A refined architecture for DRM

Koen Buyens *Sam Michiels* *Wouter Joosen*

Report CW450, June 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A refined architecture for DRM

Koen Buyens *Sam Michiels* *Wouter Joosen*

Report CW450, June 2006

Department of Computer Science, K.U.Leuven

Abstract

Although various publications confirm the need for a generic DRM software architecture, we observe that current state-of-the-art DRM architectures are not sufficiently detailed to enable the creation and management of DRM systems and content distribution applications. This is a considerable problem that implies a crucial challenge for the evolution of DRM, given the impact of a software architecture on the functional and non-functional qualities of the implementation.

This report presents (1) a critical analysis of a previously presented DRM architecture, (2) a refined DRM architecture that handles the main issues that were identified in this analysis, (3) a detailed description of the proposed architecture from four perspectives: content handling, consumer tracking, licensing, and content securing, (4) detailed interfaces of all identified components, (5) scenarios to illustrate usage of the architecture, and (6) an evaluation of the proposed architecture based on nine major quality attributes. The architecture is compared with related work in two DRM projects: Digital Media Project (DMP) and Automating Production of Cross Media Content for Multi-channel Distribution (AXMEDIS). To the best of our knowledge, the proposed architecture is more sophisticated than related work published so far.

Contents

1 Introduction 7

2 Starting point: a generic layered DRM architecture 7
- 2.1 Architecture . . . 7
  - 2.1.1 Users . . . 8
  - 2.1.2 A distributed view . . . 9
  - 2.1.3 Identifying layers . . . 10
  - 2.1.4 Service layer . . . 10
  - 2.1.5 Negotiation layer . . . 12
  - 2.1.6 Rights enforcement layers . . . 12
- 2.2 Issues . . . 12
  - 2.2.1 Tracking content transparently . . . 12
  - 2.2.2 Traversing content non-sequentially . . . 13
  - 2.2.3 Separating publishing components . . . 13
- 2.3 Towards a refined software architecture . . . 13

3 Publishing context 14

4 High level architectural overview 15
- 4.1 Component groups . . . 15
  - 4.1.1 Exposed router/dispatcher . . . 18
  - 4.1.2 Exposed broker . . . 19
  - 4.1.3 Blackboard . . . 20
  - 4.1.4 Solution . . . 21

5 Detailed architecture 22
- 5.1 Architecture perspectives . . . 22
- 5.2 Content handling perspective . . . 23
  - 5.2.1 Component diagram . . . 23
  - 5.2.2 Service component group . . . 23
  - 5.2.3 RI component group . . . 27
  - 5.2.4 External components . . . 28
  - 5.2.5 Scenario . . . 28
- 5.3 Tracking perspective . . . 30
  - 5.3.1 Component diagram . . . 30
  - 5.3.2 Service component group . . . 30
  - 5.3.3 RI component group . . . 32
  - 5.3.4 Organizing the internal RI component group components . . . 34
  - 5.3.5 Scenario . . . 38

5.4 Licensing perspective . . . 40
5.4.1 Component diagram . . . 40
5.4.2 Service component group . . . 42
5.4.3 RI component group . . . 42
5.4.4 Scenario . . . 42
5.5 Security perspective . . . 42
5.5.1 Architecture . . . 43
5.5.2 Component diagram . . . 46
5.5.3 Security component group . . . 46
5.5.4 Scenario . . . 49
5.6 Putting the components together . . . 49
5.7 Deployment view . . . 49
5.7.1 Conclusion . . . 50
5.8 Application Layer . . . 53
5.8.1 Content oriented . . . 53
5.8.2 Statistics oriented . . . 54

6 Evaluation and related work 54
6.1 Evaluation criteria and quality attributes . . . 54
6.2 Evaluation of the Distrinet DRM architecture . . . 58
6.3 Evaluation of the Digital Media Project architecture . . . 59
6.3.1 Interoperability . . . 59
6.3.2 Accountability . . . 60
6.3.3 Modifiability . . . 60
6.3.4 Extendability . . . 60
6.3.5 Usability for the developer . . . 61
6.3.6 Testability . . . 61
6.3.7 Availability . . . 61
6.3.8 Performance . . . 61
6.3.9 Reusability . . . 61
6.4 Evaluation of the AXMEDIS architecture . . . 62
6.4.1 Interoperability . . . 62
6.4.2 Accountability . . . 62
6.4.3 Modifiability . . . 62
6.4.4 Extendability . . . 63
6.4.5 Usability for the developer . . . 63
6.4.6 Testability . . . 63
6.4.7 Availability . . . 63
6.4.8 Performance . . . 63
6.4.9 Reusability . . . 63
6.5 Conclusion . . . 64

7 Future work 64
7.1 Securing the proposed architecture . . . 64
7.2 Anonymizing consumer tracking . . . 64
7.3 Adapting DRM protected content . . . 64
7.4 Managing domains . . . 65
7.5 Broadcasting personalized content . . . 65
7.5.1 Problem . . . 65
7.5.2 Possible solution . . . 65
7.5.3 Extensions . . . 66

8 Conclusion 67

A Detailed use cases 68
A.1 Content handling perspective . . . 68
A.1.1 Consumer . . . 68
A.1.2 Producer . . . 70
A.1.3 Publisher . . . 70
A.2 Tracking perspective . . . 70
A.2.1 Consumer . . . 70
A.3 Producer . . . 73
A.4 Publisher . . . 73
A.5 Licensing perspective . . . 74
A.5.1 Consumer . . . 74
A.5.2 Producer . . . 77
A.6 Security perspective . . . 78
A.6.1 Consumer . . . 78
A.6.2 Producer . . . 80

B Interfaces of the service component group components 83
B.1 Content Service . . . 83
B.1.1 Consumer Delivery . . . 83
B.1.2 Service Interface . . . 84
B.2 Input Management System . . . 85
B.2.1 Content Input . . . 86
B.3 Content Management System . . . 86
B.3.1 Ready Content . . . 87
B.3.2 CMS Input Interface . . . 87
B.3.3 Content In Progress . . . 88
B.4 User Management System . . . 88
B.4.1 Current Identity . . . 88
B.4.2 User Profile . . . 89

B.5 Identification Service . . . 89
B.5.1 Identification . . . 90
B.6 Payment Service . . . 91
B.7 License Service . . . 91
B.7.1 Consumer Interface . . . 91
B.7.2 License Management Interface . . . 95
B.8 Accounting Service . . . 97
B.8.1 Tracking Interface . . . 97
B.8.2 Accounting Interface . . . 98

C Interfaces of the RI component group components 100
C.1 Dispatcher Component . . . 100
C.1.1 Internal Message Interface . . . 100
C.1.2 External Message Interface . . . 102
C.2 Logging Component . . . 102
C.2.1 Message Interface . . . 102
C.3 Policy Engine Component . . . 103
C.3.1 Policy Interpretation Interface . . . 103
C.3.2 Message Interface . . . 103
C.4 Representation Component . . . 104
C.4.1 Convert/Create . . . 104
C.4.2 Representation Policy . . . 104
C.4.3 Message Interface . . . 104
C.5 Context Component . . . 105
C.5.1 Message Interface . . . 105

D Interfaces of the security component group components 106
D.1 Fingerprinting Component . . . 106
D.1.1 Generate Fingerprint . . . 106
D.2 Watermarking Component . . . 107
D.2.1 Read Watermark . . . 107
D.2.2 Add Watermark . . . 108
D.3 Certificate Management Component . . . 110
D.3.1 Certificate Management Interface . . . 110
D.4 Key Management Component . . . 111
D.4.1 Key Management Interface . . . 111
D.5 Digital Signature Component . . . 113
D.5.1 Interface . . . 113
D.6 Key store . . . 115
D.6.1 Store/Load . . . 115
D.7 Key generation . . . 115

D.7.1 Generate . . . 115
D.8 Encryption . . . 115
D.8.1 Encryption/Decryption . . . 115
D.9 Hash . . . 116
D.9.1 Hash . . . 116
D.10 Secure Storage . . . 116
D.10.1 Store/Retrieve/Delete . . . 116

E DMP architecture **117**
E.1 Context . . . 117
E.2 Users . . . 117
E.3 Architecture . . . 118
E.3.1 DMP Functions . . . 118
E.3.2 DMP Distribution and consumption Model . . . 123
E.3.3 DMP Tool Model . . . 126
E.4 Mapping onto proposed architecture . . . 128
E.4.1 Users . . . 128
E.4.2 Distributed View . . . 130
E.4.3 Components . . . 130
E.4.4 Interfaces . . . 131

F AXMEDIS architecture **135**
F.1 Context . . . 135
F.2 Users . . . 136
F.3 Architecture . . . 138
F.3.1 AXMEDIS factory . . . 142
F.3.2 AXMEDIS Distribution Area . . . 144
F.3.3 AXMEDIS Protection and Supervising Tools . . . 144
F.3.4 AXMEDIS Players . . . 145
F.4 Framework . . . 145
F.5 Mapping onto proposed architecture . . . 146
F.5.1 Users . . . 146
F.5.2 Distributed view . . . 149
F.5.3 Components . . . 149
F.5.4 Interfaces . . . 154

A refined architecture for DRM

Koen Buyens, Sam Michiels and Wouter Joosen
Research group DistriNet
Department of Computer Science
K.U.Leuven, Belgium
{koen.buyens, sam.michiels, wouter.joosen}@cs.kuleuven.be

June 30, 2006

Abstract

Although various publications confirm the need for a generic DRM software architecture, we observe that current state-of-the-art DRM architectures are not sufficiently detailed to enable the creation and management of DRM systems and content distribution applications. This is a considerable problem that implies a crucial challenge for the evolution of DRM, given the impact of a software architecture on the functional and non-functional qualities of the implementation.

