RFC-141 DMT Admin Extension

Public Draft

63 Pages

Abstract

Different industries are interested in applying OSGi to advance their businesses, in which remote management is a key issue. In the residential area, the TR-069 is the one of the de-facto standard protocol for remote management. The best way to realize remote management based on the TR-069 on OSGi is utilizing DMT Admin service, which has been defined in the OSGi Alliance for the mobile device management. In this case, TR-069 is implemented as a protocol adapter of the DMT Admin. The DMT Admin service, however, is designed mainly for OMA-DM, which is the de-facto standard protocol in mobile area. Although these protocols have the similar objectives and functionality, there are several differences. One of which is the data types definition. Therefore, this RFC defines new data types in the DMT Admin service.

1 Document Information

1.1 Table of Contents

1 Document Information....2
1.1 Table of Contents....2
1.2 Terminology and Document Conventions....3
1.3 Revision History....3

2 Introduction....8

3 Application Domain....8

4 Problem Description....9
4.1 Management agent making use of OSGi standardized Java interfaces....9
4.2 Mobile specification approach....10
4.2.1 Support for TR-069 notifications....10
4.2.2 Limitations in the number of services available in the DMT....11
4.2.3 Mapping TR-069 to the OMA-DM inspired DMT model....12
4.3 Management of overlapping trees....12
4.4 Management of lists....13
4.5 Limitations in Event handling....13
4.6 Conclusion....13

5 Requirements....14

6 Technical Solution....14
6.1 Additional data types....15
6.2 Additional system properties....15
6.3 Scaffold nodes....15
6.3.1 Meta data of scaffold nodes....16
6.3.2 Node type of scaffold nodes....17
6.3.3 Examples....17
6.3.4 Scaffold nodes and Events....18
6.4 Mount points....18
6.4.1 Definitions....18
6.4.2 How to specify mount points?....19
6.4.3 Required checks during plugin mapping....20
6.4.4 API Extensions....21
6.4.5 Mapping Examples....21
6.5 Support for value lists and plugin tables....25
6.5.1 List Subtree Node....25
6.5.2 Plugin tables....26
6.5.3 Examples for plugin tables....27
6.6 Extension for event handling....29

6.6.1 Eventing for internal changes in a plugins subtree 29
6.6.2 Eventing for internal changes of list subtrees 30
6.6.3 Eventing sequence for atomic sessions 31
6.6.4 Destructive Operations 31
6.7 The MountPlugin and MountPoint Interfaces 34
6.7.1 The MountPlugin interface 34
6.7.2 The MountPoint interface 35
6.8 Javadoc 36
6.8.1 Definition of constants for literals 36
6.8.2 Additional Data Types 44
6.8.3 Mount Plugin API 50
6.8.4 DmtException.METADATA_MISMATCH 55
6.8.5 DmtException.COMMAND_NOT_ALLOWED 55
6.8.6 info.dmtree.DmtEvent 55
6.8.7 Interface info.dmtree.SynchronousDmtEventListener 57
changeOccurred 58
6.8.8 Class info.dmtree.Uri 58
6.8.9 Interface info.dmtree.DmtAdmin 59
getSessionCreationTimeout 60
getSessionInactivityTimeout 61

7 Considered Alternatives 61

8 Security Considerations 61

9 Document Support 62
9.1 References 62
9.2 Author's Address 62
9.3 Acronyms and Abbreviations 63
9.4 End of Document 63

1.2 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 9.1.

`Source code is shown in this typeface.`

1.3 Revision History

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

Revision	Date	Comments
Initial	*Sep. 16 2008*	Initial Draft Koya Mori, NTT Corporation, mori.kouya@lab.ntt.co.jp

Revision	Date	Comments
2nd	*Jan. 20 2009*	2nd Draft Generic setter/getter methods are added. Limitation of notification is described. Koya Mori, NTT Corporation, mori.kouya@lab.ntt.co.jp
1.0.0	*Oct. 16 2009*	Public Draft Javadoc and Considered Alternatives are added. Notification property definition is changed. Koya Mori, NTT Corporation, mori.kouya@lab.ntt.co.jp Ikuo Yamasaki, NTT Corporation, yamasaki.ikuo@lab.ntt.co.jp Shigekuni Kondo, NTT Corporation, kondo.shigekuni@lab.ntt.co.jp
1.1.0	*April 3 2010*	For 3rd Public Draft Based on the discussion of REG F2F in Mountain View, the following modifications are done; • abstract was modified. • Requirements which this RFC cannot meet are removed in Section 5. • all descriptions on notifications are removed in Section 6. Section 7.1 is added for future work on notification. Koya Mori, NTT Corporation, mori.kouya@lab.ntt.co.jp Ikuo Yamasaki, NTT Corporation, yamasaki.ikuo@lab.ntt.co.jp
1.2.0	*08.07.2010*	Based on discussions from REG F2F & Bundlefest in Girona : • overlapping of plugins • support for lists • description of mount points • required api extensions Steffen Druesedow, Deutsche Telekom AG
1.4.0	*01.09.2010*	• re-structured chapter 6 • added lots of figures and step-by-step examples • integrated decisions from a number of Bugzilla discussions • added placeholders for results of pending discussions Steffen Druesedow, DTAG, steffen.druesedow@telekom.de

Revision	Date	Comments
1.5.0	*02.09.2010*	Updates according to comments from Ikuo and Evgeni Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.6.0	*03.09.2010*	Updates according to comments from Ikuo and Evgeni. Additionally: • some changes to MountPlugin section (incl. API) • changed start-index of plugins lists to 1 Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.7.0	*10.09.2010*	• Added chapters for format and meta data of scaffold nodes • described retrieval of meta data for list subtree nodes • added definition of several string literals to the Javadoc section • added properties-parameter to the MountPoint.postEvent() methods • added some postEvent related stuff to security section • (still misses other eventing related updates) Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.8.0	*06.10.2010*	• Filled chapter for eventing extensions for subtree lists with latest decisions • summarized discussions about eventing sequence in atomic sessions • changed FORMAT_REFERENCE to FORMAT_NODE_URI, including corresponding API's • added more constants for string laterals to API section Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.9.0	*08.11.2010*	• Fixed a lot of inconsistencies in the Javadocs • added description of URI and SEGMENT related system properties Steffen Druesedow, DTAG, steffen.druesedow@telekom.de

Revision	Date	Comments
1.10.0	*17.11.2010*	• Added statement about immutability of system properties for the URI format during VM lifetime. • changed description of scaffold metadata to table format • added remark about required specification improvents for Eventing in atomic sessions • enhanced description of MountPoint Eventing including required DmtEvent API changes • fixed a number of formatting issues • added description of destructive operation event • rewrote chapter about eventing of internal changes of list subtree nodes Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.11.0	*26,11.2010*	• Ikuos additions to eventing for internal changes of subtrees • minor changes in destructive operations section • clarification added how DmtAdmin detects MountPlugins • added statement about synchronous event delivery in MountPoint section • updated toString() method in Javadoc of DmtData • major update of Javadocs of MountPoint • added DESTRUCTIVE_OPERATIONS to JavaDoc of DmtEvent • updated authors section Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.12.0	*01.12.2010*	• Added DmtData fields for TRUE and FALSE • added String as allowed property type in “How to specify MountPoints?” • updated javadoc of DataPlugin and ExecPlugin to make clear that also String values are allowed for dataRootURIs and execRootURIs • some minor modifications to address latest comments from Evgeni Steffen Druesedow, DTAG, steffen.druesedow@telekom.de

Revision	Date	Comments
1.12.1		• Fixed inconsistency with RFC 149 in description of FORMAT_LONG (section 6.1) • Corrected figures and description for mount point in chapters 6.4.2 and 6.5.3 according to John's comments • repaired some parts of Javadoc in section 6.8.3 Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.13	*14.12.2010*	• Fixed some wording in chapter 6.1 • moved a chapter to a better place in chapter 6.3 • fixed wording in chapter 6.4.3 • added reference to javadocs to chapter 6.4.4 • replaced a number of strings with corresponding constants • fixed lots of typos and formatting • added NullPointerException to methods of MountPoint interface Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.13.1	*16.12.2010*	• Added section about metadata for enumerated plugins to section 6.5.2 (related to bug 1834) • added range-definition for enumeration ids in section 6.5.2 (related to bug 1835) • Modified statement in section 6.6.2 to match agreement in bug 1836 Steffen Druesedow, DTAG, steffen.druesedow@telekom.de
1.13.2	*15.02.2011*	• Changed wording in section 6.5 from “Enumerated Plugins” to “Tables” (related to bug 1854) • fixed types of parameters from Dictionary to Map in postEvent and sendEvent methods in javadoc (related to bug 1891) • added literal constants for session-opening and session-closing to the DmtConstants (related to bug 1890) • added literal constants to info.dmtree.Uri for path separator and root (related to bug 1866) • added property constants and getter methods for session-timeout options (related to bug 1853)

2 Introduction

Traditionally, fixed telecommunication operators don't have knowledge about what runs in the customer's local area network (LAN). They provide connectivity and manage the wide area network (WAN) that provides this connectivity, but they do not know anything about the devices and networks behind the gateway (xDSL mainly) that interconnect WAN and LAN. Recently the need for management of customer networks and devices is increasing in order to make the deployment of new complex services at home (home automation, tele-health, VoIP, IPTV, surveillance, etc) feasible with reasonable costs. For example to avoid sending technicians to the customer premises for solving problems.

There are two main kinds of devices that need to be managed in operator's business: those which come from the fixed business managed through TR-069 and standardized by the Broadband FORUM, and those which come from the mobile business, managed through OMA-DM and standardized by the Open Mobile Alliance. The DMT Admin specification of OSGi covers the OMA-DM ones. In addition, the design of DMT Admin Service specification potentially allows adaption of other remote management protocols other than the OMA-DM.

The best way to realize remote management based on TR-069 on OSGi is utilizing the DMT Admin service. In this case, TR-069 is implemented as a protocol adapter of the DMT Admin. However, there are several architectural differences between the TR-069 and the OMA-DM, although these two protocols have the similar objectives and functionality.

3 Application Domain

Driven by triple play service delivery in the home network, fixed line access service providers have the need to configure home devices to ensure the proper service delivery. Broadband Forum's CPE WAN Management Protocol (CWMP, alias TR-069) enables them to do this. By using a remote management server (Auto Configuration Server, ACS), they are able to manage TR-069 enabled devices. TR-069 provides them with possibilities to configure parameters, be informed of parameter changes (notification), start diagnostic tools, update firmware, etc.

Similarly, for the mobile world, the OMA defined the OMA-Device Management specification for remote management of mobile devices. OMA-DM offers similar tools to the mobile service providers as TR-069 to fixed line service provider, but OMA-DM is of course tailored to the specifics of the mobile environment.

As OSGi technology offers a flexible software environment for all these different devices, the remote management of the platform is of interest for both fixed and mobile service providers. As such, it should be possible to integrate the remote management of the OSGi platform, and the applications running on top of it, in the existing management infrastructure.

The DMT Admin service with its mobile management tree in the Mobile specification for OSGi R4 standardizes the remote management of an OSGi platform. As it is largely inspired by OMA-DM, it needs to be evaluated for multi protocol support.

4 Problem Description

In a scenario in which service providers offer a growing number of services, to use specific solutions for the management of those services is not the most suitable option. To speed up the deployment of these services, such as triple play, home automation or tele-health, it is essential to offer general management solutions that allow for the management of a large number of services and the flexible life-cycle management of applications.

These devices usually are already managed by a standard protocol, so it makes sense that an OSGi framework, which hosts the services, running on a device could be managed in the same way as the other resources of the device. Of course, the remote management should be fully integrated in the existing remote management solutions of the service provider to avoid duplicating management infrastructure and to increase performance on the devices.

Currently, there are two options in OSGi for remote management:

- create a management agent bundle making use of the Java object interfaces,

- create a protocol adapter bundle that interacts with the DMT Admin service, as defined in the OSGi Mobile specification.

4.1 Management agent making use of OSGi standardized Java interfaces

Currently, for the management of a bundle, the OSGi specifications define different Java objects with which a management application can interact. Using this approach, a management agent can implement extensive management of the OSGi framework, as well as any service standardized. Mapping the Java interfaces to the specific remote management protocol and data model tree is up to the management agent.

For runtime interaction with a bundle, a bundle can register a service in the service registry. However, this service interface is not standardized. Also, mapping the service interface to a general management model is not standardized. A current approach is to implement a proprietary service interface on all bundles to be managed. By tailoring this interface so that it easily maps to the management protocol primitives, it is simple for the management agent to map remote management commands to the bundle’s service interface. The disadvantage is the proprietary service interface, so that 3rd party bundles might not be compliant.

As a conclusion we can say that this current approach allows for extensive remote management of any aspect of the OSGi platform, but lacks a standardized service interface definition for bundles to implement.

