                while (j.hasNext()) {
                    item = (Item)j.next();
                    if (item.getSource() != null) {
                        validItems.add(item);
                    }
                }
                List oldItems = r.getItems();
                oldItems.clear();
                oldItems.addAll(validItems);
            }
        }
    }
}
```

Scope of the synclet is to remove from the incoming message all the items with no Source element in it. In fact, the motorola V500 phone, sends sometimes erroneous <Item></Item> elements which are not allowed in SyncML. With the code above those Items will be removed before the message is actually processed by the management engine.

6.2.2. Configuring an Input Synclet

The input synclet so created, is configured telling the Pipeline Manager to insert the new synclet in the input pipeline. This is done like in the following server side JavaBenas.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.engine.pipeline.PipelineManager">
  <void property="inputProcessors">
    <array class="sync4j.framework.engine.pipeline.InputMessageProcessor" length="1">
      <void index="0">
        <object class="sync4j.dm.synclet.MotorolaV500"/>
      </void>
    </array>
  </void>
  <void property="outputProcessors">
    <array class="sync4j.framework.engine.pipeline.OutputMessageProcessor" length="0"/>
  </void>
 </object>
</java>
```

6.3. Postprocessing an Outgoing Message

To postprocess an outgoing message we have to create an output processor component and to configure the pipeline manger accordingly. This is described below.

6.3.1. Creating an Output Synclet

An output synclet is a class that implements the *sync4j.framework.engine.pipeline.OutputMessageProcessor* interface. This interface defines just one method: *void postProcessMessage(MessageProcessingContext context, SyncML msg).*
The concepts behind the output message processing are the same as per input processing.

An example of an output synclet is the class *sync4j.server.engine.RespURISynclet*. The scope of this synclet is to inject into the outgoing message the RespURI element that tells the client to which URL send the next message. The code is the following:

```
public class RespURISynclet
implements OutputMessageProcessor {
    // ------------------------------------------------------------------ Constants

    public static final String PARAM_SESSION_ID = "sid";


    // ---------------------------------------------------- OutputMessageProcessor

    public void postProcessMessage(MessageProcessingContext processingContext,
                                   SyncML                   message          )
    throws Sync4jException {
        Configuration config = Configuration.getConfiguration();

        String sessionId =
            (String)processingContext.getProperty(processingContext.PROPERTY_SESSIONID);

        if (sessionId == null) {
            if (log.isLoggable(Level.INFO)) {
                log.info(processingContext.PROPERTY_SESSIONID + " is null! Synclet ignored");
            }
            return;
        }

        String serverUri =
            config.getStringValue(ConfigurationConstants.CFG_SERVER_URI);

        message.getSyncHdr().setRespURI(
            serverUri                   +
            '?'                         +
            PARAM_SESSION_ID            +
            '='                         +
            sessionId
        );
    }
```

```
}
```

6.3.2. Configuring an Output Synclet

The output synclet so created, is configured telling the Pipeline Manager to insert the new synclet in the outut pipeline. This is done like in the following server side JavaBenas (keeping the same configuration of the input pipeline as the previous example).

