SCE Agent To Agent Transfer

Approved Version 1.0 – 05 Jul 2011

Open Mobile Alliance

OMA-TS-SCE_A2A-V1_0-20110705-A

Contents

Figures

Tables

1. Scope

Open Mobile Alliance (OMA) specifications are the result of continuous work to define industry-wide interoperable mechanisms for developing applications and services that are deployed over wireless communication networks[1].

The scope of OMA "Digital Rights Management" (DRM) is to enable the distribution and consumption of digital content in a controlled manner. The content is distributed and consumed on authenticated devices per the usage rights expressed by the content owners. OMA DRM work addresses the various technical aspects of this system by providing appropriate specifications for content formats, protocols, and a rights expression language.

A number of DRM specifications have already been defined within the OMA. The latest accepted release of the OMA DRM enabler ([DRM-v2.1], including [DRM-DRM-v2.1], [DRM-DCF-v2.1], [DRM-REL-v2.1]), is referred to within this document as "OMA DRM 2.1".

The scope of this specification is to define the mechanisms and protocols necessary to implement the moving and sharing of content (via the appropriate rights), as required per [SCE-RD]. Specifically, this document specifies the interface SCE-7-A2AP as defined in [SCE-AD] and thus limited to communications between a DRM Requester and a DRM Agent (that has implemented this specification).

1.1 Content vs. Rights

The reader should be aware that the terms "content" and "rights" are sometimes used interchangeably. However, it should be clarified that what is really being moved or shared is the rights which control the use of a particular content. In OMA DRM, a particular content is encrypted in a file and can only be rendered if the corresponding rights object is available to the Device. Since the content file is encrypted and hence protected, the movement or transfer of content files ([DRM-DCF-v2.1]) is outside the scope of DRM.

[1] Although many of the mechanisms can be applied to wired communication networks, including this specification.

2. References

2.1 Normative References

[AES-MODES] "Recommendation for Block Cipher Modes of Operation", NIST Special Publication 800-38A, 2001. URL:http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[AES-WRAP] Advanced Encryption Standard (AES) Key Wrap Algorithm. RFC 3394, J. Schaad and R. Housley, September 2002. URL:http://tools.ietf.org/html/rfc3394

[DRM-v2.1] The OMA DRM 2.1 enabler as described in "Enabler Release Definition for DRM V2.1, Approved Version 2.1", OMA-TS-DRM-DRM-V2_0-20060303-A, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[DRM-DCF-v2.1] "DRM Content Format, Approved Version 2.1", OMA-TS-DRM-DCF-V2_0-20060303-A, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[DRM-DRM-v2.1] "DRM Specification, Approved Version 2.1", OMA-TS-DRM-DRM-V2_0-20060303-A, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[DRM-REL-v2.1] "DRM Rights Expression Language, Approved Version 2.1", OMA-TS-DRM-REL-V2_0-20060303-A, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[ISO8601] "Data elements and interchange formats -- Information interchange -- Representation of dates and times", ISO 8601:2004, URL:http://www.iso.org

[RFC2104] "HMAC: Keyed-Hashing for Message Authentication", H. Krawczyk, M. Bellare, and R. Canetti, February 1997. URL:http://tools.ietf.org/html/rfc2104

[RFC2119] "Key words for use in RFCs to Indicate Requirement Levels", S. Bradner, March 1997, URL:http://tools.ietf.org/html/rfc2119

[RFC3280] "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", R. Housley, W. Polk, W. Ford, and D. Solo, April 2002, http://tools.ietf.org/html/rfc3280

[RFC3447] "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", J. Jonsson, B. Kaliski, February 2003, URL:http://tools.ietf.org/html/rfc3447

[SCE-AD] "Secure Content Exchange Architecture, Draft Version", OMA-AD-SCE-Vx_y-D, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[SCE-DOM] "SCE User Domains", OMA-TS-SCE-DOM-Vx_y-D, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[SCE-LRM] "Local Rights Manager for Secure Content Exchange", OMA-TS-SCE-LRM-Vx_y-D, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[SCE-RD] "Secure Content Exchange Requirements, Draft Version 1.0", OMA-RD-SCE-V1_0-20060908-D, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[SCE-REL] "DRM Rights Expression Language – SCE Extensions", OMA-TS-SCE-REL-Vx_y-D, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[SCR-RULES] "SCR Rules and Procedures", Open Mobile Alliance™, OMA-ORG-SCR_Rules_and_Procedures, URL:http://www.openmobilealliance.org/

[SHA1] NIST FIPS 180-2: Secure Hash Standard. August 2002. URL:http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

[SRM-TS] "Secure Removable Media Specification, Candidate Version 1.0", OMA-TS-SRM-V1_0-20080128-C, Open Mobile Alliance™, URL:http://www.openmobilealliance.org/

[XC14N] Exclusive XML Canonicalization: Version 1.0, John Boyer, Donald E. Eastlake 3rd and Joseph Reagle, W3C Recommendation 18 July 2002. URL:http://www.w3.org/TR/xml-exc-c14n/

2.2 Informative References

[Bluetooth]	A short-range wireless communications technology intended to replace the cables connecting portable and/or fixed devices while maintaining high levels of security. For more information, see URL:http://www.bluetooth.com/Bluetooth/Technology/
[Bonjour]	A networking discovery protocol from Apple Computer. For more information, see URL:http://developer.apple.com/networking/bonjour/
[ISO13818-1]	"Information technology – Generic coding of moving pictures and associated audio information: Systems", ISO/IEC 13818-1, URL:http://www.iso.org
[UpnP]	A set of networking protocols that include discovery from the UpnP Forum. For more information see URL:http://www.upnp.org

3. Terminology and Conventions

3.1 Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

All sections and appendixes, except “Scope” and “Introduction”, are normative, unless they are explicitly indicated to be informative.

3.2 Definitions

Ad Hoc Sharing	Sharing that is intended to allow a source Device to share specified Rights with a recipient Device in spontaneous, unplanned situations (e.g. sharing a song with a new group of friends at a party or playing a video on a hotel room TV while travelling).
Constraint	A restriction on a Permission over DRM Content (DRM V2.1).
Consume	To Play, Display, Print or Execute DRM Content on a Device or to render DRM Content on a Render Client.
Content	One or more Media Objects (DRM V2.1).
Copy	To make Rights existing on a source Device available for use by a recipient Device, without affecting availability on the source Device. Rights may be restricted on the recipient Device. Note: this is different from the V2.1 definition.
Device	A Device is the entity (hardware/software or combination thereof) within a user equipment that implements a DRM Agent. The Device is also conformant to the OMA DRM specifications. The Device may include a smart card module (e.g. a SIM) (DRM V2.1).
Device Rights Object	A Rights Object that is initially targeted to a specific entity. Subsequently, the Rights Object may be allowed to be targeted to other entities to be consumed, serially or in parallel, independently of membership in a Domain or User Domain.
Domain	A set of v2.x and/or SCE DRM Agents that can consume Domain Rights Objects.
Domain Rights Object	A Rights Object that is targeted to a specific v2.x Domain. The Rights Object can be consumed independently by each v2.x or SCE DRM Agent that is a member of the Domain.
DRM Agent	The entity in the Device that manages Permissions for Media Objects on the Device (DRM V2.1). In this document, the DRM Agent implements some or all the functionality defined in this specification.
DRM/Render Agent	An entity that is either a DRM Agent or a Render Agent.
DRM Content	Media Objects that are consumed according to a set of Permissions in a Rights Object (DRM V2.1).
DRM Requester	An entity that uses the interface defined by this specification.
DRM Time	A secure, non user-changeable time source. The DRM Time is measured in the UTC time scale (DRM V2.1).
Lending	The act of sharing such that the Shared Rights cannot be used on the source Device as long as the recipient Device is able to render the shared Content associated with the Shared Rights.
lsb	Least significant bit.
Media Object	A digital work e.g. a ring tone, a screen saver, or a Java game (DRM V2.1).

Move	To make Rights existing initially on a source Device fully or partially available for use by a recipient Device, such that the Rights or parts thereof that become usable on the recipient Device can no longer be used on the source Device.
Moving	The act of performing a Move.
msb	Most significant bit.
Partial Rights	A subset of a set of Rights, such that the Partial Rights are equally or more restrictive than those in the set.
Permission	Actual usages or activities allowed (by the Rights Issuer) over DRM Content.
Render Agent	The entity in a Render Client that manages the secure rendering of DRM Content on the Render Client.
Render Client	The entity (hardware, software or combination thereof) within a user equipment that implements a Render Agent. The Render Client is used to transiently render DRM Content.
Restore	Transferring the DRM Content and/or Rights Objects from an external location back to the Device from which they were backed up (DRM V2.1).
Rights	The collection of permissions and constraints defining under which circumstances access is granted to DRM Content.
Rights Issuer	An entity that issues Rights Objects to OMA DRM conformant Devices (DRM V2.1).
Rights Object	A collection of Permissions and other attributes that are linked to DRM Content. When used in the context of a Rights Object transfer, it also includes the State Information (for stateful Rights Objects) and other related meta data.
Shared Rights	Rights that can be consumed on multiple Devices, where the allowed distribution and consumption of the Rights among the Devices are specified by permissions in the Rights themselves or in the Domain Policy of the Domain for which the Rights were obtained.
Sharing	The act of providing Shared Rights from a source Device to a recipient Device, such that the recipient Device is able to render the shared content associated with the Shared Rights.
State Information	A set of values representing current state associated with Rights. It is managed by the DRM Agent only when the Rights contain any of the stateful constraints (e.g. interval, count, timed-count, accumulated, etc.).
User	The human user of a Device. The User does not necessarily own the Device (DRM V2.1).
User Domain	A set of v2.x and/or SCE DRM Agents that can consume User Domain Rights Objects.
User Domain Rights Object	A Rights Object that is targeted to a specific User Domain. Besides requiring membership in the User Domain, consumption may require being targeted to an SCE DRM Agent.

3.3 Abbreviations

A2A	Agent to Agent
AES	Advanced Encryption Standard
CD	Compact Disc
CEK	Content Encryption Key
CRL	Certificate Revocation List
DCF	DRM Content Format
DER	Distinguished Encoding Rules
DRM	Digital Rights Management
DVD	"Digital Versatile Disc" or "Digital Video Disc"
HMAC	Keyed-Hash Message Authentication Code
HTTP	Hyper Text Transfer Protocol
IV	Initialisation Vector
KDF	Key Derivation Function
MAC	Message Authentication Code
MAKE	Mutual Authentication and Key Exchange
MK	MAC Key
N/A	Not applicable
OMA	Open Mobile Alliance
OMNA	Open Mobile Naming Authority
(P)DCF	A DCF or a PDCF
REK	Rights Object Encryption Key
REL	Rights Expression Language
RFC	Request For Comments
RFU	Reserved for Future Use
RI	Rights Issuer
RO	Rights Object
ROID	Rights Object Identifier
RSA	Rivest-Shamir-Adelman public key algorithm
RSA-OAEP	RSA encryption scheme - Optimal Asymmetric Encryption Padding
RSA-PSS	RSA Probabilistic Signature Scheme
SAC	Secure Authenticated Channel
SCE	Secure Content Exchange
SCR	Static Conformance Requirement
SHA1	Secure Hash Algorithm
SK	Session Key
SRM	Secure Removable Media
URI	Uniform Resource Indicator
URL	Uniform Resource Locator
USB	Universal Serial Bus
WBXML	Wireless Binary XML
WiFi	Wireless Fidelity, also Wi-fi, Wifi, or wifi

3.4 Syntax Descriptions

The syntax descriptions used in this document are based on the method of syntax descriptions used in [ISO13818-1] and follow the conventions defined in Appendix B of [SRM-TS].

3.5 Conventions

The following conventions are used in this document:

```
Syntax definitions are described in this font.
```

Status codes are listed in this font.

Messages, data structures and fields are italized, e.g. *A2ARequest*. Fields within other fields are indicated by separating the names with a period (‘.’), e.g. *A2ARequest.MessageID*.

4. Introduction (Informative)

One of the goals of the Secure Content Exchange (SCE) Enabler is to extend OMA DRM V2.1 [DRM-v2.1] to enable the moving of Rights Objects from one Device to another Device (without the involvement of any network entity) and the ad hoc sharing of DRM Content with Devices the User encounters in unplanned or impromptu situations. Examples of when ad hoc sharing may be applicable include Users who want to render their content on a television set at a friend's house or in a hotel room while the User is travelling, or a User who wants to borrow DRM Content for a period of time. The ad hoc sharing part of the SCE Enabler enforces temporal and proximity-based restrictions that are defined by the Rights Issuer (RI), e.g. DRM Content can only be shared with a Device that is in close proximity to the subscriber's Device.

The SCE Enabler extends DRM V2.1 with the flexible sharing of DRM Content between Devices. Some of these enhancements provide the following benefits to subscribers, content providers and operators:

- Subscribers benefit from increased flexibility to share and render their Content in ways that were previously not possible. They perceive a level of convenience in their digital media service that rivals the user experience offered by physical media such as CDs and DVDs, which can be played on any device available.
- Content providers benefit from an increase in content purchases, while enjoying the protection against content piracy that DRM provides.
- The added appeal of flexible sharing to subscribers makes the operator's mobile digital media service competitive with wireline-based services and physical media, resulting in an increase in the number of service subscribers and content purchases (and hence an increase in operator revenue).

This specification reuses as many common items from the Secure Removable Meda (SRM) specification [SRM-TS] as possible. For Devices supporting both this specification and SRM, this will minimize the amount of software code needed to implement both specifications. However, there are some differences and developers should be aware of those differences.

5. Overview

5.1 Architecture

This document only specifies the SCE-7-A2AP interface which is defined in [SCE-AD]. Other interfaces are defined in other specifications of the SCE Enabler. The SCE-7-A2AP interface is illustrated in the figure below.

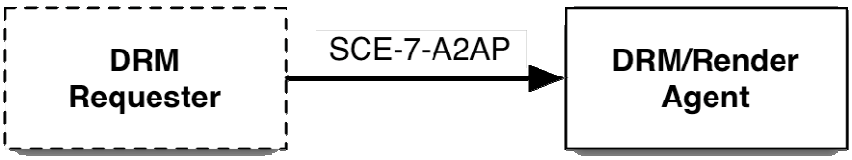

Figure 1: SCE-7-A2AP

A Render Agent is explained in section 5.4. For conciseness, the term "A2A" is used to refer to the SCE-7-A2AP interface throughout the remainder of this document. From a logical design point of view, the DRM Requester can be any entity. However, because of certain security requirements, a DRM Requester has to be a DRM Agent to fully participate in the protocol.

5.2 Trust Model

This specification follows the approach taken by [DRM-v2.1] for trust models with the addition that the trust model MUST make Certificate Revocation Lists (CRLs) available to DRM Agents. Therefore, other than providing CRLs, any trust model used by [DRM-v2.1] can be used to support this specification.

Devices supporting this specification can belong to multiple trust models. However, this specification follows the approach taken by [SRM-TS] where the exchange of Rights Objects is performed using credentials from the same trust model.

A trust model is identified by its root of trust, i.e. the certificate of the Root Certificate Authority for the trust model.

5.2.1 Revocation Status Checking

There is a requirement for the DRM Requester and the DRM/Render Agent to mutually authenticate themselves. Part of this process is to check the revocation status of the other entity. For this specification, the revocation status checking is done via a Certificate Revocation List (CRL). The trust model MUST provide one or more repositories where a DRM Requester, a Render Agent (see section 5.4) or a DRM Agent can get a current CRL. How to access the CRL repositories is outside the scope of this specification. DRM Requesters, Render Agents and DRM Agents MUST get a new CRL when the CRL they have expires. If a DRM Reqeustor, Render Agent or DRM Agent has an expired CRL, it MUST NOT perform the mutual authentication and key exchange process (see section 9.2).

This document provides a mechanism for the "viral" distribution of CRLs between a DRM Requester and a DRM Agent (see sections 9.4, 9.5 and 9.6).

5.3 Partial Rights

The term Partial Rights applies only to Stateful Rights which have either the <count> or <timed-count> constraint elements. In this case, if any count remains, then a portion of the remaining count can be Moved from a DRM Requester to a DRM Agent.

Prior to this enabler, Devices did not have to manage Rights Objects with the same ROID. Because the A2A Interface allows the Moving of Partial Rights, Devices can receive Partial Rights that have an ROID that is identical to the ROID of a Rights

Object that is currently installed in the Device. In this case, the received remaining count is added to the existing remaining count (unless adding the received remaining count would exceed the original count in the Rights Object).

5.4 Render Client

A Render Client is a device that is somewhat similar to an OMA DRM Device, but has the following characteristics:

- It has no capability to handle (e.g. receive, parse, etc) Rights Objects.
- It has no capability to store Rights Objects.
- It can decrypt a (P)DCF when given the Content Encryption Key (CEK).
- After rendering the DRM Content once, it loses all knowledge of the CEK.
- It has a Render Agent (that supports DRM Time).

A DRM Agent and a Render Agent can be differentiated via their certificates.

Some trust models may not allow Render Clients.

5.5 State Information Consistency

A Stateful Rights Object (or portions of) may be Moved to another Device. The recipient Device MUST check that the current *StateInformation* (see section 8.21) that is transferred is consistent with the actual Rights Object. This consistency check means the following:

- For the <interval> constraint, the date-time field in the *StateInformation* structure MUST be all zeros or the specified date MUST be less than or equal to the current DRM Time plus the <interval> value.
- For the <count> and <timed-count> constraint, the *remainingCount* field in the *StateInformation* structure MUST be less than or equal to the corresponding <count> or <timed-count> value.
- For the <accumulated> constraint, the *accumulatedTime* field in the *StateInformation* structure MUST be less than the <accumulated> value.

