RFC 87 Device Management Tree Structure

Confidential, Draft

22 Pages

Abstract

This document specifies the structure of the Managed Objects defined by OSGi. All information related to management of an OSGi platform is available through the Device Management Tree. Also all device management operations can be issued by manipulation of the Device Management Tree.

0 Document Information

0.1 Table of Contents

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [1].

The terminology used in this RFC is defined in [9].

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	06 08 2004	Initial version. Balázs Gődény, Nokia, balazs.godeny@nokia.com
0.2	07 20 2004	Major rework in configuration and logging Gábor Pécsy, Nokia, gabor.pecsy@nokia.com
0.3	08 12 2004	Rework of configuration: changed XML representation to node structure Small modifications in the logging section. Balázs Gődény
0.4	09 01 2004	Comments from the Boston meeting addressed Balázs Gődény
0.5	09 27 2004	Added monitoring and app model chapters. Corrections in config chapter: nested arrays and mixed type vectors not supported. Corrections in log chapter: response is sent asynchronously Balázs Gődény

Revision	Date	Comments
0.6	11 19 2004	Log search response structure changed to match alertitems Added KPI node DTD, clarifications to Monitoring chapter Plugin boundaries are not mandated Changed representation of factory configurations. Changed config structure to allow the storage of location binding. Balázs Gődény
0.7	12 16 2004	Log search response structure changed to use the OMA BLOB spec Corrections in the app model chapter. App containers and content handler section removed. App properties merged to MEG app properties. Removed category, container_id, required_applications, added venor and singleton properties. MIDP properties removed. PAUSE, RESUME removed. Config: OMA DM types and other Java types have the same representation Node name limits in configuration and monitorable PID names addressed Balázs Gődény

1 Introduction

As described in the MEG High Level Architecture document [4], the only way a remote manager can issue management operations and access management information stored on devices running the OSGi MEG platform is via manipulation of the Device Management Tree. Therefore the API for remote management is the description of the managed objects defined by OSGi, that is the structure of the OSGi subtree. Note that the protocol used between the remote manager and the device is not specified.

The OSGi subtree holds information about all management aspects of the platform. This document specifies how information related to configuration, logging, monitoring, deployment, applications and policy is mapped to the Device Management Tree.

2 Application Domain

See the DMT API RFC [7] for a good summary on why the OMA DMT model was chosen as a device management meta data model in OSGi MEG.

3 Problem Description

[TODO]

4 Requirements

Requirements against different areas of the device management can be found in their respective RFPs. The only relevant additional requirement is that all management operations MUST be available through DMT operations.

Quote from the Device Management RFP [5] :

“Most of the use cases […] apply to both the user of the device and the remote administrator who are expected to have both read and write access to DM data and operations. […] Such a multitude of actors and access patterns result in a set of requirements for a uniform access mechanism on the device which can be used by all actors.”

5 Technical Solution

5.1 OSGi Managed Object URIs

This specification does not mandate where the OSGi managed objects must be placed in the DMT. We assume however that there is a common root somewher in the tree for all the objects defined in this document. The URI of this root is referred to as `[OSGi_root]`, even in example texts.

5.2 Configuration

This section describes how configuration data is represented in the DMT. Modifying configuration is achieved by directly manipulating the DMT nodes representing configuration. All configuration data is stored and can be accessed from the `[OSGi_root]/cfg/` subtree. Description of the OSGi Configuration Service can be found in Chapter 10 of [8].

5.2.1 Representing the Persistent Identity

5.2.1.1 Managed Services

Individual configuration dictionaries are stored in the subtrees under the following URI:

```
        [OSGi_root]/cfg/<id>
```

where in case of Managed Services the `<id>` is a unique identifier of the configuration dictionary. It is the persistent identity of the Managed Service (as defined in section 10.3 of [8]) provided the persistent identity is a valid OMA DM node name. To be a valid node name it must not contain the Reserved characters described in 2.2 of RFC-2396 (URI Generic Syntax) and also it's length must be shorter than the URI segment length limit defined for the given device. As the Configuration Admin specification does not contain these limitations for service PID strings it can not be always guaranteed that the PID is a valid node name. Because of this the PID is always present as the value of the `[OSGi_root]/cfg/<id>/pid` node. The type of this node is `chr`, the valid operations are: Add, Get, Replace. The entity responsible for providing the DMT view of the configuration database must make sure that the `<id>` is the same as the PID if it is a valid node name on the device, or it is a unique identifier otherwise. In the latter case it is not specified how the `<id>` should be generated.