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
 <object class="sync4j.framework.engine.pipeline.PipelineManager">
  <void property="inputProcessors">
    <array class="sync4j.framework.engine.pipeline.InputMessageProcessor" length="1">
      <void index="0">
        <object class="sync4j.dm.synclet.MotorolaV500"/>
      </void>
    </array>
  </void>
  <void property="outputProcessors">
    <array class="sync4j.framework.engine.pipeline.OutputMessageProcessor" length="1">
      <void index="0">
        <object class="sync4j.server.engine.RespURISynclet"/>
      </void>
    </array>
  </void>
 </object>
</java>
```

7. Implementing Management Operations

Another mechanism to extend the Sync4j DM Server is based on developing custom *management operations*. This section describes how to do it.

7.1. Overview

A *management operation* is a sequence of management commands that the server sends to the device in order to perform a higher level task. For example, in the case of the client settings provisioning, a “setBrowserSettings” operation is translated into a sequence of Get/Replace commands that will result in setting the phone browser configuration.

The Management Engine is the core component that handles device management sessions and then operations. It implements the protocol requirements (as defined in [1]) but delegates to external *management processors* the accomplishment of the management actions to perform during a management session.
When a client starts a new management session, the Management Engine selects the Management Processor to use by the means of the *Manager Selector*. The selector will make its choice based on the content of the first device information sent by the client in the SyncML DM PKG 1.

The architecture of the management engine is shown in Figure 9.
The orange colored components are the ones that can be customized. In this section we will focus on management selector and processor developments.

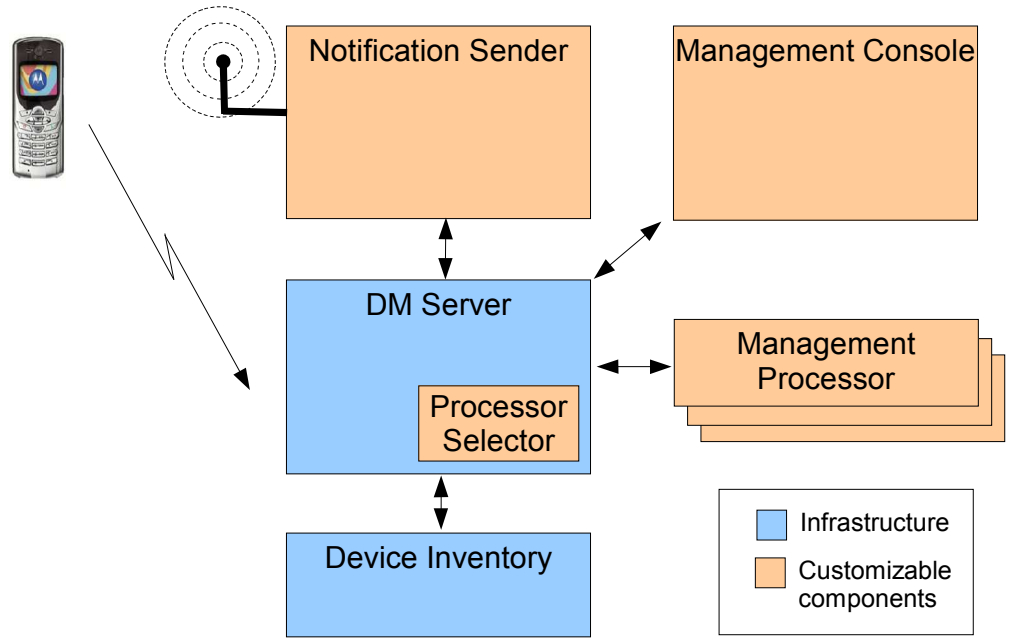

Figure 9 - Management engine architecture

7.2. Creating a Processor Selector

The Processor Selector component is represented by an implementation of the interface *sync4j.framework.server.dm.ProcessorSelector*, which is defined by the following methods:

method	***description***
ManagementProcessor getProcessor(DeviceDMSession dms, DevInfo devInfo)	Called by the Management Engine at the beginning of a management session to determine the manager that must handle the session.

Two simple ProcessorSelector implementations are provided out of the box: *DeviceIdProcessorSelector*, which associates management processors to sets of device identifiers; and *OperationProcessorSelector*, which associates a management processor to the operation stored in the device management state.

7.2.1. DeviceIdProcessorSelector

This is represented by the class *sync4j.server.dm.DeviceIdProcessorSelector*. It is configured with an array of associations <regexp>-<management_processor>, where <regexp> is a regular expression interpreted by the JDK package *java.util.regex* and used to match the device id; <management_processor> is a server side bean configuration path. If no device id matches any of the given regexp, a default processor will be returned; otherwise the first match is returned.

An example of DeviceIdProcessorSelector configuration file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <object class="sync4j.server.dm.DeviceIdProcessorSelector">
  <void property="defaultProcessor">
   <string>sync4j/server/dm/manager/DefaultProcessor.xml</string>
  </void>
```

```
  <void property="patterns">
   <array class="sync4j.framework.tools.PatternPair" length="2">
    <void index="0">
     <object class="sync4j.framework.tools.PatternPair">
      <void property="manager">
       <string>sync4j/server/dm/manager/DeviceDetailProcessor.xml</string>
      </void>
      <void property="pattern">
       <string>IMEI:333*</string>
      </void>
     </object>
    </void>
    <void index="1">
     <object class="sync4j.framework.tools.PatternPair">
      <void property="manager">
       <string>sync4j/server/dm/manager/SettingsProcessor.xml</string>
      </void>
      <void property="pattern">
       <string>IMEI:3335{3}1*</string>
      </void>
     </object>
    </void>
   </array>
  </void>
 </object>
</java>
```