6. The A2A Protocol

The A2A interface uses a client-server communications model (similar to the Internet Web). This specification defines a protocol using a set of requests and responses between a DRM Requester (the client) and a DRM Agent (the server). A DRM Requester sends a request to the DRM Agent. The DRM Agent processes the request and sends a response to the DRM Requester. Once a DRM Requester sends a request, it waits for a response from the DRM Agent before sending another request.

All OMA Devices supporting the A2A Interface MUST implement both the DRM Agent and the DRM Requester functionality. A Device is not required to function as both a DRM Agent and a DRM Requester simultaneously.

How the requests and responses are transported between the DRM Requester and the DRM Agent are outside the scope of this specification. Potential transports include, but are not limited to, USB, Bluetooth, IrDA and WiFi.

Two Devices must discover each other before any A2A functionality can take place. It is during the discovery phase that the roles of DRM Requester and DRM Agent are assigned. How the discovery is performed and how the roles are assigned are outside the scope of this specification. Potential discovery mechanisms include, but are not limited to, [UPnP] and [Bonjour].

6.1 Messages, Operations and Transactions

A *message* is either a request or a response. The sequence of sending a request and getting the response is called an *operation*. Certain A2A functionality requires two or more operations. This set of operations is called a *transaction*. Certain implementations of a DRM Agent MAY require that transactions be performed as defined in this document, i.e. that the set of operations be done in sequence. Other implementations MAY allow the interleaving of operations that are not part of a transaction.

The operations and transactions are described in section 8. Whenever an operation or a transaction is not successfully completed, the User MAY be informed. Error recovery is described in section 6.3.

6.2 Message Syntax

The syntax for all A2A messages is defined in the following sub-sections.

6.2.1 Request Syntax

All A2A requests follow one of two generic syntaxes:

- Plain request – this type of request does not require any integrity protection.
- Protected request – this type of request requires integrity protection.

The syntax for a plain request is defined as follows:

```
A2ARequest(){
   MessageType()
   Body()
   ExtensionsContainer()
}
```

The fields are defined as follows:

- *Message*Type – this field identifies the request type. The list of valid message types is described in section 6.2.3.
- *Body* – this field contains the body of the request. There is not just one definition of *Body*, but each request defined in section 9 defines the data structure for this field. Some requests may have an empty *Body* field.
- *ExtensionsContainer* – this field allows for extending a request in future revisions of this specification. This field is described in section 6.2.5.

The syntax for a protected request is defined as follows:

```
A2AProtectedRequest(){
   MessageType()
   replayCounter       32     uimsbf
   Body()
   Extensions()
   Hmac()
}
```

The fields are defined as follows:

- *MessageType* – this field contains the request type. The list of valid message types is described in section 6.2.3.
- *replayCounter* – this field constains a 32 bit unsigned integer that is used to provide replay protection. The use of this field is described in section 7.3.
- *Body* – this field contains the body of the request. There is not just one definition of *Body*, but each request defined in section 9 defines the data structure for this field. Some requests may have an empty *Body* field.*A2ARequest* – This field contains a plain request as defined above.
- *ExtensionsContainer* – this field allows for extending a request in future revisions of this specification. This field is described in section 6.2.5.
- *Hmac* – this field contains an HMAC value that provides integrity over the *MessageType, replayCounter*, *Body* and the *Extensions* fields. This field is defined in section 8.5.

An *A2AProtectedRequest* MUST be sent using a Secure Authenticated Channel (see section 9.2).

Example requests are provided in Appendix C.

6.2.2 Response Syntax

All A2A responses follow one of two generic syntaxes:

- Plain response – this type of response does not require any integrity protection.
- Protected response – this type of response requires integrity protection.

The syntax for a plain response is defined as follows:

```
A2AResponse(){
   MessageType()
   Status()
   if( Status == 0 ){
      Body()
   }
   ExtensionsContainer()
}
```

The fields are defined as follows:

- *MessageType* – this field contains the response type. The list of valid message types is described in section 6.2.3.
- *Status* – this field contains the result of processing a request. The list of allowed values are described in section 6.2.4.
- *Body* – this field contains the body of the response. There is not just one definition of *Body* but each response defined in section 9 defines the data structure for this field. This field will exist only if *Status* is set to Success (see section 6.2.4). Some responses may have an empty *Body* field.
- *ExtensionsContainer* – this field allows for extending a response in future revisions of this specification. This field is defined in section 6.2.5.

The syntax for a protected response is defined as follows:

```
A2AProtectedResponse(){
   MessageType()
   replayCounter       32     uimsbf
   Status()
   if( Status == 0 ){
```

```
        Body()
    }
    Extensions()
    Hmac()
}
```

The fields are defined as follows:

- *MessageType* – this field contains the request type. The list of valid message types is described in section 6.2.3.
- *replayCounter* – this field constains a 32 bit unsigned integer that is used to provide replay protection. The use of this field is described in section 7.3.
- *Status* – this field contains the result of processing a request. The list of allowed values is described in section 6.2.4.
- *Body* – this field contains the body of the response. There is not just one definition of *Body* but each response defined in section 9 defines the data structure for this field. Some responses may have an empty *Body* field. This field will exist only if *Status* is set to Success (see section 6.2.4).
- *ExtensionsContainer* – this field allows for extending a response in future revisions of this specification. This field is defined in section 6.2.5.
- *Hmac* – this field contains an HMAC value that provides integrity over the *MessageType*, *replayCounter*, *Status*, *Body* and *Extensions* fields. This field is defined in section 8.5.

An *A2AProtectedResponse* MUST be sent using a Secure Authenticated Channel (see section 9.2).

Example responses are provided in Appendix C.

6.2.3 Message Types

The type of each A2A message is determined via the *MessageType* field. This field is defined as follows:

```
MessageType(){
   messageType          8      uimsbf
}
```

The *messageType* field is an 8 bit, unsigned integer that contains the message type. The following table lists all the message types defined in this version of this specification.

Table 1: Message Types

Value	Description
0	A2A Agent Hello Request
1	A2A Agent Hello Response
2	Authentication Request
3	Authentication Response
4	Key Exchange Request
5	Key Exchange Response
6	Change SAC Request
7	Change SAC Response
8	CRL Query Request
9	CRL Query Response
10	Put CRL Request
11	Put CRL Response
12	Get CRL Request
13	Get CRL Response
14	Move RO Request
15	Move RO Response

16	Move REK Request
17	Move REK Response
18	Share RO Request
19	Share RO Response
20	Lend RO Request
21	Lend RO Response
22	Lend Release Request
23	Lend Release Response
24	Render Request
25	Render Response
26	Copy RO Request
27	Copy RO Response
28 – 255	RFU

By definition, a DRM/Render Agent only receives requests. If it receives a message with a response type or a type marked as "RFU", the DRM/Render Agent SHALL send an *A2AResponse* with *Status* set to RequestNotSupported (see section 6.2.4).

By definition, a DRM Requester only receives responses. If it receives a message with a request type or a type marked as "RFU", the DRM Requester SHALL treat the message as an error to the operation or transaction.

6.2.4 Status

The *Status* field of a response indicates the result of the DRM Agent processing a request. It is defined as follows:

```
Status(){
   status              8     uimsbf
}
```

The following table lists all the values that are valid for this version of this document.

Table 2: Status Values and Names

Value	Name	Description
0	Success	The request was successfully processed.
1	TrustAnchorNotSupported	The trust anchor is not supported.
2	CertificateChainVerificationFailed	The verification of a certificate chain failed.
3	FieldDecryptionFailed	The decryption of a field failed.
4	RandomNumberMismatched	A random number did not match an expected value.
5	VersionMismatched	A version did not match an expected value.
6	SACNotEstablished	A SAC have not been established under the requested trust model.
7	OldCrl	A newly received CRL is older than the current CRL.
8	CrlVerificationFailed	The verification of a CRL failed.
11	CrlNotFound	CRL Not Found
12	IntegrityVerificationFailed	The integrity verification of the request failed.
13	NotEnoughSpace	Not Enough Space
17	RequestNotSupported	The DRM/Render Agent does not support the request.
18	RiCertificateChainNotFound	RI Certificate Chain Not Found
21	InvalidField	The request contains an invalid field.
22	UnexpectedRequest	The request was not expected.

Value	Name	Description
23	NotADomainMember	The Device is not a member of the User Domain for which the operation was meant.
24	NoCommonTrustAnchor	A common trust anchor was not found.
25	CrlExpired	The DRM/Render Agent has an expired CRL.
26	DrmRequesterRevoked	The DRM Requester is listed on a CRL.
27	InvalidRightsObject	The DRM Agent considers the Rights Object to be invalid.
28	UnknownHandle	The DRM Agent has no knowledge of the handle.
29	RequestNotSupported	The DRM/Render Agent does not support the request type.
30 – 255	RFU	Reserved For Future Use

6.2.5 Extending a Message

Future specifications MAY use the *ExtensionsContainer* field to extend messages defined in this document without changing the definitions specified in this document. The ExtentionsContainer is defined as follows:

```
ExtensionsContainer(){
   nbrOfEntries          8     uimsbf
   for( i = 0; i < nbrOfEntries; i++ ){
      extensionType      8     uimsbf
      size              16     uimsbf
      Extension()
   }
}
```

The fields are defined as follows:

- *nbrOfEntries* – this field contains the number of extensions present in this container as an 8 bit unsigned integer.
- *extensionType* – this field identifies the extension. This value MUST be unique in the context of a message. There are no extensions defined in this version of this specification.
- *size* – this field contains the size (or length) of the *Extension* field in a 16 bit unsigned integer.
- *Extension* – this field contains the actual extension. The content of this field will depend on the particular extension (as identified by the particular message and *extensionType*) and will be defined in future versions of this specification.

If a DRM Requester or a DRM Agent conformant to this specification receives an *ExtensionContainer* with one or more *Extension*s, the DRM Requester or DRM Agent SHALL ignore the *Extension*s.

Extensibility in Future Specifications (Informative)

When an extension is specified in a future specification, the extension can either be included in all messages independent of the version of the DRM Requesters and DRM Agents involved or only included when communication between DRM Requesters and DRM Agents of appropriate versions occurs. The decision on when and where a certain extension is to be included will be taken when the new specification is written.

Extensions can be mandated in future specifications. This means DRM Requesters and DRM Agents conformant to those specifications must include the extensions, even though older DRM Requesters and DRM Agents will ignore them. The extensions have to be designed in such a way that this does not open an attack opportunity.

6.3 Error Recovery

Under normal circumstances, a DRM Requester sends a request to a DRM/Render Agent and a short time later receives a response from the DRM/Render Agent. However, under certain circumstances, the DRM Requester may not receive a response after sending a request. It is anticipated that a DRM Requester will time out if a response from the DRM/Render Agent is not received within a certain wait period. Upon such time-out, the DRM Requester MAY terminate the

operation/transaction and inform the User, or it MAY try sending the request again. If the DRM Requester is going to retry sending the request and the request is an *A2AProtectedRequest*, then the DRM Requester MUST increment the *replayCounter* field (see section 7.3). This document does not specify the wait period or how many times the DRM Requester re-sends a request. These are left as implementation choices or may be specified by a trust model.

7. Secure Authenticated Channel

Certain operations and transactions require integrity and confidentiality of the data being exchanged. For this to take place, a secure logical channel, called a Secure Authenticated Channel (SAC), must be established between the DRM Requester and the DRM Agent. A SAC provides the following characteristics:

- Entity authentication – an entity that does not represent itself with a valid credential cannot participate in the communications.
- Message integrity – a message cannot be modified without detection.
- Replay protection – an attacker cannot capture a message and then replay it without detection.
- Confidentiality – an entity that did not establish the SAC cannot understand the portions of the message that are encrypted.

The Mutual Authentication and Key Exchange (MAKE) transaction, described in section 9.2, is used to establish a SAC. Once established, it is possible to exchange a protected message, i.e. either an *A2AProtectedRequest* or an *A2AProtectedResponse*.

7.1 Entity Authentication

Both the DRM Requester and the DRM Agent MUST have their own X.509 certificate that is used to authenticate themselves to each other as part of establishing the SAC.

7.2 Message Integrity

Once the SAC is established, when an *A2AProtectedRequest* or an *A2AProtectedResponse* is sent, it is integrity protected. If the protected message is modified between the sender and the receiver, the receiver will detect that the message has been modified.

7.3 Replay Protection

Replay protection prevents an attacker from recording protected message and then attempting to send them again at a later time without detection. Replay protection is based on counters kept by both entities involved in the SAC. The counter is maintained in the SAC context (see section 9.2.6). The value of a counter is sent in a protected message via the *replayCounter* field (see sections 6.2.1 and 6.2.2).

The replay protection mechanism works as follows.

1. Sending an *A2AProtectedRequest*
   - The DRM Requester copies its *currentReplayCounterR* to the *A2AProtectedRequest.replayCounter* field. After the protected request is sent, the DRM Requester increments its *currentReplayCounterR*value by 1.
   - When an *A2AProtectedRequest* is received, the DRM/Render Agent checks the *A2AProtectedRequest.replayCounter* field against its *currentReplayCounterA* value. This check of the replay counter can be performed before the integrity of the message is checked. But the integrity check MUST be performed before further processing, in particular, before any decryption is performed. If *replayCounter* ≥ *currentReplayCounterA*, the DRM/Render Agent sets *currentReplayCounterA* = *A2AProtectedRequest.replayCounter* + 1. Then the DRM/Render Agent proceeds with processing the request. If *replayCounter* < *currentReplayCounterA*, the DRM/Render Agent SHALL return an *A2AProtectedResponse* with *Status* set to IntegrityVerificationFailed.
2. Sending an *A2AProtectedResponse*
   - The DRM/Render Agent copies its *currentReplayCounterA* to the *A2AProtectedResponse.replayCounter* field. After the protected response is sent, the DRM/Render Agent increments its *currentReplayCounterA* value by 1.
   - When an *A2AProtectedResponse* is received, the DRM Requester checks the *A2AProtectedResponse.replayCounter* field against its *currentReplayCounterR* value. This check of the replay counter can be performed before the integrity of the message is checked. But the integrity check MUST be performed before further processing, in particular, before any decryption is performed. If *replayCounter* ≥ *currentReplayCounterR*, the DRM Requester sets

currentReplayCounterR = *A2AProtectedResponse.replayCounter* + 1. Then the DRM Requester proceeds with processing the protected response. If *replayCounter* < *currentReplayCounterR*, the DRM Requester SHALL initiate a new MAKE transaction in order to resync the replay counters.

Note that the counters will roll over to 0 after 4,294,967,295 messages are sent under a SAC. A trust model MAY require that a new SAC be established when this happens.

7.4 Confidentiality

When a message (or a portion of a message) requires confidentiality, the data is encrypted using an encryption algorithm and key established as part of the SAC (see section 9.2.5).

8. Common Data Structures

The messages defined in this specification have a number of common data structures. These are defined in the following sub-sections.

8.1 Octet Strings

An octet string holds variable length binary data. There are two types, one for "short" octet strings and the other for "long" octet strings. The short octet string is defined as follows:

```
OctetString8(){
   length               8     uimsbf
   for( i = 0; i < length; i++ ){
      octet             8     uimsbf
   }
}
```

The long octet string is defined as follows:

```
OctetString16(){
   length              16     uimsbf
   for( i = 0; i < length; i++ ){
      octet             8     uimsbf
   }
}
```

The fields are defined as follows:

- *length* – This field contains the number of octets in the octet string. For an *OctetString8*, the range is 0 – 255. For an *OctetString16*, the range is 0 – 65535.
- *octet* – This field contains one octet (8 bits) of the octet string.

8.2 Version

Version is used to represent a version number. It is defined as follows:

```
Version(){
   major                4     uimsbf
   minor                4     uimsbf
}
```

The fields are defined as follows:

- *major* – This field contains the major portion of the version. The range is 1 – 15.
- *minor* – This field contains the minor portion of the version. The range is 0 – 15.

For this version of this specification, major = 1 and minor = 0.

8.3 Hash

Hash is used to hold a hash value. It is defined as follows:

```
Hash(){
   OctetString8()
}
```

The fields are defined as follows:

- *OctetString8* – This field contains the hash value as an *OctetString8* which is defined in section 8.1. The hash algorithm is either specified in this document or it is negotiated between the DRM Requester and the DRM Agent.

8.4 Trust Anchor

TrustAnchor is used to represent the identification of a trust model. It is defined as follows:

```
TrustAnchor(){
   Hash()
}
```

The fields are defined as follows:

- *Hash* – This field contains the SHA-1 hash of the DER-encoded subjectPublicKeyInfo component of the trust model's Root CA certificate. It is of type *Hash* which is defined in section 8.3.

8.5 Entity ID

EntityID is used to represent the identification of an entity. It is defined as follows:

```
EntityID(){
   Hash()
}
```

The fields are defined as follows:

- *Hash* – This field contains the SHA-1 hash of the DER-encoded subjectPublicKeyInfo component of the entity's certificate. This field is defined in section 8.3.