Valid operations for the `[OSGi_root]/cfg/<id>` node: Add, Get, Delete.

An example when the id and the pid are the same (see 10.5 of [8]):

```
        The configuration dictionary is under [OSGi_root]/cfg/com.acme.fudd.
        The value of the [OSGi_root]/cfg/com.acme.fudd/pid node is : “com.acme.fudd”
```

An example when the id and the pid are not the same:

The configuration dictionary is under [OSGi_root]/cfg/longname.
The value of the [OSGi_root]/cfg/longname/pid node is : “a_very_long_pid/with_forbidden_characters”

5.2.1.2 Managed Service Factories

The configuration dictionaries belonging to Managed Service Factories are represented as defined above for Managed Services with the following two differences.

Factory PIDs. The dictionary contains the factory PID property which is stored under the following URI:

```
        [OSGi_root]/cfg/<id>/keys/service.factoryPid
```

Creating this node from the remote server is equivalent to calling the `createFactoryConfiguration` method of the ConfigurationAdmin service. Valid operations for the factory PID node: Add, Get.

PID. The primary key of the dictionary in the factory case is generated by the Configuration Admin. Using this key as a node name has the following two drawbacks:

- Transferring this ID to the remote server would require an additional remote protocol iteration.
- As this ID can not be known before the configuration dictionary is created it is not possible to write a script without using variables which creates the dictionary and manipulates the data in it later.

For these reasons the following solution is used. The server creates the subtree representing the configuration dictionary as in the Managed Service case, but the PID it is using as primary key will be mapped to the PID the ConfigurationAdmin service creates when the dictionary is created. It is the task of the entity responsible for the configuration subtree of the DMT (typically a Dmt Plugin) to store this mapping persistently. This way the PID of the factory configuration becomes a well known name for the server without needing an additional protocol iteration.

An example. The server creates a new configuration dictionary using “`myapp.id:12`” as PID. I.e. the dictionary is created under `[OSGi_root]/cfg/myapp.id:12`. It sets up a factory pid: “`com.what.ever.myapp`”. I.e. the value of the `[OSGi_root]/cfg/myapp.id:12/keys/service.factoryPid` node is `com.what.ever.myapp`. The Configuration Admin creates the configuration dictionary and sets its PID to some machine generated string, like “`confadm.12345`”. The Dmt Plugin maps persistently `myapp.id:12` to `confadm.12345`, so that the remote server can always access the dictionary by the key it knows, but the plugin uses the `confadm.12345` key when it manipulates the dictionary through the Configuration Admin.

How the mapping is stored is implementation specific, however a simple way could be to store the key given by the server in the configuration dictionary itself under a fixed name. Another solution is to shortcut this indirection and use the key given by the server as the key in the Configuration Admin.

5.2.2 Location

An important property of the configuration dictionary is the location of the bundle to which it is bound (see 10.4.1 of [8]). This location string is stored in a node of type `chr` under the following URI:

```
        [OSGi_root]/cfg/<id>/location
```

If the configuration dictionary has no associated location, then this node is not present. Valid operations for the location node: Add, Get.

5.2.3 Configuration dictionary nodes

The configuration dictionary of a configuration target consists of key-value pairs. The configuration dictionary is mapped to the DMT. The URI for a configuration item is the following:

```
        [OSGi_root]/cfg/<id>/keys/<key>
```

For example, the `portNumber` property of the previous `com.acme.fudd` Managed Service can be found in the subtree under the following URI:

```
        [OSGi_root]/cfg/com.acme.fudd/keys/portNumber
```

Valid operations for the nodes representing configuration keys: Add, Get, Delete, Replace. These nodes are internal nodes.