This report presents (1) a critical analysis of a previously presented DRM architecture, (2) a refined DRM architecture that handles the main issues that were identified in this analysis, (3) a detailed description of the proposed architecture from four perspectives: content handling, consumer tracking, licensing, and content securing, (4) detailed interfaces of all identified components, (5) scenarios to illustrate usage of the architecture, and (6) an evaluation of the proposed architecture based on nine major quality attributes. The architecture is compared with related work in two DRM projects: Digital Media Project (DMP) and Automating Production of Cross Media Content for Multi-channel Distribution (AXMEDIS). To the best of our knowledge, the proposed architecture is more sophisticated than related work published so far.

1 Introduction

The domain of Digital Rights Management (DRM) is currently lacking a generic architecture that supports interoperability between and reuse of specific DRM technologies. This lack of architectural support is a serious drawback in light of the rapid evolution of a complex domain like DRM. It is highly unlikely that a single DRM technology or standard will be able to support the diversity of devices, users, platforms, and media, or the wide variety of system requirements concerning security, flexibility, and efficiency.

In a previous report[14] and paper[15, 13], we introduced a general layered architecture. This report extends this architecture by solving known issues and adding new features. Section 2 gives a brief description of our original generic layered architecture. Sections 3, 4 and 5 refine this architecture. Section 6 validates our architecture by comparing it to AXMEDIS and DMP, two recent DRM architectures. Section 7 elaborates open issues for future work. The last section 8 concludes our work.

2 Starting point: a generic layered DRM architecture

Systems that provide DRM are highly complex and extensive [8]: DRM technologies must support a diversity of devices, users, platforms, and media, and a wide variety of system requirements concerning security, flexibility, and manageability. This complexity and extensiveness poses three major challenges to DRM development: fragmentation of individual solutions, limited reuse and interoperability of DRM systems, and lack of a domain-specific structure that supports and guides the design and implementation of DRM systems and their applications.

Previous research[14, 15, 13] has tackled these problems by presenting a generic layered DRM architecture. The remainder of this section gives a brief overview of this architecture.

2.1 Architecture

This section integrates the three main roles identified in a DRM system (consumer, producer, and publisher), the seven most important DRM services (content, license, access, tracking, payment, import, and identification), and the various cryptographic primitives to implement security services (a.o. encryption, digital signatures, certificates, watermarks, and secure clocks). Identifying key DRM services and locating them in an overall structure brings us one step closer to a software architecture for DRM.

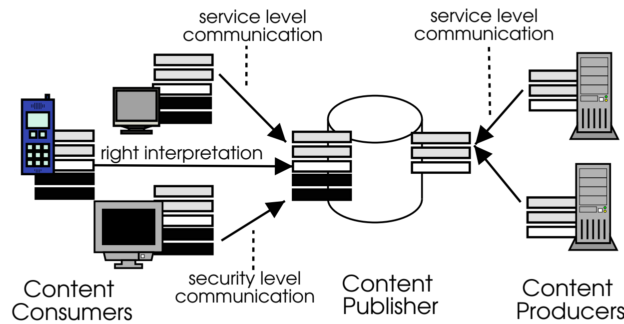


Figure 1: A distributed view on a DRM architecture with content consumers, producers and a publisher. The figure shows different levels of communication between layered architectures in each party. The gray levels represent application services (e.g. payment, tracking, or licensing). The white level represents right expression. The black levels represent rights enforcement technologies (e.g. watermarking, certificates, or digital signatures). Each party can provide a different subset of layers.

The section starts by defining the content consumer, producer, and publisher and positioning them in a high level distributed view on the DRM architecture. Subsequently, it identifies the layers for each party of the distributed view. After this high level overview, the section zooms in on the top layers of the architecture, which represent application services, and the bottom layers, providing security and rights enforcement technologies.

2.1.1 Users

What is needed at application level to have a complete DRM system? In order to answer this question, three main roles are distinguished: the consumer, the producer and the publisher. In addition, some external roles are briefly discussed. Identifying multiple roles in a DRM system is crucial to view the complete picture.

Consumer. Consumers want to consume protected content in a user-friendly way. They want to be able to browse the content catalog of the on-line DRM system where the content at stake can be obtained. Since consumers also need a license, they must be able to select a license type and view the usage rules associated with it. Generally, consumers first have to pay, one way or another; different business models should be possible (e.g. subscription, pay-per-license, or pay-per-use). When time-based licenses expire, it must be possible to update them, which may also require some financial transaction. Consumers also want to browse their obtained licenses locally and view the usage rules in a human readable format. Finally, consumers want to consume the protected content, according to

the usage rules associated with the corresponding license. In order to fetch licenses (and sometimes also protected content), consumers need to authenticate to the on-line DRM system.

Producer. Producers want to easily compose a contract. Both content and contract must be submitted to the on-line DRM system. After some time, they may want to update the contract or maybe even cancel it, i.e. stop the distribution of the content. Content producers expect a financial compensation from the DRM service for the trade of their content. Therefore, they want to receive statistical information from the DRM service about the number of downloads or content usage patterns. In order to query or submit content to the on-line DRM system, content producers need to authenticate themselves.

Publisher. When one or more DRM clients are no longer secure, their right to consume content must be revoked. It may also be necessary to update some parts of the DRM system (and the DRM client). Content publishers may want an overview of system usage patterns. When content is found mass-distributed, the source of abuse must be identifiable. Publishers need to authenticate to the DRM system first.

External roles. Some other roles include the financial institution that offers support for billing issues, and the owner of the network that is used to distribute content and licenses. When certificates are used, an external Certification Authority (CA) is needed to obtain certificates and to check the validity of certificates

2.1.2 A distributed view

Figure 1 illustrates three aspects of DRM. First of all, it gives a high-level overview of the distributed DRM architecture, with content consumers, producers and a publisher.

Secondly, the figure shows parties interacting at different levels, which clearly matches a layered architecture. Content consumers, for instance, communicate with the publisher at both service and security level. Service level communication relates to issues such as licensing, tracking, importing, or paying. Security level communication enforces protection of published content by using techniques such as watermarking, encryption, or certificates. In addition to multi-level communication, the figure illustrates that not every level is relevant for each party. Content producers, for instance, interact with the publishing system by exclusively using service level communication, while consumers communicate at both the service and the security level.

Finally, the figure highlights a client-server interaction pattern between consumer and publisher, and producer and publisher. This implies that both sides must provide (part of) the interaction protocol or technology in use.

In summary, the figure intuitively motivates why the three parties should pro-

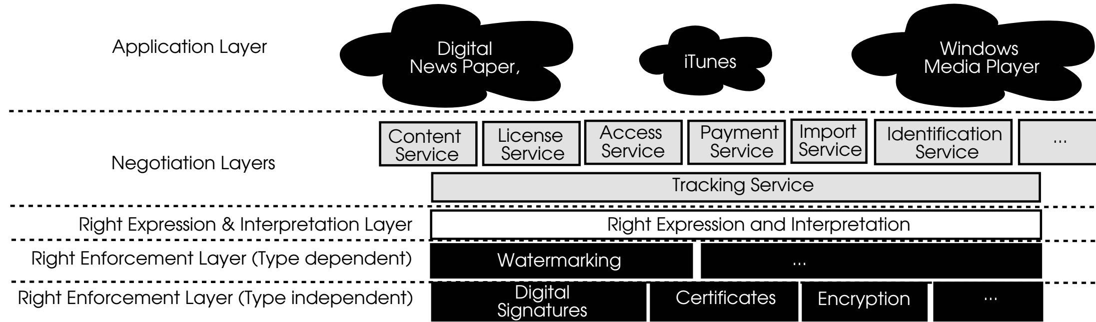


Figure 2: DRM as a layered architecture identifying key services at the negotiation layer, such as content, license, access, payment, import, identification, and tracking. Security technologies, such as watermarking, digital signatures, certificates, and encryption, are located at the rights enforcement layers.

vide a layered DRM architecture, while not every party has to offer each level of communication.

2.1.3 Identifying layers

Based on the architecture presented by Jamkhedkar and Heileman [11], we identified five layers, from top to bottom: the application layer, the negotiations layer, the rights interpretation layer, the upper rights enforcement layer (REL), and the lower REL (see also figure 2 on page 10).

We classified these layers into three main groups: the service level (shown in gray), the right interpretation level (in white), and the security level (in black). At the upper two services layers, content is used and manipulated according to the rights associated with it. In the middle, the rights interpretation layer is concerned with how rights are specified and how they should be interpreted in particular environments. The lowest layers are responsible for rights enforcement on the consumer side.

The following sections explores the security and service layer.

2.1.4 Service layer

Four key DRM services with respect to the content consumer are identified: the *Content Service*, the *License Service*, the *Access Service*, and the *Tracking Service*. In addition, we describe two external services for payment and certification.

Content Service The *Content Service* is a component which manages the obtaining, protecting and submitting of content. The consumers contact the *Content*

Service for obtaining (including searching) content. The *Content Service* possibly protects content before sending it to the consumer.