8.6 Trust Anchor and Entity ID Pair List

TrustAnchorAndEntityIdPairList is a list of *TrustAnchor* and *EntityID* pairs. It is defined as follows:

```
TrustAnchorAndEntityIdPairList(){
   nbrOfEntries         8     uimsbf
   for( i = 0; i < nbrOfEntries; i++ ){
      TrustAnchor()
      EntityID()
   }
}
```

The fields are defined as follows:

- *nbrOfEntries* – this field contains the number of *TrustAnchor* and *EntityID* pairs as an 8 bit unsigned integer. There MUST be at least 1 pair; therefore the range is 1 – 255.
- *TrustAnchor* – this field identifies a trust model. This field is defined in section 8.4.
- *EntityID* – this field identifies the entity under the *TrustAnchor*. If an entity has more than one identifier under a particular trust model, only one identifier is allowed in the list. This field is defined in section 8.5.

8.7 HMAC

Hmac is used to hold an HMAC value. *Hmac* is defined as follows:

```
Hmac(){
   OctetString8() //Defined in section 8.1
}
```

The fields are defined as follows:

- *OctetString8* – This field contains the HMAC value as an *OctetString8* which is defined in section 8.1. The HMAC algorithm is either specified in this document or it is negotiated between the DRM Requester and the DRM Agent.

8.8 X.509 Certificates

CertificateChain is used to hold the certificate chain of an entity. It is defined as follows:

```
CertificateChain(){
   nbrOfEntries          8     uimsbf
   for( i = 0; i < nbrOfEntries; i++ ){
      Certificate()
   }
}

Certificate(){
   OctetString16()
}
```

The fields are defined as follows:

- *nbrOfEntries* – This field contains the number of certificates in the chain as an 8 bit unsigned integer.
- *Certificate* – This field contains one X.509 certificate as an *OctetString16* which is defined in section 8.1.

The order of the certificates MUST be as follows: first certificate is the entity's certificate. The first certificate is followed by any intermediate CA certificates, in order of signing, up to but not including the root certificate.

8.9 Algorithm

Algorithm is used to identify a security related algorithm. It is defined as follows:

```
Algorithm(){
   algorithmId           8     uimsbf
}
```

The following table contains the list of algorithms defined in this specification.

Table 3: Supported Algorithms

Value	Description	Reference
0	SHA-1 – This is the default hash algorithm.	[SHA1]
1	HMAC-SHA1 – This is the default HMAC algorithm.	[RFC2104]
2	AES-128-CBC – Defined to be compatible with [SRM-TS]. It is not used in this specification.	[AES-MODES]
3	RSA-OAEP – This is the default asymmetric key encryption algorithm.	[RFC3447]
4	DRMV2-KDF – This is the default KDF algorithm, the KDF from OMA DRM v2.1.	[DRM-DRM-v2.1]
5	AES-128-CTR – This is the default symmetric key algorithm.	[AES-MODES]

8.10 Algorithm List

AlgorithmList contains a list of algorithms. It is defined as follows:

```
AlgorithmList(){
   nbrOfEntries          8      uimsbf
   for( i = 0; i < nbrOfEntries; i++ ){
      Algorithm()
   }
}
```

The fields are defined as follows:

- *nbrOfEntries* – This field contains the number of algorithms in the list as an 8 bit unsigned integer. If the number is 0, then the default algorithm is used.
- *Algorithm* – This field contains one algorithm as defined in section 8.9.

8.11 Encrypted Data

EncryptedData is used to hold data that has been encrypted. The encryption algorithm is negotiated between the DRM Requester and the DRM Agent. *EncryptedData* is defined as follows:

```
EncryptedData(){
   Iv()
   CipherText()
}

Iv(){
   OctetString8()
}

CipherText(){
   OctetString16()
}
```

The fields are defined as follows:

- *Iv* – This field contains the IV if required by the encryption algorithm as an *OctetString8* that is defined in section 8.1. If not required, the length is set to 0.
- *CipherText* – This field contains the actual encrypted data as an *OctetString16* which is defined in section 8.1.

8.12 Encrypted CEK

EncryptedCek is used to send a CEK that is encrypted using an algorithm and key established during a MAKE transaction (see section 9.2). It is defined as follows:

```
EncryptedCek(){
   EncryptedData()
}
```

8.13 Hashed CEK

HashedCek is used to send the SHA-1 hash over the CEK. It is defined as follows:

```
HashedCek(){
   Hash()
}
```

8.14 Random Number

RandomNumber contains a string of random octets. It is defined as follows:

```
RandomNumber(){
   OctetString8()  //Defined in section 8.1
}
```

8.15 Rights Object ID

RoID contains an ROID. It is defined as follows:

```
RoID(){
   OctetString16()  //Defined in section 8.1
}
```

8.16 String80

String80 contains a variable length string with a maximum length of 80 bytes. It is defined as follows:

```
String80(){
   OctetString8()  //Defined in section 8.1
}
```

8.17 X.509 Certificate Revocation Lists (CRLs)

Crl is used to hold one X.509 CRL. It is defined as follows:

```
Crl(){
   OctetString16() //Defined in section 8.1
}
```

CrlList is used to hold a list of CRLs. It is defined as follows:

```
CrlList(){
   nbrOfEntries         8     uimsbf
   for( i = 0; i < nbrOfEntries; i++ ){
      Crl()
   }
}
```

The fields are defined as follows:

- *nbrOfEntries* – This field contains the number of CRLs in the list as an 8 bit unsigned integer.
- *Crl* – This field contains one X.509 CRL.

CrlIdList is used to hold a list of CRL identifiers. It is defined as follows:

```
CrlIdList(){
   nbrOfEntries         8     uimsbf
   for( i = 0; i < nbrOfEntries; i++ ){
      CrlIssuerID()  //Defined below
      CrlNumber()    //Defined below
   }
}

CrlIssuerID(){
   EntityID()  //Defined in section 8.5
```

```
}

CrlNumber(){
   OctetString8()  //Defined in section 8.1
}
```

The fields are defined as follows:

- *nbrOfEntries* – this field contains the number of CRL Issuer ID and Number pairs as an 8 bit unsigned integer.
- *CrlIssuerID* – this field identifies the issuer of the CRL. It is of type *EntityID* which is defined in section 8.5.
- *CrlNumber* – this field contains the number of the CRL per [RFC3280].

8.18 Asset ID

AssetID contains an identifier for a DRM Content. It is defined as follows:

```
AssetID(){
   OctetString16()  //Defined in section 8.1
}
```

8.19 CEK Info

CekInfo is used to send for each asset either the CEK in an *EncryptedCek* field (see section 8.12), or the SHA-1 hash over the CEK in an *HashedCEK* field. It has the following definition:

```
CekInfo() {
   noCEKs              16     uimsbf
   for( i=0; i<noAssets; i++ ) {
      AssetID()
      cekOrCekHash      1     bslbf
      rfu               7     bslbf
      if( cekOrCekHash == 0 )
         EncryptedCek()
      else
         HashedCek()
   }
}
```

The fields are defined as follows:

- *noCEKs* - this field indicates the number of CEKs the CekInfo field describes.

For each CEK (or asset), there are the following fields:

- *AssetID* - this field contains the Asset ID from the Asset associated with the CEK.
- *cekOrCekHash* - if this bit has the value 0, an *EncryptedCek* field follows. If this bit has the value 1, a *Hash* field follows.
- *rfu* – this field is reserved for future use. It MUST be set to all zeros.
- *EncryptedCek* - this field contains the encrypted CEK for this asset. See section 8.12 for more details.
- *HashedCek* - this field contains the hash over the CEK for this asset. See section 8.13 for more details.

8.20 Rights Object Container

A *RightsObjectContainer* holds a Rights Object that is being sent from a DRM Requester to a DRM Agent. Consistent with the structure of a Rights Object, the Rights Object Container consists of the <rights> element and the <signature> element in the ROPayload [DRM-v2.1]. Unlike the ROPayload, the *RightsObjectContainer* does not include an <encKey> element. Unlike the ProtectedRO [DRM-v2.1], the *RightsObjectContainer* does not include a <mac> element (generated by the RI over the ROPayload) that is checked as part of [DRM-v2.1] RO installation. Consequently, “install” as used in this Technical

Specification does not include such <mac> element verification. The contents of the <rights> element of the Rights Object Container MUST be canonicalised as Exclusive Canonical XML format, as specified in [XC14N].

This field is defined as follows:

```
RightsObjectContainer(){
   OctetString16()  //Defined in section 8.1
}
```

The fields are defined as follows:

- *OctetString16* – this field (see section 8.1) contains an XML document of type oma-dd:RightsObjectContainer. It is instantiated as a <oma-dd:roContainer> element and contains the <rights> element and the <signature> element from the ROPayload as described above. The XML schema is as follows:

```
<!--Rights Object Container Definitions -->
<element name="roContainer" type="oma-dd:RightsObjectContainer">
<complexType name=”RightsObjectContainer”>
   <sequence>
      <element name=”rights” type=”o-ex:rightsType”/>
      <element name=”signature” type=”ds:SignatureType”/>
   </sequence>
</complexType>
```

8.21 State Information

StateInformation holds the state information of Rights to be Moved from a DRM Requester to a DRM Agent, i.e. the Rights that become available to the DRM Agent after the Move. It may represent the current remaining Rights on the DRM Requester (in case of a full Move), or it may be a subset of the remaining Rights (in case of a Partial Move). It is defined as follows:

```
StateInformation(){
   // Length of StateInfo
   length              16     uimsbf
   StateInfo()  //Defined below
}

StateInfo(){
   nbrOfAssetIDs        8     uimsbf
   for( i = 0; i < nbrOfAssetIDs; i++ ){  //<asset> elements
      AssetID()  //Defined in section 8.18
   }
   nbrOfPermissions     8     uimsbf
   for( i = 0; i < nbrOfPermissions; i++ ){  //<permission> elements
      PermissionState()  //Defined below
   }
}

PermissionState() {
   constraintPresent    1     bslbf
   assetPresent         1     bslbf
   playPresent          1     bslbf
   displayPresent       1     bslbf
   executePresent       1     bslbf
   printPresent         1     bslbf
   exportPresent        1     bslbf
   movePresent          1     bslbf
   copyPresent          1     bslbf

   // for future extension: all zeros now
   rfu                  7     bslbf

   if( constraintPresent ){
      ConstraintState()  //Defined below
   }
   if( assetPresent ){
       AssetID()  //Defined in section 8.18
   }
   if( playPresent ){
      ConstraintState()  //Defined below
   }
   if( displayPresent ){
      ConstraintState()  //Defined below
   }
   if( executePresent ){
      ConstraintState()  //Defined below
   }
   if( printPresent ){
      ConstraintState()  //Defined below
   }
   if( exportPresent ){
      ConstraintState()  //Defined below
   }
   if( movePresent ){
```

```
      ConstraintState()  //Defined below
   }
}
   if( copyPresent ){
      ConstraintState()  //Defined below
   }

ConstraintState() {
   countPresent         1     bslbf
   timedCountPresent    1     bslbf
   intervalPresent      1     bslbf
   accumulatedPresent   1     bslbf
   permissionLost       1     bslbf
   rfu                  3     bslbf

   if( countPresent ){         // For <count>
      remainingCount   32     uimsbf
   }
   if( timedCountPresent ){    // For <timed-count>
      remainingCount   32     uimsbf
   }
   if( intervalPresent ){      //For <interval>
      // YYYY-MM-DDThh:mm:ssZ [ISO8601]
      // All zeros if the asset has NOT been rendered
      for( i = 0; i < 20; i++ ) {
         char                 8  uimsbf
      }
   }
   if( accumulatedPresent ){   //For <accumulated>
      accumulatedTime  32     uimsbf //upto 2^32 seconds
   }
}
```

The fields are defined as follows:

- *length* – this field contains the length of the *StateInfo* structure in a 16 bit unsigned integer.
- *nbrOfAssetIDs* – this field contains the number of *AssetID*s in an 8 bit unsigned integer.
- *AssetID* – this field contains one Asset ID and is defined in section 8.18.
- *nbrOfPermissions* – this field contains the number of *PermissionState*s in an 8 bit unsigned integer.
- *constraintPresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to all permissions in the Rights Object.
- *assetPresent* – this is a boolean field, that if true, indicates that an *AssetID* field is present.
- *playPresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <play> permission.
- *displayPresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <display> permission.
- *executePresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <execute> permission.
- *printPresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <print> permission.
- *exportPresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <export> permission.

- *movePresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <move> permission.
- *copyPresent* – this is a boolean field, that if true, indicates that a *ConstraintState* field is present that is applicable to the <copy> permission.
- *rfu* – this field is reserved for future use. It MUST be set to all zeros.
- *ConstraintState* – this field contains constraint and state information. It is defined below.
- *countPresent* – this a boolean field, that if true, indicates that a *remainingCount* field is present that is applicable to the <count> constraint.
- *timedCountPresent* – this a boolean field, that if true, indicates that a *remainingCount* field (a 20 byte string) is present that is applicable to the <timed-count> constraint.
- *intervalPresent* – this a boolean field, that if true, indicates that a 20 character string is present that is applicable to the <interval> constraint.
- *accumulatedPresent* – this a boolean field, that if true, indicates that a *accumulatedTime* field is present that is applicable to the <accumulated> constraint.
- *permissionLost* – this boolean field, if true, indicates that the associated permission is lost and cannot be exercised.
- *remainingCount* – this field contains the remaining count value for a <count> constrain as a 32 bit unsigned integer.
- *char* – this field contains one ASCII character of a 20 character string that represents an end date after which the permission SHALL NOT be granted. The format of the string is “YYYY-MM-DDThh:mm:ssZ” as specified in [ISO8601].
- *accumulatedTime* – this field contains the accumulated time value, in seconds, for an <accumulated> constraint as a 32 bit unsigned integer.

9. A2A Operations and Transactions

The operations and transactions of the A2A interface are defined in the following sub-sections.

9.1 A2A Hello Operation

The A2A Hello operation allows the DRM Requester and the DRM/Render Agent to exchange information about each other. This operation can be performed at any time. However, it will normally be performed immediately after the DRM Requester discovers the DRM/Render Agent (note that discovery is outside the scope of this specification). Completion of the A2A Hello operation is a prerequisite for performing the MAKE transaction. The following figure illustrates the A2A Hello operation.

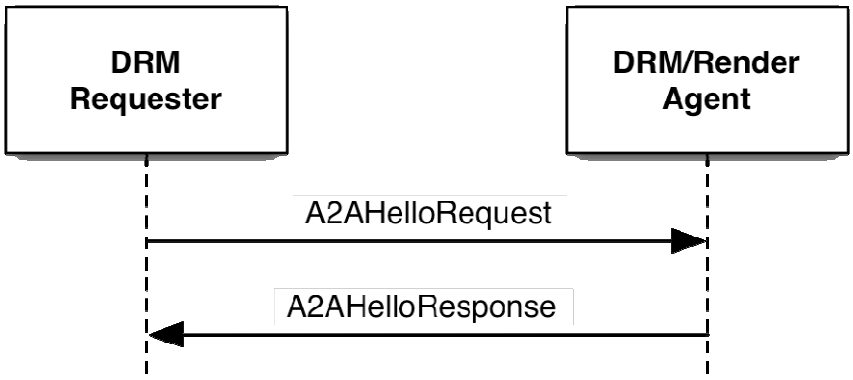

Figure 2: A2A Hello Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester generates an *A2AHelloRequest* using the highest interface version supported by the DRM Requester and the trust anchors and IDs it has.
2. The DRM Requester sends the *A2AHelloRequest* to the DRM/Render Agent.
3. The DRM/Render Agent processes the request as follows:
    a. It validates the fields of the *A2AHelloRequest*. If any field is invalid, it sets *A2AHelloResponse.Status* to InvalidField and continues with step 4.
    b. It checks if it has a *TrustAnchor* from the *A2AHelloRequest.Body.TrustAnchorAndEntityIdPairList* field in common. If it does not have a common *TrustAnchor*, then it sets *A2AHelloResponse.Status* to NoCommonTrustAnchor and continues with step 4.
    c. It sets *A2AHelloResponse.Body.SelectedVersion* to the minimum of *A2AHelloRequest.Body.Version* and highest interface version supported by the DRM/Render Agent. It also saves this value for use in a MAKE transaction.
    d. It fills out the *A2AHelloResponse.Body.TrustAnchorAndEntityIdPairList* field using the trust anchors and IDs for itself.
    e. It sets *A2AHelloResponse.Status* to Success.
4. The DRM/Render Agent sends the *A2AHelloResponse* to the DRM Requester.
5. The DRM Requester processes the response as follows:
    a. If *A2AHelloResponse.Status* is InvalidField, then it may either restart the A2A Hello operation at step 1 or terminate the operation.
    b. If *A2AHelloResponse.Status* is NoCommonTrustAnchor, then no communications that require a SAC is possible with the DRM/Render Agent.
    c. At this point (*A2AHelloResponse.Status* is Success), the A2A Hello operation has successfully completed.

9.1.1 A2AHelloRequest

An *A2AHelloRequest* is sent as a plain request and its body is defined as follows:

```
Body(){
   Version()
   TrustAnchorAndEntityIdPairList()
}
```

The fields are defined as follows:

- *Version* – this field contains the largest version number of the A2A interface supported by the DRM Requester. This field is defined in section 8.2. For this version of the A2A interface, *Version* SHALL be set to 0x10, indicating version "1.0".
- *TrustAnchorAndEntityIdPairList* – this field contains a list of *TrustAnchor* and *EntityID* pairs for the DRM Requester. This field is defined in section 8.6.

9.1.2 A2AHelloResponse

An *A2AHelloResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 4: A2AHelloResponse Status Values

Status Values
Success
InvalidField
NoCommonTrustAnchor

The body of an *A2AHelloResponse* is defined as follows:

```
Body(){
   Version()
   TrustAnchorAndEntityIdPairList()
}
```

The fields are defined as follows:

- *Version* – this field contains the interface version selected by the DRM Agent. This field is of type *Version*, which is defined in section 8.2. The value MUST be minimum of the *Version* from the A2AHelloRequest and the highest interface version supported by the DRM/Render Agent.
- *TrustAnchorAndEntityIdPairList* – this field contains a list of *TrustAnchor* and *EntityID* pairs for the DRM/Render Agent. This field is defined in section 8.6.