The value of a configuration item is typed. The mapping of the value to the DMT depends on the type of the data. Configuration Admin currently permits the following data types:

```
        type ::=
               String | Integer | Long | Float
             | Double | Byte | Short | Character
             | Boolean
             | vector
             | arrays

        primitive ::=
               long | int | short | char
             | byte | boolean | double | float

        arrays ::=
               primitive '[]' | type '[]'
        vector = <Vector of type>
```

5.2.3.1 Representation of scalar types, arrays and vectors

Scalar data types, arrays and vectors are represented as a multi-node structure. As mentioned above the `[OSGi_root]/cfg/<id>/keys/<key>` node is an internal node (having node format). It has 3 children nodes as follows:

- `[OSGi_root]/cfg/<id>/keys/<key>/type` contains the Java type name like `Float, char)` in a `chr` node
- `[OSGi_root]/cfg/<id>/keys/<key>/cardinality` is a `chr` node containing
  - o "`scalar`" in case of a scalar value
  - o "`array`" where the configuration item is an array
  - o "`vector`" where the configuration item is a vector
- `[OSGi_root]/cfg/<id>/keys/<key>/value`
  - o is a `chr` node in case the cardinality is scalar and there is no corresponding OMA DM type of the Java type. The node contains the value in a string which is parseable to the corresponding Java type.

- o is a node with the corresponding OMA DM format if there is a good mapping from the Java type. The table in 5.2.3.2 contains the list of types where the corresponding OMA type must be used. The value of the node in this case must be the value of the configuration item.

- o it is an internal node if the cardinality is array or vector. (With the exception of `byte[]` which is mapped to the `bin` DMT type.) It contains the items of the array or vector as its children leaf nodes. The name property of these leaf nodes must be the index of the corresponding element in the array or vector. As both arrays and vectors are indexed with integer numbers, the name must be parseable to Java Integer type. The type of the leaf nodes must be `chr` unless a corresponding OMA DM data type exists.

5.2.3.2 Scalar types supported by OMA DM

If the data type is supported by OMA DM – i.e. an applicable format exists – then the value of the configuration item is stored in the value of the corresponding DMT node and the standard DMT format must be used. The following table summarizes the mapping between data types supported by Configuration Admin and the formats supported by OMA DM.

Configuration Admin Data Type	**OMA DM Data Format**
`String`	`chr`
`Boolean`	`bool`
`byte[]`	`bin`
`Null`	`null`, and the corresponding value must be empty.

Note that the OMA `int` data type is unsigned, so it can not be mapped to the Java integer type. Integers must be stored in String format as described in the previous section.

Note also that the list of types supported by OMA DM is expected to grow in the future.

5.2.4 Restrictions

The specified DMT structure is very flexible and permits the representation of Configuration objects that are not considered as valid according to the Configuration Admin Service specification. Following is a list of restrictions and rules that should be obeyed in order to maintain valid Configuration objects. If these rules are violated the entity responsible for the `[OSGi_root]/cfg` subtree must throw a DmtException (see [7]).

- This representation rule could be applied to arrays of arrays or other, more complex data structures; however, Configuration Admin does not support them so they must not be used in the DMT either.

- Vectors cannot store primitive types (long, int, short, char, byte, boolean, double and float); therefore, they must not be used in subtrees representing a vector.

- Null (`null`) values are permitted in vectors. Null values can be used in arrays, which store object types (String, Long, Integer, Short, Character, Byte, Boolean, Double, Float). Null values are not permitted in arrays of primitive types.

- Java arrays and vectors are indexed from 0 to some n continuously. If an index is missing, the array or vector is in an inconsistent state. When deleting elements (an unlikely but possible use case) care must be taken to maintain the continuous indexing of elements at the end of the modification session.

- The node names in DMT are case sensitive; however, in a configuration dictionary the keys are searched case insensitively. Therefore, the node names in the DMT representation of a configuration dictionary must be different even if compared case insensitively.

Nodes in the `[OSGi_root]/cfg` subtree support the add, get, replace and delete commands, controlled by the ACL of the node. These nodes do not support the exec command. Modification of more than one entry under the same `service_pid` must be done atomically, i.e. the whole dictionary is updated in one step. In OMA DM, this corresponds to the `Atomic` operation.

5.2.5 Examples

A scalar value having a corresponding OMA DM type.