The DeviceIdProcessorSelector class is a good example of how to develop a selector. A simplified version of the source code is reported below.

```
public class DeviceIdProcessorSelector
implements ProcessorSelector, LazyInitBean {
    // ------------------------------------------------------------ Private data

    private Pattern[] regexps;
    // ---------------------------------------------------------------- Properties

    /**
     * The pattern-pairs used to metch device ids
     */
    private PatternPair[] patterns;

    /**
     * Sets patterns
     *
     * @param patterns the new patterns
     */
    public void setPatterns(PatternPair[] patterns) {
        this.patterns = patterns;
    }

    /**
     * Gets patterns
     *
     * @return the patterns property
     */
    public PatternPair[] getPatterns() {
        return patterns;
    }

    /**
     * The default processor server bean name
     */
    private String defaultProcessor;

    /**
     * Sets defaultProcessor
     *
     * @param defaultProcessor the new default processor name
     */
```

```
public void setDefaultProcessor(String defaultProcessor) {
    this.defaultProcessor = defaultProcessor;
}

/**
 * Returns defaultProcessor
 *
 * @return defaultProcessor property value
 */
public String getDefaultProcessor() {
    return this.defaultProcessor;
}

// ------------------------------------------------------ ProcessorSelector

/**
 * @param sessionId the management session id: ignored
 * @param devInfo the device info
 *
 * @see ProcessorSelector
 */
public ManagementProcessor getProcessor(DeviceDMState dms, DevInfo devInfo) {
    String beanName = defaultProcessor;

    String device = devInfo.getDevId();

    Matcher m;
    for (int i=0; i<regexps.length; ++i) {
        m = regexps[i].matcher(device);

        if (m.matches()) {
            beanName = patterns[i].processor;
            break;
        }
    }

    ManagementProcessor processor = null;
    try {
        processor = (ManagementProcessor)
                    Configuration.getConfiguration().getBeanInstanceByName(beanName);
    } catch (Exception e) {
        // error handling
    }

    return processor;
}

// ------------------------------------------------------------ LazyInitBean

/**
 * During bean initialization all the given regular expressions are compiled.
 * If there are errors, a BeanInitializationException is thrown.
 *
 * @throws BeanInitializationException if one of the patterns cannot be compiled
 */
public void init() throws BeanInitializationException {
    if ((patterns == null) || (patterns.length == 0)) {
        regexps = new Pattern[0];
        return;
    }

    regexps = new Pattern[patterns.length];
    for (int i=0; i<patterns.length; ++i) {
        try {
            regexps[i] = Pattern.compile(patterns[i].pattern);
        } catch (Exception e) {
            if (log.isLoggable(Level.SEVERE)) {
                log.severe( "Error compiling pattern '"
                          + patterns[i].pattern
                          + "': "
                          + e.getMessage()
                          );
            }
```

```
                throw new BeanInitializationException(
                    "Error compiling pattern '" + patterns[i].pattern + "'", e
                );
            }
        }
    }
}
```

Most of the methods are getters and setters for the properties that we want to be able to configure through the XML document seen above. The core of the class is the *getSelector(...)* method, which tries a match between the device id (extracted from the DevInfo object) and the given regular expressions. If a match is found, the corresponding processor name is considered a server side JavaBean and then is instantiated by the means of the Configuration object.

Note also the use of lazy initialization: *init()* is called after the instance is created and all properties have been set. It gives DeviceIdProcessorSelector the opportunity to compile the regular expressions specified in the configuration file as strings.

7.2.1.1. OperationProcessorSelector

This is implemented by the class *sync4j.server.engine.dm.OperationProcessorSelector*. The idea is to use this selector as a dispatcher: it reads the operation to perform on a given device and uses that operation to build the name of the processor that should process the request. This processor is configured with an error processor name (to be used if the device state is 'E' - error) and a default processor name (to be used when no other processors could be selected).

The management processor name is constructed as follows:

prefix + *operation* + *suffix*

Where *prefix* and *suffix* are configurable values and *operation* is read from the device management state.

The algorithm used to select the correct management state for the device currently under management is represented in Figure 10. If the device management session is in an error state, the error processor is selected. If instead, the device management session is in any other state, the operation field, if specified, is used to select the right processor. Otherwise, the default selector will be used.