9.2 Mutual Authentication and Key Exchange Transaction

The Mutual Authentication and Key Exchange (MAKE) transaction is used to establish a SAC between the DRM Requester and the DRM/Render Agent. If a DRM Requester, Render Agent or DRM Agent has an expired CRL, it MUST NOT perform a MAKE transaction. If a DRM Requester has previously performed a MAKE transaction with the DRM/Render Agent and still has a valid SAC context, it can reuse the SAC Context and does not need to perform a new MAKE transaction (the exact details are described in section 9.2.6). The following figure illustrates the MAKE transaction.

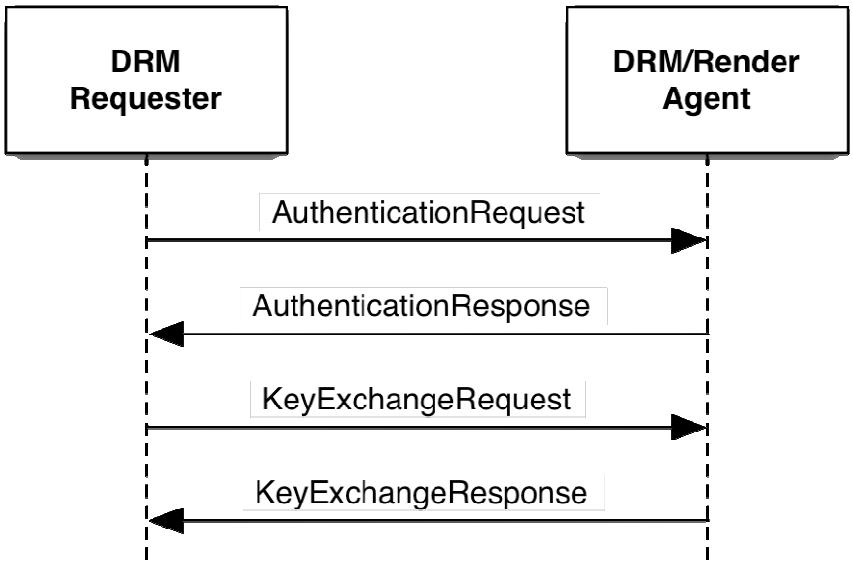

Figure 3, MAKE Transaction

The following table lists the operations that cannot be performed without a successful MAKE transaction.

Table 5: Operations Requiring MAKE

Operation
Move RO
Move REK
Share RO
Lend RO
Copy RO
Render

If the MAKE transaction is terminated for any reason, the User MAY be informed.

In order for this transaction to take place, the following MUST be performed:

1. The DRM Requester checks the validity dates of its current CRL. If the CRL has expired, it MUST NOT perform this transaction. The DRM Requester could check if the DRM/Render Agent has a current CRL by performing a CRL Query operation.
2. The DRM Requester generates an *AuthenticationRequest* using the following:
   - A trust anchor from the *A2AHelloResponse.Body.TrustAnchorAndEntityIdPairList* received from the DRM/Render Agent.
   - Its certificate chain under the selected trust anchor (the root certificate is NOT included).
   - The security algorithms it supports.
3. The DRM Requester sends the *AuthenticationRequest* to the DRM/Render Agent.
4. The DRM/Render Agent processes the request as follows:
   a. It validates the fields of the *AuthenticationRequest*. If any field is invalid, it sets *AuthenticationResponse.Status* to InvalidField and proceeds to step 5.
   b. It checks that it supports the *AuthenticationRequest.Body.TrustAnchor*. If it does not, it sets *AuthenticationResponse.Status* to TrustAnchorNotSupported and proceeds to step 5.
   c. It verifies the *AuthenticationRequest.Body.CertificateChain* (the chain MUST end at the root certificate identified by the *AuthenticationRequest.Body.TrustAnchor*). The verification includes the following:
      i. The DRM Requester's certificate MUST NOT be expired. If the certificate has expired, the DRM/Render Agent sets *AuthenticationResponse.Status* to CertificateChainVerificationFailed and proceeds to step 5.

ii. The DRM Requester's certificate has an `extKeyUsage` extension with the `oma-kp-sceDrmAgent` key purpose. If the certificate does not have the key purpose, the DRM/Render Agent sets *AuthenticationResponse.Status* to CertificateChainVerificationFailed and proceeds to step 5.

d. It checks the validity dates of its current CRL. If the CRL has expired, it sets *AuthenticationResponse.Status* to CrlExpired and proceeds to step 5.
e. It checks if the DRM Requester is listed in its current CRL. If the DRM Requester is listed, the DRM/Render Agent sets *AuthenticationResponse.Status* to DrmRequesterRevoked and proceeds to step 5.
f. It sets *AuthenticationResponse.Status* to Success.
g. It copies its certificate chain under the *AuthenticationRequest.Body.TrustAnchor* to *AuthenticationResponse.Body.CertificateChain*.
h. It sets *AuthenticationResponseData.RandomNumberS* to a freshly generated 16-byte random number.
i. It sets *AuthenticationResponseData.Version* equal to *A2AHelloRequest.Body.Version*.
j. It sets *AuthenticationResponseData.SelectedAlgorithms* to the security algorithms it wants to use from those sent in the *AuthenticationRequest.Body.SupportedAlgorithms*.
k. It sets *AuthenticationResponseData.HashOfSupportedAlgorithms* to the hash of the *AuthenticationRequest.Body.SupportedAlgorithms* field. The algorithm is the selected hash algorithm from *AuthenticationResponseData.SelectedAlgorithms*.
l. It sets *AuthenticationResponse.Body.EncryptedData* to the RSA-OAEP encryption of *AuthenticationResponseData*. The encryption key is the DRM Requester's public key from the *AuthenticationRequest.Body.CertificateChain*.

5. The DRM/Render Agent sends the *AuthenticationResponse* to the DRM Requester.
6. The DRM Requester processes the response as follows:
   a. If *AuthenticationResponse.Status* is not Success, then it determines if it can restart the MAKE transaction at step 2. Otherwise, it terminates the MAKE transaction.
   b. It verifies the *AuthenticationResponse.Body.CertificateChain* (the chain MUST end at the root certificate identified by the *AuthenticationRequest.Body.TrustAnchor*). Verification includes checking that the entity's certificate has an `extKeyUsage` extension with either the `oma-kp-sceDrmAgent` or `oma-kp-sceRenderAgent` key purpose. If the verification fails, it terminates the MAKE transaction.
   c. It checks if the DRM/Render Agent is in its current CRL. If the DRM/Render Agent is on its current CRL, the DRM Requester terminates the MAKE transaction. The User MAY be informed that the DRM/Render Agent is revoked.
   d. It decrypts *AuthenticationResponse.Body.EncryptedData* to get an *AuthenticationResponseData*. To decrypt, it uses the private key that corresponds to the certificate it sent in the *AuthenticationRequest*. If the decryption fails, it terminates the MAKE transaction.
   e. It performs the following:
      1. Checks that *AuthenticationResponseData.Version* matches what it sent in the *A2AHelloRequest.Body.Version*. If not equal, it terminates the MAKE transaction.
      2. Checks that *AuthenticationResponseData.HashOfSupportedAlgorithms* matches the hash of *AuthenticationRequest.Body.SupportedAlgorithms*. If the hashes do not match, it terminates the MAKE transaction.
      3. Checks that *AuthenticationResponseData.SelectedAlgorithms* correspond to the algorithms in *AuthenticationRequest.Body.SupportedAlgorithms*. If they do not, it terminates the MAKE transaction.
   f. It sets *KeyExchangeData.RandomNumberR* to a freshly generated 16-byte random number.
   g. It sets *KeyExchangeData.HashOfRandomNumberS* to the hash of *AuthenticationResponseData.RandomNumberS*, using the hash specified in *AuthenticationResponseData.SelectedAlgorithms*.
   h. It sets *KeyExchangeData.SelectedVersion* to a copy of *AuthenticationResponseData.Version*.
   i. It encrypts *KeyExchangeData* with the public key of the DRM/Render Agent taken from the *AuthenticationResponse.CertificateChain*. The encrypted *KeyExchangeData* is put into the *KeyExchangeRequest.Body*.
7. The DRM Requester sends the *KeyExchangeRequest* to the DRM/Render Agent.
8. The DRM/Render Agent processes the request as follows:
   a. It validates the fields of the *KeyExchangeRequest*. If any field is invalid, it sets *AuthenticationResponse.Status* to InvalidField and proceeds to step 9.

b. It decrypts *KeyExchangeRequest.Body* with its private key to get *KeyExchangeData*. If there is an error with the decryption, it sets *KeyExchangeResponse.Status* to FieldDecryptionFailed and continues with step 9.
c. It checks that *KeyExchangeData.HashOfRandomNumberS* matches the hash, using the hash algorithm it sent in the *AuthenticationResponse*, of the *RandomNumberS* it sent in the *AuthenticationResponse*. If it does not match, it sets *KeyExchangeResponse.Status* to RandomNumberMismatched and continues with step 9.
d. It checks that *KeyExchangeData.SelectedVersion* matches what it sent in the *AuthenticationResponse*. If it does not match, it sets *KeyExchangeResponse.Status* to VersionMismatched and continues with step 9.
e. It sets *KeyExchangeResponse.Status* to Success.
f. It sets *KeyExchangeResponse.Body.Hash* to the hash, using the selected algorithm, of the concatenation of *KeyExchangeData.RandomNumberR* and *RandomNumberS* that it sent in the *AuthenticationResponse*.

9. The DRM/Render Agent sends the *KeyExchangeResponse* to the DRM Requester. After sending the *KeyExchangeResponse*, the DRM/Render Agent generates the keys for the SAC (see section 9.2.5) and sets up its SAC context (see section 9.2.6).
10. The DRM Requester processes the response as follows:
    a. If *KeyExchangeResponse.Status* is not Success, then, based on the error, it can either retry the MAKE transaction (at either step 2 or step 6.e) or it can terminate the MAKE transation.
    b. It verifies that *KeyExchangeResponse.Body.Hash* matches the hash it calculates over the concatenation of *RandomNumberR* and *RandomeNumberS*. If the hashes do not match, it terminates the MAKE transaction.
    c. It generates the keys for the SAC (see section 9.2.5) and sets up its SAC context (see section 9.2.6).
    d. At this point, the MAKE transaction has successfully completed and a SAC is established.

9.2.1 AuthenticationRequest

An *AuthenticationRequest* is sent as a plain request and its body is defined as follows:

```
Body(){
   TrustAnchor()
   CertificateChain()
   SupportedAlgorithms()
}

SupportedAlgorithms(){
   // Hash algorithms
   AlgorithmList()
   // HMAC algorithms
   AlgorithmList()
   // Symmetric algorithms
   AlgorithmList()
   // Asymmetric algorithms
   AlgorithmList()
   // KDF algorithms
   AlgorithmList()
}
```

The fields are defined as follows:

- *TrustAnchor* – this field identifies the trust model under which the DRM Requester wants to establish the SAC. This field is defined in section 8.4.
- *CertificateChain* – this field contains the DRM Requester's certificate chain under the trust model in the previous field. This field is defined in section 8.8.
- *SupportedAlgorithms* – this field contains the security algorithms (hash algorithms, HMAC algorithms, symmetric encryption algorithms, asymmetric encryption algorithms and key derivation functions) that are supported by the DRM Requester.
- *AlgorithmList* – this field is a list of algorithms and is defined in section 8.10.

9.2.2 AuthenticationResponse

An *AuthenticationResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 6: AuthenticationResponse Status Values

Status Values
Success
InvalidField
TrustAnchorNotSupported
CertificateChainVerificationFailed
CrlExpired
DrmRequesterRevoked

The body of an *AuthenticationResponse* is defined as follows:

```
Body(){
   CertificateChain()
   EncryptedData()   //Contains an encrypted AuthenticationResponseData
}

AuthenticationResponseData(){
   RandomNumberS()
   Version()
   SelectedAlgorithms()
   HashOfSupportedAlgorithms()
}

RandomNumberS(){
   RandomNumber()
}

SelectedAlgorithms(){
   // Hash algorithm
   Algorithm()
   // HMAC algorithms
   Algorithm()
   // Symmetric algorithms
   Algorithm()
   // Asymmetric algorithms
   Algorithm()
   // KDF algorithms
   Algorithm()
}

HashOfSupportedAlgorithms() {
   Hash()
}
```

The fields are defined as follows:

- *CertificateChain* – this field contains the DRM/Render Agent's certificate chain under the trust model identified by the *AuthenticationRequest.Body.TrustAnchor*. This field is defined in section 8.8.
- *EncryptedData* – this field contains an *AuthenticationResponseData* data structure that is encrypted by the DRM Requester's public key (from the DRM Requester's certificate). This field is defined in section 8.11.

- *RandomNumberS* – this field contains a 16-byte random number generated by the DRM/Render Agent. This field is of type *RandomNumber* which is defined in section 8.12.
- *Version* – this field contains a copy of the *Version* field of the *A2AHelloRequest*.
- *SelectedAlgorithms* – this field contains the security algorithms selected by the DRM/Render Agent from the *SupportedAlgorithms* sent in the *AuthenticationRequest*.
- *Algorithm* – this field contains one security algorithm and is defined in section 8.10.
- *HashOfSupportedAlgorithms* – this field contains the hash, using the selected hash algorithm, of the *SupportedAlgorithms* field in the *AuthenticationRequest*. This field is of type *Hash*, which is defined in section 8.3.

9.2.3 KeyExchangeRequest

A *KeyExchangeRequest* is sent as a plain request and its body is defined as follows:

```
Body(){
   EncryptedData()   //Contains an encrypted KeyExchangeData
}

KeyExchangeData(){
   RandomNumberR()
   HashOfRandomNumberS()
   SelectedVersion()
}

RandomNumberR(){
   RandomNumber()
}

HashOfRandomNumberS(){
   Hash()
}

SelectedVersion(){
   Version()
}
```

The fields are defined as follows:

- *EncryptedData* – this field contains a *KeyExchangeData* data structure that is encrypted by the DRM/Render Agent's public key (from the DRM Agent's certificate). This field is of type *EncryptedData*, which is defined in section 8.11.
- *RandomNumberR* – this field contains a 16 byte random number generated by the DRM Requester. This field is of type *RandomNumber*, which is defined in section 8.12.
- *HashOfRandomNumberS* – this field contains the hash, using the selected hash algorithm, of the *RandomNumberS* field in the *AuthenticationResponse*. This field is of type *Hash*, which is defined in section 8.3.
- *SelectedVersion* – this field is a copy of the *SelectedVersion* sent in the *A2AHelloResponse*. This field is of type *Version*, which is defined in section 8.2.

9.2.4 KeyExchangeResponse

A *KeyExchangeResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 7: KeyExchangeResponse Status Values

Status Values
Success
InvalidField
FieldDecryptionFailed

Status Values
RandomNumberMismatched
VersionMismatched
Unexpected Request

The body of a *KeyExchangeResponse* is defined as follows:

```
Body(){
   Hash()
}
```

The fields are defined as follows:

- *Hash* – this field contains the hash, using the selected hash algorithm, of the concatenation of the random numbers *RandomNumberR* and *RandomNumberS* exchanged in this transaction. This field is defined in section 8.3.

9.2.5 SAC Key Material

As part of steps 9 and 10 of the MAKE transaction, both the DRM Requester and the DRM/Render Agent have mutually authenticated each other and have exchanged secret random numbers. By using a Key Derivation Function (KDF), key material that is required for the SAC (i.e. MAC Key, Session Key and CtrCounter) is derived from the secret random numbers.

The default KDF is the KDF specified in section 7.1.2 of [DRM-v2.1]. When using this KDF, set **Z** = *RandomNumberR* | *RandomNumberS*, set **otherInfo** = *SupportedAlgorithms* | *SelectedAlgorithms* and set **kLen** = 48 bytes (the total size of the key material in Table 8).

The SAC provides message integrity by using an HMAC algorithm. The default HMAC algorithm is HMAC-SHA1 with a 20-byte (160 bits) key. This key is referred to as the MAC Key (MK) and is equal to the 20 most significant bytes of the KDF output **T** (i.e. byte 0 to byte 19).

When encrypting portions of a message under the SAC, the negotiated symmetric encryption algorithm is used. The default symmetric encryption algorithm is AES-128-CTR. The encryption key is referred to as the Session Key (SK) and is equal to the next 16 bytes of **T** (i.e. byte 20 to byte 35). The initial value of the counter for the AES-128-CTR is equal to the next 12 bytes of **T** (i.e. byte 36 to byte 47).

The following table summarizes the key material derived from the exchanged secret random numbers for the default algorithms.

Table 8: Default SAC Key Material

Name	Description	Size	Abbreviation
MAC Key	The HMAC-SHA1 key used to provide message integrity (20 most significant bytes of **T**).	160 bits	MK
Session Key	The key used to encrypt portions of a message using AES in counter mode (next 16 bytes of **T**).	128 bits	SK
CtrCounter	The high order bits of the counter used in AES counter mode (next 12 bytes of **T**).	96 bits	CtrA

If different algorithms are defined in the future, the key material table has to be defined for the new algorithms.