URI	Node format	Node value
`[OSGi_root]/cfg/myservice/keys/autostart`	`node`	
`[OSGi_root]/cfg myservice/keys/autostart/type`	`chr`	`boolean`
`[OSGi_root]/cfg/myservice/keys/autostart/cardinality`	`chr`	`Scalar`
`[OSGi_root]/cfg/myservice/keys/autostart/value`	`bool`	`True`

A scalar value not having a corresponding OMA DM type:

URI	Node format	Node value
`[OSGi_root]/cfg/myservice/keys/myPI`	`node`	
`[OSGi_root]/cfg myservice/keys/myPI/type`	`chr`	`double`
`[OSGi_root]/cfg/myservice/keys/myPI/cardinality`	`chr`	`scalar`
`[OSGi_root]/cfg/myservice/keys/myPI/value`	`chr`	`3.14`

An array of scalar values, where the base type has a corresponding OMA DM type:

URI	Node format	Node value
`[OSGi_root]/cfg/myservice/keys/mycfg`	`node`	
`[OSGi_root]/cfg myservice/keys/mycfg/type`	`chr`	`boolean`
`[OSGi_root]/cfg/myservice/keys/mycfg/cardinality`	`chr`	`array`

`[OSGi_root]/cfg/myservice/keys/mycfg/value`	node	
`[OSGi_root]/cfg/myservice/keys/mycfg/value/0`	boolean	true
`[OSGi_root]/cfg/myservice/keys/mycfg/value/1`	boolean	false

An array of scalar values, where the base type has no corresponding OMA DM type:

URI	**Node format**	**Node value**
`[OSGi_root]/cfg/myservice/keys/mycfg`	node	
`[OSGi_root]/cfg myservice/keys/mycfg/type`	chr	double
`[OSGi_root]/cfg/myservice/keys/mycfg/cardinality`	chr	array
`[OSGi_root]/cfg/myservice/keys/mycfg/value`	node	
`[OSGi_root]/cfg/myservice/keys/mycfg/value/0`	chr	1.1
`[OSGi_root]/cfg/myservice/keys/mycfg/value/1`	chr	1.2

5.2.6 Considered alternatives

1. Representing the whole dictionary as a single XML document under the node corresponding to the pid of the configuration target. Advantage: the whole configuration dictionary can be updated in one atomic step. Drawback: coexistence of two different structures (DMT and XML).

2. Representing the whole dictionary as a single document but not using XML. Storing the dictionary as a string value. An example, which shows what kind of format could be used:

```
key=address
format=String
value=255.255.255.255

key=port
format=integer
value=1024

key=signature
format=[byte
value=jkhdsfKJhdsf89374h
```

Advantage: no dependency on the XML Service in the device. Drawback: self made, proprietary parser needed on both sides

3. Representing the data types not supported by OMA DM in a single node using XML format. Advantage: can be fetched in one step. Drawback: as in alternative 1.

4. Using the Type property to carry type information about the nodes. Drawback: only IANA registered MIME types can be used, this solution is not extensible.

5.3 Logging

This section describes how logging information is made available to the remote management server. Typically, the remote manager is interested only in a subset of the log records, like the highest severity entries originating from a specific application within the last 24 hours. Because of the high bandwidth required, transferring all the records and doing the filtering on the server side is not an option. The remote manager has to have means to issue log search requests and receive only the log records it is interested in. The standard OSGi LogReader service (see 9.4 in [8]) does not provide filtering, it supports only giving back the full list of log records. The management application, which provides remote log access through the DMT, should extend the functionality of the LogReader service in a significant way.

Applications creating log entries use the Log Service as specified in Chapter 9. of [8]. The Log Service specification need not be modified in any way to fulfill the requirements of remote log access.

The solution specified below does not require the device to store the results of a log search. The DMT is used to set up log search requests, the result comes back to the server in an asynchronous notification.

5.3.1 URIs

Log searchs can be initiated using the `[OSGi_root]/log/` subtree. Note that the node structure described here can be used without modifications also in other execution environments, not only in the OSGi service platform. In OSGi we mandate that the log subtree's root is `[OSGi_root]/log` but in other environment the subtree might be located elsewhere in the DMT. The tree structure described here might be standardized later on OMA.