4.2 Mobile specification approach

The Mobile Expert Group has provided its own solution based on the OMA [3] Device Management [4] specification to provide a remote management solution. The OSGi Mobile specification contains two chapters related to remote management:

- chapter 3: detailing the mobile management tree
- chapter 117: detailing: the DMT Admin service, bundle plugin interface specification, the notification service

The Device Management Tree model of OMA-DM was chosen as meta-data model and operational model. However, it was intended to be mappable to other protocols.

An analysis of mapping the Mobile specification DMT model to TR-069, however, shows that the current DMT model approach (as defined in the OSGi R4 Mobile Specification) introduces some issues. For example:

- Limitations for active or passive notifications on any parameter in the object tree
- A limited number of services have been mapped to the DMT model
- The complexity of mapping a new protocol to the OMA-inspired DMT model, which could imply performance issues on limited devices.

4.2.1 Support for TR-069 notifications

TR-069 offers the feature of active and passive notifications. By setting a parameter's notification attribute, a remote manager requests to be notified with the parameter's new value at the time the value changes (active notification) or at the next periodic inform (passive notification). Notification can be configured on any parameter of the TR-069 object tree. This approach enables the remote manager to be informed not only of changes in status variables of the platform, but also of configuration changes performed by a local manager, e.g. through a local Web interface.

The Mobile specification offers a few features that could help to implement TR-069 notification support:

- The DMT Admin service sends events using the Event Admin service when operations have been performed on nodes (nodes added, removed or copied; node values changed etc.)
- The OSGi Notification service defines a way to alert a remote management server. Protocol adaptors on their turn have to implement a RemoteAlertSender interface (and register it) for use by the notification service. Notifications are sent by calling sendNotification on the notification service:
- The Monitor Admin service: A bundle can register a Monitorable service, to be used by the Monitor Admin service. By registering a Monitorable service, the bundle exposes access to a number of status variables. Notification can be implemented by the StatusVariable provider. If it does, it will call the update method on the Monitor Listener. The Monitor Admin service then generates an event on the Event Admin service. The Monitor service is currently also represented in the DMT.

Two problems arise when trying to map the current approach to TR-069:

- TR-069 defines that notification is applicable to any parameter in the object tree.

  Currently, the DMT Admin service only send events for operations on DMT nodes that were performed using the DMT Admin API. For example: if configuration changes are performed by using the Configuration Admin service API, no events will be sent. Most of the current implementations do not perform all changes via the DMT Admin service. Therefore, the events sent by the DMT Admin service are an only subset and thus not very reliable as single source of events (and thus as single source of TR-069 notifications).

The OSGi Monitor service only supports notification of changes on Status Variables, exposed through a Monitorable service, and enabled by the bundle to support on-change notification (i.e. dynamic Status Variables).

- Requesting notification is not fully under the control of the remote manager. In the case of a bundle using the notification service, there is no standardized way to configure the bundle to send alerts when the value of one of the implemented DMT nodes changes. In the case of the monitor service, the sending of events can be controlled, but is limited to dynamic Status Variables.

The current DMT Admin service has no attributes properties on its nodes to be used to configure notification behavior, such as active notification and passive notification defined in TR-069. Therefore, a remote manager cannot control the notification behavior of DMT nodes in a standardized way.

To conclude, the current options, as provided in the Mobile specification, limit notification of parameter changes to StatusVariables, explicitly enabled for monitoring. There is no standardized approach available to monitor changes on any node in the DMT.

4.2.2 Limitations in the number of services available in the DMT

The OSGi R4 Mobile specification mapped a number of services to the DMT. However, these services are limited to the services listed in the Mobile Specification. Other interesting services, as listed in the OSGi R4 Service Compendium are not yet mapped (standardized) in the DMT:

The DMT defined in the Mobile Specification contains objects for the following services:

- Configuration Admin service
- Log service
- Monitor Admin service
- Application Admin service
- Conditional Permission Admin service
- Deployment service

A number of areas that could be of interest to a remote manager are currently missing in the DMT:

- Startlevel management
- Bundle management: managing individual bundles as opposed to deployment packages (inventory, life-cycle management, exported services, …)
- Service management: getting a remote view on services registered in the service registry
- Permission Admin management

● Home Gateway Core Function management: handling Home Gateway core functions, such as firewall configuration and port forwarding control, from bundles running on an OSGi framework

4.2.3 Mapping TR-069 to the OMA-DM inspired DMT model

Within the OSGi Mobile specification, the choice has been made to model the DMT after OMA-DM.

As a result, creating an OMA-DM protocol adapter is quite straightforward. Although no major hurdles have been identified in creating a TR-069 protocol adapter, it is less straightforward:

1. The TR-069 RPC primitives have to be translated to the DMT Admin service interface methods (which are OMA-DM RPC inspired).

2. The TR-069 tree has to be mapped to the DMT. Translating object model specific features like DMT meta nodes, or TR-069 attributes is not straightforward. It might require specific extensions to the DMT, e.g. to support TR-069 attributes, etc. the TR-069 data types have to be mapped to the DMT Admin data types. However, TR-069 data types, such as “unsignedInt” and “dateTime” (ISO 8601), cannot be translated appropriately into DMT Admin data types defined in the current specification. Translating these data types might result a limitation of the available value range and a complex object that consists of multiple nodes, respectively.

4.3 Management of overlapping trees

The DMT Admin has the restrictions that subtrees of configuration data cannot overlap. A DMT Admin Data Plugin that was registered to manage, for example, the sub-tree “*./InternetGatewayDevice*” receives all requests to data objects below that name. It is not allowed to register a second Data Plugin that, for example, manages the sub-tree “*./InternetGatewayDevice/wan”.* Because of this restriction services which register later in time are not allowed to register a Data Plugin to manage the tree or any sub-tree below “*./InternetGatewayDevice*”. Figure 1 illustrates overlapping subtrees, one of an IGD Data Plugin and the overlapping one, the Vendor IGD Driver Data Plugin that can manage the data model for *SomeVendorDevice*.

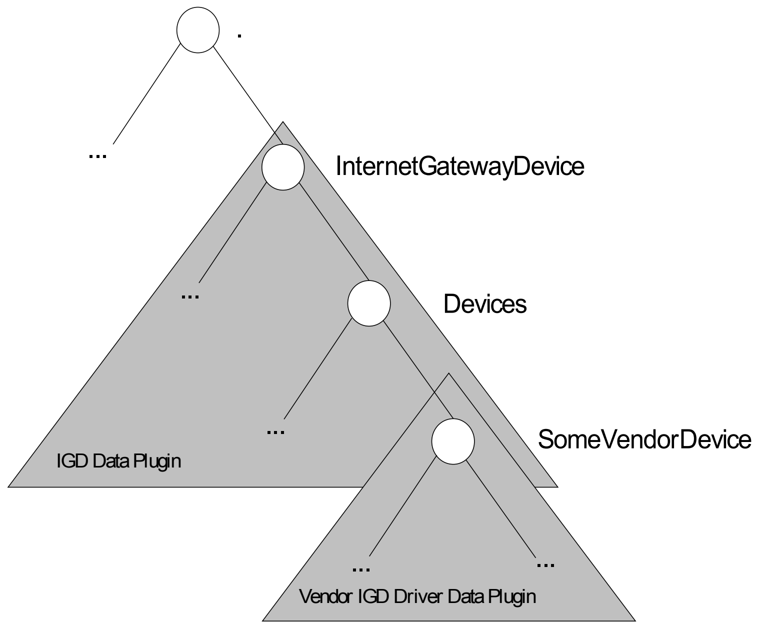


Illustration 1: Overlapping Subtrees in DMT Admin

A bundle provider can of course register the Data Plugin so that it just registers itself somewhere in the DMT Admin's configuration tree. But in this case, any Management Service that likes to set or access configuration values to or from that bundle does not know the value's path in the configuration tree.

4.4 Management of lists

Data models like TR-098 require the ability to map lists of entries to the DMT. The node "./InternetGatewayDevice.LANDevice.{i}." is an example for that. There can be more than one LANDevice present in the tree, each of them identified by its own unique id.

The DMT Admin has no mechanism specified that provides an automatic generation and management of such unique identifiers to avoid potential name-clashes between different plugins. Such a mechanism must also ensure that a plugin that registers itself gets the same index again, even in case that it was uninstalled/unregistered and then registered again. The mapping of the unique ids to the plugins should be persistent.

4.5 Limitations in Event handling

Another restriction concerns the raising of events. The DMT Admin only raises events when a part of the managed configuration is changed via a Data Plugin. A bundle that likes its configuration been managed by the DMT Admin and therefore registers a Data Plugin cannot use the DMT Admin to notify interested bundles if some part of its configuration has changed through other means (low-level event). An example for this is that in an IGD the connection on the WAN interface was established or disconnected. This event arises in the IGD Core Functions layer. The responsible driver bundle can reflect this in its internal configuration mapping, but there are no means to trigger the DMT Admin to raise an event.

4.6 Conclusion

The OSGi Mobile specification delivers a standardized data model (the DMT), and standardized interface (on the DMT Admin service) to enable remote management through a protocol adapter. However, the current specification lacks management objects for a number of interesting areas. Also, there is some support lacking for TR-069 notifications. Furthermore, since the DMT model is OMA-DM inspired, implementing a TR-069 protocol adapter is not straightforward, although not impossible.

5 Requirements

REQUIREMENT[9]: The solution should be able to handle the data types of “unsignedInt”, “unsignedLong”, “long”, “hexBinary”, “dateTime” (ISO 8601) and “nodeURI” to accommodate TR-069 protocol.

REQUIREMENT[10]: The solution must allow overlapping DataPlugins and overlapping ExecPlugins.

REQUIREMENT[11]: The solution must provide a mechanism to generate and manage unique ids for list entries.

REQUIREMENT[12]: The solution must allow DataPlugins to send Events for changes in its managed subtree data that where not performed via a DmtSession.

6 Technical Solution

This RFC describes how DMT Admin service is extended to accommodate the TR-069 protocol. Since DMT Admin service intends to use the OMA-DM protocol as a remote management protocol, some key features such as data types are missing which makes it difficult to support the TR-069 protocol.

This RFC describes the following changes to DMT Admin service.

- Addition of new data types to DMT Admin service
- Definition of scaffold nodes
- Definition of mount points and overlapping of trees
- Mechanism to ~~enumerate~~ map tables of plugins
- Mechanism to handle list subtree nodes
- Extensions for event handling

- minor API-changes for DmtException and

There is no need to modify the basic architecture.

6.1 Additional data types

Some data types are missing in the DMT Admin specification and are needed to accommodate the TR-069 protocol. The following data types should be added to the format of the DmtData class.

- FORMAT_LONG – A long value for general purpose..

- FORMAT_DATETIME – A String object that is interpreted as the *dateTime* type defined in ISO 8601. it is used as the value of date and time in TR-069.

- FORMAT_UNSIGNED_LONG – A String object that is interpreted as an unsigned long value. It is used as the value of unsigned long in TR-069.

- FORMAT_UNSIGNED_INTEGER– A String object that is interpreted as an unsigned integer value. It is used as the value of unsigned int in TR-069.

- FORMAT_HEXBINARY – A byte[] value. The node holds an hex binary value. The value of the node corresponds to the Java `byte[]` type.

- FORMAT_NODE_URI – A string value representing a reference to a uri.

6.2 Additional system properties

The following OSGi environment properties are used to specify the following tree parameters:

- org.osgi.dmtree.max.segment.name.length
  This property specifies the maximum allowed length of a URI segment. The value can be requested at runtime via Uri.getMaxSegmentNameLength().

- org.osgi.dmtree.max.uri.length
  This property specifies the maximum allowed length of a URI. The value can be requested at runtime via Uri.getMaxUriLength().

- org.osgi.dmtree.max.uri.segments
  This property specifies the maximum allowed number of Uri segments. The value can be requested at runtime via Uri.getMaxUriSegments().

The above values are immutable after their first initialization. Any subsequent changes of the the system properties after their first initialization are ignored.

This RFC does not define any mechanism to specify default or minimum values for the corresponding URI attributes. This is considered as implementation specific..

6.3 Scaffold nodes

Plugins can be mapped anywhere into the DMT. An ancestor node of a plugin does not necessarily need to exist at the time of registration.

Nevertheless, the DMT Admin must ensure that browse operations in the tree (e.g. starting from the root node ".") are always possible and all mapped plugins are reachable at any point in time.

In order to ensure this, the DMT Admin must "emulate" these missing nodes. Such emulated nodes are called "scaffold" nodes. All operations of an info.dmtree.spi.ReadableDataSession must be possible on a scaffold node. Any other operation is forbidden and must lead to a DmtException of error code COMMAND_NOT_ALLOWED.

6.3.1 Meta data of scaffold nodes

Scaffold nodes are just "structural" nodes, which are emulated by the DMT Admin to allow sophisticated mappings of plugins in the DMT. No data plugin is directly responsible for them. That's why it is the DMT Admin who must provide the meta data for them.