9.2.6 SAC Context

Once a SAC has been established, a logical SAC context will exist. At a minimum, the context consists of the following information:

- Trust Anchor – this contains the trust anchor under which the SAC was established. Used when multiple SACs are available and the DRM Requester wants to switch to a different SAC as specified in section 9.3.
- Entity ID – for the DRM Requester, this contains the DRM/Render Agent's ID (under the trust anchor); for the DRM/Render Agent, this contains the DRM Requester's ID (under the trust anchor).

- Selected Algorithms – this contains the algorithms that were negotiated during the MAKE transaction.
- MAC Key (MK) – this contains the derived key for the negotiated HMAC algorithm.
- Session Key (SK) – this contains the derived key for the negotiated symmetric encryption algorithm.
- CtrCounter – this contains the current message counter when a symmetric algorithm in counter mode has been negotiated.
- currentReplayCounterR – this contains the current replay counter when acting as a DRM Requester. Its use is described in section 7.3. This counter is set to 0 when the SAC context is established.
- currentReplayCounterA – This contains the current replay counter when acting as a DRM/Render Agent. Its use is described in section 7.3. This counter is set to 0 when the SAC context is established.

The SAC context exists until a new SAC with the same DRM Requester and DRM/Render Agent, and under the same trust model, is established. By using the A2A Hello operation, a DRM Requester can determine if it is communicating with the same DRM/Render Agent. If it is communicating with the same DRM/Render Agent, the DRM Requester can reuse the SAC context. If the DRM Requester reuses the SAC context, sends a protected request and gets back an *IntegrityVerificationFailed* error, this probably indicates that the SAC context is no longer valid. In this case, the DRM Requester SHOULD establish a new SAC.

9.2.7 Data Encryption

Any portion of a protected message that needs confidentiality must be encrypted using the symmetric key algorithm that was negotiated during the MAKE transaction. The key used to encrypt is the key derived using the KDF per section 9.2.5.

The default encryption algorithm is AES in counter mode. The initial value of the AES counter is shown in the following table.

Table 9: Intial AES Counter Value

Counter Portion	Bits	Description
CtrCounter	80	The msb's of the counter. Taken from the KDF.
CtrR	32	A copy of the *replayCounter* of the message being sent.
CtrB	16	The lsb's of the counter. Initially set to 0 and then incremented for each block.

Because the least significant bits of the counter are used for the blocks, the maximum field size that can be encrypted is 1048576 bytes, although *EncryptedData* only allows for a maximum field size of 65535 (see section 8.11).

9.3 Change SAC Operation

The Change SAC operation is used by the DRM Requester to change to a different SAC with the DRM/Render Agent. The following figure illustrates the Change SAC operation.

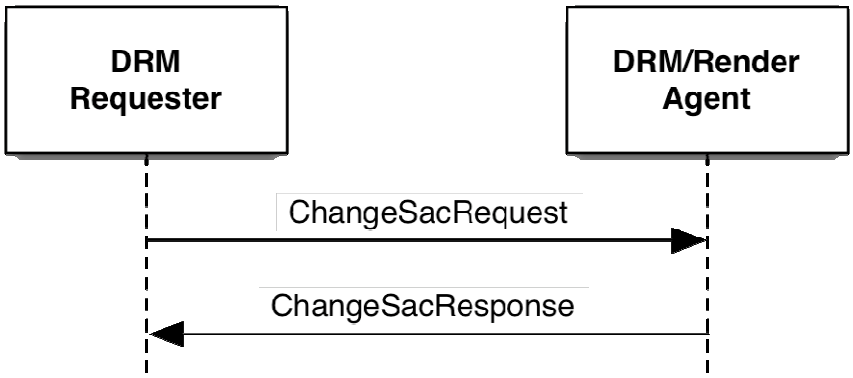

Figure 4: Change SAC Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester generates a *ChangeSacRequest* using the trust anchor that was used to establish the SAC.
2. The DRM Requester sends the *ChangeSacRequest* to the DRM/Render Agent.
3. The DRM/Render Agent processes the request as follows:
    a. It validates the fields of the *ChangeSacRequest*. If any field is invalid, it sets *ChangeSacResponse.Status* to InvalidField and proceeds to step 4.
    b. It checks if it has a SAC context that corresponds to *ChangeSacRequest.Body.TrustAnchor*. If it does not, it sets *ChangeSacResponse.Status* to SACNotEstablished and proceeds to step 4.
    c. It sets *ChangeSacResponse.Status* to Success and changes to the SAC context.
4. The DRM/Render Agent sends the *ChangeSacResponse* to the DRM Requester.
5. The DRM Requester processes the response as follows:
    a. If *ChangeSacResponse.Status* is not Success, it determines if it can restart the Change SAC operation at step 1. If it does not restart, it terminates the Change SAC operation.
    b. It changes to the SAC context, identified by the trust anchor sent in the *ChangeSacRequest*.
    c. At this point, the Change SAC operation has successfully completed.

9.3.1 ChangeSacRequest

A *ChangeSacRequest* is sent as a plain request and its body is defined as follows:

```
Body(){
   TrustAnchor()
}
```

The fields are defined as follows:

- *TrustAnchor* – this field identifies the trust model of the SAC to change to. This field is defined in section 8.4.

9.3.2 ChangeSacResponse

A *ChangeSacResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 10: ChangeSacResponse Status Values

Status Values
Success
InvalidField
SACNotEstablished

The body of a *ChangeSacResponse* is empty and is defined as follows:

```
Body(){
}
```

9.4 CRL Query Operation

The CRL Query operation is used by the DRM Requester to query what CRLs the DRM/Render Agent has. The following figure illustrates the CRL Query operation.

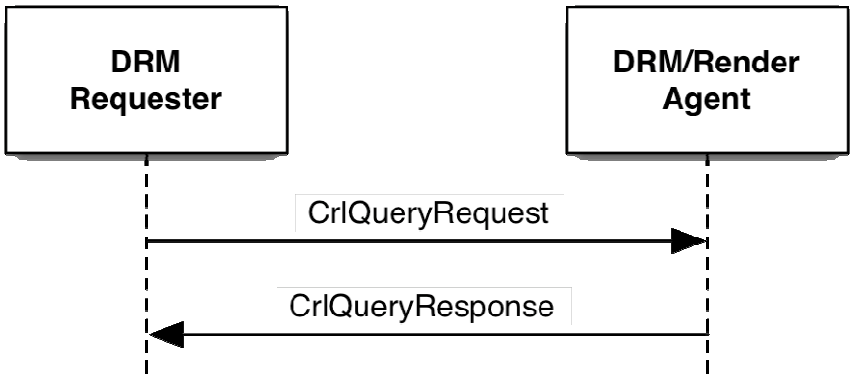

Figure 5: CRL Query Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester sends the *CrlQueryRequest* to the DRM/Render Agent.
2. The DRM/Render Agent processes the request as follows:
    a. It sets *CrlQueryResponse.Body.CrlIdList* with the list of CRLs it has.
    b. It sets *CrlQueryResponse.Status* to Success.
3. The DRM/Render Agent sends the *CrlQueryResponse* to the DRM Requester.
4. The DRM Requester processes the response as follows:
    a. It checks the CRLs it has against the CRLs the DRM/Render Agent has. If the DRM Requester has CRLs that are more recent than the CRLs the DRM/Render Agent has, then it SHOULD send the most recent CRLs to the DRM/Render Agent via the Put CRL operation. If the DRM/Render Agent has more recent CRLs, the DRM Requester SHOULD request those CRLs via the Get CRL Operation (see section 9.6).
    b. At this point, the CRL Query operation has successfully completed.

9.4.1 CrlQueryRequest

A *CrlQueryRequest* is sent as a plain request. It has an empty body that is defined as follows:

```
Body(){
}
```

9.4.2 CrlQueryResponse

A *CrlQueryResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 11: CrlQueryResponse Status Values

Status Values
Success

The body of a *CrlQueryResponse* is defined as follows:

```
Body(){
   CrlIdList()
```

```
}
```

The fields are defined as follows:

- *CrlIdList* – this field contains a list CRLs that the DRM Requester is requesting from the DRM/Render Agent. It is defined in section 8.17.

9.5 Put CRL Operation

The Put CRL operation is used by the DRM Requester to send one or more CRLs to the DRM/Render Agent. The following figure illustrates the Put CRL operation.

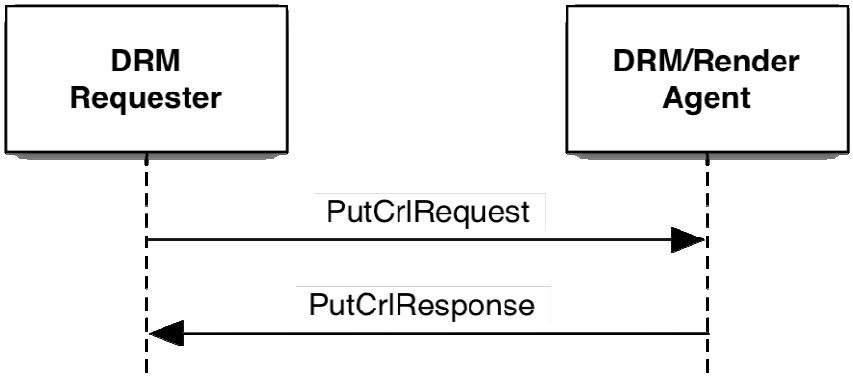

Figure 6: Put CRL Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester sets *PutCrlRequest.Body.CrlList* with the CRLs it wants to send to the DRM/Render Agent. Usually, the CRLs are chosen as part of performing a CRL Query Operation (see section 9.4).
2. The DRM Requester sends the *PutCrlRequest* to the DRM/Render Agent.
3. The DRM/Render Agent processes the request as follows:
    a. It validates the fields of the *PutCrlRequest*. If any field is invalid, it sets *PutCrlResponse.Status* to InvalidField and proceeds to step 4.
    b. It sets *PutCrlResponse.Status* to Success.
    c. For each CRL received, it does the following:
        1. It verifies the CRL. If the verification fails, it sets *PutCrlResponse.Status* to CrlVerificationFailed and does not save the CRL. If there's another CRL, it continues at step 3.c.1 with the next CRL.
        2. It checks if the CRL is a more recent CRL for the same CRL issuer. If it is more recent, it overwrites the older CRL.
4. The DRM/Render Agent sends the *PutCrlResponse* to the DRM Requester.
5. The DRM Requester processes the response as follows:
    a. If *PutCrlResponse.Status* is not Success, it determines if it can restart the Put CRL operation at step 1. If it does not restart, it terminates the Put CRL operation.
    b. At this point, the Put CRL operation has successfully completed.

9.5.1 PutCrlRequest

A *PutCrlRequest* is sent as a plain request and its body is defined as follows:

```
Body(){
   CrlList()
}
```

The fields are defined as follows:

- *CrlList* – this field contains a list of CRLs being sent to the DRM/Render Agent. It is defined in section 8.17.

9.5.2 PutCrlResponse

A *PutCrlResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 12: PutCrlResponse Status Values

Status Values
Success
InvalidField
CrlVerificationFailed

The body of a *PutCrlResponse* is empty and is defined as follows:

```
Body(){
}
```

9.6 Get CRL Operation

The Get CRL operation is used by the DRM Requester to get one or more CRLs from the DRM/Render Agent. The following figure illustrates the Get CRL operation.

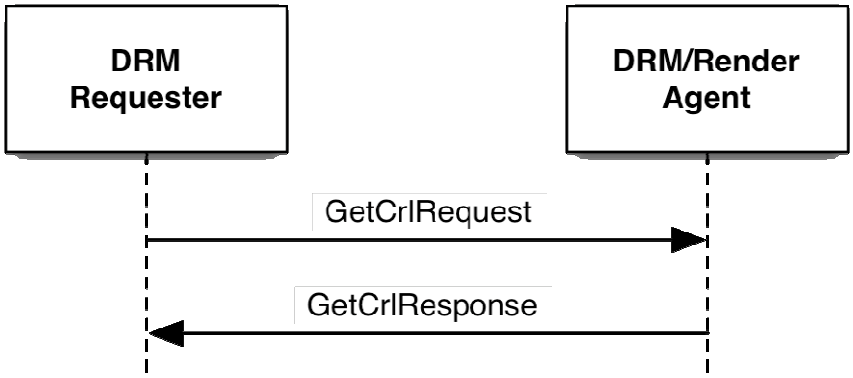

Figure 7: Get CRL Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester sets *GetCrlRequest.Body.CrlIdList* with the CRL IDs it wants from the DRM/Render Agent. Usually, the CRLs are chosen as part of performing a CRL Query Operation (see section 9.4).
2. The DRM Requester sends the *GetCrlRequest* to the DRM/Render Agent.
3. The DRM/Render Agent processes the request as follows:
    a. It validates the fields of the *GetCrlRequest*. If any field is invalid, it sets *GetCrlResponse.Status* to InvalidField and proceeds to step 4.
    b. It checks if it has all the requested CRLs. If it does not, it sets *GetCrlResponse.Status* to CrlNotFound and proceeds to step 4.
    c. It sets *GetCrlResponse.Status* to Success.
    d. It sets *GetCrlResponse.Body.CrlList* with the requested CRLs.
4. The DRM/Render Agent sends the *GetCrlResponse* to the DRM Requester.
5. The DRM Requester processes the response as follows:

a. If *GetCrlResponse.Status* is not Success, it determines if it can restart the Get CRL operation at step 1. If it does not restart, it terminates the Get CRL operation.
b. For each CRL in *GetCrlResponse.Body.CrlList*, it does the following:
    1. It verifies the CRL. If the CRL does not verify and there is another CRL, continue with step 5.b.1 with the next CRL.
    2. It checks if the CRL is more recent than the CRL it has. If it is more recent, overwrite the older CRL with the more recent CRL.
c. At this point the Get CRL operation has successfully completed.

9.6.1 GetCrlRequest

A *GetCrlRequest* is sent as a plain request and its body is defined as follows:

```
Body(){
   CrlIdList()
}
```

The fields are defined as follows:

- *CrlIdList* – this field contains a list of CRLs that the DRM Requester is requesting from the DRM/Render Agent. It is defined in section 8.17.

9.6.2 GetCrlResponse

A *GetCrlResponse* is sent as a plain response. The following table lists the valid *Status* values for this response.

Table 13: GetCrlResponse Status Values

Status Values
Success
InvalidField
CrlNotFound

The body of a *GetCrlReponse* is defined as follows:

```
Body(){
   CrlList()
}
```

The fields are defined as follows:

- *CrlList* – This field contains the requested CRLs. It is defined in section 8.17.

9.7 Move RO Transaction

The Move RO transaction is used by the DRM Requester to Move a Rights Object (RO) with a <move> permission to a DRM Agent. This transaction MUST take place using a SAC. This transaction MUST NOT be performed if the DRM Requester's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set or the DRM Agent's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set (see section A.1). The DRM Agent MUST reject the RO if the <signature> element over the <rights> element has been generated by an entity other than an RI or an LRM. The following figure illustrates the Move RO transaction.

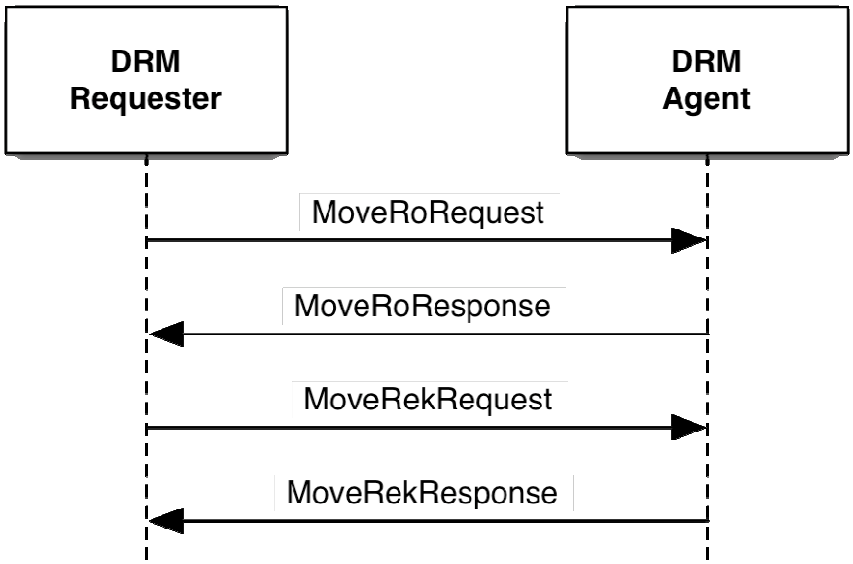

Figure 8: Move RO Transaction

In order for this transaction to take place, the following MUST be performed:

1. The DRM Requester performs the following:
    a. It checks if the RO has the <move> permission. The "allowPartial" attribute MUST be "true" if a Partial Move is to be performed. If the <move> permission is not present, the Move RO transaction is terminated. Otherwise, the following is performed:
        i. If there is a <system> constraint, then it checks the <context> child element(s) of the <system> constraint. If no <context> child element identifies the (A2A) Move RO transaction, then the Move RO transaction is terminated.
        ii. If there is a <count> constraint, then it checks the current count value in the state information of the RO. If the current move count is 0, then the Move RO transaction is terminated. Otherwise, the DRM Requester decrements the current move count value in the state information of the RO.
    b. It checks the entity type that created the RO. If the RO was created by an RI, the DRM Requester proceeds to step 1.c. Otherwise, the following is performed:
        i. If the LRM's certificate does not have the *localRightsManagerDevice* extended key purpose (see [SCE-LRM]), then the RO MUST have a <userDomain> constraint. If the constraint is not present, the Move RO transaction is terminated.
        ii. If the LRM's certificate has the *localRightsManagerDevice* extended key purpose, the RO MUST be a Device RO. If it is not a Device RO, the Move RO transaction is terminated.
    c. It checks if the RO has a <userDomain> constraint. If the constraint is present, the DRM Requester checks its own User Domain Authorization (see [SCE-DOM]). If the User Domain Authorization is expired, the Move RO transaction is terminated.
    d. It marks the RO being Moved as unusable. If the RO is stateful and just a portion of the RO is being Moved (Partial Rights, see section 5.3), then that portion being Moved is marked as unusable.
    e. It generates a random *moveHandle* and creates a Move context with the *moveHandle*, the REK of the RO being Moved, and the DRM Agent ID.
2. The DRM Requester generates a *MoveRoRequest* with the information for the RO (or portion) being Moved to the DRM Agent and *moveHandle* (from step 1.e). If the RO has a <copy> permission, but the <copy> permission was lost (see also section 9.8, point 4.q), the DRM Requester MUST signalise this to the DRM Agent by including a ConstraintState() field for the <copy> permission in the MoveRoRequest, with the permissionLost field set to true.
3. The DRM Requester sends the *MoveRoRequest* to the DRM Agent, applying the replay protection mechanism described in section 7.3.
4. The DRM Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.

b. It verifies the integrity of the request. If the integrity check fails, it sets *MoveRoResponse.Status* to IntegrityVerificationFailed and proceeds to step 5.
c. It validates the fields of the *MoveRoRequest*. If any field is invalid, it sets *MoveRoResponse.Status* to InvalidField and proceeds to step 5.
d. It verifies the signature on the RO, including the *SourceCertificateChain* field. If any of the verifications fails, it sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
e. It checks that the RO has the <move> permission. If it does not, it sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
f. If the RO is stateful, it validates that the *StateInformation* is consistent with the original state in the RO (see section 5.5). If any state is invalid, it sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
g. It checks the entity that created the RO. If the RO was created by an RI, the DRM Requester proceeds to step 4.i.
h. If the LRM's certificate does not have the *localRightsManagerDevice* extended key purpose (see [SCE-LRM]), then the RO MUST have a <userDomain> constraint. If the constraint is not present, the DRM Agent sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
i. It checks whether the RO has a <userDomain> constraint. If not, the DRM Agent proceeds to step 4.j. Otherwise, the DRM Agent performs the following checks:
    i. It checks whether the RO has a <copy> permission. If not, the DRM Agent proceeds to step 4.i.iii.
    ii. It checks if it has a current record (whether installed or waiting to be installed) of an RO with the same ROID. If the duplicate RO exists, the DRM Agent sets *MoveRoResponse.Status* to DuplicateRightsObject and proceeds to step 5.
    iii. It checks if an LRM created the RO. If an LRM created the RO, the DRM Agent checks if the LRM's certificate has the *localRightsManagerDomain* extended key purpose. If the certificate does not, the DRM Agent sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    iv. It validates the *UserDomainAuthorization* for the DRM Requester. If the validation fails, the DRM Agent sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5. Validation MUST include the following:
        a. Verifying the signature
        b. User Domain Authorization is not expired
        c. Entity ID of User Domain Authorization matches ID of DRM Requester
    v. It checks that the User Domain Authorization of the <party> element of the RO corresponds to the RI/LRM that signed the <rights> element, and verifies the DEA's signature on the User Domain Authorization. If the correspondence check or DEA signature verification fails, the DRM Agent sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    vi. It checks that the User Domain baseID of the <userDomainID> element within the User Domain Authorization in the <party> element of the RO is the same as the User Domain baseID of the <userDomainID> element within the *UserDomainAuthorization* field. If not, the DRM Agent sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    vii. It checks that the User Domain generation of the *UserDomainAuthorization* field is greater than or equal to the User Domain generation of the User Domain Authorization in the <party> element of the RO. If not, the DRM Agent sets *MoveRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    viii. If the DRM Agent is already a member of the User Domain, it checks that the User Domain generation of the *UserDomainAuthorization* field is greater than or equal to the User Domain generation of the DRM Agent's User Domain Authorization. If not, the DRM Agent sets *MoveRoResponse.Status* to LowUserDomainGeneration and proceeds to step 5.
j. It checks if it has enough room to install the RO. If it does not, it sets *MoveRoResponse.Status* to NotEnoughSpace and proceeds to step 5.
k. It saves *moveHandle* and associates *moveHandle* with the RO (which cannot be installed yet).
l. It sets *MoveRoResponse.Status* to Success.

5. The DRM Agent sends the *MoveRoResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
6. The DRM Requester processes the response as follows:
    a. It processes the response for replay as described in section 7.3.

b. If the integrity verification of the response fails or *MoveRoResponse.Status* is not Success, it determines if it can restart the Move RO transaction at step 2. If it does not restart the transaction, the DRM Requester performs the following:
    i. It marks the RO (or portion) as usable.
    ii. If the <move> permission had a <count> constraint, it increments the current move counter of the state information.
    iii. It terminates the Move RO transaction.
c. It deletes the RO (or portion) that was Moved (but still keeps the corresponding Move context). Note: if the RO being Moved has been backed up, the Backed Up RO MUST NOT be restored.

7. The DRM Requester generates a *MoveRekRequest* with the data from the Move context.
8. The DRM Requester sends the *MoveRekRequest* to the DRM Agent, applying the replay protection mechanism described in section 7.3.
9. The DRM Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.
    b. It validates the fields of the *MoveRekRequest*. If any field is invalid, it sets *MoveRoResponse.Status* to InvalidField and proceeds to step 10.
    c. It verifies the integrity of the request. If the integrity check fails, it sets *MoveRekResponse.Status* to IntegrityVerificationFailed and proceeds to step 10.
    d. It checks if it has an RO that corresponds to the *moveHandle*. If it does not have a corresponding RO, it sets *MoveRekResponse.Status* to UnknownHandle and continues with step 10.
    It decrypts *MoveRekRequest.Body.EncryptedRek*. Note: if the RO is a User Domain RO with a <userDomain> constraint, and the DRM Agent is not yet a member of the User Domain (i.e. it does not have the UDK), the DRM Agent MUST join the User Domain to receive a copy of the UDK in order to fully decrypt the REK.
    e. If the User Domain Context ([SCE-DOM]) has expired (as indicated by the User Domain Context Expiry Time) the DRM Agent MUST NOT install the RO.
    f. It checks whether the RO has a <contextRequired> constraint element. If not, it proceeds to step 9.h.
    g. It tags the RO that corresponds to the *moveHandle* as 'pending RI/LRM Context verification' removes the *moveHandle* from the RO, and proceeds to step 9.i.
    h. It marks the RO that corresponds to the *moveHandle* as usable, and removes the *moveHandle* from the RO.
    i. It sets *MoveRekResponse.Status* to Success.
10. The DRM Agent sends the *MoveRekResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
11. If the RO has been tagged as 'pending RI/LRM Context verification', upon successful verification of an active/current Context with the RI or LRM that generated the <signature> element of the RO, the DRM Agent removes the tag and marks the RO as usable. If the RO has a 'pending RI/LRM Context verification' tag, the DRM Agent MUST NOT grant any permissions other than <move>.
12. The DRM Requester processes the response as follows:
    a. It processes the response for replay as described in section 7.3.
    b. It verifies the integrity of the response. If the integrity check failed, the DRM Requester determines if it can restart the Move RO transaction at step 7. If it does not restart the transaction, the DRM Requester MUST leave the RO marked as unusable and terminate the Move RO transaction.
    c. If *MoveRekResponse.Status* is not Success, it determines if it can restart the Move RO transaction at step 7. If it does not restart the transaction, the DRM Requester performs the following:
        i. It marks the RO (or portion) as usable.
        ii. If the <move> permission had a <count> constraint, it increments the current move counter of the state information.
        iii. It terminates the Move RO transaction.
    d. It removes the cached corresponding Move context.
    e. At this point the Move RO transaction has successfully completed.

9.7.1 MoveRoRequest

A *MoveRoRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   timeStampPresent     1     bslbf
   stateInfoPresent     1     bslbf
   udaPresent           1     bslbf
   rfu                  5     bslbf
   moveHandle          64     uimsbf
   RoAlias()
   SourceAlias()
   SourceID()
   if( timeStampPresent ){
      SourceTimeStamp()
   }
   RightsObjectContainer()
   if( stateInfoPresent ){
      StateInformation()
   }
   CertificateChain()
   if( udaPresent ){
      UserDomainAuthorization()
   }
}

RoAlias(){
   String80()
}

DomainAlias(){
   String80()
}

SourceAlias(){
   String80()
}

SourceID(){
   EntityID()
}

TimeStamp(){
   year                14     uimsbf
   month                4     uimsbf
   day                  5     uimsbf
   hour                 5     uimsbf
   minute               6     uimsbf
   second               6     uimsbf
}

UserDomainAuthorization(){
   OctetString16()
}
```

The fields are defined as follows:

- *timeStampPresent* – this is a boolean field, that if true, indicates that the source (RI or LRM) *TimeStamp* field is present.
- *stateInfoPresent* – this is a boolean field, that if true, indicates that the *StateInformation* field is present.
- *rfu* – this is a 5 bit field that is reserved for future use. When sending the request, this field MUST be set to 0. When processing this field, its value MUST be ignored.
- *moveHandle* – this field contains a random 64 bit unsigned integer that is used to correlate the *MoveRoRequest* with the *MoveRekRequest*.
- *RoAlias* – this field contains an optional alias for the RO. It is of type *String80* which is defined in section 8.16.
- *DomainAlias* – this field contains an optional alias for the domain if the RO is a domain RO. It is of type *String80* which is defined in section 8.16.
- *SourceAlias* – this field contains an optional alias for the Rights Issuer or LRM that created the RO. It is of type *String80* which is defined in section 8.16.
- *SourceID* – this field contains the identity of the Rights Issuer or LRM that created the RO. It is of type *EntityID* which is defined in section 8.5.
- *RightsObjectContainer* – this field contains a RO as defined in section 8.19.
- *StateInformation* – this field, if present, contains the state information for the Rights being Moved. This field is defined in section 8.21. This field MUST be present if the RO is stateful.
- *year* – this field contains the year – 2000 of the timestamp. Range is 0 – 16383, corresponding to the years 2000 – 18,383.
- *month* – this field contains the month of the timestamp, with 0 representing January. Range is 0 – 11.
- *day* – this field contains the day – 1 of the month of the timestamp. Range is 0 – 30.
- *hour* – this field contains the hour of the timestamp. Range is 0 – 23.
- *minute* – this field contains the minute of the timestamp. Range is 0 – 59.
- *second* – this field contains the seconds of the timestamp. Range is 0 – 59.
- *CertificateChain* – this field contains the certificate chain for the Rights Issuer or LRM that created the RO. This field is defined in section 8.8.
- *UserDomainAuthorization* – this field, if present, contains the User Domain Authorization for the DRM Requester. T*his field MUST be presen*t if the RO being Moved has a <userDomain> constraint.

9.7.2 MoveRoResponse

A *MoveRoResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 14: MoveRoResponse Status Values

Status Values
Success
InvalidField
InvalidRightsObject
NotEnoughSpace
NotADomainMember

The body of a *MoveRoResponse* is empty and is defined as follows:

```
Body(){
}
```

9.7.3 MoveRekRequest

A *MoveRekRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   moveHandle          64      uimsbf
   EncryptedRek()
}
```

```
EncryptedRek (){
   EncryptedData()  //Contains an encrypted REK
}

Rek(){
   for( i = 0; i < 16; i++ ){
      byte              8     uimsbf
   }
}
```

The fields are defined as follows:

- *moveHandle* – this field contains a random 64 bit unsigned integer that is used to correlate the *MoveRoRequest* with the *MoveRekRequest*.
- *EncryptedRek* – this field contains an encrypted REK. If the RO has a <userDomain> constraint, the REK is first encrypted with the (current generation of the) UDK (for the User Domain) using [AES-WRAP] and then the wrapped REK is encrypted with the SK using the negotiated algorithm. If the RO does not have a <userDomain> constraint, the REK is encrypted by the SK using the negotiated algorithm. The field is of type *EncryptedData* which is defined in section 8.11.
- *Rek* – this field contains an REK.

9.7.4 MoveRekResponse

A *MoveRekResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 15: MoveRekResponse Status Values

Status Values
Success
InvalidField
UnknownHandle
IntegrityVerificationFailed

The *MoveRekResponse* is empty and is defined as follows:

```
Body(){
}
```

9.8 Copy RO Operation

The Copy RO operation is only used by a DRM Requester to Copy a <userDomain>-constrained Rights Object (RO) with a <copy> permission to a DRM Agent. This operation MUST take place using a SAC. This operation MUST NOT be performed if the DRM Requester's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set or the DRM Agent's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set (see section A.1). The DRM Agent MUST reject the RO if the <signature> element over the <rights> element has been generated by an entity other than an RI or an LRM. The following figure illustrates the Copy RO operation.

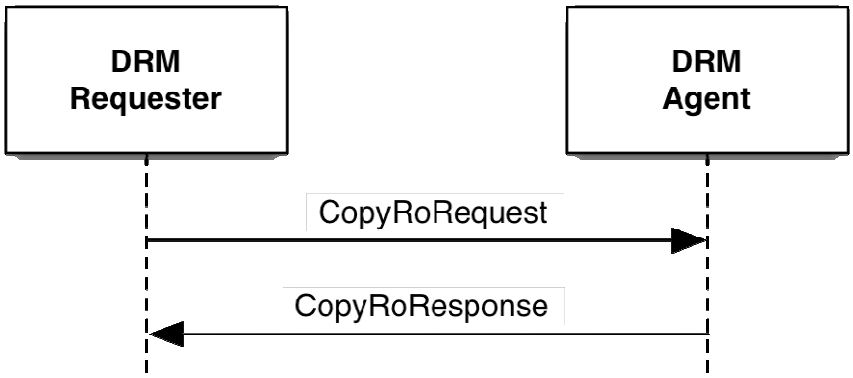

Figure 9: Copy RO Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester performs the following:
    a. It checks if the RO has the <copy> permission, and that the <copy> permission was not lost. If the <copy> permission is not present or the <copy> permission was lost, the Copy RO operation is terminated. Otherwise, the following is performed:
        i. If there is a <system> constraint, the DRM Requester checks the <context> child element(s) of the <system> constraint. If no <context> child element identifies the (A2A) Copy RO operation, the Copy RO operation is terminated.
        ii. If there is a <count> constraint, then it checks the current count value in the state information of the RO. If the current copy count is 0, the DRM Requester terminates the Copy RO operation. Otherwise, it decrements the current copy count value in the state information of the RO.
    b. It checks the entity type that created the RO. If an RI created the RO, the DRM Requester proceeds to step 1.d.
    c. If the LRM's certificate does not have the *localRightsManagerDomain* extended key purpose (see [SCE-LRM]), the Copy RO operation is terminated.
    d. It checks its User Domain Authorization ([SCE-DOM]). If the User Domain Authorization is expired, the Copy RO operation is terminated.
    e. It checks if the RO contains a <userDomain> constraint. If there is no <userDomain> constraint, it terminates the Copy RO operation.
2. The DRM Requester generates a *CopyRoRequest* with the information for the RO being Copied to the DRM Agent.
3. The DRM Requester sends the *CopyRoRequest* to the DRM Agent, applying the replay protection mechanism described in section 7.3.
4. The DRM Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.
    b. It verifies the integrity of the request. If the integrity check fails, the DRM Agent sets *CopyRoResponse.Status* to IntegrityVerificationFailed and proceeds to step 5.
    c. It validates the fields of the *CopyRoRequest*. If any field is invalid, the DRM Agent sets *CopyRoResponse.Status* to InvalidField and proceeds to step 5.
    d. It verifies the signature on the RO, including the *SourceCertificateChain* field. If any of the verifications fails, the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    e. It checks if it has a current record (whether installed or waiting to be installed) of an RO with the same ROID. If the duplicate RO exists, the DRM Agent sets *CopyRoResponse.Status* to DuplicateRightsObject and proceeds to step 5.
    f. It checks that the RO has the <copy> permission. If it does not, the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    g. It checks the entity that created the RO. If the RO was created by an RI, the DRM Requester proceeds to step 4.h.
    h. If the LRM's certificate does not have the *localRightsManagerDomain* extended key purpose (see [SCE-LRM]), the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    i. It checks if the RO has a <userDomain> constraint. If the constraint is not present, the DRM Agent sets *CopyROResponse.Status* to InvalidRightsObject and proceeds to step 5.

j. It validates the *UserDomainAuthorization* for the DRM Requester. If the validation fails, the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5. Validation MUST include the following:
    a. Verifying the signature
    b. User Domain Authorization is not expired
    c. Entity ID of User Domain Authorization matches ID of DRM Requester
k. It checks that the User Domain Authorization of the <party> element of the RO corresponds to the RI/LRM that signed the <rights> element, and verifies the DEA's signature on the User Domain Authorization. If the correspondence check or DEA signature verification fails, the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
l. It checks that the User Domain baseID of the <userDomainID> element within the User Domain Authorization in the <party> element of the RO is the same as the User Domain baseID of the <userDomainID> element within the UserDomainAuthorization field. If not, the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
m. It checks that the User Domain generation of the *UserDomainAuthorization* field is greater or equal to the User Domain generation of the User Domain Authorization in the <party> element of the RO. If not, the DRM Agent sets *CopyRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
n. If the DRM Agent is already a member of the User Domain, it checks that the User Domain generation of the UserDomainAuthorization field is greater than or equal to the User Domain generation of the User Domain Authorization in the <party> element of the RO. If not, the DRM Agent sets *CopyRoResponse.Status* to LowUserDomainGeneration and proceeds to step 5.
o. It checks if it has enough room to install the RO. If it does not, it sets *CopyRoResponse.Status* to NotEnoughSpace and proceeds to step 5.
p. It decrypts *CopyRoRequest.Body.EncryptedRek*.
q. **Note 1**: If the DRM Agent is already a member of the User Domain, it installs the RO per [DRM-v2.1] except that the replay cache is not considered. When installed, this RO loses the <copy> permission, i.e. the DRM Agent, acting as a DRM Requester, SHALL NOT Copy the RO to another DRM Agent. Note: if the DRM Agent is not a member of the User Domain, it will not be able to fully decrypt the REK and install the RO until it joins the User Domain and receives a copy of the UDK. If the User Domain Context ([SCE-DOM]) has expired (as indicated by the User Domain Context Expiry Time) the DRM Agent MUST NOT install the RO.
   Note 2: if the RO is afterwards moved to another Device, the <copy> permission remains lost. The information that the <copy> permission is lost is included in the State Information (via the permissionLost field associated with the <copy> permission) that is transmitted via a Move RO transaction as described in Section 9.7. This ensures that once an RO loses the <copy> permission, that permission remains lost, even if the RO is transmitted via one or multiple Move RO transaction(s).
r. It sets *CopyRoResponse.Status* to Success.