All data related to a log search requests are stored in the log subtree under the following URI:

```
        [OSGi_root]/log/<search_id>
```

where `<search_id>` is a unique string identifier of the search request, given by the management server when it creates the node representing the search request. The search identifier string must not contain the '/' character.

A search request contains the following elements (`$REQ` denotes the root of the search request i.e. `[OSGi_root]/log/<search_id>`):

URI	Format	Mandatory / Optional	Description

`$REQ/filter`	`chr`	Optional	Contains the filtering expression. The result must include only those log entries, which satisfy the specified condition. The filter should be given in the usual OSGi filter format. The filter can contain conditions with any attribute of a log entry. For example:
			
			`(&(severity>=2)`
			`  (time>=20040720T194223Z))`
			
			If the filter is not present, all log entries are included in the result.

URI	Format	Mandatory / Optional	Description
`$REQ/exclude`	`chr`	Optional	A comma-separated list of log entry attributes (see Table 1 Log entry attributes for the available attribute names). If specified, the listed attributes must not be included in the search result. Note that the filter expression may contain (and filtering should be done against) condition for an attribute even if it is added to the exclude list. Example: `"severity, data"` If not present, all log entry attributes are included in the search result.
`$REQ/maxrecords`	`int`	Optional	The maximum number of log records to be included in the search result.
`$REQ/maxsize`	`int`	Optional	The maximum size of the search result. See the interpretation of this attribute in 0.

When the server creates the $REQ node it can not assume that all the leaf nodes under it are available, they have to be created manually if they are needed.

5.3.2 Using log search

To prepare a log search, management server should first create a new node in DMT in the `[OSGi_root]/log` subtree and fill in the filter, exclude, maxrecord and maxsize nodes.

A prepared log search can be executed by an EXEC command to the $REQ URI. The EXEC command itself is synchronous, but the result of the log search is sent back asynchronously. If the returned status of the EXEC command indicates success, it means that the log search was initiated on the device and the search will arrive later asynchronously. If the command fails, the search was not started on the device. Nevertheless the device may send an asynchronous reply even in this case to indicate the reason of the failure.

It is not required that the device stores the search results in the DMT.

5.3.3 Search response format

Although the result of a successful log search operation is not stored in the DMT, the format of the search response data is equivalent to the WBXML representation of an imaginary subtree using the `BLOB` feature of OMA DM 1.2 (see the BLOB specifications in 22). Note that it would be possible to define a much simpler XML structure to carry the same information but we did not want to introduce any proprietary format where a standard is available.

This imaginary subtree is the following (Note once more that this subtree is not present in the DMT, this is used only as a structured representation of the log search results.):

Log records are stored under the `./logresult/<record_id>` interior nodes, where `<record_id>` is an implementation dependent unique identifier of the log record. This identifier is expected to be unique in the scope of the result set of a single execution of this log search request. Under these nodes, the attributes of the corresponding log entries are stored in leaf nodes. Note that only those attributes are present which are not included in the exclude list of the log search request.

The following table summarizes the log entry attributes, their format and value. The name of the leaf node that contains the attribute value is the same as the attribute name. The table also specifies how to map the content of a LogEntry (9.5 in [8]) to the attributes:

Table 1 Log entry attributes

Attribute name	Description
`time`	The value is the UTC base date and time of the creation of the log entry in basic representation, complete format as defined by ISO-8601. Example: `“20040720T221011Z”`
`severity`	The associated severity level of the log entry. See 9.3 of [8] for the description of log level usage in OSGi.
`system`	The name of the large-scale functional unit that generated the entry. In OSGi, this attribute should be the ID of the originator bundle.
`subsystem`	The name of the small-scale functional unit that generated the entry. In OSGi, this attribute should be the name of the originator service.
`message`	Textual, human readable description of the event.
`data`	Supplementary data for the log entry, it may be empty. The content is event specific. In OSGi this attribute must contain exception associated with the log entry, if any. The attribute must contain the name of the exception class, the message and the stack trace associated with the exception object.
`id`	A device generated unique identifier of the log record.

