Version 0.84

Access Control Lists

Proposal for Access Control Lists in CMIS

05/14/2009

Versions

Version	Authors	Date	Changes
0.1	Paul Goetz, SAP	12/09/2008	document created
0.2	Florian Müller, Open Text Jens Hübel, Open Text Viktor Gavrysh, EMC Martin Hermes, SAP Paul Goetz, SAP	12/16/2008	completely restructured
0.3	Paul Goetz, SAP	12/18/2008	motivation added, minor changes, maps for required extend permission per operation added
0.4	Paul Goetz, SAP	02/18/2009	adapted to findings from CMIS Faces2Face 28/01/2008, extended permission model removed
0.5	Paul Goetz, SAP	03/19/2009	included proposals from Viktor Gavrysh, EMC
0.6	Paul Goetz, SAP	03/27/2009	Principal as structure with optional attributes (e.g. type), addACEs and removeACEs combined to applyACEs
0.7	Paul Goetz, SAP	04/05/2009	reworked to reflect input from Oracle, references to CMIS specificiation now updated to v0.6
0.8	Al Brown, IBM Paul Goetz, SAP	04/24/2009	removed permission sets, added bindings, applyACEs now applyACL, some definitions copied to a specific version of CMIS Part I – Domain Model v0.61
0.81	Paul Goetz, SAP	04/29/2009	tried to clean up wording for levels and tiers
0.82	Paul Goetz, SAP	05/05/2009	applyACL: merging replaced by add/remove lists, capabilityACL: read and discover combined to discover getACL: added “exact” flag
0.83	Paul Goetz, SAP	05/10/2009	getProperties now also returns ACLs if requested, direct-Flag not optional, additional comments for permission model
0.84	Paul Goetz, SAP	05/15/2009	added mapping for permissions to allowable actions

CONTENTS

INTRODUCTION

This document's primary location is the OASIS TC's *Documents* area (URL: http://www.oasis-open.org/apps/org/workgroup/cmis/download.php/32262/ACL%20Proposal%20v0.83.doc).

This document should be considered as an addendum to the "*CMIS Part I – Domain Model v0.61c with ACLs.doc*" document.

While the "*CMIS Part I – Domain Model v0.61c with ACLs.doc*" contains the normative parts for the ACL proposal, this document "*ACL Proposal v.081.doc*" tries to explain the motivation, use cases and a bit of history.

RECAP: SECURITY IN CMIS SPECIFICATION 0.6

Version 0.6 of the CMIS specification draft contains a concept for policy objects (see Section ***2.6 Policy Object***). Access to certain aspects of an object can be restricted by a policy.

Policies – like other primary entities of the CMIS specification – are typed, have an object ID and have properties (see *General Concepts* below).

A policy is created using the Object Service's *createPolicy* method. Input for this method is a description of the policy (name, type, properties, etc.), output is an ID of the created policy instance. Providing this ID, a policy can be applied to a controllable object (*applyPolicy*), removed (*removePolicy*), or retrieved from an object (*getAppliedPolicies*) via the Policy Service.

A controllable object can have zero or more policies applied. Not having a policy applied means that there is no restriction accessing the object.

ACL DESIGN OBJECTIVES AND ASSUMPTIONS

The basic requirements for ACLs can be grouped by the following three levels:

Level 1 – **Unified Search** → getRepositoryInfo: capabilityACL = Discover, permissionNames = { CMIS.BasicPermission.Read }

This requires the ablity to discover who is allowed to read the content and properties of a document or folder. The scenario is that data from a CMIS repository is to be indexed by an external search engine: In order to filter relevant results for a given user efficiently, the search engine needs to add index information about "who is allowed to read the search result", e.g. by extending the query to something like `WHERE … AND currentuser IS IN read-acl-entries` when searching.

Level 2 – **Reporting Permissions** → getRepositoryInfo: capabilityACL = Discover, permissionNames = { CMIS.BasicPermission.Read, … }

The requirement is to distinguish different permissions, like READ, WRITE or DELETE.
The scenario is that a user is able to figure out which other users she or he can collaborate on a shared document or folder (e.g. who can read, who can modify, and who can manage the permissions of a document).

Level 3 – **Managing Permissions** → getRepositoryInfo: capabilityACL = Manage

Like Level 2, plus the requirement to be able to modify the ACL for a document or folder.
Like for Level 2, the scenario is that a user would like to allow other users (e.g. his or her team) to get access to a document or folder (e.g. that all the members of a team can collaborate on the same folder).

Thus, we assume that ACLs will be used mainly for collaborative user scenarios, where an end user needs to be able to control the permissions to be applied to documents or folders at runtime at least to allow content sharing and collaboration. E.g. "My working drafts for the documentation should only be editable by my co-workers John and Mary, be visible to my team, but they must not to be seen by someone else outside the team".
In addition, we assume that at least a minimal set of permissions should be predefined, such that specific applications can rely on known semantics for this predefined permissions (like for Level 1 the indexing engine relies on the READ semantics).

Another assumption is that for enterprise level security constraints, ***Policies*** are more appropriate than ACLs. Policies are intended to express security constraints more on an enterprise level and that are shared by several objects (e.g. "job references in folder EMEA can be read by members of the HR department in the EMEA region only"). ACLs are intended as an additional mechanism for collaborative scenarios.