The *Content Service* may need to send protection data, specific to that piece of content, to the *License Service*, to allow this service to issue associated licenses. The *License Service* possibly asks the *Content Service* to give the protection data.

The *Content Service* receives its content and potentially some additional data from the producer. The producer can also remove content from the *Content Service*.

License Service The *License Service* is able to issue corresponding licenses upon consumer request, once it has received the protection data from the *Content Service*. The *License Service* may offer different license types corresponding to the same content. The consumer requires a description of the different types and selects one. The consumer's DRM client provides some system or user specific parameters to generate a license that is only usable by that consumer. Before the license is issued, the consumer usually has to pay for it. Therefore, the *License Service* asks the *Access Service* if it may issue the license. The *Access Service* in turn contacts the external *Payment Service* and will only answer positively if the consumer has effectively paid. Expired licenses can be updated by sending them to the *License Service*, which will issue new ones to the consumer, often after a new payment. The *License Service* may also send statistical information about the license issuance to the *Tracking Service*. The *License Service* receives its contracts (licenses) and potentially some additional data from the *Import Service*, which is explained in the content producer part.

Access Service The *Access Service* is responsible for the authentication of the content producers, consumers (or its DRM clients) and publisher. It is also responsible for checking payments of both consumers and producers before allowing particular actions on the DRM system. The *Access Service* may, for example, deny access if some bills are not paid or if consumers fail to identify themselves. The *Access Service* also allows the publisher to revoke a single DRM client or a whole class of clients, for example all DRM clients that have not yet been renewed after a security threat.

Before content consumers are able to obtain licenses or even content, some registration procedure may be necessary beforehand. This procedure usually involves the *Access Service* and, if some financial transaction is needed, the *External Payment Service*.

Tracking Service The DRM Client, the *License Service* and the *Content Service* can generate statistical usage information (e.g. the number of times a piece

of content is downloaded, the type of licenses issued for it, or the number of times it is consumed). These components (or a subset) can forward such information to the *Tracking Service*, which in turn can produce usage statistics. The publisher and the producer have the possibility to request statistics.

Others The publisher can send content that is found on mass-distribution networks to the *Identification Service*, which will reveal the identity of the source of abuse.

The financial aspects are taken care of by an external *Payment Service*, typically a bank or another financial institution. Obviously, some interaction is needed between the external Payment Service, the Access Service and the DRM client, producer tool or consumer tool.

Certification Authority (external) is only necessary if certificates are used. The CA is responsible for the issuance, distribution and revocation of certificates.

2.1.5 Negotiation layer

The diversity of services is located at the negotiation layer. This layer provides a selection of services to satisfy the varying needs of different applications [11]. Figure 2 shows the main services that have been discussed in detail in the previous section. The *Tracking Service* can be seen as a sub-layer that intercepts and logs relevant communication of upper services.

2.1.6 Rights enforcement layers

The security technologies reside at the two bottom layers of the architecture. The upper layer is responsible for type dependent manipulations, while the lower layer handles content irrespectively of its type. Figure 2 shows rights enforcement technologies such as watermarking, digital signatures, certificates, and encryption.

2.2 Issues

The previously defined architecture suffers from the following key problems: (i) content tracking is not transparent, (ii) the architecture cannot be layered since content does not always travel sequentially through the stack of DRM layers, and (iii) generic content publishing components are not separated from the DRM architecture.

2.2.1 Tracking content transparently

The original architecture supports content tracking in an ad-hoc manner: it provides a *Tracking Service* component that must be called explicitly by each other

service that needs logging. Since tracking is so crucial for DRM systems and since it is a non-functional service, it should be transparent for functional services. In this way, tracking behavior can easily be changed independently from the components making use of it.

2.2.2 Traversing content non-sequentially

The relationship between the layers sketched in the architecture is not top-down but rather cyclic for the following reason. At the publisher side, content is sent by the service layer through the RI layer and the security layer, while at the consumer side, the reverse progress takes place. However, content may consist of multiple parts and each part can have a different license associated with it, which can result in different ways of protection. Think, for instance, of a video stream with subtitles in multiple languages and extra bonus tracks that can only be played after a certain date. The representation layer knows how composed content is represented and thus how these parts are glued together and protected. The security layer does only know how to protect a particular content part, not the composed content. Therefore, the representation layer must consult the security layer several times (for each part of the content) in order to construct composed content.

2.2.3 Separating publishing components

It is impossible to embed the DRM architecture in another than a publishing system, because the architecture does not make a distinction between DRM components and publishing components. A DRM system is typically not stand-alone but embedded within, for instance, a publishing system. In this way, it can make use of the main services of a typical publishing system, i.e. the *Content Management System (CMS)*, the *User Management System (UMS)*, and the *Input Management System (IMS)*. Typically, the publishing side of a DRM architecture receives content via the *IMS*, stores it in the *Content Management System*, and possibly personalizes it by using user information obtained from the *User Management System*. The needed context components depend on the wishes of the the publisher.

2.3 Towards a refined software architecture

The remainder of this report refines the previously described architecture. Section 3 addresses the issue of separating publishing components from the architecture. Section 4 identifies three coherent collections of components and handles two other issues by the use of architectural patterns: the need for transparent consumer

tracking and non-sequential content traversal through the system, are addressed. Section 5 explains interactions between the three component collections.

3 Publishing context

A DRM system can be deployed in the context of some kind of publishing context in which three main features are typically offered: content management, user management, and input management. A *Content Management System* (*CMS*) is responsible for storing and retrieving content items, a *User Management System* (*UMS*) takes care of authenticating, authorizing, and accounting users who are using the services of the publishing system, and an *Input Management System* (*IMS*) is used to (automatically) annotate and prepare produced content to be stored in the *CMS* so that it is ready to be published.

The main roles involved in a publishing system, and thus also in DRM, are the producer, the consumer and the publisher. The *producer* is the entity that produces content, or at least owns the rights to distribute and sell it, and represents the starting point of a DRM chain. This can be, for instance, an author, a musician, or a record label (such as EMI, Sony and AOL Time Warner). At the other end of the DRM chain, the *consumer* is the entity who wants to obtain and consume content. For example, a home user who wants to download and play music. The *publisher* is the entity that owns and manages the DRM system used to distribute content, and forms the bridge between consumers and producers. Most DRM technologies available today focus on the consumer aspect.

A DRM system is typically not a stand-alone application but is embedded within a publishing system. In this way, it can make use of the *CMS*, the *UMS*, and the *IMS* for protecting content or authenticating users. Typically, the publishing side of a DRM architecture obtains content from the *CMS*, possibly personalizes it by using user information obtained from the *UMS*, and finally protects it. The needed context components depend on the wishes of the the publisher. Table 1 summarizes the relationship between the needs of the publisher, the needed context components and their influence on the DRM architecture. For instance, a *UMS* is not needed if the publisher offers anonymous content obtaining. If a publisher wants to know which customer mass-distributed its content, the DRM system needs to interweave personal user information with the content. Therefore, it obtains the malicious consumer's identity from an *Access Service* (Authentication System) and extra user information, e.g. a phone-number, from a *UMS*. In the remainder of this report, we assume the presence of the publishing components.

Goal	Needed Context
Non-anonymous consumption	Authentication System
Personalized Content	UMS Authentication System
Mass Distribution Detection	UMS Authentication System

Table 1: What context components does a DRM-system need in order to achieve a certain goal.

4 High level architectural overview

This section revises the architecture presented in section 2. It presents a high level overview of the proposed DRM architecture. It identifies three coherent collections of components and explains key interactions between them. Two issues, the need for transparent consumer tracking and non-sequential content traversal through the system, are addressed by the use of architectural patterns.

4.1 Component groups

A component group represents a coherent collection of components that offers a key functional aspect of DRM. We identify three component groups that represent (1) DRM services, (2) content representation and interpretation (RI), and (3) security algorithms. Figure 3 on page 16 illustrates key relationships between component groups at consumer, producer, and publisher side. Since a DRM architecture is embedded within a publishing context, functionalities related to user, content, or input management can sometimes be reused.[1]

- **Service component group (DRM and Context):** This component group offers the main services of a DRM system: content handling, licensing, access control, and input management. Some of these services can be reused at the publisher side.
- **RI component group (DRM):** This component group is used to represent content, licenses and other entities, and to interpret policies, for instance for licensing. It contains the *Representation Component*, the *Logging Component*, and the *Interpretation Component*.

[1]Text between parentheses indicates to what extent the component group reuses components from the publishing context (Context) or not (DRM).

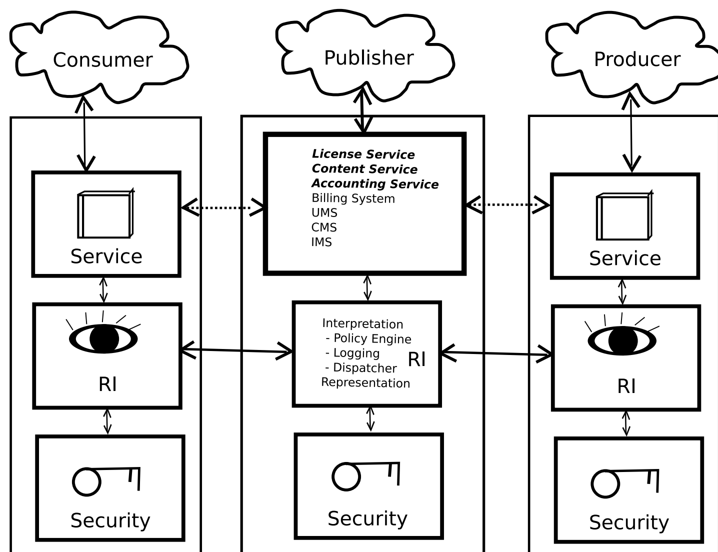


Figure 3: Architectural overview that illustrates three main roles: consumer, producer, and publisher. Each part of the architecture consists of three component groups. The service component contains the main services of the publishing context. The RI component group is used to represent content and licenses, and to interpret policies. It acts as a dispatcher that delegates requests according to a policy. The security component group provides various security techniques.