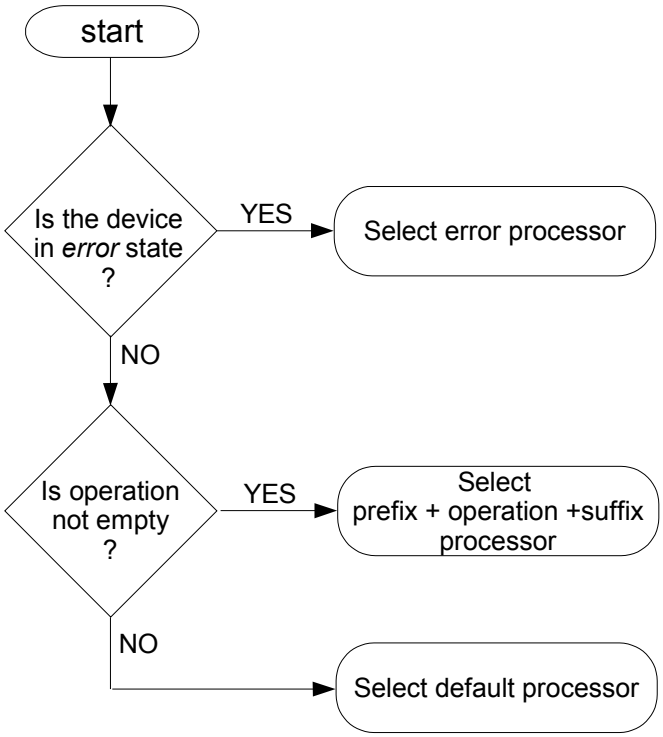

Figure 10 - OperationProcessorSelector selection flow chart

An example of an OperationProcessorSelector configuration file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
 <object class="sync4j.server.dm.OperationProcessorSelector">
  <void property="defaultProcessor">
   <string>sync4j/server/dm/processor/DefaultProcessor.xml</string>
  </void>
  <void property="errorProcessor">
   <string>sync4j/server/dm/processor/ErrorProcessor.xml</string>
  </void>
  <void property="namePrefix">
   <string>sync4j/server/dm/processor/</string>
  </void>
  <void property="namePostfix">
   <string>.xml</string>
  </void>
 </object>
</java>
```

7.2.2. Configuring the Management Engine

To configure the management engine in order to use a specific processor selector, set the Sync4j.properties's *server.dm.selector* property. For example:

server.dm.selector=sync4j/server/dm/OperationProcessorSelector.xml

7.3. Creating a Management Processor

A Management Processor is represented by the interface *sync4j.framework.engine.dm.ManagementProcessor*, which has the following methods:

method	*description*
void beginSession(String sessionId, Principal principal, int type, DevInfo info, DeviceDMState dmstate)	Called when a management session is started for the given principal. SessionId is the content of the SessionID element of the OTA DM message; type is the management session type (as specified in the message Alert); info is the device info of the device under management; dmstate is the *device management state*, which represents a row of the sync4j_dm_state table.
void endSession(int completionCode)	Called when the management session is closed. CompletionCode can be one of: • DM_SESSION_SUCCESS • DM_SESSION_ABORT • DM_SESSION_FAILED
ManagementOperation[] getNextOperations()	Called to retrieve the next management operations to be performed.
void setOperationResults (ManagementOperationResult[] results)	Called to set the results of a set of management operations.
String getName()	Name to uniquely identify the management processor instance (each installed management processor should have a different name in its configuration file).

7.3.1. ManagementOperation

This class represents an action that can be performed on a client management tree such as a Get, Replace, Exec and so on. It belongs to the *sync4j.framework.engine.dm package*. *ManagementOperation* can represent one of the following actions (see [7]):

- Add
- Atomic
- Copy
- Delete
- Exec
- Get
- Replace
- Sequence

To represent all possible operations, the hierarchy of Figure 11 is implemented; gray boxes are abstract classes, black boxes concrete implementations. *sync4j.framework.engine.dm.ManagementOperation* is an abstract class used for abstraction purposes only.

AggregatedManagementOperation adds the following methods:

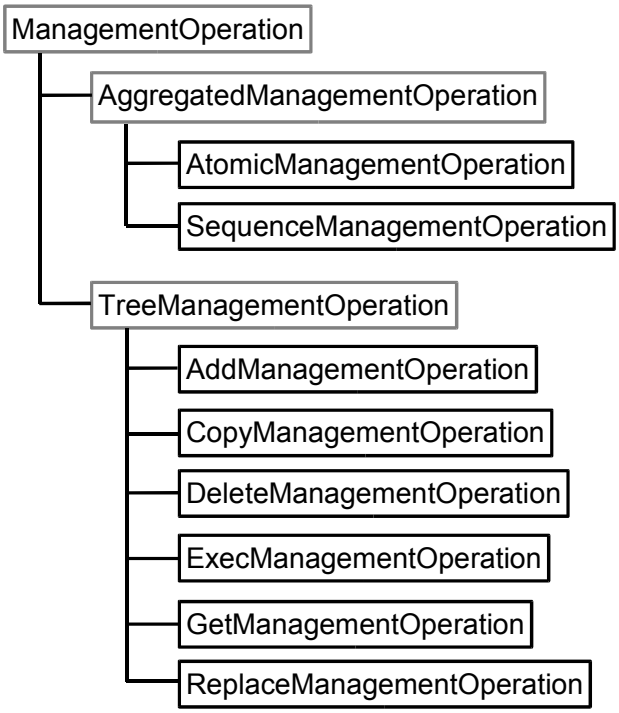

Figure 11 - ManagementOperation hierarchy

Method	***Description***
ManagementCommand[] getCommands()	Returns the contained commands.
void setCommands(ManagementCommand[] commands)	Sets the commands aggregation.

TreeManagementOperation adds the following methods:

Method	***Description***
Map getNodes()	Returns the management nodes affected by the operation.
void setNodes(Map)	Sets the management nodes affected by the operation.

7.3.2. ManagementOperationResult

When a management action is performed on the client, result status and maybe data are returned. This information is wrapped into a *sync4j.framework.engine.dm.ManagementOperationResult* object. It represents a combination of the following SyncML DM commands (see [7]):

- Results
- Status

sync4j.framework.engine.dm.ManagementOperation has the following methods:

Method	*Description*
int getStatusCode()	Returns the corresponding status for the operation.
void setStatusCode(int statusCode)	Sets the operation status code.
Map getNodes()	Returns the nodes property.
void setNodes(Map nodes)	Sets the nodes property.
String getCommand()	Returns the requested command (i.e. Add, Replace, Delete and so on).
void setCommand(String command)	Sets the requested command (i.e. Add, Replace, Delete and so on).

Note that the nodes properties may contain results if the *ManagementOperationStatus* regards a Get, or a set of nodes if it relates to a status of any command with items in it.
For example, if the following status is returned for a Delete command:

```
<Status>
    <CmdID>2</CmdID>
    <MsgRef>1</MsgRef>
    <CmdRef>1</CmdRef>
    <Cmd>Delete</Cmd>
    <TargetRef>./DevInfo/Lang</TargetRef>
    <Data>405</Data>
</Status>
```

The corresponding *ManagementOperationResults* would have:

statusCode: 405
command: Delete
nodes: {./DevInfo/Lang}

7.4. Using Scripting Management Processors

The ability to implement your own management processing is a powerful tool for the development of device management applications. However, the fact that you have to develop a java class, compile it and embed it into the server may be somehow complicated. For this reason, Sync4j DM Server provides out-of-the-box a concrete implementation of a management processor that makes things much simpler.

The class *sync4j.server.engine.dm.ScriptManagementProcessor* is a concrete implementation of the *ManagementProcessor* interface that uses a scripting language to carry on the required management logic. The scripting language supported by the current Sync4j DM Server implementation is BeanShell[9].

The interpreter is created once in the ManagementProcessor's *beginSession()* method and is initialized setting the scripting variable listed below and running the script specified in the *scriptFile* property. Scripts are located under the config path *sync4j/server/dm/processor/bsh*.

The script specified in *scriptFile* must have four entry points: *init()*, *getNextOperations(), setOperationResults()* and *endSession()*. In order to keep the interaction between ScriptManagementProcessor and the the underlying scripting engine, input and output values are passed by variables and not as input parameters and return values.

7.4.1. Scripting Variables

The following scripting variables are set in the interpreter environment:

Variable	*Description*
processor	The ManagementProcessor instance reference.
principal	User principal who is going to be managed.
devInfo	Device info of the device which is going to be managed.
managementType	Value given by the device when starting the management session (such as server or client initiated management session).
config	The Configuration object used to get server side JavaBeans objects and other configuration parameters.
sessionId	The current session identifier.
log	The Sync4jLogger to use for logging.
dmstate	The DeviceDMState object associated to the session.

The following scripting variables are input/output variables that the management script and the management processor share:

Variable	*Type*	*Description*
operations	OUT	ManagementOperation[] to be returned to the device management engine.
results	IN	ManagementOperationResult[] returned by the device management engine.

7.4.2. Helper Commands

Some utility commands have been developed. Those commands are available to any scripting management processor. They are:

Command	*Description*
addErrorMessage	Adds an error message to the given errorString accordingly to the given management operation result.
getBSURL	Returns the URL to use to invoke BS web services.
getInventory	Returns the DeviceInventory object configured with the server bean / sync4j/server/dm/DeviceInventory.xml.
getStatusMessage	Returns a text message associated to the given status code. If the code is unknown, an empty string is returned.
getStore	Returns the PersistentStore object used by the server.
InvokeWS	Invokes a GLUE WebServices given its wdsl, name and parameters.
valueOf	Null-safe version of String.valueOf().

7.4.3. Scripting Processor Example

A good example of how to develop a management processor script is represented by the GetDeviceDetails script used to retrieve from a device some of its ./DevDetail parameters. This script implements the GetDeviceDetails management operation.

The code is the following.