5. The DRM Agent sends the *CopyRoResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
6. The DRM Requester processes the response as follows:
    a. It processes the response for replay as described in section 7.3.
    b. It verifies the integrity of the response. If the integrity check failed, the DRM Requester MUST NOT increment the copy counter, if any, of the state information, and MUST terminate the Copy RO operation.
    c. If *CopyRoResponse.Status* is not Success, it determines if it can restart the Copy RO operation at step 2. If it does not restart the operation, the DRM Requester performs the following:
        i. If the <copy> permission had a <count> constraint, it increments the current copy counter of the state information.
        ii. It terminates the Copy RO operation.
    d. At this point the Copy RO operation has successfully completed.

9.8.1 CopyRoRequest

A *CopyRoRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   timeStampPresent     1     bslbf
   rfu                  7     bslbf
```

```
    RoAlias()
    SourceAlias()
    SourceID()
    if( timeStampPresent ){
       TimeStamp()
    }
    RightsObjectContainer()
    EncryptedRek()
    CertificateChain()
    UserDomainAuthorization()
}

EncryptedRek(){
    EncryptedData()  //Contains an encrypted REK
}
```

The fields are defined as follows:

- *timeStampPresent* – this is a boolean field, that if true, indicates that the source (RI or LRM) *TimeStamp* field is present.
- *rfu* – this is a 7 bit field that is reserved for future use. When sending the request, MUST be set to 0. When processing this field, its value MUST be ignored.
- *RoAlias* – this field contains an optional alias for the RO. It is defined in section 9.7.1.
- *DomainAlias* – this field contains an optional alias for the domain if the RO is a domain RO. It is defined in section 9.7.1.
- *SourceAlias* – this field contains an optional alias for the Rights Issuer or LRM that created the RO. It is defined in section 9.7.1.
- *SourceID* – this field contains the identity of the Rights Issuer or LRM that created the RO. It is defined in section 9.7.1.
- *RightsObjectContainer* – this field contains an RO as defined in section 8.19.
- *EncryptedRek* – this field contains an REK that has been encrypted twice. The REK is first encrypted with the UDK (for the User Domain) using [AES-WRAP] and then the wrapped REK is encrypted with the SK using the negotiated algorithm. The field is of type *EncryptedData* which is defined in section 8.11. A *Rek* field is defined in section 9.7.1.
- *TimeStamp* – this field constains the timestamp of the Rights Issuer or LRM that created the RO. It is defined in section 9.7.1.
- *CertificateChain* – this field contains the certificate chain for the Rights Issuer or LRM that created the RO. This field is defined in section 8.8.
- *UserDomainAuthorization* – this field contains the User Domain Authorization for the DRM Requester.

9.8.2 CopyRoResponse

A *CopyRoResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 16: CopyRoResponse Status Values

Status Values
Success
InvalidField
InvalidRightsObject
DuplicateRightsObject
NotEnoughSpace
IntegrityVerificationFailed

The body of a *CopyRoResponse* is empty and is defined as follows:

```
Body(){
}
```

9.9 Share RO Operation

The Share RO operation is used by the DRM Requester to do Ad Hoc Sharing of a RO. This operation MUST take place using a SAC. This operation MUST NOT be performed if the DRM Requester's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set or the DRM Agent's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set (see section A.1). The DRM Agent MUST reject the RO if the <signature> element over the <rights> element has been generated by an entity other than an RI or an LRM. The following figure illustrates the Share RO operation.

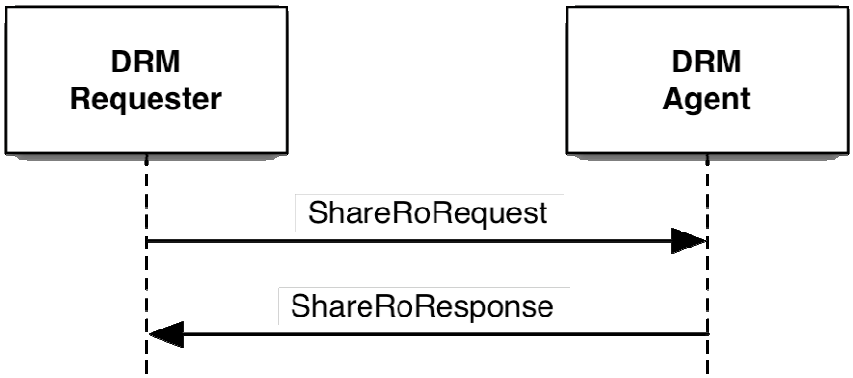

Figure 10: Share RO Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester checks if the RO has the <adhoc-share> permission and any constraints. If the RO cannot be Ad Hoc Shared, the Share RO operation is terminated. In particular, the DRM Requester MUST check the following:
    a. If the <banning-interval> constraint is present, then it ensures that the banning interval timer for this DRM Agent has elapsed. If the banning interval timer has elapsed, then the DRM Requester starts the banning interval timer for this DRM Agent with the value of the <banning-interval> constraint.
    b. If the <max-concurrent> constraint is present, then it ensures that the number of DRM Agents it is currently performing Adhoc Sharing with is less than the <max-concurrent> value. If the number of DRM Agents is less, then the DRM Requester increments the concurrent counter of DRM Agents for this RO.
2. The DRM Requester generates a *ShareRoRequest*.
3. The DRM Requester sends the *ShareRoRequest* to the DRM Agent, applying the replay protection mechanism described in section 7.3.
4. The DRM Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.
    b. It validates the fields of the *ShareRoRequest*. If any field is invalid, it sets *ShareRoResponse.Status* to InvalidField and proceeds to step 5.
    c. It verifies the integrity of the request. If the integrity check fails, it sets *ShareRoResponse.Status* to IntegrityVerificationFailed and proceeds to step 5.
    d. It verifies the signature on the RO, including the *SourceCertificateChain* field. If any of the verifications fails, it sets *ShareRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    e. It checks that the RO has the <adhoc-share> permission. If it does not, it sets *ShareRoResponse.Status* to InvalidRightsObject and proceeds to step 5.
    f. It checks that the RO contains a <cekHash> element in the <context> element in the <party> element. If it doesn't, the DRM Agent sets *ShareROResponse.Status* to InvalidRightsObject and proceeds to step 5.
    g. It calculates, using the CEKs and CEK hashes, the CEKhash as defined in [SCE-REL]. If the value is different from the value in the <cekHash> element, the DRM Agent sets *ShareROResponse.Status* to InvalidRightsObject and proceeds to step 5.

h. It checks if it has enough room to install the RO. If it does not, it sets *ShareRoResponse.Status* to NotEnoughSpace and proceeds to step 5.
i. It installs the RO per [DRM-DRM-V2.1] except that the replay cache is not considered. It marks the RO as "shared", meaning that only the permissions under the <adhoc-share> permission can be granted.
j. It sets *ShareRoResponse.Status* to Success.

5. The DRM Agent sends the *ShareRoResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
6. The DRM Requester processes the response as follows:
   a. It processes the response for replay as described in section 7.3.
   b. If *ShareRoResponse.Status* is not Success, it determines if it can restart the Share RO operation at step 2. If it does not restart the operation, it performs the following:
      i. If the RO contains the <banning-interval> constraint, it causes the banning interval timer for this DRM Agent to elapse.
      ii. If the RO contains the <max-concurrent> constraint, it decrements the concurrent counter of DRM Agents.
      iii. It terminates the Share RO operation.
   c. At this point the Share RO operation has successfully completed.

9.9.1 ShareRoRequest

A *ShareRoRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   RightsObjectContainer()
   CertificateChain()
   CekInfo()
}
```

The fields are defined as follows:

- *RightsObjectContainer* – this field contains a RO as defined in section 8.19.
- *CertificateChain* – this field contains the certificate chain for the Rights Issuer or LRM that created the original RO. This field is defined in section 8.8
- *CekInfo* – this field contains, per asset, the Content Encryption Key (CEK) , encrypted with the SK, or the SHA-1 hash over the CEK. The field is defined in section 8.19. Note that the encrypted CEKs are only delivered for those assets that are Ad Hoc Shared.

9.9.2 ShareRoResponse

A *ShareRoResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 17: ShareRoResponse Status Values

Status Values
Success
InvalidField
NotEnoughSpace
IntegrityVerificationFailed
InvalidRightsObject

The body of a *ShareRoResponse* is empty and is defined as follows:

```
Body(){
}
```

9.10 Lend RO Operation

The Lend RO operation is used by the DRM Requester to do Lending of a RO. This operation MUST take place using a SAC. This operation MUST NOT be performed if the DRM Requester's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set or the DRM Agent's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set (see section A.1). The DRM Agent MUST reject the RO if the <signature> element over the <rights> element has been generated by an entity other than an RI or an LRM. The following figure illustrates the Lend RO operation.

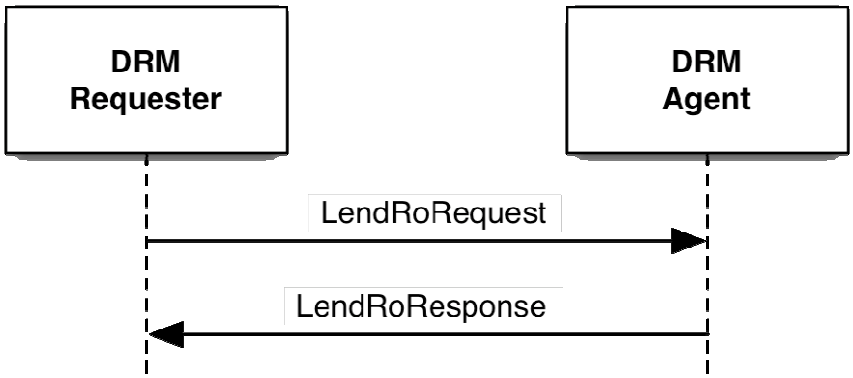

Figure 11: Lend RO Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester does the following:
    a. It checks if the RO has the <lend> permission and any constraints. If the RO contains stateful constraints for consumption by the DRM Requester or if the RO cannot be Lent, the Lend RO operation is terminated.
    b. It marks the RO as unusable.
    c. It creates a Lending context for this RO that includes the ROID, the *lendingHandle*, the DRM Agent's ID and a lending interval timer.
    d. It generates a random *lendingHandle* and copies it to the Lending context and the *LendRoRequest*.
2. It starts the lending interval timer in the Lending context using the value of the <lending-interval> constraint. Note that once this lending interval timer expires, the DRM Requester marks the RO as usable again.
3. The DRM Requester generates a *LendRoRequest*.
4. The DRM Requester sends the *LendRoRequest* to the DRM Agent, applying the replay protection mechanism described in section 7.3.
5. The DRM Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.
    b. It validates the fields of the *LendRoRequest*. If any field is invalid, it sets *LendRoResponse.Status* to InvalidField and proceeds to step 6.
    c. It verifies the integrity of the request. If the integrity check fails, it sets *LendRoResponse.Status* to IntegrityVerificationFailed and proceeds to step 6.
    d. It verifies the signature on the RO, including the *SourceCertificateChain* field. If any of the verifications fails, it sets *LendRoResponse.Status* to InvalidRightsObject and proceeds to step 6.
    e. It checks that the RO has the <lend> permission. If it does not, it sets *LendRoResponse.Status* to InvalidRightsObject and proceeds to step 6.
    f. It checks that the RO does not contain stateful constraints for consumption by the DRM Requester. If it does, it sets *LendRoResponse.Status* to InvalidRightsObject and proceeds to step 6.
    g. It checks that the RO contains a <cekHash> element in the <context> element in the <party> element. If it doesn't, the DRM Agent sets *LendROResponse.Status* to InvalidRightsObject and proceeds to step 6.

h. It calculates, using the CEKs and CEK hashes, the CEKhash as defined in [SCE-REL]. If the value is different from the value in the <cekHash> element, the DRM Agent sets *LendROResponse.Status* to InvalidRightsObject and proceeds to step 6.
i. It checks that the <lend> permission has an <lending-interval> constraint. If it does not, it sets *LendRoResponse.Status* to InvalidRightsObject and proceeds to step 6.
j. It checks if it has enough room to install the RO. If it does not, it sets *LendRoResponse.Status* to NotEnoughSpace and proceeds to step 6.
k. It installs the RO per [DRM-DRM-V2.1] except that the replay cache is not considered. It marks the RO as "lent".
l. It creates a Lent context for this RO that includes the ROID, the *lendingHandle*, the DRM Requester's ID and a lending timer.
m. It starts the lending timer in the Lent context with the value of the <lending-interval> constraint of the <lend> permission.
n. It sets *LendRoResponse.Status* to Success.

6. The DRM Agent sends the *LendRoResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
7. The DRM Requester processes the response as follows:
   a. It processes the response for replay as described in section 7.3.
   b. If *LendRoResponse.Status* is not Success, it determines if it can restart the Lend RO operation at step 2. If it does not restart the operation, the DRM Requester performs the following:
      i. It marks the RO as usable.
      ii. It removes the Lending context, stopping the lending interval timer.
      iii. It terminates the Lend RO operation.
   c. At this point the Lend RO operation has successfully completed.

After the successful execution of the Lend RO operation, the DRM Agent MAY grant the following permissions (if present and subject to any constraints): <play>, <display> and <execute>. Other permissions that are present MUST NOT be granted.

9.10.1 LendRoRequest

A *LendRoRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   lendingHandle       32      uimsbf
   RightsObjectContainer()
   CertificateChain()
   CekInfo()
}
```

The fields are defined as follows:

- *lendingHandle* – this field contains a 32 bit unsigned integer assigned by the DRM Requester to identify the RO being Lent. The DRM Requester can use this value in the Lend Release operation (see section 9.11) to release the RO.
- *RightsObjectContainer* – this field contains a RO as defined in section 8.19.
- *CertificateChain* – this field contains the certificate chain for the Rights Issuer or LRM that created the original RO. This field is defined in section 8.8
- *CekInfo* – this field contains, per asset, the Content Encryption Key (CEK), encrypted with the SK, or the SHA-1 hash over the CEK. The field is defined in section 8.19. Note that the encrypted CEKs are only delivered for those assets that are Lent.

9.10.2 LendRoResponse

A *LendRoResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 18: LendRoResponse Status Values

Status Values
Success

Status Values
InvalidField
IntegrityVerificationFailed
InvalidRightsObject
NotEnoughSpace

The body of a *LendRoResponse* is empty and is defined as follows:

```
Body(){
}
```

9.11 Lend Release Operation

The Lend Release operation is used by the DRM Requester to release a RO it had previously received via a Lend operation (see section 9.10). This operation MUST take place using a SAC. This operation MUST NOT be performed if the DRM Requester's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set or the DRM Agent's certificate does not have an `extKeyUsage` extension with `oma-kp-sceDrmAgent` key purpose set (see section A.1). Note that for this operation to succeed, the DRM Requester (for this operation) MUST be the DRM Agent that received the Lent RO and the DRM Agent (for this operation) MUST be the DRM Requester that Lent the RO. The following figure illustrates the Lend Release operation.

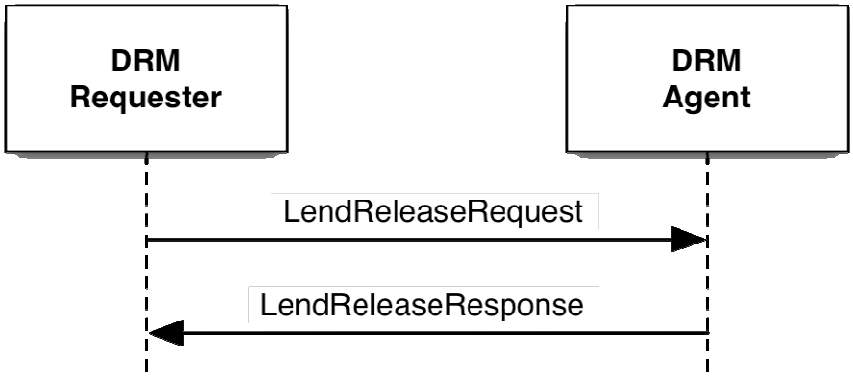

Figure 12: Lend Release Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester generates a *LendReleaseRequest* using the data from the Lent context for the RO.
2. The DRM Requester sends the *LendReleaseRequest* to the DRM Agent, applying the replay protection mechanism described in section 7.3.
3. The DRM Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.
    b. It validates the fields of the *LendReleaseRequest*. If any field is invalid, it sets *LendReleaseResponse.Status* to InvalidField and proceeds to step 4.
    c. It verifies the integrity of the request. If the integrity check fails, it sets *LendReleaseResponse.Status* to IntegrityVerificationFailed and proceeds to step 4.
    d. It checks if it has a Lending context for the *lendingHandle* and DRM Requester ID. If it does not have a Lending context, it sets *LendReleaseResponse.Status* to UnknownHandle and proceeds to step 4.
    e. It marks the RO corresponding to the *lendingHandle* as usable and removes the Lending context.
    f. It sets *LendReleaseResponse.Status* to Success.
4. The DRM Agent sends the *LendReleaseResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
5. The DRM Requester processes the response as follows:
    a. It processes the response for replay as described in section 7.3.

b. If *LendReleaseResponse.Status* is not Success, it determines if it can restart the Lend Release RO operation at step 1. If it does not restart the operation, it terminates the Lend Release operation.
c. It deletes the Lent RO it just released and removes the Lent context.
d. At this point the Lend Release operation has successfully completed.

9.11.1 LendReleaseRequest

A *LendReleaseRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   lendingHandle       32      uimsbf
}
```

The fields are defined as follows:

- *lendingHandle* – this field contains a 32 bit unsigned integer that was previously assigned by the DRM Requester (that Lent the RO) to identify the RO being released.