- **Security component group (DRM):** This component group provides various security algorithms to protect content against abuse, by encrypting, signing, or watermarking. These components are used by the RI component group, for instance, to enforce a licensing policy.

In order to allow consumer tracking at publisher side, consumer requests should be intercepted before they are handled by the service component group. Yet, in order to offer tracking at consumer side, user requests should be intercepted before reaching the security component group. For example, if a publisher is interested in the number of consumers that has requested content of a particular producer, it should[2]: (i) intercept consumer requests intended for the *Content Service*, (ii) interpret these requests to find out which content has been requested, (iii) log each request for content of that particular producer, and (iv) forward each request to the *Content Service*.

These component groups cannot be structured as layers with a sequential interaction flow for the following reason. At the publisher side, content is sent by the service component group, through the RI group and the security component group, while at the consumer side, the reverse progress takes place. However, content may consist of multiple parts, and each part can have a different license associated with it which can result in different ways of protection. Think for instance of a video stream with subtitles in multiple languages and extra bonus tracks that can only be played after a certain date. The RI component group knows how composed content is represented and, by consequence, how these parts are glued together and protected. The security component group, however, cannot distinguish between various content items with different security requirements. Since content cannot always be processed in one pass through the security component group, a cyclic processing scheme seems more natural than a layered (sequential) scheme.

We identified three architectural styles[3], which solves the two aforementioned issues – the need for transparent consumer tracking and non-sequential content traversal through the system. An architectural style expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

1. **Exposed broker:** The exposed broker is a slight variation of the exposed router.
2. **Exposed router/dispatcher:** Source components initiates an interaction that is forwarded to, at most, one of multiple target components.
3. **Blackboard:** The blackboard pattern uses a central repository.

[2]We assume that the *Content Service* successfully serves the request.

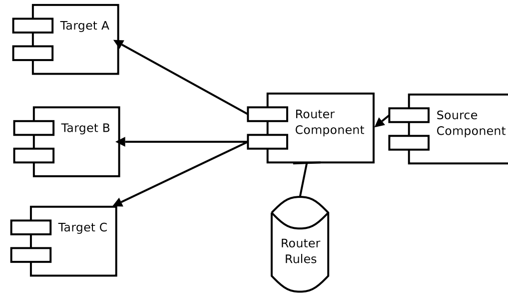


Figure 4: Exposed router pattern. Source components initiates an interaction that is forwarded to, at most, one of multiple target components. Selection of the target component is controlled by distribution rules that govern functioning of the connector component.

4.1.1 Exposed router/dispatcher

The router pattern applies to solutions where the source components initiate an interaction that is forwarded to, at most, one of multiple target components. The selection of the target component is controlled by the distribution rules.

This application pattern, as shown in figure 4 on page 18, is divided into a number of logical components:

1. *Source Components* (*Source*) represent one or more applications that are interested in interacting with target applications.

2. *Router Rules* represent any rules associated with message handling, such as routing, transformation, and logging.

3. *Router* receives requests from multiple source components and routes them to the appropriate target components.

4. *Target Components* (*Target A*, *Target B*, *Target C*) are responsible for implementing (business) services.

Benefits of this application pattern are (i) that it intercepts communication between different component groups, (ii) that it allows integration of multiple, diverse component groups, (iii) that it provides transformation services that allow source and target to use different communication protocols, (iv) that it minimizes impact on existing services, (iv) and that it minimizes impact of changes in location of component groups.

Drawbacks of this pattern are (i) that it has a limited ability to manipulate requests, (ii) that it does not have the ability to send simultaneous requests to the

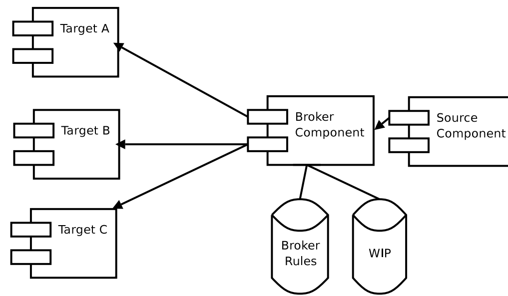


Figure 5: The exposed broker is a slight variation of the exposed router pattern. The broker also provides decomposition and recomposition of messages.

target components based on incoming requests, (iii) and that it does not have the ability to do intensive manipulations of messages.

4.1.2 Exposed broker

The exposed broker is a slight variation of the exposed router pattern described in the previous section.

This application pattern, as shown in figure 5 on page 19 is divided into a number of logical components:

1. *Source Component*s represents one or more components that are interested in interacting with *Target Component*s.

2. The *Broker Rules Component* reduce the proliferation of direct connections. In addition, it supports message routing, decomposition and recomposition, message enhancement and transformation. These rules are often captured as business rules. This component also uses a work-in-progress data store to retain the intermediate results from responses coming back from target components until all the necessary responses are received.

3. *Target Component*s are responsible for implementing services.

Benefits of this pattern are comparable to those of the router pattern, including an additional advantage: The *Broker* provides decomposition and recomposition of messages, allowing one request from the source to be satisfied using multiple target applications. The fact that the response is a composite of multiple requests and responses is hidden from the source application.

The major drawback is that the router is a bottleneck, if it handles a lot of messages.

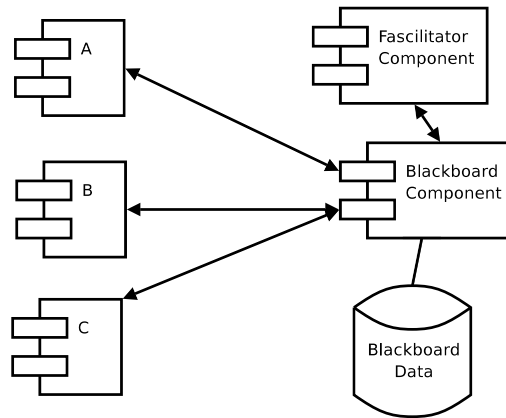


Figure 6: The blackboard pattern has a central repository, *Blackboard Component*, for all information.

4.1.3 Blackboard

The blackboard architectural pattern is useful for problems for which no deterministic solution strategies are known. It is used as a central repository for all information, which represents data, facts, assumptions, and deductions made by the system.

This application pattern, as shown in figure 6 on page 20 is divided into a number of logical components:

1. *Knowledge Sources* (A,B,C): components with application dependent knowledge. Interactions through *Knowledge Sources* take place through the *Blackboard Component.*

2. *Blackboard Component* data structure is used to put information on it. *Knowledge sources* make changes to information, that will lead to a solution.

3. *Control (facilitator) Component* controls the chalk, mediating among experts competing to write on the *Blackboard Component.* The *Facilitator Component*'s function is to determine which *Knowledge Source*, at a given point it time, has the most important insight or information to contribute to the problem's solution. This component can be in the *Blackboard Component*, in the *Knowledge Sources*, a separate component of a combination of these.

Our architecture requires a slight modification of this pattern. Information on the *Blackboard Component* should not be visible to all sources. The RI component

group, *Facilitator Component*, controls who has access to the *Blackboard Component* (Interception Points). The *Tracking Component* simply reads everything what is on the *Blackboard Component*.

Benefits of this pattern are (i) that it allows integration of multiple, diverse component groups, (ii) and that it minimizes the impact to existing services.

4.1.4 Solution

The solutions described above are compared by tabulating (see Table 2) the following design considerations[3]:

- **Adding extra components**: how easily can we add extra component groups?
- **Change in data representation**: how easily can we change representation of messages between component groups?
- **Reuse**: how easily cam we reuse component groups?
- **Changing interaction between components**: how easily can we change interactions between component groups?
- **Granularity**: is our solution (too) fine-grained or not?
- **Performance**: is our solution performant enough? Consumers does not want to spend a long time waiting before consuming.

The blackboard is weak in changes for the overall processing algorithm, reuse and representation, because all experts communicate with the *Facilitator Component* and the *Blackboard Component* using the same communication representation. However, its performance is relatively good, because all components have access to 'raw' data. New components can also be relatively easily added, because they use the *Blackboard Component*.

Data and message representations of different components in router and broker patterns can be easily changed, as long as their key components (*Broker Component* and *Router Component*) support all data representations. Performance is negatively affected, if a lot of components use different representations. New components can be easily added, because they just have to register to the *Router*

[3]These design considerations are aspects of quality scenario's (see section 6). Change in data representation and change in interaction are modifiability aspects. Adding extra components is extensability. Minimizing granularity aids usability and reuse aids interoperability. Performance is a quality attribute on its own.

	Router	Broker	Blackboard
Change in data representation	+	+	-
Change in processing algorithm	-	-	-
Adding functions	+	+	+
Performance	-	-	+
Reuse	+	+	-
Leakage of information	+	+	-

Table 2: Comparison between the different architectural patterns

Component. Reuse of each individual component group is possible, however interactions between them will change.