E.g. a business application scenario, like attaching scanned images of an invoice to ERP data would rather add a policy like "Invoices with a total of more than 1Million EURO should not be visible by anyone who's not a member of the controlling team and has doesn't have at least a clearance level 2", than applying an ACL.

Theoretically, there would be different options on where the put the knowledge about the semantics for permissions. However, in the discussions it turned out that there shouldn't be too much semantics within the CMIS specification. This implies, that – except for the predefined permissions – it will be up to the client (usually the user then) to "know" about the semantics of an ACL. Thus, the focus for this proposal is on abilities to marshall the information required to discover and manage ACLs for a user – and only to a minor extend to help applications to "understand" the ACL (except for the predefined CMIS permissions).

As the ACEs of an ACL define *who* is allowed to do *what*, two additional (technical) assumptions:

1. Regarding the *who*:
   We assume that all the systems share a common understanding of the principals to be checked. In an enterprise or intranet scenario, this is more likely to be the case, as a central LDAP or other kind of directory service will most probably be available. For extranet/internet scenarios, we assume that more generic authentication standards will be relevant (in the worst case, the CMIS consumer would have to do the user mapping by means beyond the scope of CMIS).
   ➔ We assume that principals are known to both, consumer and provider – thus user/group discovery is not within the scope of this document.

2. Regarding the *what*:
   We assume that ACLs are applied to folder- and document-like objects only, and that checks against ACLs are performed for operations on those objects only.
   ➔ We assume that ACLs are appropriate for the basic object types folder and document (not for relationship,

policy) as this concept is known from existing file system implementations – other CMIS objects would have to be secured by policies then.

RELATION TO OTHER STANDARDS

Content Repository for Java – JSR 283

Reference: http://jcp.org/en/jsr/detail?id=283

As we expect that JCR might serve as a local Java API for the CMIS protocol (either for consumers – using JCR to access a CMIS provider – or for providers – using CMIS to expose a JCR repository, like Apache Chemistry), the ACL concepts proposed by CMIS should be mappeable to JCR:

Policies can be mapped to the JCR's **AccessControlPolicy** objects and handling of policies can be mapped to the **AccessControlManager**'s `get…`/`set…`/`delete…` methods (while CMIS' **addPolicy** could be mapped to a **getApplicablePolicies** on a specific system path, or createing a node with a specific structure (e.g. using XACML)).

ACLs with ACEs for arbitrary permissions are not covered by JCR – they would have to be mapped to JCR policies as well.

ACLs with ACEs for CMIS-defined permissions should be mappable by taking care that the semantics defined in this proposal are compliant with the JCR's standard privileges **jcr:read**, **jcr:setProperties**, **jcr:addChildNodes**, **jcr:removeChildNodes**, **jcr:write**, **jcr:getAccessControlPolicy**, **jcr:setAccessControlPolicy**, and **jcr:all**. See *Permissions* below.

HTTP Extensions for Distributed Authoring – WebDAV

Reference: http://www.ietf.org/rfc/rfc2518.txt, http://www.webdav.org/acl/, and http://www.webdav.org/specs/rfc3744.html

As we assume that a CMIS provider might also expose its repository via WebDAV, the proposed ACL concept should be mappeable to WebDAV.

Policies are not covered by WebDAV.

ACLs with ACEs for arbitrary permissions are pretty much the same as specified in WebDAV, using specific privileges (aka permissions).

ACLs with ACEs for CMIS-defined permissions should use a simplified set of privileges which could then be mapped as well. See *Permissions* below.

XACML

Reference: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20

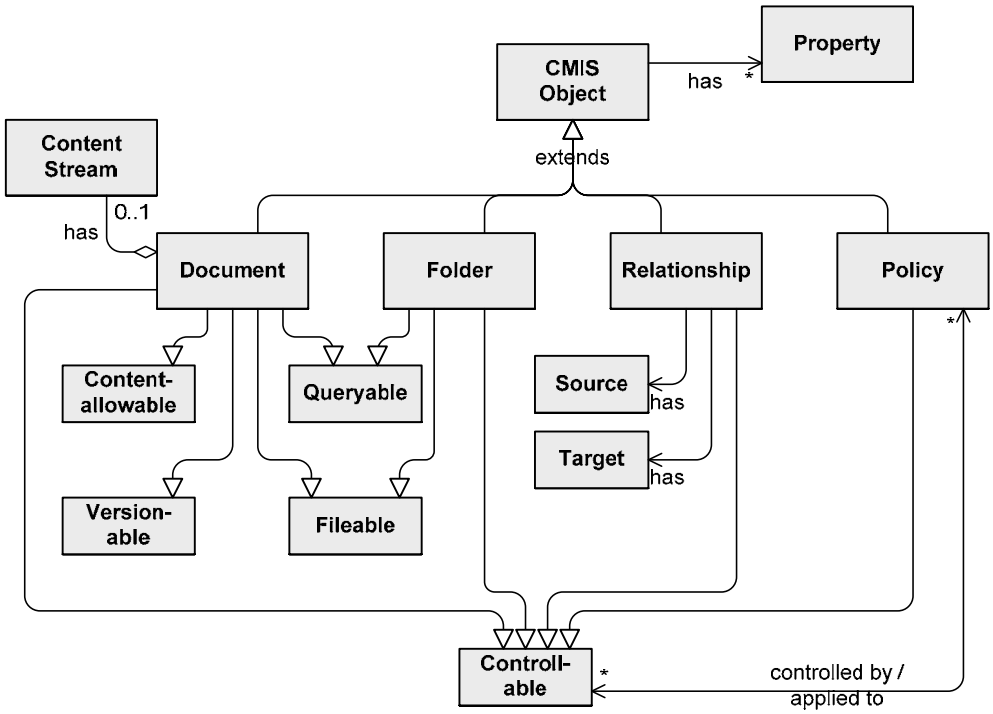
XACML is much more generic and flexible then the proposed ACLs. As outlined above, we consider more generic security handling being related to ***Policies*** and therefore as out of scope for this proposal on ACLs.