Following is a description of the values that the meta data for a scaffold node should provide:

Operation	Returned value
can()	CMD_GET
isLeaf()	False
getScope()	MetaNode.PERMANENT
getDescription()	null
GetMaxOccurrence()	1
`isZeroOccurrenceAllowed()`	true
`getDefault()`	null
`getValidNames()`	null
`getValidValues()`	null
`getMax()`	Double.MAX_VALUE
`getMin()`	Double.MIN_VALUE
`getRawFormatNames()`	null
`getMimeTypes()`	null
`isValidName()`	true
`isValidValue()`	false
`getFormat()`	DmtData.FORMAT_NODE

NOTE:
As shown and described later in chapter 6.3 it is possible that the root of a newly registered plugin is mapped to a scaffold node. In such a case, the meta data for this node will not be provided by the DMT Admin anymore but rather by the newly mapped plugin.
Please refer to chapter 6.4.5 in part "Mounted Plugin registered before Mounting Plugin" for an example and a more detailed description.

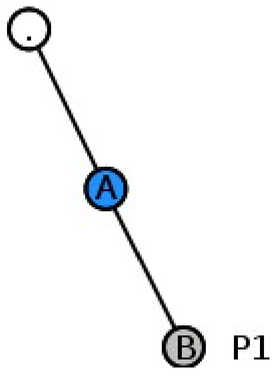
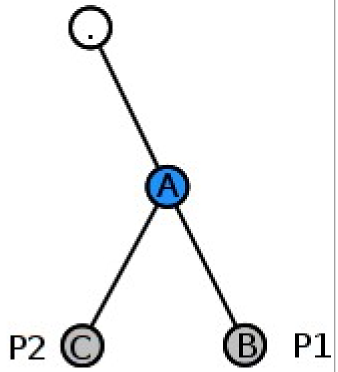
6.3.2 Node type of scaffold nodes

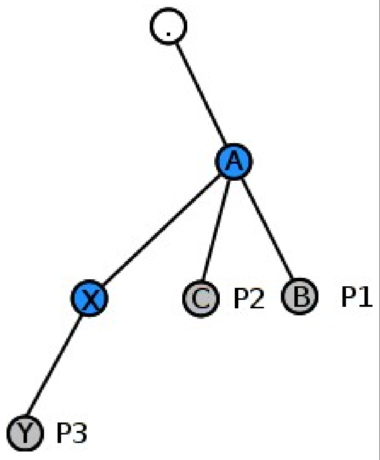
The node type of scaffold nodes must be provided by the DMT Admin as URI info.dmtree.spi.DmtConstants.DDF_SCAFFOLD. Therefore it is not allowed and possible that scaffold nodes are list subtree nodes (see also 6.5.1).

NOTE:
The same remark as for meta data is valid also for the node type. As soon as another plugin is mapped to the scaffold node, this plugin takes over the responsibility for this URI and can return another node type.

6.3.3 Examples

The following table below shows some exemplary cases. The dataRootURIs in the examples indicate under which URIs the Data Plugins (P1, P2, etc) are registered.

dataRootURIs	**Illustration**	**Description**
P1: ./A/B		This example maps the root of data plugin P1 to URI ./A/B. The node with URI ./A is a scaffold node. An invocation of getChildNodeNames() on URI ./A must return [“B”].
P1: ./A/B P2: ./A/C		This example maps the root of data plugins P1 to URI ./A/B and P2 to URI ./A/C The node with URI ./A is a scaffold node. An invocation of getChildNodeNames() on URI ./A must now return [“B”, “C”].

P1: ./A/B P2: ./A/C P3: ./A/X/Y		This example maps the root of data plugins P1 to URI ./A/B, P2 to URI ./A/C and P3 to URI ./A/X/Y The nodes with URIs ./A and ./A/X are scaffold nodes. An invocation of a getChildNodeNames() operation on URI ./A must now return [“B”, “C”, “X”]. An invocation of a getChildNodeNames() operation on URI ./A/X must return [“Y”].

Scaffold nodes must be removed by the DMT Admin as soon as they are not required anymore, i.e. when all plugins were unregistered that caused the creation of the scaffold node.

6.3.4 Scaffold nodes and Events

The DmtAdmin must not send Dmt Events when scaffold nodes are created or deleted.

6.4 Mount points

Allowing overlapping subtrees can easily lead to quite complex situations. The mount point mechanism is an approach to limit this complexity and to provide a level of control to the implementers of plugins.

The main idea behind mount points is that only a plugin itself has the knowledge and the right to define the points in its own subtree where other plugins are allowed to “mount” themselves.

The described mount point mechanisms apply for Data Plugins as well as for Exec Plugins.

6.4.1 Definitions

Mount Point:
A mount point is a URI relative to the plugin's root. It must not start with a “.” or “/”.

Mounting Plugin:
A mounting plugin is a plugin that defines mount points in its own subtree.

Mounted Plugin:
A mounted plugin is a plugin that is mapped (mounted) to one of the defined mount points of a mounting plugin.

Ancestor Plugin:
A plugin P1 is an ancestor plugin of a plugin P2 if P1it is mapped to a URI, which is an ancestor of P2's root URI.

Descendant Plugin:

A plugin P2 is a descendant plugin of a plugin P1 if P2's root URI is part of the descendants of P1's root URI.

Root Plugin:
The root plugin is a “virtual” plugin which is implicitly mapped to the URI “.”. The root plugin does not specify any mount points. A mapping of a plugin is always possible, if its only ancestor plugin is the root plugin.

A Plugin can be a mounting and mounted plugin at the same time. That means it can specify own mount points while being mounted in another plugin's subtree.

6.4.2 How to specify mount points?

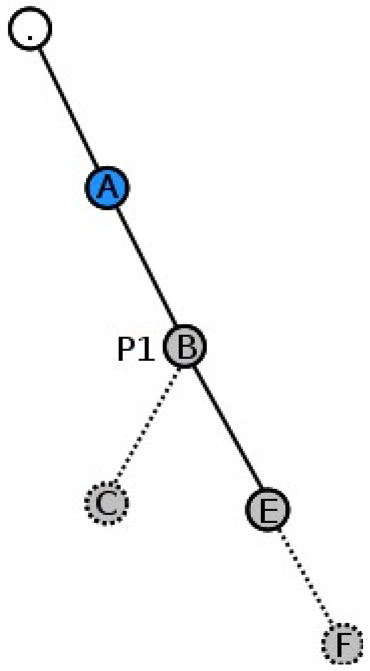
Mount points are specified as an optional property during the registration of a plugin. As name for this property the constant info.dmtree.spi.MountPlugin.MOUNT_POINTS must be used. The value of this property is a String or an array of Strings, containing the mount point URIs relative to the plugin's dataRootURI (its own “root”).

NOTE:
In order to avoid assignment problems, mount points are only allowed if the plugin was registered with exactly one one uri in dataRootURIs or execRootURIs. If this is not the case and the plugin was registered with the “mountPoints” property then the plugin registration must be ignored and an error must be logged.

Code Example:

```
Dictionary props = new Hashtable(2, 1f);
props.put( info.dmtree.spi.DataPlugin.DATA_ROOT_URIS, “./A/B” );
props.put( info.dmtree.spi.MountPlugin.MOUNT_POINTS, new String[] {“C”, “E/F”} );
context.registerService( DataPlugin.class.getName(), this, props );
```

This code registers a Plugin at URI ./A/B and specifies that other plugins are allowed to mount themselves under the (absolute) URIs ./A/B/C and ./A/B/E/F in its own subtree.

Illustration	Description
	A plugin (P1) with dataRootURI ./A/B was mapped. The plugin has specified a mount point “C” and “E/F”. These two nodes are the only points in P1's subtree where other plugins can be mapped. The node with uri ./A/B/E is under control of the plugin P1. NOTE that these specified mount points are just markers to help the DMT Admin. They are not visible while browsing through the tree. Therefore a call to the getChildNodeNames() method on the node with URI ./A/B would return just [E] and the child nodes of node with uri ./A/B/E would be an empty array, if no Mounted Plugins for the mount points exist.

6.4.3 Required checks during plugin mapping

When a plugin is registered, then the DMT Admin has to perform several mount-point-related checks before the plugin can be mapped into the DMT.

Do the plugin roots point to valid mount points of other plugins?

Does the plugin agree with the locations of all direct descendant plugins? The DMT Admin must check that all already mapped descendant plugins conform to the new plugins' mount points.

The DMT Admin must perform following checks for each dataRootURIs of the plugin:

1. Is there already an ancestor plugin mapped except the root plugin ?

   a) if yes: then continue with step 2).

   b) If not: then continue with step 3).

2. Does the nearest ancestor plugin (i.e. the first mapped plugin that is found, when the tree is traversed upwards from the plugins root) define a mount point that is equal to the plugin root?

   a) if yes: then continue with step 3).

b) if not: then the plugin cannot be mapped to this root. This plugin root is ignored and an error is logged.

3. Are there any descendant plugins for the plugin root?

    a) if yes, then continue with step 4).

    b) if not, then the plugin can be mounted a the given root without further checks.

4. Does the plugin specify any mount points?

    a) if yes, then continue with step 5).

    b) if not, then this plugin root must be ignored, because it would otherwise tolerate another plugin that it does not agree with. In this case an error must be logged.

5. Is the root of each descendant plugin matching any of the specified mount points?

    a) if yes, then the plugin can be mounted at the given root without further checks.

    b) if not, then this plugin root must be ignored, because it would otherwise tolerate another plugin that it does not agree with. In this case an error must be logged.

NOTE:

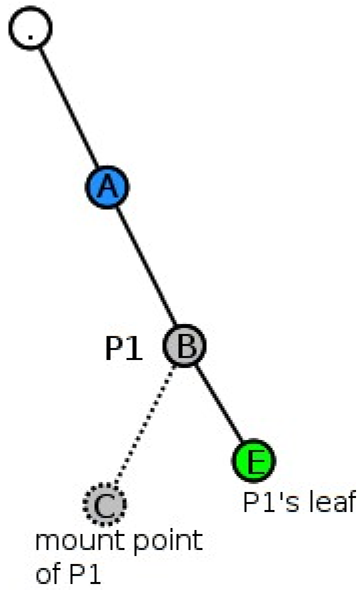
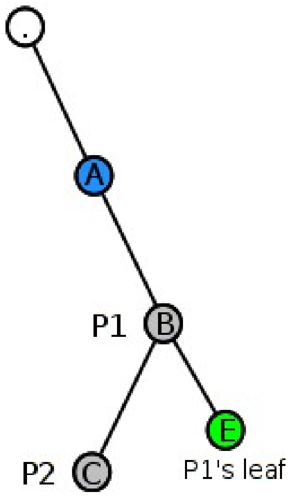
If a plugin is registered with more than one roots at once, then the above checks are executed for each root individually. Only those roots which failed the check are ignored and lead to an error log.

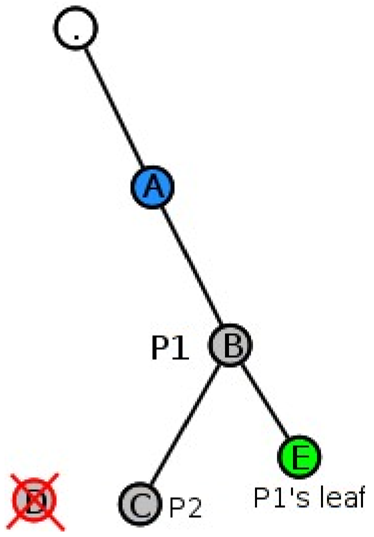
6.4.4 API Extensions

In order to support the mount point mechanism some API extensions have been made. Please refer to chapter 6.8.3 Mount Plugin API for details.