The exposed broker is valid for the DRM client, the online DRM system, and the producer tool. It allows modification, change in data representation, adding functions and reuse. Performance depends on the usage of different representations and processing power of the *Broker Component/Router Component*.

Each component group sends messages to the RI component group (the *Broker Component*), which interprets and manipulates the messages and subsequently routes them to the right component group.

5 Detailed architecture

After having sketched the overall structure of component groups at consumer, producer, and publisher side, we zoom in on the details of each component group. Since the architecture is quite extended, we follow an incremental approach in presenting it. Figure 21 on page 51 shows the detailed architecture.

This section will present the architecture from four perspectives: content handling, license issuing, consumer tracking, and content securing. After a short introduction of these perspectives, some scenarios related to content obtaining and consuming are used to explain key aspects of each perspective and to demonstrate how the architecture can be used.

5.1 Architecture perspectives

The content handling perspective focuses on core components of a DRM system, the licensing perspective shows how licensing is added, the consumer tracking perspective shows how the architecture can be extended with logging functionality,

and the security perspective closes the content publishing loop by illustrating how content can be secured by various techniques.

- **The content handling perspective** involves all DRM services related to content manipulation. By consequence, this perspective focuses on core DRM services and the integration with the *CMS*, the *UMS*, and the *IMS* of the embedding publishing system.
- **The tracking perspective** extends the core perspective and deals with extracting and accounting consumer activities.
- **The licensing perspective** extends the previous perspective with licensing.
- **The security perspective**, finally, focuses on the integration of various security algorithms and techniques that are used for enforcing licenses and protecting content.

Each perspective can be seen as a filter that focuses on a particular level of functionality. The lowest level introduces the core functionality that is needed for distributing content. The other levels extend this functionality with tracking and security. The combination is a full-blown architecture which supports creation, distribution and consumption of DRM-protected content, distribution of licenses, and content tracking.

5.2 Content handling perspective

This perspective presents the key architectural elements needed to provide the core business of the DRM publisher. This core business is translated into a core service, which is more specifically the service of providing content to consumers, and content management. This perspective identifies the needed context for a DRM system. Billing is not worked out, because most publishing systems already offer support for billing.

5.2.1 Component diagram

The architecture component diagram for the core functionalities from publisher perspective can be found in figure 7.

5.2.2 Service component group

To get a better understanding of each of the internal components, we give a brief overview of each of them in which we explain their role and responsibilities within

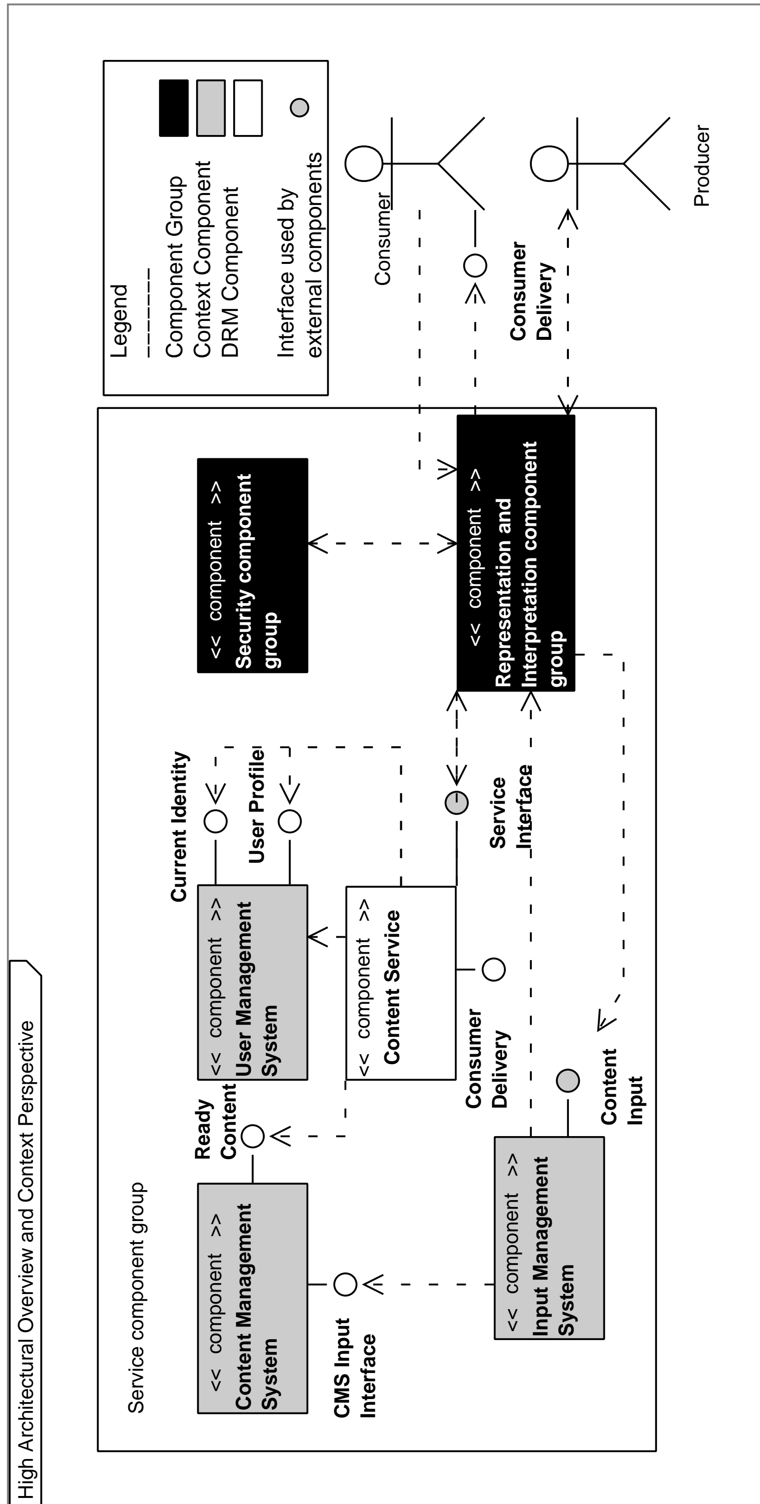


Figure 7: Architecture component diagram of the content handling perspective from publisher side.

the architecture, and give a high level description of their interfaces. Completely elaborated interfaces can be found in appendices B, C and D.

In a previous report[14] we have identified seven key DRM service components: *License Service*, *Access Service*, *Content Service*, *Identification Service*, *Import Service*, *Tracking Service*, and *Payment Service*. However, DRM developers should not reinvent the wheel, but make use of existing components instead. For instance, a publishing architecture typically provides a *Content Service*, a *CMS*, a *UMS* (*Access Service*), an *IMS* and a *Payment Service*.

Content Service (DRM and Context) The *Content Service* offers the publisher's content to the consumer and is thus its core service. The publisher can offer a centralized *CMS* for obtaining content. However, the *Content Service* can use any other system to retrieve content, and content can be pushed by any system to the *Content Service*.

Content can be pushed to or pulled by consumers. For instance, a new version of a newsreport can automatically be sent to consumers, while electronic books can be downloaded. The *Content Service* can use the *Representation Component* if it (automatically) wants to convert content to a representation more suitable for its customers. For instance, the *Content Service* can convert content if the DRM client only supports MPEG-21, while the *Content Management System* only offers MPEG-2. This conversion can be done in two ways: (i) by explicitly invoking the *Representation Component*, or (ii) transparently by the *Representation Component*.

In case of explicitly invoking the *Representation Component*, the *Content Service* sends content to the RI component group, which interprets and changes its representation depending on a policy. The major advantages are (i) that each *Content Service* can have its own DRM policy and (ii) that protecting content can be boarded out. After all, another company can take care of the *Representation Component*. The key drawback is that the *Content Service* should know the location of the RI component group.

In case of transparent conversion ,the RI component group intercepts the content, converts it and sends the converted content to the DRM client. The major advantage is that DRM can be integrated seamlessly with existing publishing systems (context).

The *Content Service* component offers the following interfaces:

- **Consumer Delivery:** This interface is used to trigger the release of content. It is a notification that certain content was added. In a push-case, the *Content Service* fetches content and send it to all consumers that are subscribed to this *Content Service*.

- **Service Interface:** This interface is offered to the consumer for browsing through content, obtaining content, etc.

Identification Service (DRM) The *Identification Service* identifies users, content, etc. For instance, consumers who mass-distribute content can be identified. This component offers one interface:

- **Identification**: Content, e.g. found on mass distribution networks, is passed via this interface.

Content Management System (Context) All content, such as books, music, movies are stored and organized centrally in the *Content Management System*. This system offers content to the *Content Service*, which makes it available to consumers.

This component offers the following interfaces:

- **CMS Input Interface:** New content is passed to the *Content Management System* via this interface.

- **Ready Content:** Content is retrieved by the DRM *Content Service* via this interface.

- **Content In Progress:** Content is manipulated by the publisher via this interface. For instance, he indicates that content is ready for publication.

This contextual component is needed, because the DRM *Content Service* has to obtain its content from somewhere (in case the publisher does not give content to the *Content Service*). Therefore, it uses the **Ready Content Interface**.