```
import java.util.*;
import java.util.logging.*;

import sync4j.framework.core.*;
import sync4j.framework.engine.dm.*;

// ---------------------------------------------------------------------------

final String DEVDETAIL_FWV = "./DevDetail/FwV";
final String DEVDETAIL_SWV = "./DevDetail/SwV";
final String DEVDETAIL_HWV = "./DevDetail/HwV";

// ---------------------------------------------------------------------------

String buildDetailString(HashMap nodes) {
    StringBuffer xml = new StringBuffer();

    xml.append("<DevDetail>")
       .append("<DevId>").append(devInfo.devId).append("</DevId>")
       .append("<Man>").append(devInfo.man).append("</Man>")
       .append("<Mod>").append(devInfo.mod).append("</Mod>")
       .append("<Lang>").append(devInfo.lang).append("</Lang>")
       .append("<FwV>").append(valueOf(nodes{DEVDETAIL_FWV})).append("</FwV>")
       .append("<SwV>").append(valueOf(nodes{DEVDETAIL_SWV})).append("</SwV>")
       .append("<HwV>").append(valueOf(nodes{DEVDETAIL_HWV})).append("</HwV>")
       .append("</DevDetail>");

    return xml.toString();
}

void sendDeviceDetail(String bssid, String status, String details) {
    log.info( "sendDeviceDetail("
            + bssid
            + ", "
            + status
            + ", "
            + details
            + ")"
            );

     invokeWS(getBSURL(), "setDeviceDetails", new Object[] { bssid, status, details });
}

// ---------------------------------------------------------------------------

void init() {
    log.info("Management script initialization");
    cont = true;
}

void getNextOperations() {
    log.info("getNextOperations!");

    nodes = new HashMap();

    nodes.put(DEVDETAIL_FWV, "");
    nodes.put(DEVDETAIL_HWV, "");
    nodes.put(DEVDETAIL_SWV, "");

    o = new GetManagementOperation();
    o.nodes = nodes;
    if (cont) {
        operations = new ManagementOperation[] { o };
        cont = false;
    } else {
        operations = new ManagementOperation[0];
    }
}

void setOperationResults() {
    log.info("setOperationResults!");
```

```
    String fwv = null;
    String swv = null;
    String hwv = null;

    details = "";
    for (result: results) {
        if (log.isLoggable(Level.FINE)) {
            log.fine("status code: " + result.statusCode);
            log.fine("for: " + result.nodes);
            log.fine("command: " + result.command);
        }

        if (Get.COMMAND_NAME.equals(result.command)) {
            if (result.statusCode != 200) {
                if (log.isLoggable(Level.INFO)) {
                    log.info( "Received error code "
                            + result.statusCode
                            + " for nodes "
                            + result.nodes
                            );
                    log.info( "Device: "
                            + devInfo.devId
                            + "; operation: GetDeviceDetail; sessionId: "
                            + sessionId
                            );
                }
                addErrorMessage(status, result);
            } else {
                details = buildDetailString(result.nodes);
            }
        }
    }

    //
    // If any error occurred, error contains the error message
    //
    if (status.length() == 0) {
        status = "0:"; // it means ok!
    }

    if (log.isLoggable(Level.FINE)) {
        log.fine("Device detail: " + details);
    }

    sendDeviceDetail(dmstate.info, status.toString(), details);

    //
    // Reset the operation so that GetDeviceDetails won't be erroneously
    // called again
    //
    dmstate.operation = null;
    dmstate.state = DeviceDMState.STATE_COMPLETED;
}

void endSession(int code) {
    log.info("endSession with code: " + (char)code);
}

// ---------------------------------------------------------------------------


log.info("Global script!");

importCommands("sync4j/server/dm/processor/bsh/command");

cont = true;