Thus – although XACML might become relevant when getting into more details for policies – we won't take XACML into account for this proposal on ACLs.

GENERAL CONCEPTS

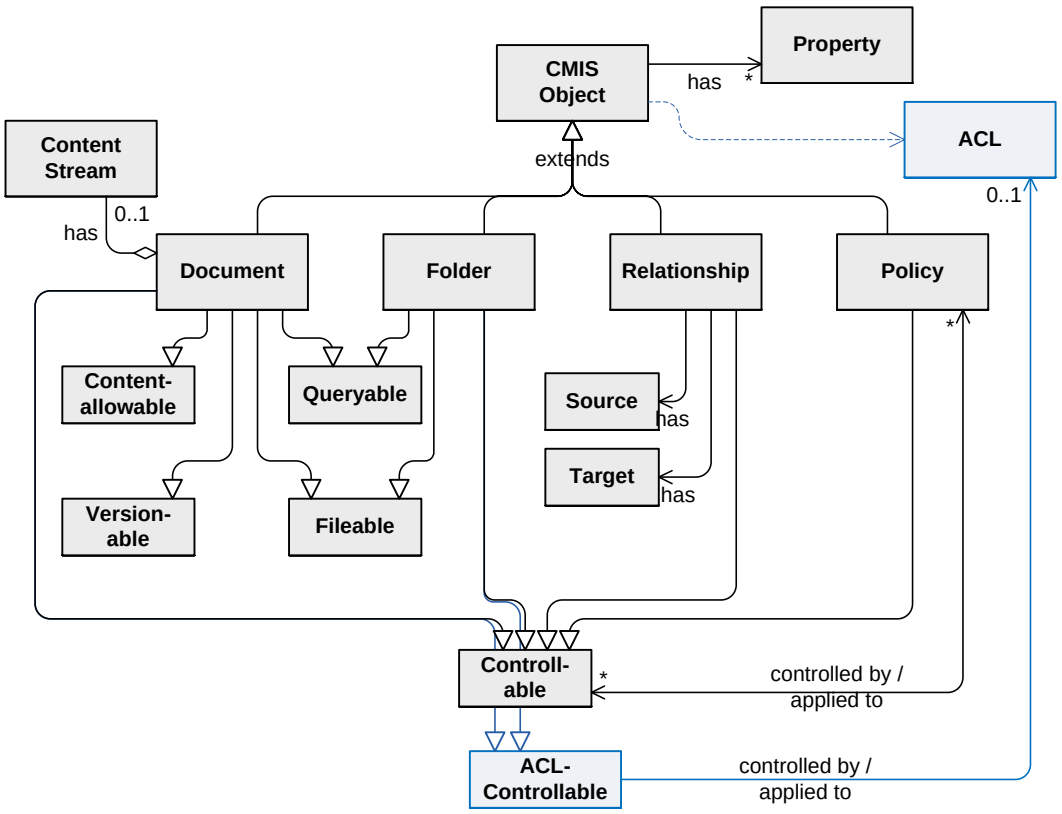
To provide an overview about the general concept and to provide a more formal naming, the following sections use a Java style pseudo code to illustrate the proposal.

The CMIS specification currently defines the following object hierarchy:



A [CMIS] ***Object*** is either a ***Document***, a ***Folder***, a ***Relationship*** or a ***Policy***. All of them may have ***Properties*** assigned, while only Documents may have ***Content*** [Streams] (i.e. have *contentStreamAllowed* <> *notAllowed*) and are *Version-able*. Folders and Documents are *Query-able* and *File-able*. Relations can reference to a a Source Object and a Target Object. Documents, Folders, Relationsships and Policies might be *Control-able* – and may then be controlled by zero or more Policies. Vice versa, a Policy may be assigned to zero or several control-able Objects.

During the first CMIS TC's F2F Meeting in January 2009 it was decided that ACLs shall be added as a specific "dependend" object type (like Propertie), thus the propsed changed object hierarchy would then look like:



Only ***Documents*** and ***Folders*** may be *ACLControl-able*. An *ACLControl-able* may then be controlled by one ACL (can have an ACL assigned).

An **ACL** in turn is implicitly assigned to its object – an ACL is not a [CMIS] *Object* on its own, an ACL depends on the [CMIS] Object it belongs to (like a Property, but in contrast to Policies).

Only *Document Objects* and *Folder Objects* may have an ACL assigned – but only if they are "tagged" as being *ACLControl-able*.