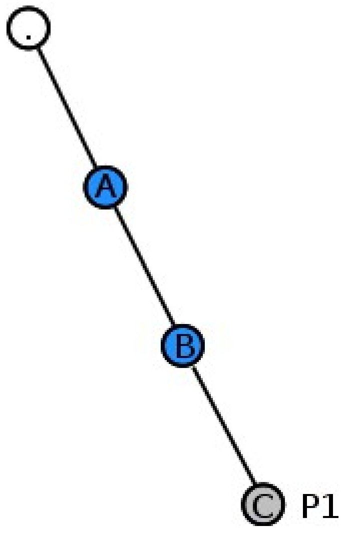
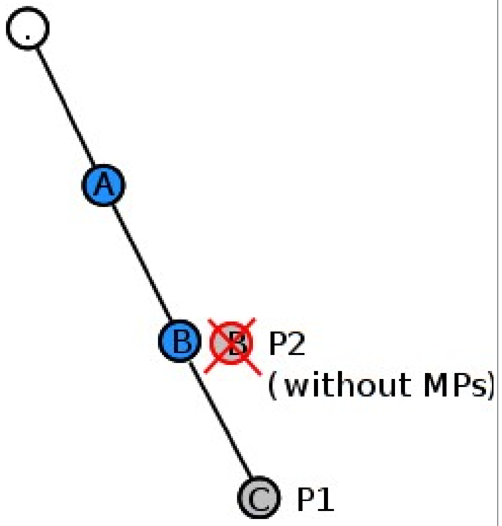
6.4.5 Mapping Examples

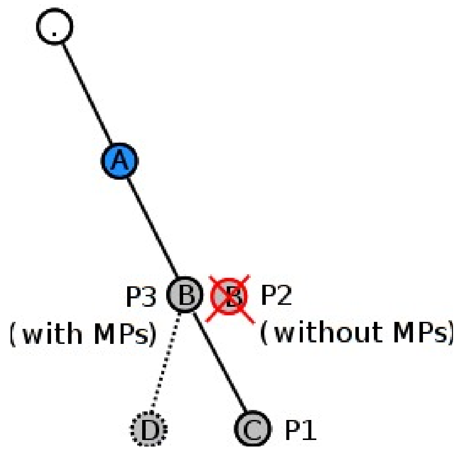
Mounting Plugin registered before Mounted Plugin:

Step No.	Illustration	Description
1	 . A P1 B C E P1's leaf mount point of P1	A plugin P1 is registered with: P1: dataRootURIs: [./A/B] mountPoints: [C] The Plugin P1 is mapped to URI ./A/B. The node with URI ./A is a scaffold node. The node C is defined as a mount point. The node with URI ./A/B/E is a leaf node, managed by P1.
2		A second plugin P2 is registered with: P2: dataRootURIs: [./A/B/C] mountPoints: [ ] The plugin P2 is successfully mapped, because its root URI corresponds to a mount point of P1, i.e. it is valid and P1 agrees with it.

Step No.	Illustration	Description
3		A third plugin P3 is registered with: P3: dataRootURIs: [./A/B/D] mountPoints: [ ] The Plugin is not mapped, because there is no corresponding mount point for the URI ./A/B/D.

Mounted Plugin registered before Mounting Plugin:

Step No.	Illustration	Description
1		A plugin P1 is registered with: P1: dataRootURIs: [./A/B/C] mountPoints: [ ] The Plugin P1 is mapped to URI ./A/B/C, because it has no other ancestor than the root node. The nodes with URIs ./A and ./A/B are scaffold nodes.
2		A second plugin P2 is registered with: P2: dataRootURIs: [./A/B] mountPoints: [ ] The plugin P2 is not mapped, because it has not specified C (the root of P1) as mount point, i.e. it does not tolerate/agree with P1. That's why it must be ignored and an error must be logged.

Step No.	Illustration	Description
3	 A P3 B B P2 (with MPs) (without MPs) D C P1	A third plugin P3 is registered with: P3: dataRootURIs: [./A/B] mountPoints: [C, D] The Plugin is successfully mapped, because it defines the mount point C and therefore agrees with the location of the already mapped plugin P1. Furthermore it defines a second mount point D. NOTE: Until step 2 the node with URI ./A/B is a pure scaffold node. The meta data for this node was therefore provided by the DmtAdmin so far. Now P3 was mapped successfully to ./A/B. From now on this node is P3's root as well as a scaffold node for P1, otherwise P1 would not be “reachable” anymore. The meta data for node with URI ./A/B however, will now be provided by plugin P3 and can therefore differ from the meta data for the same URI in step 2.

6.5 Support for value lists and plugin tables

6.5.1 List Subtree Node

List Subtree Node consist of:

- an interior node that must return the URI info.dmtree.spi.DMTConstants.DDF_LIST_SUBTREE (“org.osgi/1.0/ListSubtree”) as result of its ReadableDataSession.getNodeType method. This URI indicates that this node should be treated as a list of values by a protocol adapter or a local manager.
- a number of leaf nodes with the same meta data, which represent the individual entries of the list
- NOTE: Only leaf nodes are allowed as child nodes of the List subtree node.

The meta data of the leaf nodes should be provided as follows:

- a data plugin that wants to support a list subtree node should ensure that meta data for the child leaf nodes of the list are supported, even if no child leaf exists at this point in time.
- In order to retrieve the meta data for a child leaf node of a list, ANY name can be used as the leaf node name.
  Example: in order to get the meta data for the elements of the list subtree node with URI ./A/B all following invocations of the getMetaNode() method are valid:
  - getMetaNode(new String[] { “.”, “A”, “B”, “0” }

- getMetaNode(new String[] { ".", "A", "B", "anyName" }
- getMetaNode(new String[] { ".", "A", "B", "whatever" }

When changes to the nodes of internal lists are applied in an atomic session, events will be fired by the DmtAdmin as described in 117.14.6 of the DmtAdmin specification.

6.5.2 Plugin tables

This sections describes a mechanism that allows different plugins to register with the same root-uri and nevertheless be successfully mapped to the DMTree. Without such an extension, just the first registered root-uri would be mapped and all subsequent registrations would be ignored.

The DmtAdmin therefore creates and maintains an index for each of these root-uris and appends this index to the uri before it is mapped, i.e. it maintains a table of plugins with identic root-uris. In order to indicate to the DMT Admin that a data plugin registers itself as an element of a~~n enumeration~~table, an extended syntax rule for the root-uri~~dataRootURIs~~ is introduced. (Note: The term "root-uri" stands here for dataRootURIs as well as execRootURIs, indicating that this mechanism is valid for DataPlugins as well as ExecPlugins.) In such cases the ~~dataRootURIs~~root-uri must end with a "#" as placeholder for the index. Note, that "#" as table~~list~~ indicator is only allowed as last character in the root-uri~~dataRootURIs~~.

For instance ~~Therefore~~ a plugin that wants to register itself as element of the table~~list~~ "InternetGatewayDevice.LANDevice.{i}." in TR-098 has to use the dataRootURIs "./InternetGatewayDevice/LANDevice/#". (see examples section below)

NOTE: This syntax must also be used for the definition of mount points to indicate that the mount point is a table~~list~~. Also here the "#" is allowed only as last character of the URI.

Persistence of assigned IDs

The DMT Admin is responsible to assign a unique id and to ensure that the same id is re-used in case that the same plugin is "known" already. This id consists of the String representation of an integer value n with n >= 1 and n < Integer.MAX_VALUE.

A plugin is considered as "known" if it uses a known value for the optional registration property "service.pid". If a plugin registers without this property then the DMT Admin assigns a free unique id (an id that is not assigned to a currently mapped plugin and not persistently stored yet), but then this id-assignment will not be made persistent. If this plugin is later-on unregistered and registered again, it might get a different id.

Metadata for the root of ~~enumerated~~ plugins that are elements of a table

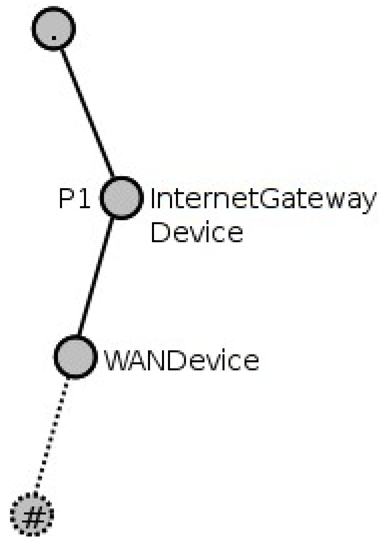
It is the responsibility of the plugin to provide the correct metadata for its own root node, if this root is part of a~~n enumeration~~table. The data plugin must ensure that the operations of the MetaNode return the correct values as shown in the following table.

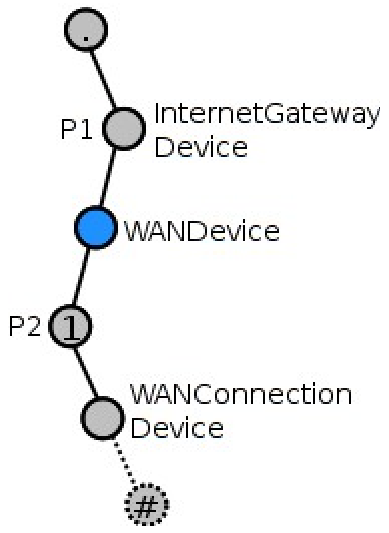
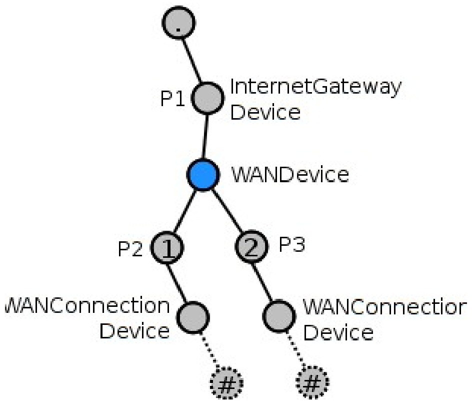
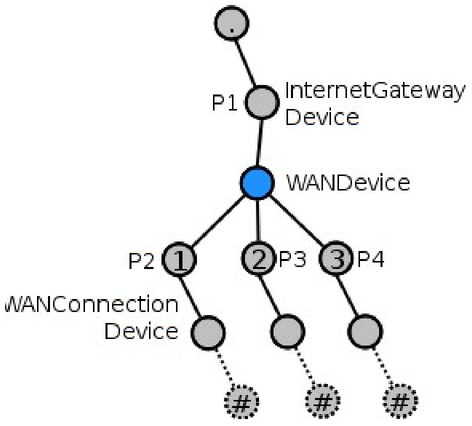
Operation	Returned value
can()	CMD_GET

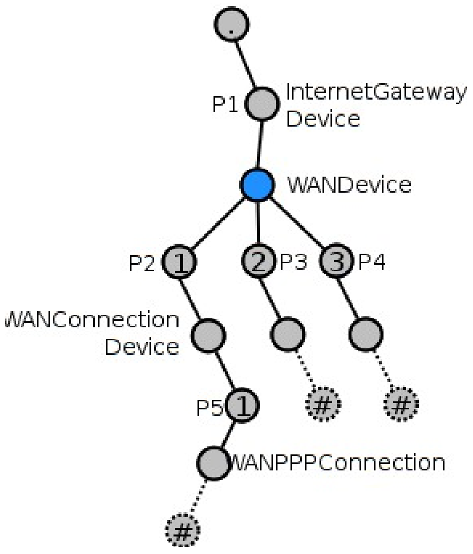
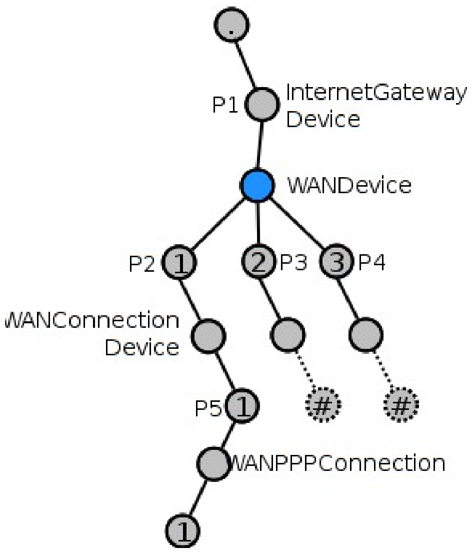
isLeaf()	false
getScope()	depends on tree structure
getDescription()	responsibility of the plugin
GetMaxOccurrence()	Integer.MAX_VALUE
`isZeroOccurrenceAllowed()`	true
`getDefault()`	null
`getValidNames()`	null
`getValidValues()`	null
`getMax()`	Double.MAX_VALUE
`getMin()`	Double.MIN_VALUE
`getRawFormatNames()`	null
`getMimeTypes()`	null
`isValidName()`	Only valid integer values are allowed
`isValidValue()`	false
`getFormat()`	DmtData.FORMAT_NODE

6.5.3 Examples for plugin tables

The following examples illustrate the use of the table~~list~~ indicator "#" in the dataRootURIs and shows the resulting mappings on an example taken from the TR-098 data model.

dataRootURIs and mount points	**Illustration**	**Description**
P1: (IGD - Plugin) **dataRootURI:** ./InternetGatewayDevice **mount point:** WANDevice/#		This example maps the root of data plugin P1 to URI ./InternetGatewayDevice. P1 furthermore specifies the mount point WANDevice/#, indicating that other plugins are allowed to occupy this place as elements of a list. The node with uri ./InternetGatewayDevice/WANDevice is still under control of P1.