5.3.3.1 OMA DM binding

If the management protocol is OMA DM then the log results must be sent in an Alert of type 1226. The WBXML encoded binary data is placed into the data element of the alert.

An example of a log search result alert.

```
<Alert>
   <CmdID>1</CmdID>
   <Data>1226</Data>
   <Item>
      <Source><LocURI>[OSGi root]/log/myrequest</LocURI></Source>
      <Meta>
         <Format xmlns=’syncml:metinf’>xml</Format>
         <Type xmlns=’syncml:metinf’>
             application/vnd.syncml.dmblob+xml
         </Type>
      </Meta>
      <Data>  *** Payload XML-BLOB data here *** </Data>
   </Item>
</Alert>
```

5.3.4 Considered alternatives

1. Giving back the whole list of logs as the current LogReader does. Advantage: simple. Drawback: very expensive in terms of bandwidth

2. Using a special XML language for log search requests. The general practice is to have the parameters of an exec command available in a subtree.

3. Storing the result in the DMT. Drawback: requires storage space on the device. Unclear when to delete the data.

4. Using the `?list=StructData` feature of OMA DM (see [3]). to describe the search result. Disadvantage: very verbose.

5.4 Monitoring

This section describes how the monitoring related features (see [12]) are made available to the remote management server through DMT operations.

The OMA Device Management work group defined a management object structure for Traps (event based reporting of faults and performance data), see the Trap-MO documents in [11] . The structure defined on OMA can be used without modifications in OSGi to represent monitoring data in the DMT. The tree structure is not described in this document, refer to [11] for the definition.

The Java interfaces of the MonitorAdmin service were designed in such way that there is a straightforward mapping from the API to the tree structure. There are some features however which are available only to the user of the API but not for the remote server:

- The API allows to specify a given number of measurements to be made. The remote server has to explicitely stop a measurement job, it will never expire.

- The API allows to group KPIs into measurement jobs. The remote server has to manage the KPIs individually.

A monitorable service is identified by its service_pid. The KPIs provided by the monitorable service are stored in the DMT under the

```
[OSGi_root]/mon/<service_pid>/<KPI_name>
```

node. This is an internal node which has a subtructure as defined in [11]. The subtree holds information about the collection method, the reporting schedule and the current value of the KPI.

In [12] it is mandated that the service_pid of a monitorable service must not contain contain the Reserved characters described in 2.2 of RFC-2396 (URI Generic Syntax) and also it is recommended to keep the service_pid string as short as possible.

5.4.1 Result node format

[11] does not define how the KPI values should be stored in the tree, but it prohibits the use of structured representation, it allows only a single node of any allowed OMA DM format. For the representation of KPIs we use a simple XML format. Each KPI will be stored in a node having XML format, conforming to the following DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT kpi (value+)>
<!ATTLIST kpi
   type (string | long | integer | short | character | byte | double |
         float | boolean) #REQUIRED
    cardinality (scalar | array ) #IMPLIED
>
<!ELEMENT value (#PCDATA)>
```

If the cardinality attribute is not present then the KPI is a scalar. If the cardinality is `vector` and the current vector is empty then the value of the type attribute is unspecified.

The following is an example for a scalar and an array KPI value.

```
<kpi type="integer">
    <value>5</value>
</kpi>
```

```
<kpi type="integer" cardinality="array">
    <value>5</value>
    <value>3</value>
</kpi>
```

5.4.2 Notifications

[11] mandates that a notification must be sent to the server when a KPI is updated or when the predefined sampling time expires. In case of OMA DM management protocol the notification must be an alert of type 1226 having the new KPI value in its Data element. The type of the alert must be `x-oma-trap:<monitorable_id>/<KPI_id>`. An example of an OMA DM KPI trap follows.