OVERVIEW OF ACLS

Access Control Lists

An Access Control List (***ACL***) is just a list of Access Control Entires (***ACE***s)

```
public List<AccessControlEntry> AccessControlList;
```

```
<xs:complexType name="cmisAccessControlListType">
      <xs:sequence>
            <xs:element name="permission"
                        type="cmis:cmisAccessControlEntryType" />
            <xs:any namespace="##other" />
      </xs:sequence>
</xs:complexType>
```

Access Control Entries

An Access Control Entry (***ACE***) specifies a ***Permission*** and a ***Principal***., and holds a boolean flag ***direct***, which indicates if the ACE is applied directly to this object, or derived/inherited from some other object.

```
public class AccessControlEntry {
  public Principal principal;
  public Permission permission;
  public Boolean direct;
}
```

```
<xs:complexType name="cmisAccessControlEntryType">
      <xs:sequence>
            <xs:element name="principal"
                        type="cmis:cmisAccessControlPrincipalType" />
            <xs:element name="permission"
                        type="cmis:cmisAccessControlPermissionType" />
            <xs:element name="direct" type="xs:boolean" />
            <xs:any namespace="##other" />
      </xs:sequence>
</xs:complexType>
```

This proposal restricts to positive ACEs. Thus, no *negative* flag is required.

See also *Permissions* below for more details on the ***PermissionDefinition***.

Since an ACE is specified by it's prinicpal's ID and the permission's name, "changing" an ACE means either removing an existing ACE (*PrincipalID,PermissionName*) or adding a new ACE (*PrincipalID,PermissionName*).

The boolean ***direct*** flag indicates that the ACE is applied to the Document or Folder object itself (*direct=TRUE*), or is derived or inherited from another object (*direct=FALSE*). The *direct* flag is provided for reporting purposes only – it is relevant when the ACL is retrieved from the repository, it is not relevant (can be omitted) when a client specifies an ACE to be applied to an document or folder and will be ignored by the repository.

For more details, see *Discovering ACL Capabilities* and *Applying ACEs* below.

Principals

A principal represents either a single user, or a set of users – this can be a group, or some other notion for a "set of users" like a "role". As we assume that user management is outside the scope of CMIS and to be handled by the client and the repository, it might be beneficial if the repository and the client could use the principal as a container for some additional data. Therefore, a principal might have some additional (optional) attributes:

```
public class Principal {
  public String principalId;
  public HashMap<String,String> attributes;
}
```

```
<xs:complexType name="cmisAccessControlPrincipalAttributeType">
      <xs:sequence>
```

```
            <xs:element name="key" type="xs:string" />
            <xs:element name="value" type="xs:string" />
      </xs:sequence>
</xs:complexType>

<xs:complexType name="cmisAccessControlPrincipalType">
      <xs:sequence>
            <xs:element name="principalId" type="xs:string" />
            <xs:element name="attributes"
                        type="cmisAccessControlPrincipalAttributeType"
                        minOccurs="0" maxOccurs="unbounded" />
            <xs:any namespace="##other" />
      </xs:sequence>
</xs:complexType>
```

The attributes are optional key-value-pairs which can be used to add more information about the principal, e.g. “type”=”user” or “datasource”=”LDAP”.

Permissions

As outlined in the *ACL Design Objectives and Assumptions*, the main focus for this proposal is to be able to expose the permissions a repository supports to clients. Therefore, the permissions are basically strings.

```
<xs:complexType name="cmisAccessControlPermissionType">
      <xs:sequence>
            <xs:element name="permission" type="xs:string" />
            <xs:element name="description" type="xs:string"
                        minOccurs="0" />
            <xs:any namespace="##other" />
      </xs:sequence>
</xs:complexType>
```

The *permission* string uniquely identifies the permission in the repositories set of permissions. The *description* is an (optional) string containing a “name” for the permission to be displayed to the user.

For some scenarios, a minimal set of “meaningful” permissions might be required by the application:

The absolute minimal permission to discover is ***Read*** (see the Level 1 – Unified Search scenario above).

To support Level 3 scenarios (managing permissions) for background applications, we propose that at least the permissions ***All***, ***Write***, ***Read*** should defined by CMIS:

- **All**: includes all permissions supported by the repository
  (corresponding to *jcr:all*, or *DAV:all*)
- **Write**: when granted, it should permit the following operations: delete the object, write properties and content of the object, filing or unfiling of the object (plus ***Read***, see below).
  (corresponding to *jcr:write* + *jcr:setProperty* + *jcr:addChildNodes* + *jcr:removeChildNodes*, or *DAV:write*)
- **Read**: when granted, it should permit reading of properties and content for the given object.
  (corresponding to *jcr:read*, or *DAV:read*)

```
<xs:simpleType name="enumBasicPrivileges">
```

```
      <xs:restriction base="xs:string">
            <xs:enumeration value="CMIS.BasicPermission.Read" />
            <xs:enumeration value="CMIS.BasicPermission.Write" />
            <xs:enumeration value="CMIS.BasicPermission.All" />
      </xs:restriction>
</xs:simpleType>
```

We assume that this are the most basic *Permissions* to be defined by CMIS. As a repository might add repository specific permissions, the permissions defined by CMIS should be identified by a CMIS-specific namespace or prefix: ***CMIS.BasicPermission.All***, ***CMIS.BasicPermission.Write***, and ***CMIS.BasicPermission.Read***.