dataRootURIs and mount points	**Illustration**	**Description**
P2: (DSL – Plugin 1) **dataRootURI:** ./InternetGatewayDevice/WANDevice/# **mount point:** WANConnectionDevice/#	 . P1 InternetGateway Device WANDevice P2 1 WANConnection Device #	Plugin P2 is mapped as first list element to P1's mount point. The index 1 has automatically been assigned by the DMT Admin. The resulting root URI of P2 is then: ./InternetGatewayDevice/WANDevice/1 P2 furthermore specifies the mount point WANConnectionDevice/#, indicating that other plugins are allowed to occupy this place as elements of a ~~list~~table. The node with uri ./InternetGatewayDevice/1/WANConnection Device is under control of P2.
P3: (DSL – Plugin 2) **dataRootURI:** ./InternetGatewayDevice/WANDevice/# **mount point:** WANConnectionDevice/#	 . P1 InternetGateway Device WANDevice P2 1 2 P3 WANConnection Device WANConnection Device # #	Plugin P3 is mapped as second list element to P1's mount point. The index 2 has automatically been assigned by the DMT Admin. The resulting root URI of P3 is then: ./InternetGatewayDevice/WANDevice/2 P3 furthermore specifies the mount point WANConnectionDevice/#, indicating that other plugins are allowed to occupy this place as elements of a ~~list~~table.
P4: (ETH – Plugin) **dataRootURI:** ./InternetGatewayDevice/WANDevice/# **mount point:** WANConnectionDevice/#	 . P1 InternetGateway Device WANDevice P2 1 2 P3 3 P4 WANConnection Device # # #	Plugin P4 is mapped as third list element to P1's mount point. The index 3 has automatically been assigned by the DMT Admin. The resulting root URI of P4 is then: ./InternetGatewayDevice/WANDevice/3 P4 furthermore specifies the mount point WANConnectionDevice/#, indicating that other plugins are allowed to occupy this place as elements of a table~~list~~.

dataRootURIs and mount points	Illustration	Description
P5: (ATM – Plugin) **dataRootURI:** ./InternetGatewayDevice/WANDevice/1/WANConnectionDevice/# **mount point:** WANPPPConnection/#	 . P1 InternetGateway Device WANDevice P2 1 2 P3 3 P4 WANConnection Device P5 1 # # WANPPPConnection #	Plugin P5 is mapped as first list element to P2's mount point. The index 1 has automatically been assigned by the DMT Admin. The resulting root URI of P5 is then: ./InternetGatewayDevice/WANDevice/1/WANConnectionDevice/1 P5 furthermore specifies the mount point WANPPPConnection/#, indicating that other plugins are allowed to occupy this place as elements of a table~~list~~.
P6: (PPP – Plugin) **dataRootURI:** ./InternetGatewayDevice/WANDevice/1/WANConnectionDevice/1/WANPPPConnection/# **mount point:** []	 . P1 InternetGateway Device WANDevice P2 1 2 P3 3 P4 WANConnection Device P5 1 # # WANPPPConnection 1	Plugin P6 is mapped as first list element to P5's mount point. The index 1 has automatically been assigned by the DMT Admin. The resulting root URI of P6 is then: ./InternetGatewayDevice/WANDevice/1/WANConnectionDevice/1/WANPPPConnection/1

6.6 Extension for event handling

6.6.1 Eventing for internal changes in a plugins subtree

Changes that influence the state of a plugins subtree can not only occur through a DmtSession but also by “low level” changes on the plugins data model. There can - for instance - new devices be attached which must be reflected in the data plugins subtree.

In the existing specification there is no mechanism defined, that allows a plugin to notify such internal changes to interested entities.

This RFC defines a mechanism that also supports internal events. The main difference of internal events compared to normal session based ones is, that internal events don't have an attached session id, because they are caused outside of any session.

The details of the mechanism are described in chapter 6.7.2 The MountPoint interface.

It is strongly recommended that DataPlugins use the postEvent() methods of the MountPoint interface to notify internal changes. Using this interface is the only way to ensure that also DmtEventListeners are able to receive such events.

6.6.2 Eventing for internal changes of list subtrees

The described extension is related to DataPlugins which implement lists. As described in chapter 6.5.1, the individual elements of a list are represented as leaf nodes of an interior node of type info.dmtree.spi.DMTConstants.DDF_LIST_SUBTREE.

This leads to a situation where the outside view on such a node just sees an atomic value while the internal representation is much more granular. It is in the responsibility of the Protocol Adapters (PA) to harmonize both views.

If internal changes to the granular elements of a list subtree node happen, then the resulting Events must be first DELETED, then ADDED.

The problem is that the PA needs to know which DmtEvent DELETED and ADDED for siblings belong together and should be assumed as “list modification”. Otherwise, if for example a TR-069 PA receives one DELETED and one ADDED event for a list subtree node with a notification attribute of “active notification”, then it could erroneously fire two notifications instead of just one.

In order to provide a hint to the event receiver, telling that the current event is just the first one of a pair, a new optional event property is introduced:

`EVENT_PROPERTY_LIST_UPCOMING_EVENT`

The only valid value of this property key is “true”.

Furthermore the event property:

`EVENT_PROPERTY_LIST__NODE`

is introduced. This property is filled with the URI of the list subtree node and must be identically in both corresponding events.

`If an event receiver gets an event of topic` info/dmtree/DmtEvent/ADDED with a given property

`EVENT_PROPERTY_LIST_NODE` and the **`EVENT_PROPERTY_LIST_UPCOMING_EVENT`** set to “true”, then this receiver should wait for a corresponding event of topic info/dmtree/DmtEvent/DELETED, before it takes the corresponding actions.

In case of the Event with the topic of info/dmtree/DmtEvent/DELETED for the internal change of a list subtree, the value to be set to “nodes” property will be special meaning. The “nodes” property must have only one entry which is the same as **`EVENT_PROPERTY_LIST_NODE`** In some cases, a DataPlugin cannot know the old value for the replaced or deleted node when it detects internal change. For example, the DataPlugin provides the data model for monitoring OSGi services and it can detect the change of the service properties by receiving ServiceEvent.

However, at the moment, the DataPlugin cannot know the old service properties. It is possible for the DataPlugin to cache the service properties, however, it can cause a problem by consuming a lot of memory.

Example:

For explanation, let's assume the following situation:

- The node "./A/B" has the node type of info.dmtree.spi.DMTConstants.DDF_LIST_SUBTREE.
- The leaf node "./A/B/0" has the value of v0, and the leaf node "./A/B/1" has the value of v1.
- Now, internal change of the list subtree occurs: the list should be set to [v2,v1,v3].

Then two events will be thrown in the following order:

```
info/dmtree/DmtEvent/ADDED {
  nodes = [./A/B/0, ./A/B/1, ./A/B/2 ]
  list.node="./A/B"
  list.upcoming.event="true"
}
```

nodes:	must contain all node uris as a result of the modification. If no nodes exist, empty array is set.
list.node:	the internal list node uri
list.upcoming.event:	indicates that this event is part of a sequence, value must be “true”

```
info/dmtree/DmtEvent/DELETED {
  nodes = [ ./A/B]
  list.node="./A/B"
}
```

nodes:	must be the same as list.node property
list.node:	the internal list node uri

Even if there existed no leaf node before this internal change occurs, both events must be fired. Even if there exists no leaf node after this internal change occurs, both events must be fired. In that case, the “nodes” property must contain an empty array.

6.6.3 Eventing sequence for atomic sessions

The specification in section 117.10.1 Event Admin based Events must be improved to make explicitly clear that DmtEvents must not provide information about transaction internal states. That means only those changes that describe the delta between beginning of the session and commit must be signaled as Dmt Events.

It is the responsibility of the DmtAdmin to determine this delta and to send the corresponding events.

6.6.4 Destructive Operations

There can be situations where a Protocol Adapter (PA) invokes an operation in a DmtSession which would lead to immediate changes in the framework state.

In such a case it would be impossible for the PA to wait for a proper result for its method invocation and the PA would therefore not be able to respond as required, e.g. to send back a response to a “SetParameterValue” request to an ACS in TR-069 cases.

Required is a mechanism that allows the DataPlugin to delay the execution of the destructive operation until it gets a signal that it is safe to perform it.

In order to achieve that, a new event topic of name “info/dmtree/DmtEvent/DESTRUCTIVE_OPERATION” is introduced. This topic can be used by the DataPlugin for sending events indicating that it intends to perform a destructive operation, i.e. an operation that can change the state of the system bundle and therefore the whole framework.
In contrast to all other internal events that the DataPlugin can emit, this event must be forwarded synchronously to all potential listeners. The listeners (e.g. Protocol Adapters) then have the chance to take the required actions before they return from the handler method.
The DataPlugin must wait until the synchronous sending of this event is finished before it can perform its destructive operation.

The diagram on the following page shows an exemplary case in form of a message sequence chart.

Components that want to receive that event, where an OSGi service platform is not supported, must register themselves with the new interface SynchronousDmtEventListener, which inherits from DmtEventListener, otherwise they are not able to receive this synchronous event.
It is strongly recommended that DataPlugins use the sendEvent() methods of the MountPoint interface to send such events synchronously to potential receivers. Otherwise SynchronousDmtEventListeners will not be able to receive this event.

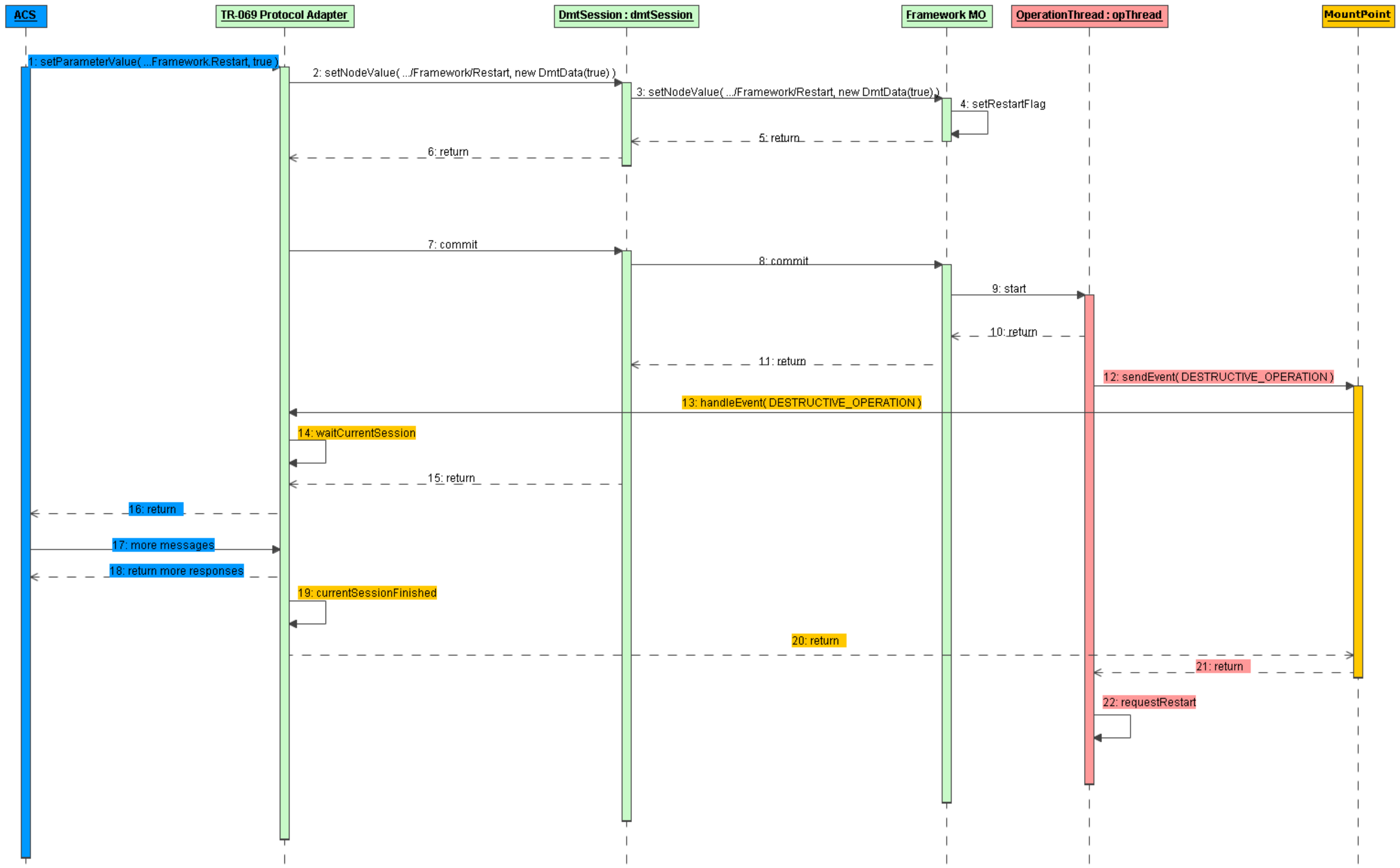

ACS
TR-069 Protocol Adapter
DmtSession : dmtSession
Framework MO
OperationThread : opThread
MountPoint
1: setParameterValue(...Framework.Restart, true)
2: setNodeValue(.../Framework/Restart, new DmtData(true))
3: setNodeValue(.../Framework/Restart, new DmtData(true))
4: setRestartFlag
5: return
6: return
7: commit
8: commit
9: start
10: return
11: return
12: sendEvent(DESTRUCTIVE_OPERATION)
13: handleEvent(DESTRUCTIVE_OPERATION)
14: waitCurrentSession
15: return
16: return
17: more messages
18: return more responses
19: currentSessionFinished
20: return
21: return
22: requestRestart

Figure 1: Example sequence with a DESTRUCTIVE_OPERATION event

6.7 The MountPlugin and MountPoint Interfaces

As a consequence of the new possibility to register plugins as elements of lists (by using "#" in the plugin root), such plugins don't know their final mapped URI in advance. The list Ids are assigned by the DMT Admin during the mapping process and the plugin has no special means to investigate them.