status = new StringBuffer();
```

The script looks very similar to a Java class without *main()*. As said before, when the interpreter is first created, this script is evaluated; this makes the global part of the script (the statements in the outermost scope) to be interpreted and executed. In the case above, the utility commands are

imported and some variables are initialized. Plus, remember that ScriptManagementProcessor will have set the scripting variables in the table above.

Before calling any other method of the script, ScripManagementProcessor calls *init()*; this is a good point where to put initialization code. Note that in our example there is only the initialization of the *cont* variable. It is done again for two reasons:

1. the global cont=true is done so that the variable cont will be created in the interpreter global scope (it is like a declaration)
2. the init() method could be called more then once (but always once per management session) – for example when initialization is retried in the case of a failed authentication.

The management processor asks the script processors which commands to send to the client calling *getNextOperation()*. In our case, we have to send a Get command for the three parameters ./DevDetail/FwV, ./DevDetail/SwV, ./DevDetail/HwV; therefore, the needed parameters are set in the *nodes* map and a new GetManagementOperation is created. Note that a simplified syntax is used to set the operation's nodes.
The management operation so created is returned to the management processor as an array of ManagementOperation objects setting the output variable *operations*. *cont* is then set to false to remember that the Get command has already been returned to the processor.

The processor will then processes all the returned commands and will collect the results from the device. Those results are translated to ManagementOperationsResult objects and *setOperationResults()* is called. Again, note that the ManagementOperationResult[] array is passed to the script in the input scripting variable *results*.

setOperationResults() processes the status and returned data and builds the device detail string calling *buildDetailString()*. If everything was correct, the captured device details are sent to the management console calling *sendDeviceDetail()*. Lastly, the method returns (after changing the device management status to complete).

Since a DM session is intended to be an iterative process, the processor will ask again for the next operations to send calling again *getNextOperation()*. This time *cont* is false and an empty array is returned. This tells the management processor that no more management commands are required.

At the end of the process the management processor will call *endSession()*.

8. Sync4j DM Server Interfaces to External Applications

This section describes the interfaces between the Sync4j DM Server and the external world.

8.1. Overview

The Sync4j DM Server interacts with the external world in many ways. First of all, any extensible module (synclets, message processors and so on) can interface with external software as it is needed. A good example is the Notification Sender of Figure 5. The notification sender is configured by the server JavaBeans /sync4j/server/engine/dm/NotificationSender.xml; this can be configured to use any custom class that implements the *sync4j.framework.notification.Sender* interface. For example, in the case of a WAP Push notification message, the notification of a server alerted management session is a WAP Push SMS message (or message chain). Sync4j does not implement any service to actually deliver the SMS message, therefore a custom component must be implemented. Once such component is developed, Sync4j can be configured to use just changing the corresponding server bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.2" class="java.beans.XMLDecoder">
    <object class="sync4j.dm.engine.MySender">
        <... any MySender specific property ...>
    </object>
</java>
```

This type of integration is mainly used in the Sync4j DM Server -> external system direction.

In order to make Sync4j DM Server accessible to other networked applications an EJB layer is provided. This can be accessed by any EJB client, on the same host or remotely.

This EJB layer can be easily wrapped by a web services layer using a WS toolkit such as Apache Axis [13].

The following sections describe the EJB and WS layers.

8.2. The EJB Layer

The management server can be accessed through the ManagementBean stateless session EJB. This is implemented by the classes: sync4j.server.engine.dm.ejb.ManagementBean, *sync4j.server.engine.dm.ejb.ManagementLocal*, *sync4j.server.engine.dm.ejb.ManagementRemote*, *sync4j.server.engine.dm.ejb.ManagementHomeLocal*, *sync4j.server.engine.dm.ejb.ManagementRemote*.

The interface exposed by the ManagementBean is the following:

Method	*Description*
bootstrap(int messageType, int transportType, String deviceUri, String phoneNumber, String username, String password, String info)	This method is used to create a new device into the Sync4j DM Server system. It may result in sending a bootstrap message to the physical device or not, depending by the configured sender. messageType: the type of the bootstrap message as define in *NotificationConstants* transportType: the type of the transport as define in *NotificationConstants* deviceUri: the device id phoneNumber: the phone number of the device username: the user name with which the device will do the next device management password: the password with which the device will do the next device management info: application specific info
sendNotification(int messageType, int transportType, int sessionId, String phoneNumber, String operation, String info)	Sends a notification message to the device with the given phoneNumber messageType: the type of the notification message as define in *NotificationConstants* transportType: the type of the transport as define in *NotificationConstants* sessionId: the session id to be stored in the notification message phoneNumber: the phone number operation: the management operation to be performed info: application specific detail information
executeManagementOperation(String phoneNumber, String operation, String info)	Executes the management sequence identified by the given operation name. phoneNumber: the phone number operation: the management operation name info: application specific detail information

9. Logging

This section explains how logging is performed in Sync4j DM Server.

9.1. Overview

Sync4j DM Server uses the standard Java Logging APIs introduced with the JDK 1.4.x.
For detailed information about the Java Logging APIs, see
http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/overview.html.
The output produced by the logging system can be configured in term of content and in term of writing media (the system output console, the file system, a database, etc.). To configure the JDK logging system, edit the file *{sync4j.home}/config/sync4j/logging.properties*.

Sync4j DM Server uses many logging namespaces so that it is easier to selectively enable/disable the logging of a specific module. These are:

Name	***Description***
sync4j	It is the default logging namespace, used when no other namespace is specified.
sync4j.engine	Synchronization engine logging information.
sync4j.handler	Session handling logging information.
sync4j.source	SyncSource related logging information.
sync4j.transport.http	Transport logging information
sync4j.dm	DM logging information
sync4j.server.dm.bootstrap	DM Bootstrap logging information
sync4j.server.dm.notification	DM Notification logging information
sync4j.framework.engine	Framework engine logging information

To enable the maximum of verbosity for a given module, the configuration file logging.property should have the following line (other loggers settings under sync4j should be commented out):

sync4j.level=ALL

9.2. Adding Logging for Custom Components

The Sync4j DM Server logging feature can be use by any Sync4j DM Server class or extension. It is even possible to create your own logging namespace, so that you can isolate the logging information produced by your components from all other logging.