E.g. an ACL like `{(john: CMIS.BasicPermission.All), (mary: bind), (mary:unbind), (mary:write)}` would define an ACL with one ACE using the CMIS-defined permission ***All*** for user "john", and two ACEs using repository specific permissions ***bind*** and ***unbind*** for user "mary".

Since this proposal restricts to positive ACEs, a ***Deny*** permission is not required, as this is the default permission for a principal not being listed in an ACE.

If no ACL is assigned to an object, all permissions are granted by default (unless overwritten by specific policies) – this is similar to policies (defaulting to full access for an object if no policies are applied).

DATA MODEL

See the following sections in "*CMIS Part I – Domain Model v0.61c with ACLs.doc*":

2.1.1 – Optional Capabilities (table at line 53) : *ACL*

2.6.1 – ACL (lines 374-433) : *ACL*, *ACE*, *setType*, *Permissions* and CMIS-defined Permissions

2.7.2.3 – Attributes specific to Document Object-Types and Folder Object-Types (lines 493-496) : *ACLcontrollable*

3.2.2 – getRepositoryInfo (table at line 1247) : Enum *capabilityACL*

3.4.1 – createDocument (table at line 1263) : *policies*, *addACEs* and *removeACEs*

3.4.2 – createFolder (table at line 1265) : *policies*, *addACEs* and *removeACEs*

3.10 – ACL Services (line 1340-1349) : *getACLCapabilities*, *getACL*, *applyACL*

DISCOVERING ACL CAPABILITIES

The *Repository Service* ***getRepositoryInfo*** (section 3.2.2 in the Domain Model, line 1247) returns the *enum* ***capabilityACL*** which indicates the Level (see *ACL Design Objectives and Assumptions*) the repository supports for ACLs.

```
<xs:simpleType name="enumCapabilityACL">
      <xs:restriction base="xs:string">
            <xs:enumeration value="none" />
            <xs:enumeration value="discover" />
            <xs:enumeration value="manage" />
      </xs:restriction>
</xs:simpleType>
```

If ***capabilityACL*** returns something else than ***none***, the *ACL Service* ***getACLCapabilities*** (section 3.10.1 in the Domain Model, line 1344) returns additional information about the ACL capabilities of the repository:

The multivalued *enum* ***setType*** describes the allowed values the client can use for ***applyACL*** (see below, *Applying ACEs*).

```
<xs:simpleType name="enumACLsetType">
      <xs:restriction base="xs:string">
            <xs:enumeration value="object-only" />
            <xs:enumeration value="dont-care" />
            <xs:enumeration value="propagate" />
      </xs:restriction>
</xs:simpleType>
```

The list of ***permissions*** is a list with all the permissions supported by the repository (see "Permissions" above).

The list of mappings is a list of mappings for permission name to allowable actions (see also “Required Permissions Per Operation” below). It is basically a table, with one entry per *actionName*. The ***actionName*** identifies the allowable action. The ***objectPermissionNames*** specifies the permissions that need to be applied to the “object” itself in order to allow the action (if the object already exists, or the type if the object is to be created).

```
<xs:complexType name="cmisPermissionMappingType">
      <xs:sequence>
            <xs:element name="actionName" type="xs:string" />
            <xs:element name="objectPermissionNames" type="xs:string"
                        minOccurs="1" maxOccurs="unbound" />
            <xs:element name="relatedOperands"
                        type="cmisPermissionMappingRelatedOperandType"
                        minOccurs="0" />
      </xs:sequence>
</xs:complexType>
```

The (optional) ***relatedOperands*** specifies in an additional table, what permissions need to be applied to other operands of the action (e.g. *source* and *target* for *moveObject*).

```
<xs:complexType name="cmisPermissionMappingRelatedOperandType">
      <xs:sequence>
            <xs:element name="relatedOperandType"
                        type="cmisPermissionRelatedOperandType" />
            <xs:element name="permissionNames" type="xs:string"
                        minOccurs="1" maxOccurs="unbound" />
      </xs:sequence>
</xs:complexType>
```

The ***relatedOperandType*** specifies the operand the permissions need to be applied to.

```
<xs:simpleType name="enumPermissionRelatedOperandType">
      <xs:restriction base="xs:string">
            <xs:enumeration value="folder" />
            <xs:enumeration value="policy" />
            <xs:enumeration value="source" />
            <xs:enumeration value="target" />
      </xs:restriction>
</xs:simpleType>
```

OBJECT SERVICES

The *Object Services* ***createDocument*** (section 3.4.1 in the Domain Model, line 1263) and ***createFolder*** (section 3.4.2 in the Domain Model, line 1265) take three additional optional parameters: A list of policy IDs, a list of ACEs to be added, and a list of ACEs to be removed.

“Adding” and “removing” refers to the ACL assigned to the parent folder given by *folderId*. If no folderId is specified, an empty ACL is assumed – thus, the behavior is equal to adding the ACEs from *addACEs* and ignoring the ACEs from *removeACEs*.

This would allow clients to manage security constraints for newly-created documents and folders.

ACL SERVICES

As outlined above, the *ACL Services* are optional capability, and to be supported only if ***getRepositoryInfo*** returns something else than *none* for ***capabilityACL***.

The *ACL Service* ***getACLCapibilities*** is described above in *Discovering ACL Capabilities*.