Furthermore plugins can be registered for more than one root at once, but there can be conditions that cause one or more roots to be un-mappable at this special point in time. (e.g. if the root points to a node that has no valid mountPoint in the subtree of another plugin, i.e. it has no corresponding MountingPlugin.) In such cases, not all the roots that the plugin was registered with can be mapped successfully. On the other side there can be conditions that lead to a situation where a root must be un-mapped during runtime, e.g. if the corresponding MountingPlugin was unregistered and therefore also its MountPoints become invalid.

The DmtAdmin should provide a way to make plugins aware of these mapping / un-mapping actions, so that they can adapt their behavior accordingly.

The remainder of this chapter describes such a mechanism.

6.7.1 The MountPlugin interface

The mechanism is based on a new interface – info.dmtree.spi.MountPlugin . It can be optionally implemented by plugins (DataPlugins as well as ExecPlugins).

NOTE: The Bundle that wants to make use of the MountPlugin mechanism needs the ServicePermission to register info.dmtree.spi.MountPlugin.

In order to be recognized as MountPlugin, a plugin must be registered with an array of class names including info.dmtree.spi.MountPlugin. When such a plugin is registered, the DmtAdmin checks the ServiceRegistry.getProperties(Constants.OBJECT_CLASS) for the existence of info.dmtree.spi.MountPlugin.

This interface defines a callback method "mountPointsAdded(MountPoint[])", that provides an array of all successfully mapped mount points and a method "mountPointsRemoved(MountPoint[])" that informs the plugin about removed mappings.

The sequence of operation during plugin registration will be as follows:

1. The plugin developer implements the MountPlugin interface.
2. The plugin is registered with 3 dataRootURIs (i.e. An array of 3 URI entries)
3. The DMT Admin "sees" the new Plugin registration and extracts the dataRootURIs from the registration properties.
4. The DMT Admin loops through the dataRootURIs and attemps a mapping for each one individually.
5. For each mapping attempt the DMT Admin knows the resulting mapped URI or knows that it has failed.

6. The DMTAdmin checks whether the Plugin was also registered as a MountPlugin.

   a) if not, then nothing happens

   b) if yes, then it constructs the MountPoint array with only successfully performed mapping, gets a reference to the MountPlugin interface of the Plugin and invokes the callback “mountPointsAdded()” with this array.

The plugin can use the method “getMountPath” on the provided MountPoints to get the successfully mapped URIs.

The sequence of operation when one mapping of the plugin has been removed later in time:

1. Precondition: A plugin was registered with 3 DataRootURI's and two of them where mapped successfully during registration time.

2. Now an ancestor plugin is unregistered and with this also the mount point that justified the plugins mapping has disappeared.

3. As a result, the mapping of the plugin to the disappeared mount point is removed by the Dmt Admin.

4. The DMTAdmin checks whether the Plugin was also registered as a MountPlugin.

   a) if not, then nothing happens

   b) if yes, then it constructs the MountPoint array with from the removed URI mapping, gets a reference to the MountPlugin interface of the Plugin and invokes the callback “mountPointsRemoved()” with this array.

The sequence of operation when one mapping of the plugin has been added later in time looks similar, except that then the mount points “mountPointsAdded()” method will be invoked.

6.7.2 The MountPoint interface

The new MountPoint interface provides two methods to post events via the DMT Admin. These methods can be used to signal “internal” changes in the own subtree, that means changes which occurred in their data model without a corresponding DMT Session. (see chapter 6.8.3 for API details)

The DmtAdmin is responsible for the actual forwarding of the events. All events that are posted by plugins via the MountPoint interface must be forwarded by the DmtAdmin through the EventAdmin as well as through the “changeOccured(DmtEvent)” method of the matching local DmtEventListeners

NOTE:
The use of the MountPoint.postEvent() and MountPoint.sendEvent() methods is the only possible way for Plugins to signal changes in their internal data to local event listeners For performance considerations, the use of the synchronous sendEvent() methods should be limited. (The only known use case for synchronous events is currently for destructive operations as described in chapter 6.6.4 Destructive Operations).

In chapter 6.6.2 Eventing for internal changes of list subtrees some new event properties have been introduced that allow for signaling of internal changes especially for list subtree nodes. Furthermore it was defined that such internal events do not have a session.id property in order to allow a distinction from session event on the receiver side.

The existing interface info.dmtree.DmtEvent is not prepared to handle these new properties and the fact that the session id can be missing. For that reason the DmtEvent interface is extended as follows (see chapter 6.8.6 for API details):

- a method “String[] getPropertyNames()” is added that allows to infer the names of all event properties
- a method “Object getProperty(String key)” is added that allows access to every included property
- the method “int getSessionID()” will throw a java.lang.UnsupportedOperationException in case of internal events, where no session id property exists

All event properties which are set to Events based on Event Admin are also set to DmtEvent, so that:

- The properties of “nodes” can be retrieved by DmtEvent.getNodes() or DmtEvent.getProperty(“nodes”).
- The properties of “newnodes” can be retrieved by DmtEvent.getNewNodes() or DmtEvent.getProperty(“newnodes”).
- The properties of “session.id” can be retrieved by DmtEvent.getSessionId() or DmtEvent.getProperty(“session.id”).
- The type can be retrieved only by DmtEvent.getType().

These changes make the usage - pattern for Events based on EventAdmin and Events for DmtEventListeners more aligned to each other.

The DMT Admin Service Spec Ver.1.1, Section "117.10 Events" should be restructed, in order to reflect those changes and to remove duplicated description for EventAdmin Events and DmtEvents.

6.8 Javadoc

Note that only APIs that have been modified and added are included in this document.

6.8.1 Definition of constants for literals

info.dmtree.spi.DataPlugin

`public interface`
info.dmtree.spi.DataPlugin

The second sentence in the interface description should be updated as follows to indicate that also a single String value can be provided as registration parameter for DataPlugins.

```
“In an OSGi environment such implementations should be registered at the OSGi
service registry specifying the list of root node URIs in a String array or in case
of a single value as String in the dataRootURIs registration parameter.“
```

---

public interface **DataPlugin**

Field Summary

`static final String`	**`DATA_ROOT_URIS`** The string to be used as key for the “dataRootURIs” property when a DataPlugin is registered.

Field Detail

DATA_ROOT_URIS

`public static final String` **`DATA_ROOT_URIS`**

The string to be used as key for the “dataRootURIs” property when a DataPlugin is registered.

See Also:
Constant Field Values

info.dmtree.spi.ExecPlugin

`public interface`
info.dmtree.spi.ExecPlugin

The second sentence in the interface description should be updated as follows to indicate that also a single String value can be provided as registration parameter for ExecPlugins.

```
“In an OSGi environment such implementations should be registered at the OSGi
service registry specifying the list of root node URIs in a String array or in case
of a single  value as String in the execRootURIs registration parameter.”
```

public interface Exec**Plugin**

Field Summary

`static final`	**`EXEC_ROOT_URIS`**

`String`	The string to be used as key for the “execRootURIs” property when an ExecPlugin is registered.

Field Detail

EXEC_ROOT_URIS

`public static final String EXEC_ROOT_URIS`

The string to be used as key for the “execRootURIs” property when an ExecPlugin is registered.

See Also:
Constant Field Values

info.dmtree.spi.MountPlugin

`public interface`
info.dmtree.spi.MountPlugin

public interface Mount**Plugin**

Field Summary

`static final String`	**`MOUNT_POINTS`** The string to be used as key for the mount points property when a DataPlugin or ExecPlugin is registered with mount points.

Field Detail

MOUNT_POINTS

`public static final String MOUNT_POINTS`

The string to be used as key for the mount points property when a DataPlugin or ExecPlugin is registered with mount points.

See Also:
Constant Field Values

info.dmtree.spi.DmtConstants

`public interface` **`DmtConstants`**

Field Summary	
`static java.lang.String`	**`DDF_LIST_SUBTREE`** A string defining a DDF URI, indicating that the node is a list subtree.
`static java.lang.String`	**`DDF_SCAFFOLD`** A string defining a DDF URI, indicating that the node is a scaffold node.
`static java.lang.String`	**`EVENT_PROPERTY_NODES`** A string defining the property key for the “nodes” property in node related events.
`static java.lang.String`	**`EVENT_PROPERTY_NEW_NODES`** A string defining the property key for the “newnodes” property in node related events.
`static java.lang.String`	**`EVENT_PROPERTY_LIST_NODES`** A string defining the property key for the “list.nodes” property in node related events.
`static java.lang.String`	**`EVENT_PROPERTY_LIST_UPCOMING_EVENT`** A string defining the property key for the “list.upcoming.event” property in node related events.
`static java.lang.String`	**`EVENT_PROPERTY_SESSION_ID`** A string defining the property key for the “session.id” property in node related events.
`static java.lang.String`	**`EVENT_TOPIC_ADDED`** A string defining the topic for the event that is sent for added nodes.
`static java.lang.String`	**`EVENT_TOPIC_COPIED`** A string defining the topic for the event that is sent for copied nodes.
`static java.lang.String`	**`EVENT_TOPIC_DELETED`** A string defining the topic for the event that is sent for deleted nodes.
`static java.lang.String`	**`EVENT_TOPIC_RENAMED`** A string defining the topic for the event that is sent for renamed nodes.
`static java.lang.String`	**`EVENT_TOPIC_REPLACED`** A string defining the topic for the event that is sent for replaced nodes.
`static java.lang.String`	**`EVENT_TOPIC_DESCTRUCTIVE_OPERATION`** A string defining the topic for the event that can be sent by plugins to indicate their intention to execute a destructive operation that might change the state of the system bundle, i.e. the framework.

static java.lang.String	**EVENT_TOPIC_SESSION_OPENED** A string defining the topic for the event that is sent by the DmtAdmin, if a new DmtSession has been opened.
static java.lang.String	**EVENT_TOPIC_SESSION_CLOSED** A string defining the topic for the event that is sent by the DmtAdmin, if a new DmtSession has been closed.

Field Detail

DDF_LIST_SUBTREE

`static final java.lang.String` **`DDF_LIST_SUBTREE`**

A string defining a DDF URI, indicating that the node is a list subtree.

See Also:
Constant Field Values

DDF_SCAFFOLD

`static final java.lang.String` **`DDF_SCAFFOLD`**

A string defining a DDF URI, indicating that the node is a scaffold node.

See Also:
Constant Field Values

EVENT_PROPERTY_NODES

`static final java.lang.String` **`EVENT_PROPERTY_NODES`**

a string defining the property key for the “nodes” property in node related events

See Also:
Constant Field Values

EVENT_PROPERTY_NEW_NODES

`static final java.lang.String` **`EVENT_PROPERTY_NEW_NODES`**

a string defining the property key for the “newnodes” property in node related events

See Also:
Constant Field Values

EVENT_PROPERTY_LIST_NODES

`static final java.lang.String` **`EVENT_PROPERTY_LIST_NODES`**

a string defining the property key for the “list.nodes” property in node related events

See Also:
Constant Field Values

EVENT_PROPERTY_LIST_UPCOMING_EVENT

`static final java.lang.String` **`EVENT_PROPERTY_LIST_UPCOMING_EVENT`**

a string defining the property key for the “list.upcoming.event” property in node related events

See Also:
Constant Field Values

EVENT_PROPERTY_SESSION_ID

`static final java.lang.String` **`EVENT_PROPERTY_SESSION_ID`**

a string defining the property key for the “session.id” property in node related events

See Also:
Constant Field Values

EVENT_TOPIC_ADDED

`static final java.lang.String` **`EVENT_TOPIC_ADDED`**

a string defining the topic for the event that is sent for added nodes.

The value of this field is “info/dmtree/DmtEvent/ADDED”.

See Also:
Constant Field Values

EVENT_TOPIC_DELETED

`static final java.lang.String` **`EVENT_TOPIC_DELETED`**

a string defining the topic for the event that is sent for deleted nodes.

The value of this field is “info/dmtree/DmtEvent/DELETED”.

See Also:
Constant Field Values

EVENT_TOPIC_REPLACED

`static final java.lang.String` **`EVENT_TOPIC_REPLACED`**

a string defining the topic for the event that is sent for replaced nodes.

The value of this field is “info/dmtree/DmtEvent/REPLACED”.

See Also:
Constant Field Values

EVENT_TOPIC_RENAMED

`static final java.lang.String` **`EVENT_TOPIC_RENAMED`**

a string defining the topic for the event that is sent for renamed nodes.

The value of this field is “info/dmtree/DmtEvent/RENAMED”.

See Also:
Constant Field Values

EVENT_TOPIC_COPIED

`static final java.lang.String` **`EVENT_TOPIC_COPIED`**

a string defining the topic for the event that is sent for copied nodes.