The *java.util.logging.Logger* to use for logging is acquired with the following sample code:

```
Logger log = Sync4jLogger.getLogger(name);
```

Where name is one of the standard name defined in *sync4j.framework.logging.Sync4jLoggerName* or your own logger name. Note that “sync4j.” will be prepended to the given name, so that all Sync4j DM Server loggers will be hierarchically grouped under the sync4j name space; in this way, all the Sync4j DM Server logging activity can be enabled/disabled all in once changing a single line in the logging.properties file.

10. Appendicies

10.1. WAP Headers explanation for Bootstrap Message

10.1.1. PLAIN Profile

Header
06 05 04 0B 84 C0 02 01 06 2E C2 91 80 92 45 36 34 30 37 42 37 42 30 46 37 42 46 38 39 37 43 32 45 44 37 43 43 45 46 35 31
35 43 30 37 44 42 31 44 32 34 33 39 34 00 AF 87

WDP Header
06: User-Data-Header(UDHL) Length
05: UDH IE Identifier Port Number
04: UDH port number IE length
0B: Destination port (high)
84: Destination port (low)
C0: Origination port (high)
02: Origination port (low)

WSP Header
01: Transaction ID / Push ID
06: PDU Type(push)
2E: Headerslength (content type + headers)
C2: Content-type
91: SEC
80: Auth. method = 0
92: MAC
MAC Value: 45 36 34 30 37 42 37 42 30 46 37 42 46 38 39 37 43 32 45 44 37 43 43 45 46 35 31 35 43 30 37 44 42 31 44 32 34 33 39 34 00
AF: X-WAP-Application-ID
87: x-wap-application:syncml.dm

10.1.2. WAP Profile

Header
06 05 04 0B 84 C0 02 01 06 2E B6 91 80 92 36 34 42 33 45 46 37 35 45 38 39 42 32 35 41 33 44 35 36 45 37 30 30 32 33 30 33

46 41 31 39 41 36 34 38 41 33 34 42 37 00 AF 82

WDP Header
06: User-Data-Header(UDHL) Length
05: UDH IE Identifier Port Number
04: UDH port number IE length
0B: Destination port (high)
84: Destination port (low)
C0: Origination port (high)
02: Origination port (low)

WSP Header
01: Transaction ID / Push ID
06: PDU Type(push)
2E: Headerslength (content type + headers)
B6: Content-type
91: SEC
80: Auth. method = 0
92: MAC
MAC Value: 36 34 42 33 45 46 37 35 45 38 39 42 32 35 41 33 44 35 36 45 37 30 30 32 33 30 33 46 41 31 39 41 36 34 38 41 33 34 42 37 00
AF: X-WAP-Application-ID
82: x-wap-application:wml.ua

10.2. Notification message using Wap Push

Notification message created by the DM Server:

The following examples are valid notification messages with 'sync4j' as server id.

Example 1:
81 6B 41 D3 1C 99 84 52 65 73 70 F8 C1 CC 32 C5 02 C0 00 00 00 00 04 06 73 79 6E 63 34 6A

Example 2:
47 E3 FC D5 C5 09 81 36 AF 01 4F E7 9C 1C AD F1 02 C0 00 00 00 00 03 06 73 79 6E 63 34 6A

WDP Header:
06: User-Data-Header(UDHL) Length = 6 bytes
05: UDH IE Identifier Port Number
04: UDH port number IE length
0B: Destination port (high)
84: Destination port (low)
C0: Origination port (high)
02: Origination port (low)

WSP Header:
01: Transaction ID / Push ID
06: PDU Type(push)
03: Headerslength (content type + headers)
C4: Content type
AF: X-WAP-Application-ID
87: Id for urn: x-wap-application:syncml.dm

Complete sms (WDP + WSP + Notification message):

Example 1:
06 05 04 0B 84 C0 02 01 06 03 C4 AF 87 81 6B 41 D3 1C 99 84 52 65 73 70 F8 C1 CC 32 C5 02 C0 00 00 00 00 04 06 73 79 6E 63 34 6A

Example 2:
06 05 04 0B 84 C0 02 01 06 03 C4 AF 87 47 E3 FC D5 C5 09 81 36 AF 01 4F E7 9C 1C AD F1 02 C0 00 00 00 00 03 06 73 79 6E 63 34 6A