RETRIEVING ACLS

The *ACL Service* ***getACL*** (section 3.10.2 in the Domain Model, line 1346) returns the *ACL* as a *list of ACE*s for a given object ID (and repository ID), see *Overview of ACLs* for more details.

How the ACL or ACEs are fabricated, e.g. how an inheritance mechanism applies for ACEs, is up to the repository. Therefore a client MUST NOT assume that inheritance is bound to the folder's parent-child-relationship.

Furthermore, a client MUST NOT assume that the ACL contains all the information about access control when policies are supported by a repository. In other words: Even if a given principal is granted a specific permission by ACE, this might be overwritten by a policy applied to that object. The client has to use ***getAllowableActions*** to determine the effectively allowed actions for a given object and user.

Similar, a client MUST NOT assume that a principal not listed in the ACEs of an ACL for a specific permission will not be granted access for this specific permission to the given object.

Only if the repository returns *TRUE* for the (optional) flag ***exact*** on ***getACL***, the client can assume that the ACL provided completely reflects the permissions applied to the object.

For the Level 1 – Unified Search use case, indexing retrieves the ACL via ***getACL*** for every object. When searching, the query is extended to something like `WHERE … AND currentuser IS IN read-acl-entries`. The result set might then still contain entries, where no access is granted due to applied policies, therefore this resultset has to be checked by the repository again (e.g. using a `SELECT … WHERE OrderId IS IN (`list-of-serach-result-object-ids`)`).

This implies, that changing the ACL for a hierarchy (using `setType=propagate`, see below) might result in many entries in the change log, or might require reindexing (where the "inheritance dependency" might not be bound to folders parent-child relations).

The permission names used in the ACEs returned by the repository MUST be contained in the the list of permission names returned by ***getACLCapabilties***.

APPLYING ACES

The *ACL Service* ***applyACL*** (section 3.10.3 in the Domain Model, line 1340) takes an optional enum ***setType***, the ***list of addACEs*** to be added to the ACL, and the ***list of removeACEs*** to be removed from the ACL. The provided ACEs

are merged (see below) with the security constraints already applied to the given object, and returns the resulting *ACL* as it would be returned by ***getACL***.

The *ACL Service* ***applyACL*** can be used by a client only in case ***getRepositoryInfo*** returned ***capabilityACL*** = ***manage***.

The "merging" of the ACEs is up to the repository. The only option for a client to provide some information to the repository about how the "merging" should behave (from a clients perspective) is the enum ***setType***:

- **dont-care**: is the default value and will be used by the repository, if no value is provided when calling *applyACL* by the client. This indicates that the client does not care at all about how the merging is done by the repository.
  → inheritance/propagation of the resulting security constraints is completely up to the repository.

- **object-only**: This indicates that the client want the ACEs to be applied only to the given object – without any side effects for other objects.
  → the repository MUST apply the ACEs to the given object only. It MUST NOT use inheritance/propagation.

- **propagate**: This indicates that the client wants the ACEs to be applied to the given object and all "inheriting" objects – i.e. with the intended side effect that all objects which somehow share the provided security constraints should be changed accordingly.
  → the repository MUST apply the ACEs using its internal "inheritance" mechanisms. How propagation/inheritance is defined is up to the repository.

However, a client MUST check the repository capabilities as returned in the multivalued *enum* ***setType*** from the *ACL Service* ***getACLcapability***. The *setType* provided by the client to *applyACL* MUST be contained in this map of *setType*s as returned by *getACLCapability*.

If one of the ACEs can not be added or removed, the applyACL service should fail in total, and provide the ACEs that caused the failure in the exception.

If a repository is not capable to deal with changes to inherited ACEs, it SHOULD NOT return a value for ***setType*** at all. Then, clients MUST NOT try to apply ACLs to objects having ACEs with ***direct*** = *TRUE*.

A bit more formally:

In general we assume that the repository supports ACLs (*capabilityACL* = *Manage*) , the repository supports the ACE's permission (*permissions* returned by ***getACLCapabilities*** contains *ACE.Permissions*), the repository supports the requested operation mode for *applyACL* (*setType* returned by ***getACLCapabilities*** contains the *setType* provided for ***applyACL***), and the objects type supports ACLs (*objectType.ACLControllable* = *TRUE*).

The following rules apply for ***applyACL***:

An ACE can be added to an ACL if:
exists and *direct = TRUE*
or exists and ***getACLCapabilties***.*setType* contains *applyACL.setType*

An ACE can be removed from an ACL if:
direct = TRUE
or ***getACLCapabilties***.*setType* contains *applyACL.setType*

If a repository does not support inheritance (***getACLCapabilties***.*setType* does not contain any value for *setType*), an ACE with *direct = FALSE* can not be deleted.

Rephrasing:

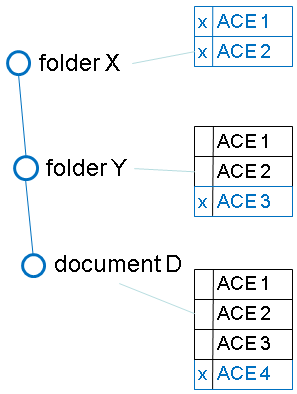
Any “inheritance” mechanism for ACEs is up to the repository – there are no means for the client to determine the inheritance mechanism (which objects are the legators of the inherited ACEs for a given object). The client has to be aware that some ACEs are computed by the repository and not “assignable” by means of the client. The ***direct*** flag is used by the repository to indicate that an ACE is applied to an object directly and not computed by some other means of “inheritance”.