User Management System (Context) The *User Management System* is a component that handles all information about all publishing system users, which are consumers, publishers and producers. This information contains amongst others contact information, such as name and address. This component also knows users who currently make use of the system. It offers the following interfaces:

- **User Profile:** This interface is used for reading and changing the profile of a user.

- **Current Identity:** This interface is used for obtaining the identity of the user that currently makes use of the system.

Input Management System (Context) The *Input Management System* (*Import Service*) is a publishing component that handles incoming content, such as books, music, and videos, that is uploaded to the system by producers. After initial processing, the publisher is notified and the content is submitted to the *Content Management System*.

This component offers the following interfaces:

- **Content Input:** This interface is used by producers to send in their content.

This contextual component is used to illustrate the *Representation Component*, which is used to convert representations.

Payment Service (Context) The *Payment Service* (Billing System) is a publishing component that keeps information about all payment methods and all *External Financial Institutions*. It also keeps track of pending payments: these are payments which are requested from the *External Financial Institution*, but not yet confirmed. It is also responsible forwarding payment notifications from *External Financial Institution*s to interested parties. It offers the following interfaces:

- **Billing Interface:** This interface is used to request payments from users. As discussed before, payment can be synchronous or asynchronous. The *Payment Service* internally knows which *External Financial Institution* to address in order to finish the payment correctly.

- **Payment Notification:** This interface is used by the *External Financial Institution*s to notify the publisher when a payment transaction has been completed successfully.

5.2.3 RI component group

Representation Component (DRM) **RI component group.** The RI component group contains a *Representation Component*, which is responsible for creating content in a certain representation and for converting content to another representation. This component can be used, for instance, to convert a representation from MPEG-2 to MPEG-21.

This component offers the following interfaces:

- **Create Representation:** This interface is used to create content in a certain representation.

- **Convert Representation:** This interface is used to convert content from one representation to another.

5.2.4 External components

In what follows an overview is given of each of the external components, with their interfaces. These external components can be seen as devices and/or software that enable physical users to interact with the publishing system.

DRM client The DRM client is a consumer's consuming device which allows him to fetch content from a publisher's *Content Service* and consume this content afterwards.

This component offers the following interfaces:

- **Consumer Delivery:** This interface is used by the system to perform a push of content to DRM clients.

Producer tool The producer tool is a producer's device which allows him to submit new content to the publishing system. More concretely, this tool interacts directly with the *Input Management System*. The interfaces are not worked out, because they are out of scope for this report.

Publisher tool The publisher tool is a publisher's device which allows him to manipulate content. It has the following interfaces:

- **Publisher Notification:** This interface is used by the publishing system to notify the publisher about new content.

5.2.5 Scenario

This section translates the most relevant use cases to a component architecture diagram. The use cases are classified based on the main stakeholders involved: consumers, producers and publishers. All use cases are worked out in full detail in appendix A.

Consumer We explain two variants of the pull content use case. The first variant sends content in an unmodified version to consumers DRM client. The second variant adapts content representation in order to support different consumer devices with different capabilities. For instance, content representation can be changed from ODT to PDF.

Assumption Content can be pulled anonymously or non-anonymously. In the latter case, we assume that the publishing system handles authentication. The *Access Service* can be used to ask the identity of a user who currently requests content, and the *UMS* can offer more information about this user, which allows personalized content.

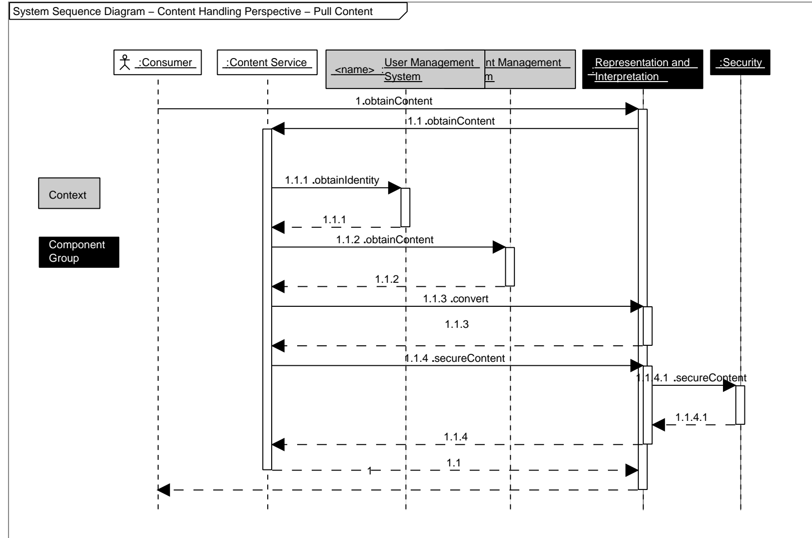


Figure 8: Sequence Diagram of the obtain content use case. The consumer requests content from the *Content Service*. This service obtains content from the *CMS* and sends it to the RI component group, which will take care of protection.

Pull Content To obtain content (see figure 8 on page 29), a DRM client first contacts the *Content Service* (1.1), via the RI component group (1). If the publisher wants non-anonymously content consumption, the *Content Service* obtains the identity of this consumer by contacting the *UMS* (1.1.1). The *Content Service* then fetches the requested content from the *CMS* (1.1.2), and possibly converts it (1.1.3) by sending the content to the *Representation Component*. Finally, the *Content Service* calls the RI component group to protect the content (1.1.4), which makes the content ready to be delivered to the DRM client. Each of the inter-group messages passes the RI component group.

Push Content To push content towards the consumer, the *Content Service* receives content from any other component, e.g. a planning system (1). Next, steps 1.1.1, 1.1.3 and 1.1.4 of the previous use case are executed.

Producer Input can be automatically converted by the *Representation component*, which is a subpart of the RI component group.

Submit Content To submit content (see figure 9 on page 30), the producer contacts the *Input Management System* (1.1) via the *RI component group* (1). The content is send towards the *CMS* (1.1.1) via the *IMS* (1.1).

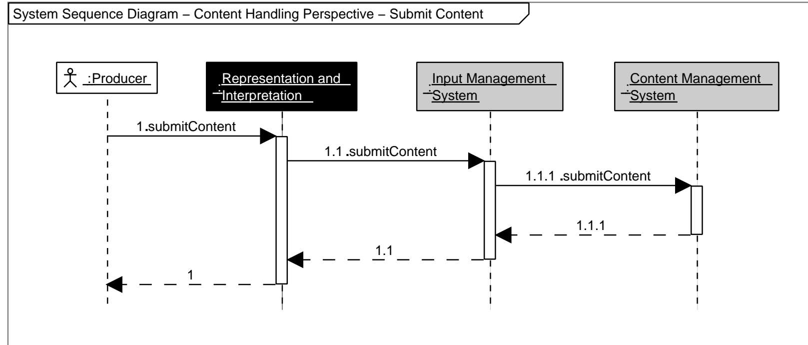


Figure 9: Sequence Diagram of the submit content use case. The producer submits content to the *Input Management System* via the RI component group. This system sends the content towards the *CMS*, which will take care of storage.

5.3 Tracking perspective

Most producers like to know how many times their content is consumed. However, the possibility of keeping track of the user's context highly depends on the technology that connects consumers to the system. For instance, the DRM client cannot submit usage statistics if it uses a one way broadcast connection.

Usage statistics can be collected at the publisher side or at the consumer side. The presented solution is similar for both sides. The RI component group logs the intercepted requests by sending them to an (Accounting Service), which stores the information. Which and when information is sent, depends on a policy located in the *Dispatcher Component*.

5.3.1 Component diagram

This section presents the architectural component diagram (see figure 10 on page 31) for the tracking perspective.

5.3.2 Service component group

Accounting Service (DRM/Context) The *Accounting Service* (*Tracking Service*) is slightly different than the one described in section 2.1 on page 7. Originally, each service collected and sent its logs to the *Accounting Service*. A more appropriate solution consists of placing a *Logging Component* in the RI component group, while processing these logs is done by the *Accounting Service*. This solution is more appropriate, because (i) the RI component group knows about every action that happens on content and (ii) there is a difference in functionality