```
<Alert>
   <CmdID>1</CmdID>
   <Data>1226</Data>
   <Item>
      <Source><LocURI>[OSGi root]/mon/service id/KPI id</LocURI></Source>
      <Meta>
         <Type>x-oma-trap:service id/KPI id</Type>
         <Format>xml</Format>
      </Meta>
      <Data>
         <kpi type="integer">
            <value>5</value>
         </kpi>
      </Data>
   </Item>
</Alert>
```

5.4.3 Considered alternatives

1. Creating a proprietary DMT representation. The Trap MO is a good enough fit for the problem. If we standardise something on OSGi which is quite similar but not the same as on OMA then there is little chance of introducing it on OMA later. Using the Trap MO for representing KPIs is a necessary but acceptable compromise.

5.5 Application model

This section describes how the features listed in the Application Model RFC [10] are made available to the remote management server through DMT operations.

5.5.1 Applications and Related Commands

Each installed application is represented in the DMT in its own subtree under the following URI

```
[OSGi_root]/apps/<unique_id>
```

where <unique_id> comes from the corresponding ApplicationDescriptor's service.pid service property. Only the GET command is valid for this node.

5.5.1.1 Creating new instances

Creating a new instance of the application can be done by issuing an EXEC command on the

```
[OSGi_root]/apps/<unique_id>/launch/<exec_id>
```

node.

The requestor is responsible for keeping the `exec_id` unique, a possible solution is that the id contains the server's identifier. To perform the EXEC command a subtree which holds the parameters of the command must be built under the `[OSGi_root]/apps/<unique_id>/launch/<exec_id>` node. Each parameter has a string name; its type and value are leaf nodes under the `[OSGi_root]/apps/<unique_id>/launch/<exec_id>/<parameter_name>` node. The name will be interpreted as a string and will act as a key in the application's startup parameter map. The type and value are leaf nodes. The allowed types are equivalent to the Java language primitive wrapper classes (e.g. "Integer", "Boolean") and "String". The value is in string representation and will be inserted into the startup parameters in the corresponding wrapper class (e.g. Integer, Boolean). Broken parameters or parameters with invalid type or value should be discarded. The following example shows a subtree holding application startup parameters for an EXEC command, which attempts to start the chess application with *player name* and *thinking time* parameters.

URI	**Value**
[OSGi_root]/apps/chess/launch/com.remotesrv.exec123/player_name/type	String
[OSGi_root]/apps/chess/launch/com.remotesrv.exec123/player_name/value	Joe
[OSGi_root]/apps/chess/launch/com.remotesrv.exec123/thinking_time/type	Integer
[OSGi_root]/apps/chess/launch/com.remotesrv.exec123/thinking_time/value	10

The `EXEC` command returns successfully (status (200) OK in OMA DM protocol) if the application was started successfully. In this case the subtree from `[OSGi_root]/apps/<unique_id>/launch/<exec_id>` may not be available any more.

Otherwise the EXEC command fails (return status is `(500) Command failed` if the protocol is OMA DM) and `reason` and `message` leaf nodes are created under the `[OSGi_root]/apps/<unique_id>/launch/<exec_id>` node. The value of `reason` is `"Singleton error"` if the application is a singleton and already has an instance, `"Application error"` if application launching has failed. The value of the `message` is the textual message of the exception – returned by the

`Exception.getMessage()` – if exception was thrown during the application startup process, otherwise empty. After performing a GET command on both `reason` and `message` the subtree from `[OSGi_root]/apps/<unique_id>/launch/<exec_id>` may not be available any more. It may happen that the performer of the EXEC command is not interested in the reason of the failure thus the stalled subtrees from `[OSGi_root]/apps/<unique_id>/launch/<exec_id>` should be removed. The removal is the responsibility of the device, it happens according to an implementation specific schedule.

5.5.1.2 Properties of applications

The following leaf nodes represent the properties of the application and these are the valid commands:

Name	Type	Description	Command
`localizedname`	`chr`	Localized name of the application according to the current locale of the device	GET: value can be obtained by retrieving the value for “application.name” key from the Map returned by the getProperties() method of the corresponding ApplicationDescriptor
`version`	`chr`	The version of the application	GET: the value can be obtained from the corresponding ApplicationDescriptor’s “application.version” service property
`vendor`	`chr`	The vendor of the application	GET: value can be retrieved from the corresponding ApplicationDescriptor’s “application.vendor” service property
`autostart`	`boolean`	Tells whether the application is started automatically after installation or when the MEG Environment starts	GET and REPLACE: value can be queried and set at the corresponding ApplicationDescriptor’s “application.autostart” service property
`locked`	`boolean`	Tells whether the application is locked	GET and REPLACE: value can be queried and set at the ApplicationAdmin’s lock/unlock/isLocked methods.
`singleton`	`chr`	Tells if the application is a singleton	GET and REPLACE: value can be queried and set at the corresponding ApplicationDescriptor’s “application.singleton”

			service property
`bundle_id`	`chr`	Refers to the long identifier of the bundle which was received from the framework when the bundle was installed.	GET: the value can be queried from the getBundle() method of the corresponding ApplicationDescriptor's ServiceReference
`required_services/<n>`	`chr`	Gives information about the required services. This dependency comes from strong service dependency or physical resource dependency. <n> is an integer starts from `1` and incremented by one. The values are the service name or service symbolic name (if exists) of the required services.	GET: the value can be counted from the dependencies listed in the application descriptor XML file and the list of the active services.

When a bundle containing applications is installed and initialized then its applications will also be installed, thus they get revealed to the rest of the system as ApplicationDescriptors. After the successful installation the corresponding subtree must become available.

When a bundle is uninstalled then its applications must also be uninstalled. As the first step of the uninstallation process the corresponding subtree must become unavailable.

5.5.2 Application Instances and Related Commands

Each application instance is represented in the DMT in its own subtree under the following URI