Vice versa, the client is only able to express if either the ACL is meant for this object directly, or if the client does not care. As a third option, repositories supporting “inheritance” for the folders parent-child relation might provide a “propagation” along paths to the client.

ACEs passed as input to ***applyACL*** are **merged** with the ACEs of the existing ACL according to the rules above.

EXAMPLES

Example for a repository with support for inheritance via parent-child relation

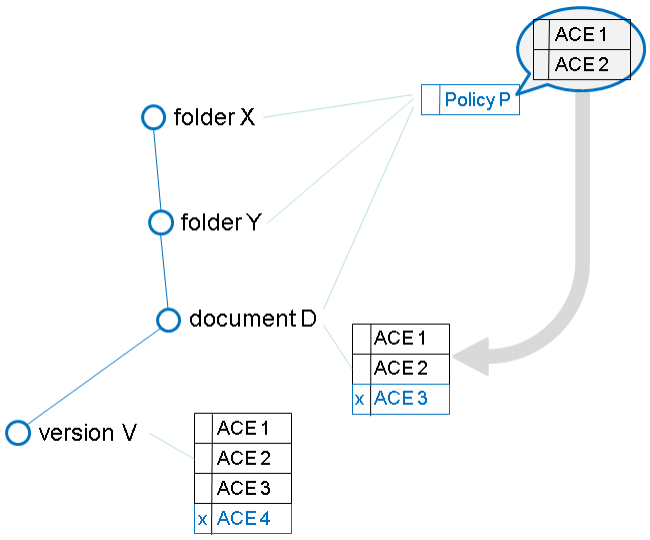


Adding ACE4 to document D results in a ACE4 with *direct = TRUE* – *setType* is not regarded.

Removing ACE4 then again from document D would work in any case, *setType* is not regarded.

Removing ACE1 from document D, which is inherited from folder X (with *direct = FALSE*) would work only, if the repository’s *getACLCapability.setType* contains *object-only* and *applyACL.setType=object-only*.

Example for a repository with policies and additional inheritance (e.g. via versions):



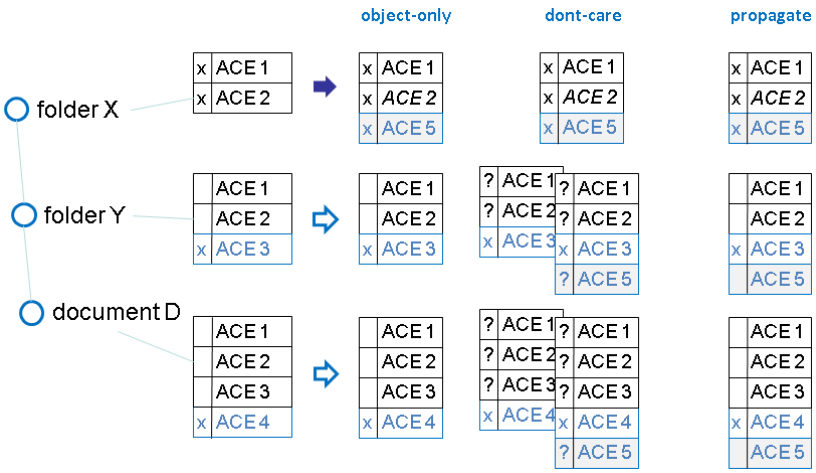
Lets assume that policy P can be reported as ACE1 and ACE2, then the repository can report this ACEs for the objects the policy is applied to (X, Y, D). As this ACEs are computed/derived, they have to be reported with direct = FALSE, and the repository can disallow changing these computed ACEs (by not reporting *object-only* for *getACLCapability.setType*).

Version V might inherit ACE1 to ACE3 from its current version document D.

Again, adding additional ACEs to V or D can be supported by the repository, but changing a non-direct ACEs is supported only if *object-only* is contained in *getACLCapability.setType*.

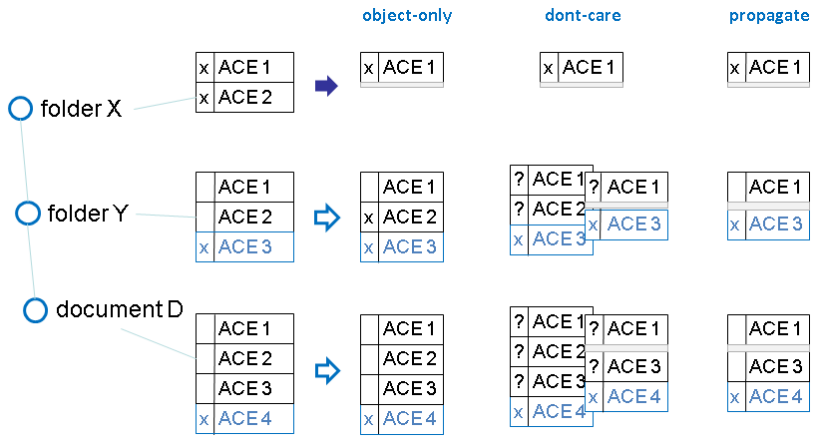
Examples for the different setTypes (using folder inheritance):

E.g. adding ACE5 to X (ACL provided to applyACEs is ACE1,ACE2,ACE5) results in ACE5 being added to X for *setType= object-only*; in ACE5 being added to X and inherited to Y and D for *setType=propagate*; in ACE5 being added to X and a repository specific behavior for *setType=dont-care*:



Removing an ACE from folder Y (or folder X) would result in an ACE with *direct = TRUE* for the given folder only – without adding ACEs to its descendants, if *setType = object-only* (or *dont-care*).