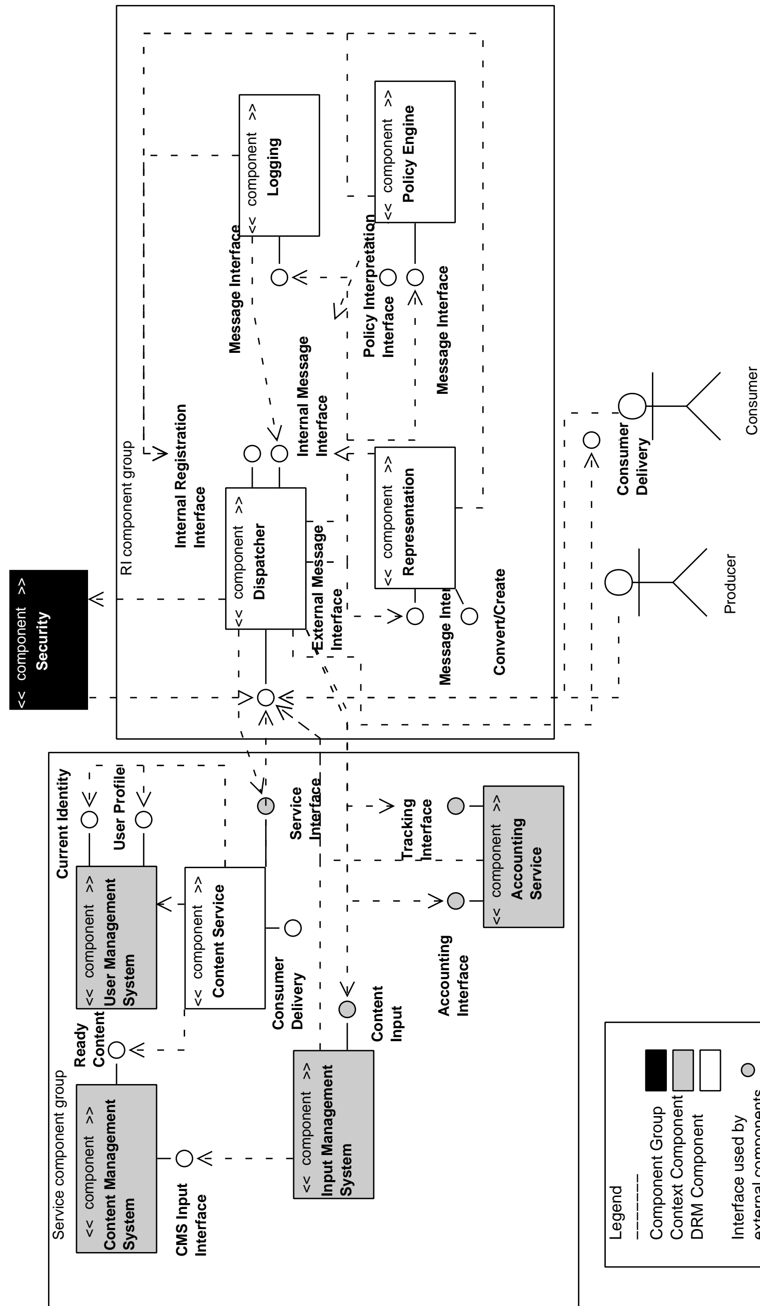


Figure 10: Architecture component diagram of the tracking perspective.

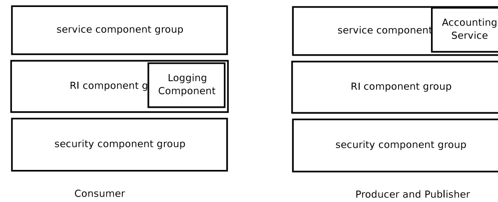


Figure 11: This figure shows tracking differences between consumers and producers or publishers. There is a functionality difference and a layering difference. Consumers are typically interested in logging, while producers and publishers are typically interested in interpreting these logs. The *Logging Component* is placed in the RI component group, because this group knows about every action on content, while the *Accounting Service* only processes those logs.

between DRM publishers and DRM consumers or DRM producers (see figure 11 on page 32).

The RI component group knows about every action that happens on content, because each service layer request (including content acquisition), passes through this layer. Therefore, logging should be placed at the RI component group, while processing of these logs is placed in the service component group.

There is a difference in functionality between DRM publishers and DRM consumers or DRM producers. A DRM consumer typically logs[4] the actions, while a DRM producer and DRM publisher typically interprets these actions.

The *Accounting Service* offers the following interfaces:

- **Tracking Interface** This interface is used by *Logging Component*s from the RI component group to submit statistical data.
- **Accounting Interface** This interface is used by other components to generate reports, obtain statistical data for billing, etc.

5.3.3 RI component group

This section elaborates the RI component group, which was described in section 4. This component group is responsible for:

- **Policy Interpretation**: online configuring of components and their interactions. This includes logging (tracking), licenses and policies.
- **Dispatching**: dispatching the messages to the right component groups.

[4]In case of a count based limitation, the client also interprets the logs, because it wants to know how many times the content has already been consumed.

- **Representation**: representing content and licenses: which parts of the content are secured with which algorithm? Where do we store the used data. Etc.

These functionalities can be implemented by using four components:

1. *Policy Engine Component*: interprets policies. At publisher side, this can contain e.g. what kind of requests should be logged.
2. *Dispatcher Component*: dispatches the message to the right component group.
3. *Logging Component*: logs and sends certain messages to the *Tracking Component*.
4. *Representation Component*: converts and transforms content and licenses.

These components can be organized by the usage of several architectural styles, including (i) broker (see section 4.1.2), (ii) pipe and filter (see section 5.3.4) and (iii) microkernel (see section 5.3.4).

Dispatcher Component This component dispatches messages to the right component group. It has the following interfaces:

- **Internal Message Interface**: this interface is used by components of the RI component group to send messages to each other.
- **External Message Interface**: this interface is used by other component groups to send messages to each other.

Logging component The *Logging Component* registers itself at the *Dispatcher Component*. By consequence it receives all messages it is interested in. Information is send to a *Tracking Service* or *Accounting Service* depending on the tracking policy. An advantage is that the *Tracking Service* and the *License Service* can be boarded out.

This component has the following interfaces:

- **Message Interface**: This interface is used to receive messages from the *Dispatcher Component*.

Policy Engine Component The *Policy Engine Component* interprets a policy. It has the following interfaces:

- **Policy Interpretation Interface**: this interface allows to add a policy, context and gives the result back.
- **Message Interface**: this interface is used to receive messages from the *Dispatcher Component*.

Representation component The *Representation Component* is responsible for converting and constructing different license and content representations. This is possible because the *Representation Component* has knowledge of how licenses and content are constructed. We assume that details of each representation are stored in a format understandable for this component.

This component sends repeatedly requests to the security component group to protect content, licenses, etc. Exact details of this process are postponed until the security section.

It has the following interfaces:

- **Representation Policy**: this interface allows the publisher to add, modify or delete available representations.
- **Convert/Create**: this interface is used to create a representation or convert a representation to another one (if possible).
- **Message Interface**: After this interface is used to receive messages from the *Dispatcher Component*.

5.3.4 Organizing the internal RI component group components

These internal components can be organized by the usage of various architectural styles, including (i) dispatcher (see section 4.1.2 on page 19), (ii) pipe and filter (see section 5.3.4 on page 37), (iii) microkernel (see section 5.3.4 on page 35) and (iv) blackboard (see section 4.1.3 on page 20), each with its own advantages and disadvantages.

Exposed router/dispatcher The existing *Dispatcher Component* takes care of the internal communication too, which is illustrated in figure 12 on page 35. Each internal RI component (*Policy Engine Component*, *Logging Component*, *Representation Component* and *Dispatcher Component*) registers itself with the *Dispatcher Component*. It informs the *Dispatcher Component* with the messages it is interested in. By consequence, the *Dispatcher Component* has a list of delivery rules. E.g. each message should pass the *Policy Engine Component* first.

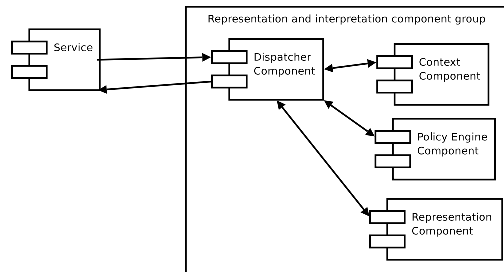


Figure 12: Application of the router architectural pattern.

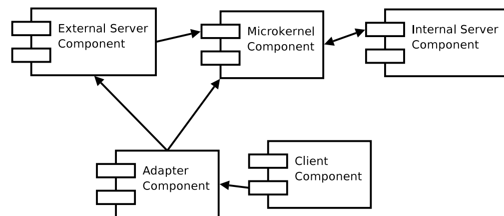


Figure 13: The microkernel architectural pattern separates a minimal functional core (*Microkernel Component*) from extended functionality (*Internal Server Component*) and customer-specific parts (*External Server Component*).

Main advantages are (i) flexibility, (ii) ability to accommodate future developments easily by just plugging in a new module, (iii) possibility to add context in different incremental steps and (iv) possibility to host each module by a third party.

Main disadvantages are (i) a single point of failure, and (ii) overhead.

Key issues are: (i) how do we deal with a blocked request?, (ii) how do we deal with different representations? (iii) how does the *Dispatcher Component* knows what context is needed? and (iv) how do we add new components.

The first issue can be solved at internal registration level. For instance, the *Policy Engine Component* says the *Dispatcher Component* to drop messages if he sends a drop-message. The second issue can be solved by sending messages to the *Representation Component* first. The third one can be solved by adding a *Context Component*, which gathers context for e.g. the *Policy Engine Component*.

Our architecture could benefit from this pattern, because (i) it allows introduction of new policies by usage of new message manipulating components, (ii) it minimizes impact on existing services, and (iii) it allows change of representations by usage of the *Representation Component*.

Microkernel The microkernel pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The *Microkernel Component* also serves as a socket for plugging in these extensions and coordinating their collaboration.

In order to make the system adaptable to changing requirements, the minimal functional core of the system *Microkernel Component* (See figure 13 on page 35) is separated from extended functionality, which can be plugged in the *Microkernel Component* through specific sockets.

The most important services of the system should be encapsulated in a *Microkernel Component*. The *Microkernel Component* maintains system resources and allows other components to interact with each other as well as to access functionality of the *Microkernel Component*. The size of the *Microkernel Component* should be kept to a minimum; therefore, only part of the core functionality can be included in it; the rest of the core functionality is deferred to separate *Internal Server Component*s.