The value of this field is “info/dmtree/DmtEvent/COPIED”.

See Also:
Constant Field Values

EVENT_TOPIC_DESCTRUCTIVE_OPERATION

`static final java.lang.String` **`EVENT_TOPIC_DESCTRUCTIVE_OPERATION`**

a string defining the topic for the event that can be sent by plugins to indicate their intention to execute a destructive operation that might change the state of the system bundle, i.e. the framework.

The value of this field is “info/dmtree/DmtEvent/DESTRUCTIVE_OPERATION”.

See Also:
Constant Field Values

EVENT_TOPIC_SESSION_OPENED

`static final java.lang.String` **`EVENT_TOPIC_SESSION_OPENED`**

A string defining the topic for the event that is sent by the DmtAdmin, if a new DmtSession has been opened.

The value of this field is “info/dmtree/DmtEvent/SESSION_OPENED”.

See Also:
Constant Field Values

EVENT_TOPIC_SESSION_CLOSED

`static final java.lang.String` **`EVENT_TOPIC_SESSION_CLOSED`**

A string defining the topic for the event that is sent by the DmtAdmin, if a DmtSession has been closed.

The value of this field is “info/dmtree/DmtEvent/SESSION_CLOSED”.

See Also:
Constant Field Values

6.8.2 Additional Data Types

info.dmtree
Class DmtData

```
java.lang.Object
  info.dmtree.DmtData
```

```
public final class DmtDataextends java.lang.Object
```

Field Summary

static int	**FORMAT_LONG** The node holds a long value
static int	**FORMAT_DATETIME** The node holds a String object that is interpreted as a dateTime type defined in ISO 8601.
static int	**FORMAT_HEXBINARY** The node holds an hex binary value.
static int	**FORMAT_NODE_URI** The node holds a string value that represents a DMT node URI.
static int	**FORMAT_UNSIGNED_INTEGER** The node holds an unsigned int value.
static int	**FORMAT_UNSIGNED_LONG** The node holds an unsigned long value.
static DmtData	**TRUE_VALUE** Constant instance representing a leaf node of boolean `true` value.
static DmtData	**FALSE_VALUE** Constant instance representing a leaf node of boolean `false` value.

Constructor Summary

DmtData(byte[] bytes, int format) Create a `DmtData` instance of the specified format and set its value based on the given `byte[]`.
DmtData(long lng) Create a `DmtData` instance of `long` format and set its value.

DmtData(java.lang.String value, int format) Create a DmtData instance of the specified format and set its value based on the given string.

Method Summary

java.lang.String	**getDateTime**() Gets the value of a node with DateTime format.
byte[]	**getHexBinary**() Gets the value of a node with hex binary format.
long	**getLong**() Gets the value of a node with long format.
java.lang.String	**getNodeUri**() Gets the value of a node uri (FORMAT_NODE_URI) format.
java.lang.String	**getUnsignedInteger**() Gets the value of a node with unsigned integer format.
java.lang.String	**getUnsignedLong**() Gets the value of a node with unsigned long format.
java.lang.String	**toString**() Gets the string representation of the DmtData.

Methods inherited from class java.lang.Object
clone, finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

FORMAT_LONG

`public static final int` **FORMAT_LONG**

The node holds a long value. The `getFormatName()` method can be used to get the actual format name.

See Also:
Constant Field Values

FORMAT_UNSIGNED_LONG

`public static final int` **FORMAT_UNSIGNED_LONG**

The node holds an unsigned long value.

See Also:
Constant Field Values

FORMAT_DATETIME

`public static final int` **`FORMAT_DATETIME`**

The node holds a String object that is interpreted as a dateTime type defined in ISO 8601.

See Also:
Constant Field Values

FORMAT_UNSIGNED_INTEGER

`public static final int` **`FORMAT_UNSIGNED_INTEGER`**

The node holds an unsigned int value.

See Also:
Constant Field Values

FORMAT_NODE_URI

`public static final int` **`FORMAT_NODE_URI`**

The node holds a string value that represents a DMT node URI.

See Also:
Constant Field Values

FORMAT_HEXBINARY

`public static final int` **`FORMAT_HEXBINARY`**

The node holds an hex binary `hexbin` value. The value of the node corresponds to the Java `byte[]` type.

See Also:
Constant Field Values

TRUE_VALUE

`public static final DmtData` **`TRUE_VALUE`**

Constant instance representing a boolean `true` value.

FALSE_VALUE

```
public static final DmtData FALSE_VALUE
```

Constant instance representing a boolean `false` value.

Constructor Detail

DmtData

```
public DmtData(java.lang.String value,
               int format)
```

Create a `DmtData` instance of the specified format and set its value based on the given string. Only the following string-based formats can be created using this constructor:

- `FORMAT_STRING` - value can be any string
- `FORMAT_XML` - value must contain an XML fragment (the validity is not checked by this constructor)
- `FORMAT_DATE` - value must be parseable to an ISO 8601 calendar date in complete representation, basic format (pattern `CCYYMMDD`)
- `FORMAT_TIME` - value must be parseable to an ISO 8601 time of day in either local time, complete representation, basic format (pattern `hhmmss`) or Coordinated Universal Time, basic format (pattern `hhmmssZ`)
- `FORMAT_DATETIME` - value must be parseable to an ISO 8601 definition of a calendar date-time in complete representation, basic format (pattern `CCYYMMDDThhmmss`) or Coordinated Universal Time, basic format (pattern `CCYYMMDDThhmmssZ`)
- `FORMAT_UNSIGNED_LONG` - value must contain a string that is parsable as unsigned long
- `FORMAT_UNSIGNED_INTEGER` - value must contain a string that is parsable as unsigned int
- `FORMAT_NODE_URI` - value must contain a string that holds a reference to a node

* The `null` string argument is only valid if the format is string or XML.

Parameters:

`value` - the string, XML, date, time, datetime, unsignedInt, unsignedLong or node-uri value to set
`format` - the format of the `DmtData` instance to be created, must be one of the formats specified above

Throws:

`java.lang.IllegalArgumentException` - if `format` is not one of the allowed formats, or `value` is not a valid string for the given format
`java.lang.NullPointerException` - if a string, XML, date, time, datetime, unsignedInt, unsignedLong or node-uri is constructed and `value` is `null`

DmtData

```
public DmtData(long lng)
```

Create a `DmtData` instance of `long` format and set its value.

Parameters:
`lng` - the long value to set

DmtData

```
public DmtData(byte[] bytes,
               int format)
```

Create a `DmtData` instance of the specified format and set its value based on the given `byte[]`. Only the following `byte[]` based formats can be created using this constructor:

- `FORMAT_BINARY`
- `FORMAT_BASE64`
- `FORMAT_HEXBINARY`

Parameters:
`bytes` - the byte array to set, must not be `null`
`format` - the format of the DmtData instance to be created, must be one of the formats specified above
Throws:
`java.lang.IllegalArgumentException` - if `format` is not one of the allowed formats
`java.lang.NullPointerException` - if `bytes` is `null`

Method Detail

getDateTime

```
public java.lang.String getDateTime()
```

Gets the value of a node with DateTime format. The returned dateTime string is formatted according to the ISO 8601 definition of a calendar date-time in complete representation. The exact format depends on the value the object was initialized with: either local time, complete representation, basic format (pattern `CCYYMMDDThhmmss`) or Coordinated Universal Time, basic format (pattern `CCYYMMDDThhmmssZ`).

Returns:
the time value
Throws:
`DmtIllegalStateException` - if the format of the node is not time

getUnsignedInteger

`public java.lang.String` **`getUnsignedInteger`**`()`

Gets the value of a node with unsigned integer (`uint`) format.

Returns:
the unsigned integer value
Throws:
`DmtIllegalStateException` - if the format of the node is not integer

---

getLong

`public long` **`getLong`**`()`

Gets the value of a node with long (`long`) format.

Returns:
the long value
Throws:
`DmtIllegalStateException` - if the format of the node is not long

---

getUnsignedLong

`public java.lang.String` **`getUnsignedLong`**`()`

Gets the value of a node with unsigned long (`ulong`) format. The value is returned as a String because the range of a long is not sufficient to hold the maximum possible unsigned value.

Returns:
the unsigned long value
Throws:
`DmtIllegalStateException` - if the format of the node is not long

getHexBinary

`public byte[]` **`getHexBinary`**`()`

Gets the value of a node with hex binary (`hexbin`) format.

Returns:
the hex binary value
Throws:
`DmtIllegalStateException` - if the format of the node is not hex binary

getNodeUri

`public java.lang.String` **`getNodeUri`**`()`

Gets the value of a node uri (`FORMAT_NODE_URI`) format.

Returns:
the data value in node uri format
Throws:
`DmtIllegalStateException` - if the format of the node is not node uri

toString

`public java.lang.String` **`toString`**`()`

Gets the string representation of the `DmtData`. This method works for all formats.

For string format data - including `FORMAT_RAW_STRING` - the string value itself is returned, while for XML, date, time, integer, float, boolean, long, unsignedInt, unsignedLong, node-uri and node formats the string form of the value is returned. Binary - including `FORMAT_RAW_BINARY` - base64 and hexBinary data is represented by two-digit hexadecimal numbers for each byte separated by spaces. The `NULL_VALUE` data has the string form of " `null`". Data of string or XML format containing the Java `null` value is represented by an empty string.

Overrides:
`toString` in class `java.lang.Object`
Returns:
the string representation of this `DmtData` instance

6.8.3 Mount Plugin API

info.dmtree.spi
Interface MountPlugin

`public interface` **`MountPlugin`**

This interface can be optionally implemented by a `DataPlugin` or `ExecPlugin` in order to get information about its absolute mount points in the overall DMT.

This is especially interesting, if the plugin is mapped to the tree as part of a list. In such a case the id for this particular plugin is determined by the DmtAdmin after the registration of the plugin and therefore unknown to the plugin in advance.

Field Summary

`static java.lang.String`	**`MOUNT_POINTS`** The string to be used as key for the mount points property when a DataPlugin or ExecPlugin is registered with mount points.

Method Summary

`void`	**`mountPointsAdded`**`(MountPoint[] mountPoints)` Provides the `MountPoint` objects describing the path where the plugin is mapped to the overall DMT.
`void`	**`mountPointsRemoved`**`(MountPoint[] mountPoints)` Informs the plugin that the provided `MountPoint` objects have been removed from the mapping.

Field Detail

MOUNT_POINTS

`static final java.lang.String` **`MOUNT_POINTS`**

The string to be used as key for the mount points property when a DataPlugin or ExecPlugin is registered with mount points.

See Also:
Constant Field Values

Method Detail

mountPointsAdded

`void` **`mountPointsAdded`**`(MountPoint[] mountPoints)`

Provides the `MountPoint` objects describing the path where the plugin is mapped to the overall DMT.

Parameters:
`mountPoints` - the newly mapped mount points

---

mountPointsRemoved

`void` **`mountPointsRemoved`**`(MountPoint[] mountPoints)`

Informs the plugin that the provided `MountPoint` objects have been removed from the mapping.

NOTE: attempts to invoke the `postEvent` or sendEvent methods on the provided `MountPoint` will be ignored.

Parameters:
`mountPoints` - array of `MountPoint` objects that have been removed from the mapping

info.dmtree.spi
Interface MountPoint

public interface MountPoint

This interface can be implemented to represent a single mount point.

It provides functionalites to get the absolute mounted uri and a shortcut method to post events via the DmtAdmin.

Method Summary

`java.lang.String[]`	**`getMountPath`**`()` Provides the absolute mount path of this `MountPoint`
`void`	**`postEvent`**`(java.lang.String topic, java.lang.String[] relativeURIs, java.util.Map properties)` Posts an event via the DmtAdmin about changes in the current plugins subtree.
`void`	**`postEvent`**`(java.lang.String topic, java.lang.String[] relativeURIs, java.lang.String[] newRelativeURIs, java.util.Map properties)` Posts an event via the DmtAdmin about changes in the current plugins subtree.
void	**`sendEvent`**`(java.lang.String topic, java.lang.String[] relativeURIs, java.util.`~~`Dictionary`~~`Map properties)` Sends an event via the DmtAdmin about changes in the current plugins subtree synchronously, i.e. blocking.

Method Detail

getMountPath

`java.lang.String[]` **`getMountPath`**`()`

Provides the absolute mount path of this `MountPoint`

Returns:
the absolute mount path of this `MountPoint`

postEvent

```
void postEvent(java.lang.String topic,
               java.lang.String[] relativeURIs,
               java.util.Map properties)
```

Posts an event via the DmtAdmin about changes in the current plugins subtree.

This method distributes Events asynchronously to the EventAdmin as well as to matching local DmtEventListeners (except SynchronousDmtEventListeners).