```
        [OSGi_root]/app_instances/<service_pid>
```

where <service_pid> is the persistent identity of the corresponding ApplicationHandle service. The GET and EXEC commands are valid for this node. The EXEC command is to change the lifecycle state of the represented application instance. The EXEC command has the following parameters:

Parameter	Description
`STOP`	Removes the subtree which represents the application instance and destroys the represented application instance by calling the ApplicationHandle.destroyApplication(). These steps must be atomic.

The EXEC command returns successfully (return status of `(200) OK` if the protocol is OMA DM) if the application instance lifecycle transition was successful. Otherwise the command fails (status is `(500) Command` `failed )` which means that the requested lifecycle transition was invalid on the application instance according to the lifecycle state diagram.

The following leaf nodes represent the properties of the application instances under its own node and the valid commands:

Name	Type	Description	Command
`state`	`int`	Represents the lifecycle state of the application instance.	GET: value can be obtained by calling the getApplicationStatus() of the corresponding ApplicationHandle. The meaning of returned value is defined in the API definitions (Application status constants in [10]).
`type`	`Chr`	Refers to the unique id of the application for which this is an instance	GET: the value can be obtained from the corresponding ApplicationHandle's getApplicationDescriptor().getUniqueID ()

After a new application instance is started successfully the subtree of the new instance must be available.

When an application instance is being stopped then just before stopping the corresponding subtree of the DMT must become unavailable.

5.5.3 Example

Let's assume that there is a chess Meglet that has been installed and it has been launched in one instance. Then the following leaf nodes will be available on the DMT for this application and application instance (node names ending with / are interior nodes.):

```
[OSGi_root]/apps/chess_app_id/

       localized_name = "Chess Game"

       version = "1.0"

       vendor = "Acme Inc."

       singleton = "true"

       autostart = false

       locked = false

       bundle_id = "12"

[OSGi_root]/app_instances/123/

       state = 0
```

```
        type = “chess_app_id”
```

6 Considered Alternatives

Each subchapter in the previous chapter contains considerations on alternatives.

7 Security Considerations

8 Document Support

8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. SyncML Device Management Tree Description

[4]. RFC 0078 MEG High-level Architecture

[5]. RFP 0058 Device Management

[6]. RFP 0055 MEG Policy Framework

[7]. RFC 0085 Device Management

[8]. OSGi R3 specification

[9]. RFP 0062 MEG Glossary

[10]. RFC 0091 Application Model

[11]. OMA DM public document repository

[12]. RFC 0084 Monitoring

[13]. RFC 0094 Deployment Configuration

8.2 Author’s Address

Name	Balázs Gődény
Company	Nokia
Address	Hungary, 1092 Budapest, Köztelek u. 6.
Voice	+36209849857
e-mail	balazs.godeny@nokia.com

8.3 Acronyms and Abbreviations

See 0.2.

8.4 End of Document