Removing an ACE from folder Y (or folder X) would result in removal of the ACE from the folder, and if *direct = FALSE* also results in removal of this ACE from all of it's descendants (document D for folder Y, folder Y and document D for folder X), if *setType = propagated* (or *dont-care*).



E.g. removing ACE2 to X (ACL provided to setACL is ACE1) results in ACE2 being removed from X for *setType=object-only*; in ACE2 being removed from X and inherently removed from Y and D for *setType=propagate*; in ACE2 being removed from X and a repository specific behavior for *setType=dont-care*.

CHECKING PRIVILEGES

The *Object Service* ***getAllowableAction*** requires more specification (e.g. is currently unclear, what has to be checked for a ***moveObject*** operation).

➔ TBD: a mechanism to define additional operands for the getAllowableAction, e.g. canMove for moveObject might not only depend on the object but also on the target folder, and optionally depend on the source folder (if specified).

REQUIRED PERMISSIONS PER OPERATION

This table needs to be discussed in more detail!

In general: The Permission model descibred below should specify the permissions to be applied in order to allow the listed "action". This means: Granting a permission will allow the listed "action", but does not necessarily result in an ACE reported by getACL for that permission.

There are two main uses cases for the manging of ACLs:

1. In a collaborative scenario (see "ACL Design Objectives and Assumptions"), a user "knows" about the semantics and side-effects when granting permission for a document or folder to another user or group. For this kind of scenarios, an application would simply present the existing ACL to the user, and allows to add or remove ACEs (principal + permission names as retrieved via getACLCapabilities) to the user. The user would then select the required permissions and principals to add a new ACE, or select the required ACE to remove.
   ➔ The permissions don't need to be known by the application.

2. In a development scenario, the developer needs to have some minimal understanding about the semantics of the permissions (e.g. to understand the READ permission for the Unified Search Indexer). For this kind of scenarios, the table below should describe the basic semantics in terms of "what permission needs to be applied to allow "action" xyz for principal abc". Additional side-effects might occur, depending on the repository.

actionName	operation	(main/first) operand	***object-Permission-Names***	***relatedOperands***
Navigation Services				
canGetDescendants	getDescendants	folder	**Read**	
canGetChildren	getChildren	folder	**Read**	
canGetFolderParent	getFolderParent	folder	**Read**	
canGetParents	getObjectParents	object	**Read**	
Object Services				
canCreateDocument	createDocument	type	**Write**	**folder: Write**
canCreateFolder	createFolder	type	**Write**	**folder: Write**
canCreateRelationship	createRelationship	type	**Write**	**source: Write** **target: Write**
canCreatePolicy	createPolicy	type	**Write**	
canGetProperties	getProperties	object	**Read**	
canViewContent	getContentStream	object	**Read**	
canUpdateProperties	updateProperties	object	**Write**	
canMove	moveObject	object	**Write**	**target: Write** **source: Write**
canDeleteVersion	deleteObject	object	**Write**	

canDeleteTree	deleteTree	folder	**Write**	
canSetContent	setContentStream	document	**Write**	
canDeleteContent	deleteContentStream	document	**Write**	
Multi-filing Services				
canAddToFolder	addObjectToFolder	object	**Write**	**folder: Write**
canRemoveFromFolder	removeObjectFromFolder	object	**Write**	**folder: Write**
Versioning Services				
canCheckout	checkOut	document	**Write**	
canCancelCheckout	cancelCheckOut	document	**Write**	
canCheckin	checkIn	document	**Write**	
canGetAllVersions	getAllVersions	versionseries	**Read**	
canDelete	deleteAllVersions	versionseries	**Write**	
Relationship Services				
canGetRelationships	getRelationships	object	**Read**	
Policy Services				
canAddPolicy	applyPolicy	object	**All**	**policy: Read**
canRemovePolicy	removePolicy	object	**All**	**policy: Read**
canGetAppliedPolicies	getAppliedPolicies	object	**Read**	
ACL Services				
canGetACL	getACL	object	**Read**	
canApplyACL	applyACL	object	**All**	

Still under discussion:

For the Unified Search: How to figure out that an ACE like { ALL, ‘mary’ } for document d implies READ access to d?

Proposal: Extend the information in the getACLCapabilities:

```
<xs:complexType name="cmisAccessControlPermissionType">
      <xs:sequence>
            <xs:element name="permission" type="xs:string" />
            <xs:element name="description" type="xs:string"
                        minOccurs="0" />
            <xs:any namespace="##other" />
            <xs:element name="parent" type="xs:string"
                        minOccurs="0" />
            <xs:element name="abstract" type="xs:Boolean”
                        minOccurs="0" />
      </xs:sequence>
</xs:complexType>
```