Parameters:

`topic` - the topic of the event to send. Valid values are:

- `info/dmtree/DmtEvent/ADDED` if the change was caused by a rename action
- `info/dmtree/DmtEvent/DELETED` if the change was caused by a copy action
- `info/dmtree/DmtEvent/REPLACED` if the change was caused by a copy action

`relativeURIs` - an array of affected node `URI`'s. All `URI`'s specified here are relative to the current `MountPoint`'s mountPath. The value of this parameter determines the value of the event property EVENT_PROPERTY_NODES. An empty array or `null` is permitted. In both cases the value of the events EVENT_PROPERTY_NODES property will be set to an empty array.

`properties` - an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain a key EVENT_PROPERTY_NODES, then the value of this property is ignored and will be overwritten by `relativeURIs`.

Throws:

java.lang.NullPointerException - if the topic is null

`java.lang.IllegalArgumentException` - if the topic has not one of the defined values

postEvent

```
void postEvent(java.lang.String topic,
               java.lang.String[] relativeURIs,
               java.lang.String[] newRelativeURIs,
               java.util.~~Dictionary~~Map properties)
```

Posts an event via the DmtAdmin about changes in the current plugins subtree.

This method distributes Events asynchronously to the EventAdmin as well as to matching local DmtEventListeners (except SynchronousDmtEventListeners).

Parameters:

`topic` - the topic of the event to send. Valid values are:

- `info/dmtree/DmtEvent/RENAMED` if the change was caused by a rename action
- `info/dmtree/DmtEvent/COPIED` if the change was caused by a copy action

`relativeURIs` - an array of affected node `URI`'s.

All `URI`'s specified here are relative to the current `MountPoint`'s mountPath. The value of this parameter determines the value of the event property EVENT_PROPERTY_NODES. An empty array or `null` is permitted. In both cases the value of the events EVENT_PROPERTY_NODES property will be set to an empty array.

`newRelativeURIs` - an array of affected node `URI`'s.The value of this parameter determines the value of the event property EVENT_PROPERTY_NEW_NODES. An empty array or `null` is permitted. In both cases the value of the events EVENT_PROPERTY_NEW_NODES property will be set to an empty array.
`properties` - an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain the keys EVENT_PROPERTY_NODES or EVENT_PROPERTY_NEW_NODES, then the values of these properties are ignored and will be overwritten by `relativeURIs` and `newRelativeURIs`.

Throws:

java.lang.NullPointerException - if the topic is null

`java.lang.IllegalArgumentException` - if the topic has not one of the defined values

sendEvent

void **sendEvent**(java.lang.String topic,
                java.lang.String[] relativeURIs,
                java.util.Map~~Dictionary~~ properties)

Sends an event via the DmtAdmin to notify listeners synchronously about a potentially destructive operation.

This method distributes Events synchronously to the EventAdmin as well as to matching local SynchronousDmtEventListeners (not to DmtEventListeners)

Parameters:

`topic` - the topic of the event to send. Valid values are:

- `info/dmtree/DmtEvent/DESTRUCTIVE_OPERATION` if the notification is about a destructive operation that causes a potential change of the system state

`relativeURIs` - an array of affected node `URI`'s. All `URI`'s specified here are relative to the current `MountPoint`'s mountPath. The value of this parameter determines the value of the event property "nodes". An empty array or `null` is permitted. In both cases the value of the events "nodes" property will be set to an empty array.
`properties` - an optional parameter that can be provided to add properties to the Event that is going to be send by the DMTAdmin. If the properties contain a key "nodes", then the value of this property is ignored and will be overwritten by `relativeURIs`.

Throws:

java.lang.NullPointerException - if the topic is null

`java.lang.IllegalArgumentException` - if the topic has not the defined value

NOTE:
The emitted events will not have a session-id, because the event is low-level and was not produced in the scope of a session.

6.8.4 DmtException.METADATA_MISMATCH

The Javadoc of DmtException.METADATA_MISMATCH must be updated as follows:

```
Section 117.13.7.8 - javadoc for DmtException.METADATA_MISTMATCH, where it is stated

that it can in an Exception when: "creating, deleting or renaming a permanent
node, or modifying its type".
```

6.8.5 DmtException.COMMAND_NOT_ALLOWED

Following situation must be added to the Javadoc of DmtException.COMMAND_NOT_ALLOWED in section 117.13.7.3 (public static final int COMMAND_NOT_ALLOWED = 405)

- an attempt to execute an operation other than the ones defined in info.dmtree.spi.ReadableDataSession was performed

6.8.6 info.dmtree.DmtEvent

Following extension are required in the javadoc of the interface DmtEvent.

Field Summary	
`static int`	**`DESTRUCTIVE_OPERATION`** Event type indicating that a destructive operation is going to be performed by a plugin.

Method Summary	
`java.lang.Object`	**`getProperty`**`(java.lang.String key)` This method can be used to get the value of a single event property.
`java.lang.String[]`	**`getPropertyNames`**`()` This method can be used to query the names of all properties of this event.

Field Detail

DESTRUCTIVE_OPERATION

`static final int` **`DESTRUCTIVE_OPERATION`**

Event type indicating that a destructive operation is going to be performed by a plugin. This can be any action that causes a change in system bundles state (like restart, shutdown etc.).

Events of this type are delivered synchronously in order to give the receiver the chance to react accordingly before the destructive operation takes place.

See Also:
Constant Field Values

Method Detail

getSessionId

`int` **`getSessionId`**`()`

This method returns the identifier of the session in which this event took place. The ID is guaranteed to be unique on a machine.

For events that are result of an internal change inside of a `DataPlugin` (not in the scope of any session), no sessionId is defined. An invocation of this method on such events will therefore throw an UnsupportedOperationException.

The availability of a session.id can also be check by using `getProperty()` with "session.id" as key.

Returns:
the unique identifier of the session that triggered the event

Throws:
`java.lang.UnsupportedOperationException` - in case that no session id is available. This indicates that the `DmtEvent` was not a result of a `DmtSession`.

getPropertyNames

`java.lang.String[]` **`getPropertyNames`**`()`

This method can be used to query the names of all properties of this event.

The returned names can be used as key value in subsequent calls to `getProperty(java.lang.String)`.

Returns:
the array of property names

See Also:
`getProperty(java.lang.String)`

getProperty

```
java.lang.Object getProperty(java.lang.String key)
```

This method can be used to get the value of a single event property.

Parameters:
`key` - the name of the requested property

Returns:
the requested property value or null, if the key is not contained in the properties

See Also:
`getPropertyNames()`

6.8.7 Interface info.dmtree.SynchronousDmtEventListener

info.dmtree
Interface SynchronousDmtEventListener

All Superinterfaces:
DmtEventListener

```
public interface SynchronousDmtEventListener extends DmtEventListener
```

Like for `DmtEventListener`, registered implementations of this class are notified via `DmtEvent` objects about important changes in the tree. The main difference is that implementers of SynchronousDmtEventListener are notified synchronously and therefore blocking.

See Also:
`DmtEventListener`

Method Summary

`void`	**`changeOccurred`**`(DmtEvent event)` `DmtAdmin` uses this method to notify the registered listeners about the change.

Method Detail

changeOccurred

```
void changeOccurred(DmtEvent event)
```

`DmtAdmin` uses this method to notify the registered listeners about the change. This method is called synchronously from the actual event occurrence.

Parameters:
`event` - the `DmtEvent` describing the change in detail

6.8.8 Class info.dmtree.Uri

info.dmtree
Class Uri

```
java.lang.Object
  └info.dmtree.Uri
```

`public final class` **`Uri`**`extends java.lang.Object`

Field Summary

`static java.lang.String`	**`PATH_SEPARATOR`** This constant stands for a string identifying the path separator in the DmTree ("/").
`static char`	**`PATH_SEPARATOR_CHAR`** This constant stands for a char identifying the path separator in the DmTree ('/').
`static java.lang.String`	**`ROOT_NODE`** This constant stands for a string identifying the root of the DmTree (".").
`static char`	**`ROOT_NODE_CHAR`** This constant stands for a char identifying the root of the DmTree ('.').

Field Detail

ROOT_NODE

`public static final java.lang.String` **`ROOT_NODE`**

This constant stands for a string identifying the root of the DmTree (".").

See Also:
Constant Field Values

ROOT_NODE_CHAR

`public static final char ROOT_NODE_CHAR`

This constant stands for a char identifying the root of the DmTree ('.').

See Also:
Constant Field Values

PATH_SEPARATOR

`public static final java.lang.String PATH_SEPARATOR`

This constant stands for a string identifying the path separator in the DmTree ("/").

See Also:
Constant Field Values

PATH_SEPARATOR_CHAR

`public static final char PATH_SEPARATOR_CHAR`

This constant stands for a char identifying the path separator in the DmTree ('/').

See Also:
Constant Field Values

6.8.9 Interface info.dmtree.DmtAdmin

info.dmtree
Interface DmtAdmin

`public interface DmtAdmin`

Field Summary

static java.lang.String	**SESSION_CREATION_TIMEOUT** This system property can be used to specify the DMT session creation timeout in milliseconds.
static java.lang.String	**SESSION_INACTIVE_TIMEOUT**

	This system property can be used to specify the DMT session inactivity timeout in milliseconds.

Field Detail

SESSION_CREATION_TIMEOUT

`static final java.lang.String` **`SESSION_CREATION_TIMEOUT`**

This system property can be used to specify the DMT session creation timeout in milliseconds. If this value is not given, then an implementation specific default value will be used.

See Also:
Constant Field Values

SESSION_INACTIVE_TIMEOUT

`static final java.lang.String` **`SESSION_INACTIVE_TIMEOUT`**

This system property can be used to specify the DMT session inactivity timeout in milliseconds. If this value is not given, then an implementation specific default value will be used.

See Also:
Constant Field Values

Method Summary

`long`	**`getSessionCreationTimeout`**`()` Returns the value for the session creation timeout in milliseconds.
`long`	**`getSessionInactivityTimeout`**`()` Returns the value for the session inactivity timeout in milliseconds.

getSessionCreationTimeout

`long` **`getSessionCreationTimeout`**`()`

Returns the value for the session creation timeout in milliseconds. The returned value reflects the value that has been set via the corresponding system property `SESSION_CREATION_TIMEOUT` or the implementation specific default value.

Returns:
the active value of the session creation timeout in milliseconds

getSessionInactivityTimeout

`long getSessionInactivityTimeout()`

Returns the value for the session inactivity timeout in milliseconds. The returned value reflects the value that has been set via the corresponding system property `SESSION_INACTIVE_TIMEOUT` or the implementation specific default value.

Returns:
the active value of the session creation timeout in milliseconds

7 Considered Alternatives

Nothing to be described.

8 Security Considerations

All security requirements follow the DMT Admin specification.

With reference to chapter 6.7 (The MountPlugins and MountPoint interfaces) the following must be stated additionally:

When a Plugin invokes one of the postEvent methods of the MountPoint interface 6.8.3 Mount Plugin API, then the DmtAdmin BundleContext will get the EventAdmin service and no permission is required for the calling bundle except:

ServicePermission[MountPlugin, REGISTER].

9 Document Support

9.1 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]. OMA, Open Mobile Alliance. The mission of the Open Mobile Alliance is to facilitate global user adoption of mobile data services by specifying market driven mobile service enablers that ensure service interoperability across devices, geographies, service providers, operators, and networks, while allowing businesses to compete through innovation and differentiation. http://www.openmobilealliance.org/

[4]. OMA Device Management specification v1.2. The goal of the Device Management Working Group is to specify protocols and mechanisms that achieve management of mobile devices including the necessary configuration to access services and management of the software on mobile devices. http://www.openmobilealliance.org/release_program/dm_v1_2C.html

[5]. CPE WAN Management Protocol, TR-069 amendment 2, December 2007. http://www.broadband-forum.org/technical/download/TR-069Amendment2.pdf

[6]. The Broadband Forum is a global consortium of nearly 200 leading industry players covering telecommunications, equipment, computing, networking and service provider companies. Established in 1994, originally as the ADSL Forum and later the DSL Forum, the Broadband Forum continues its drive for a global mass market for broadband, to deliver the benefits of this technology to end users around the world over existing copper telephone wire infrastructures. http://www.broadband-forum.org/about/forumhistory.php

[7]. Data model template for TR-069 enabled devices, TR-106 amendment 1, November 2006

9.2 Author's Address

Name	Steffen Drüsedow
Company	Deutsche Telekom AG, Laboratories
Address	Ernst-Reuter-Platz 7, 10589 Berlin
Voice	+49-30-8353-58139
e-mail	steffen.druesedow@telekom.de

Name	Ikuo Yamasaki
Company	NTT Corporation
Address	Y320C, 1-1 Hikari-no-oka, Yokosuka, Kanagawa, Japan
Voice	+81-46-859-8537
e-mail	yamasaki.ikuo@lab.ntt.co.jp

Name	Shigekuni Kondo
Company	NTT Corporation
Address	Y320C, 1-1 Hikari-no-oka, Yokosuka, Kanagawa, Japan
Voice	+81-46-859-3444
e-mail	kondo.shigekuni@lab.ntt.co.jp

9.3 Acronyms and Abbreviations

9.4 End of Document