9.11.2 LendReleaseResponse

A *LendReleaseResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 19: LendReleaseResponse Status Values

Status Values
Success
InvalidField
IntegrityVerificationFailed
UnknownHandle

The body of a *LendReleaseResponse* is empty and is defined as follows:

```
Body(){
}
```

9.11.3 Lending Expiration

If the DRM Requester does not release the Lent RO and the lending timer of the corresponding Lent context expires, it MUST perform the following:

1. Delete the Lent RO.
2. Remove the Lent context.

9.12 Render Operation

The Render operation is used by the DRM Requester to securely deliver the CEK for the DRM Content to the Render Agent so that the DRM Content can be rendered remotely. The DRM Content is identified by its Asset ID (see section 8.18). This operation MUST take place using a SAC. All ROs are implicitly allowed to be rendered remotely. Although not within the scope of this specification, it is assumed that the Render Agent will lose knowledge of the CEK after the rendering of the DRM Content is complete. In addition, the DRM Requester MUST ensure that the rendering application on the Render Client is trustworthy and securely communicates the rendering status to the DRM Requester.

This operation MUST NOT be performed if the DRM Requester's certificate does not have an `extKeyUsage` extension with `oma-kp-sceRenderSource` key purpose set or the Render Agent's certificate does not have an `extKeyUsage` extension with `oma-kp-sceRenderAgent` key purpose set (see section A.1). The following figure illustrates the Render operation.

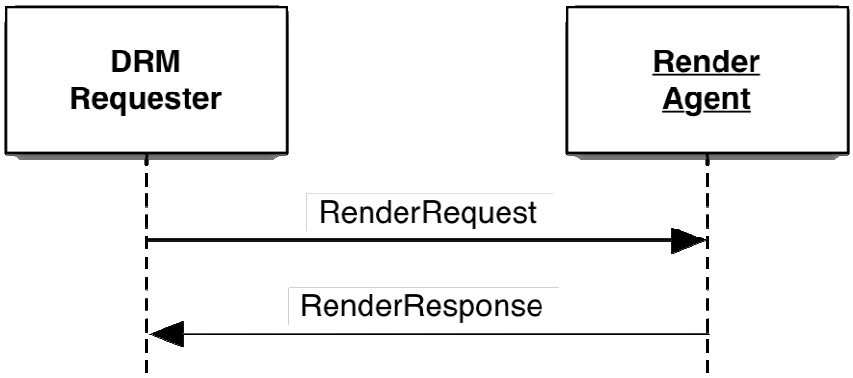

Figure 13: Render Operation

In order for this operation to take place, the following MUST be performed:

1. The DRM Requester generates a *RenderRequest*.
2. The DRM Requester sends the *RenderRequest* to the Render Agent, applying the replay protection mechanism described in section 7.3.
3. The Render Agent processes the request as follows:
    a. It processes the request for replay as described in section 7.3.
    b. It validates the fields of the *RenderRequest*. If any field is invalid, it sets *RenderResponse.Status* to InvalidField and proceeds to step 4.
    c. It verifies the integrity of the request. If the integrity check fails, it sets *RenderResponse.Status* to IntegrityVerificationFailed and proceeds to step 4.
    d. It decrypts the CEK.
    e. It creates a Render context with the *renderHandle*, *AssetId*, CEK and the DRM Requester ID.
    f. It sets *RenderResponse.Status* to Success.
4. The Render Agent sends the *RenderResponse* to the DRM Requester, applying the replay protection mechanism described in section 7.3.
5. The DRM Requester processes the response as follows:
    a. It processes the response for replay as described in section 7.3.
    b. If *RenderResponse.Status* is not Success, it determines if it can restart the Render operation at step 1. If it does not restart the operation, it terminates the Render operation.
    c. It creates a Render context, associating the *renderHandle*, *AssetID* and Render Agent ID.
    d. At this point the Render operation has successfully completed.

9.12.1 RenderRequest

A *RenderRequest* is sent as a protected request and its body is defined as follows:

```
Body(){
   renderHandle        32     uimsbf
   AssetID()
   EncryptedCek()
}
```

The fields are defined as follows:

- *renderHandle* – this field contains a 32 bit unsigned integer assigned by the DRM Requester to identify the rendering of the DRM Content. *AssetID – th*is field contains the identification of the DRM Content that the Render Agent should render. It is defined in section 8.18.
- *EncryptedCek* – this field contains the Content Encryption Key (CEK) , encrypted with the SK, for decrypting the DRM Content. It is defined in section 8.12.

9.12.2 RenderResponse

A *RenderResponse* is sent as a protected response. The following table lists the valid *Status* values for this response.

Table 20: RenderResponse Status Values

Status Values
Success
InvalidField
IntegrityVerificationFailed

The body of a *RenderResponse* is empty and is defined as follows:

```
Body(){
}
```

10.*SourceCertificateChain* Revocation Checking

When receiving an RO via an instance of a Move RO transaction, Copy RO operation, Share RO operation, or Lend RO operation, the DRM Agent MAY refuse to install the RO or consume the RO for the first time if, based on available CRL or OCSP information the (RI or LRM) entity that generated the <signature> element over the <rights> element has been revoked. The decision to implement this functionality is left to the Trust Authority.

Such CRL MAY be acquired by the Device by using the Get CRL operation (see section 9.6) or by other means. Such OCSP response MAY be acquired during registration of the Device with the specific entity. Such OCSP response MAY be acquired through unspecified communication of the Device with another Device or other entity. This checking of the revocation status of the source entity identified by the *SourceCertificateChain* field is in addition to verification of the signature on the RO, including the *SourceCertificateChain* field as specified within the processing of the transaction or operation (see section 9).

Note that to be sure that the DRM Requester does not inadvertently lose access to the content received via one of the named transactions or operations, a fresh exchange between the DRM Requester and DRM Agent of the latest revocation information MAY be performed prior to completing the transaction or operation.

11.Security Considerations (Informative)

In addition to the Security Considerations of [DRM-DRM-v2.1], several additional factors have to be considered when allowing the features of this specification, as described below. This list is not claimed to be exhaustive.

11.1 Entity Compromise

11.1.1 DRM Requester Compromise

A compromised DRM Requester may result in any of the following:

- Duplication of ROs – the DRM Requester does not remove Moved ROs or allows the restoration of ROs that have been Moved. This may not require actual restoration of the RO, i.e. only resetting of state, in the case of Partial Rights Moves.
- Move of Duplicated ROs – Rights duplicated as above are Moved by an unknown-compromised DRM Requester. Because of the allowance of Partial Rights Move, multiple Moves (even to the same DRM Agent) may not be considered suspicious.

11.1.2 DRM Agent Compromise

A compromised DRM Agent may result in any of the following:

- Duplication of ROs – the DRM Agent does not remove Shared or Lent ROs after the Sharing or Lending time has expired.

11.1.3 Render Agent Compromise

A compromised Render Agent may result in any of the following:

- Disclosure of the CEK.
- Disclosure of Protected Content – the Render Agent releases the plaintext DRM Content to a compromised rendering application on the Render Client.

11.2 DRM Time

Although Devices implementing this specification are required to support DRM Time, there is no explicit checking of DRM Time between a DRM Requester and a DRM Agent. Since Devices are not trusted sources of time, it is possible that the DRM Requester's DRM Time and the DRM/Render Agent's DRM Time will be different when A2A functionality is performed.

11.3 CRL Distribution

Revocation status checking depends on the timely distribution of CRLs. Without such distribution, Devices and Render Clients may not be aware that an entity has been revoked. Note that an entity may have a valid and current CRL but not be aware that a new CRL is available.

Appendix A. Certificates and CRLs

A.1 Certificate Profiles and Requirements

The profile for DRM/Render Agent certificates used in this specification follows the profile of the DRM Agent Certificates in [DRM-DRM-v2.1] with the exceptions and additions as described below.

Table 21: DRM Agent Certificate Profile

Fields	Values

Extensions, extKeyUsage	The `extKeyUsage` extension SHALL be present and MUST contain the `oma-kp-drmAgent` key purpose object identifier as stated in [DRM-DRM-v2.1]. In addition, the following key purpose object identifier MUST be present:
	`oma-kp-sceDrmAgent OBJECT IDENTIFIER ::= {oma-kp 4}`
	If the DRM Agent is allowed to remotely render via a Render Agent, the following key purpose object identifier MUST be present:
	`oma-kp-sceRenderSource OBJECT IDENTIFIER ::= {oma-kp 5}`

Extensions, cRLDistributionPoints	The `cRLDistributionPoints` extension SHALL be present. It MUST have at least one `DistributionPoint` that in turn MUST have a `distributionPoint` with a URL of where to obtain a CRL.

Table 22: Render Agent Certificate Profile

Fields	Values
Extensions, extKeyUsage	The `extKeyUsage` extension SHALL be present and the following key purpose object identifier MUST be present: `oma-kp-sceRenderAgent OBJECT IDENTIFIER ::= {oma-kp 6}`
Extensions, cRLDistributionPoints	The `cRLDistributionPoints` extension SHALL be present. It MUST have at least one `DistributionPoint` that in turn MUST have a `distributionPoint` with a URL of where to obtain a CRL.

If allowed by the trust model, a Device MAY contain both a DRM Agent and a Render Agent. In that case, the certificate MUST contain the oma-kp-sceDrmAgent, the oma-kp-drmAgent and the oma-kp-sceRenderAgent key purpose.

A.2 CRL Profiles and Requirements

The profile for CRLs SHALL follow the CRL profile as stated in [SRM-TS].

Appendix B. Static Conformance Requirements (Normative)

The notation used in this appendix is specified in [SCR-RULES].

B.1 SCR for DRM Agent

Item	Function	Reference	Status	Requirement
A2A-DA-001	Support CRLs	5.2.1	M	
A2A-DA-002	Support replay protection.	7.3	M	
A2A-DA-003	Support the A2A Hello operation	9.1	M	
A2A-DA-004	Support the MAKE transaction	9.2	M	A2A-DA-001
A2A-DA-005	Support AEA encryption	9.2.7	M	A2A-DA-004
A2A-DA-006	Support the Change SAC operation	9.3	O	
A2A-DA-007	Support the CRL Query operation	9.4	M	A2A-DA-001
A2A-DA-008	Support the Put CRL operation	9.5	M	A2A-DA-001
A2A-DA-009	Support the Get CRL operation	9.6	M	A2A-DA-001
A2A-DA-010	Support the Move RO operation	9.7	M	A2A-DA-004
A2A-DA-011	Support checking the `oma-kp-sceDrmAgent` key purpose of the DRM Requester	9.7, 9.8, 9.9, 9.10	M	
A2A-DA-012	Support the Share RO operation	9.8	M	A2A-DA-004
A2A-DA-013	Support the Lend RO operation	9.9	M	A2A-DA-004
A2A-DA-014	Support the Lend Release operation	9.10	M	A2A-DA-004

B.2 SCR for DRM Requester

Item	Function	Reference	Status	Requirement
A2A-DR-001	Support CRLs	5.2.1	M	
A2A-DR-002	Support replay protection.	7.3	M	
A2A-DR-003	Support the A2A Hello operation	9.1	M	
A2A-DR-004	Support the MAKE transaction	9.2	M	A2A-DR-001
A2A-DR-005	Support AEA encryption	9.2.7	M	A2A-DR-004
A2A-DR-006	Support the Change SAC operation	9.3	O	
A2A-DR-007	Support the CRL Query operation	9.4	M	A2A-DR-001

Item	Function	Reference	Status	Requirement
A2A-DR-008	Support the Put CRL operation	9.5	M	A2A-DR-001
A2A-DR-009	Support the Get CRL operation	9.6	M	A2A-DR-001
A2A-DR-010	Support the Move RO operation	9.7	M	A2A-DR-004
A2A-DR-011	Support checking the `oma-kp-sceDrmAgent` key purpose of the DRM Agent	9.7, 9.8, 9.9, 9.10	M	
A2A-DR-012	Support the Share RO operation	9.8	M	A2A-DR-004
A2A-DR-013	Support the Lend RO operation	9.9	M	A2A-DR-004
A2A-DR-014	Support the Lend Release operation	9.10	M	A2A-DR-004
A2A-DR-015	Support the Render operation	9.11	O	A2A-DR-004
A2A-DR-016	Support checking the `oma-kp-sceRenderAgent` key purpose of the Render Agent		O	

B.3 SCR for Render Agent

Item	Function	Reference	Status	Requirement
A2A-RA-001	Support CRLs	5.2.1	M	
A2A-RA-002	Support replay protection.	7.3	M	
A2A-RA-003	Support the A2A Hello operation	9.1	M	
A2A-RA -004	Support the MAKE transaction	9.2	M	A2A-RA-001
A2A-RA -005	Support AEA encryption	9.2.7	M	A2A-RA-004
A2A-RA -006	Support the Change SAC operation	9.3	O	A2A-RA-001
A2A-RA -007	Support the CRL Query operation	9.4	M	A2A-RA-001
A2A-RA -008	Support the Put CRL operation	9.5	M	A2A-RA-001
A2A-RA -009	Support the Get CRL operation	9.6	M	A2A-RA-001
A2A-RA -010	Support the Render operation	9.11	M	A2A-RA-004
A2A-RA -011	Support checking the `oma-kp-sceRenderSource` key purpose of the DRM Requester	9.11, 9.12	M	

Appendix C. Examble A2A Message (Informative)

Below are some sample A2A Messages. The last row contains the values (in hex). The second to the last row contains the offset (in decimal) from the beginning of the message.

The following is an example *A2AHelloRequest*:

A2ARequest							
MessageId	Body						ExtensionsContainer
	Version	TrustAnchorAndEntityIdPairList					nbrOfEntries
		nbrOfEntries	TrustAnchor		EntityId		
			length	octets	length	octets	
0	1	2	3	4 – 23	24	25 – 44	45
0x00	0x10	0x01	0x14	hash	0x14	hash	0x00

The following is an example *CrlQueryRequest*:

A2ARequest	
MessageId	ExtensionsContainer
	nbrOfEntries
0	5
0x08	0x00

The following is an example *LendReleaseRequest*:

A2AProtectedRequest				
MessageId	replayCounter	Body	ExtensionsContainer	Hmac
		lendingHandle	nbrOfEntries	
0	1 – 4	5 – 8	9	10 – 29
0x14	0x12345678	0x98765432	0x00	hmac

The following is an example *CrlQueryResponse* with *Status* = Success:

A2AResponse						
MessageId	Status	Body				ExtensionsContainer
		CrlIdList				
		nbrOfEntries	CrlIssuerId	CrlNumber		
				length	octets	nbrOfEntries
0	1	2	3 – 22	23	24 – 25	26
0x09	0x00	0x01	hash	0x02	0x1234	0x00

The following is an example *PutCrlResponse* with *Status* = Success:

A2AResponse		
MessageId	Status	ExtensionsContainer
		nbrOfEntries
0	1	2
0x0B	0x00	0x00

The following is an example *A2AHelloResponse* with *Status* = InvalidField:

A2AResponse		
MessageId	Status	ExtensionsContainer
		nbrOfEntries
0	1	5
0x01	0x15	0x00

The following is an example *LendReleaseResponse* with *Status* = Success:

A2AProtectedResponse				
MessageId	replayCounter	Status	ExtensionsContainer	Hmac
			nbrOfEntries	
0	1 – 4	5	6	7 – 26
0x0B	0x23455432	0x00	0x00	hmac

The following is an example *PutRoResponse* with *Status* = InvalidField:

A2AProtectedResponse				
MessageId	replayCounter	Status	ExtensionsContainer	Hmac
			nbrOfEntries	
0	1 – 4	5	6	7 – 26
0x15	0x34566543	0x15	0x00	hmac

Appendix D. Change History (Informative)

D.1 Approved Version History

Reference	Date	Description
OMA-TS-SCE_A2A-V1_0-20110705-A	05 Jul 2011	Status changed to Approved by TP: OMA-TP-2011-0233-INP_SCE_V1_0_ERP_for_Final_Approval