External services provide a more complex functionality; they are built on top of core services provided by the *Microkernel Component*. Different *External Server Component*s may be needed in the system in order to provide functionality for specific application domains. These servers run in separate processes and employ communication facilities provided by the *Microkernel Component* to receive requests from their clients and to return the results. To make clients less dependent on the interfaces of the *External Server Component*s, *Adapter Components* may be implemented between clients and their *External Server Component*s.

When a client calls (through *Adapter Component*) a service of an *External Server Component*, the *Adapter Component* asks the *Microkernel Component* to establish a communication link with the required server and sends its request to this server (using remote procedure call). Upon completion of processing of the request, the server delivers the results to the *Adapter Component*, which in turn forwards them to the client. When an *External Server Component* needs a service provided by an *Internal Server Component*, it sends the request to the *Microkernel Component*, which in turn delivers the request to the *Internal Server Component*. Upon the completion of processing, the results are delivered, through the *Microkernel Component*, back to the External Server Component.

Our architecture can use a slight modification of this pattern: the *Adapter Component*s are left out. The *Microkernel Component* offers basic communication services. It has methods for invoking Internal Server Components (*Policy Engine Component*, *Context Component* and *Representation Component*), allocating resources and initializing communications. Each *Internal Server Component* indicates for which requests it is interested. For instance, the *Policy Engine*

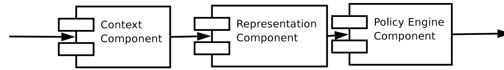


Figure 14: The pipe and filter architectural pattern is usefull for problems which have to process a stream of data, e.g. content. Each processing step is encapsulated in a *Filter Component*. Data is passed though *Pipe Component*s between adjacent *Filter Component*s.

Component indicates that it wants to handle all requests. The *External Server Component* (*Dispatcher Component*) provides the interface between clients and the *Microkernel Component*. It receives service requests from client applications using communication facilities provided by the *Microkernel Component*, forward it to process the request, and then returns the results back to the client.

Major benefits of this pattern are that it (i) is able to accommodate future developments easily by just plugging in a new module (*Internal Server Component*), (ii) offers a lot of flexibility by easy integration, evolution and enhancement of its overall functionality, (iii) separates low-level mechanisms (provided by *Microkernel Component* and *Internal Server Component*s) and higher-level policies (provided by *External Server Component*s), which improves maintainability and changeability of the system, (iv) scales easily with the number of machines in the network, when *Microkernel Component* is implemented on a network of machines, (v) achieves reliability, availability, and fault tolerance, when the *Microkernel Component* is distributed, (vi) encapsulates the location-related details of inter-component communication in the *Microkernel Component*.

Main drawbacks are (i) inferior performance, as compared with monolithic architectures, due to the need to communicate across several layers of indirection, (ii) complexity of design and implementation.

Our architecture could benefit from this pattern, because if offers a lot of flexibility.

Pipe and filter The pipe and filter architectural pattern (see figure 14 on page 37) is usefull for problems which have to process a stream of data, e.g. content. Each processing step is encapsulated in a *Filter Component*. Data is passed though *Pipe Component*s between adjacent *Filter Component*s. Recombining *Filter Component*s allows you to build families of related systems. It is important that *Filter Component*s do not know the identity of their adjacent *Filter Component*s.

Figure 14 shows the application of this pattern onto our architecture. The *Context Component* adds context to the message. The *Representation Component* transforms it to an understandable policy for the *Policy Engine Component*.

Major advantages of this pattern are that (i) intermediate files are not nec-

essary, (ii) filter exchange allows flexibility, (iii) recombination allows flexibility, (iv) filter components can be reused, (v) pipelines can be prototyped rapidly, (vi) parallel processing allows efficiency.

Main drawbacks of this patterns are that (i) sharing state information is expensive or inflexible, (ii) efficiency gain by parallel processing is often an illusion, (iii) it has data transformation overhead, (iv) error handling is difficult.

Blackboard The blackboard architectural pattern is already explained in section 4.1.3. This pattern can easily applied. The *Facilitator Component* is the *Policy Engine Component* and controls the access to the*Blackboard Component*. The *Dispatcher Component* and the *Logging Component* are source components. The *Dispatcher Component* puts the messages (the data) on the *Blackboard Component*, the *Logging Component* reads data.

This approach is comparable to the router approach. Instead of sending a message to the source components, the router sends the message directly to the right components.

A major disadvantage is that this approach introduces extra communication.

Solution Table 3 motivates our choice for the dispatcher architectural pattern. This table explains influence of the different patterns on relevant quality attributes (see section 6.1). The dispatcher and blackboard architectural pattern support interoperability, because they can automatically translate messages for different components by usage of the *Representation Component*. The dispatcher, microkernel and blackboard architectural patterns also enhance modifiability, because (i) new components can be easily plugged in (ii) and communication between components can be influenced by the blackboards *Facilitator Component* and the *Dispatcher Component*. The dispatcher and blackboard architectural pattern negatively affects performance, because they intercept each message.

The dispatcher architectural pattern and the blackboard approach perform equally well. We choose the dispatcher approach, but the blackboard approach could be chosen as well.

5.3.5 Scenario

Consumer Pull Content

The scenario described in section 5.2 is extended with logging steps, which can be done before (1.1) and after (1.3) the processing of requests.

Consume Content

This scenario focuses on content consumption at the consumer side. Content and license are supposed to be delivered to the DRM client in order to deprotect it.

Quality	Dispatcher	Microkernel	Pipe and filter	Blackboard
Interoperability	+	-	-	+
Modifiability	+	+	-	+
Integrability	+	+	-	+
Performance	-	-	+	-

Table 3: This table shows the relation between quality attributes and architectural patterns. This influences the design of our DRM architecture.

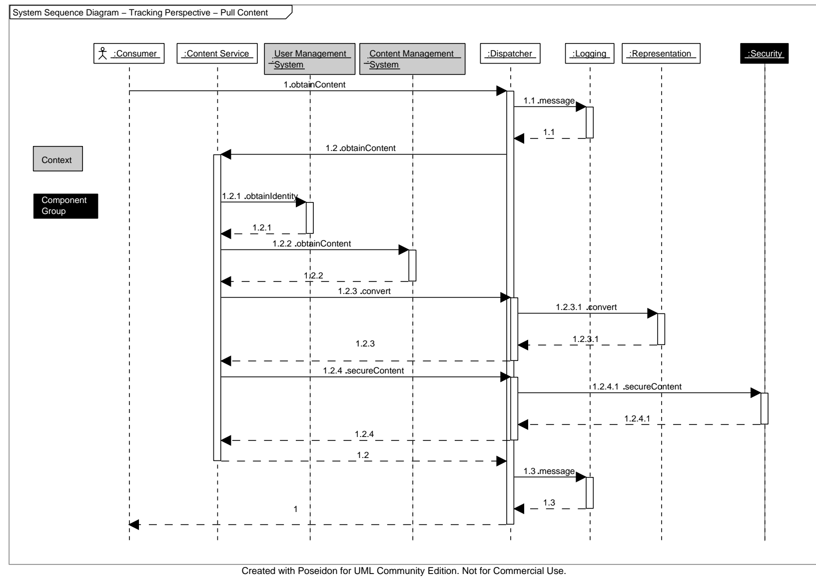


Figure 15: Sequence Diagram of the pull content use case. The pull use case is extended with logging capabilities.

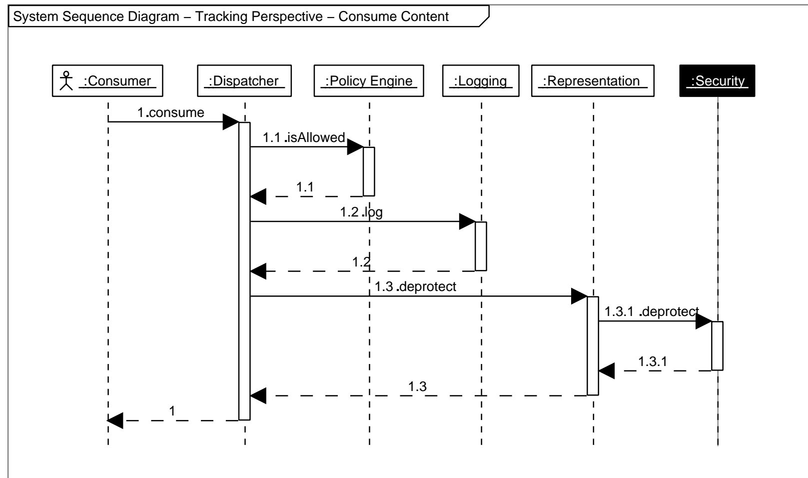


Figure 16: Sequence Diagram of the consume content use case. The consumer (or his decoder) asks to consume content. The *Dispatcher Component* sends the request to the *Policy Engine Component* for interpretation of licenses and to the *Representation Component* for converting secured content to non-secured, consumable, content.

The DRM client gathers context needed to interpret the license, interprets the license and, if allowed, deprotects the content.

The request is processed in the RI component group via the following steps (see figure 16 on page 40). The message arrives at the *Dispatcher Component* (1), which adds context needed for the *Policy Engine Component* and decides whether or not the content is allowed to be consumed (1.1). If so, the *Logging Component* logs relevant information of the request (1.2). Finally, the content is sent to the *Representation Component* (1.3), which delegates the content to the security component group for deprotection (1.3.1).

5.4 Licensing perspective

This perspective focuses on license issuing.

5.4.1 Component diagram

The architecture component diagram for the licensing perspective can be found in figure 17 on page 41.

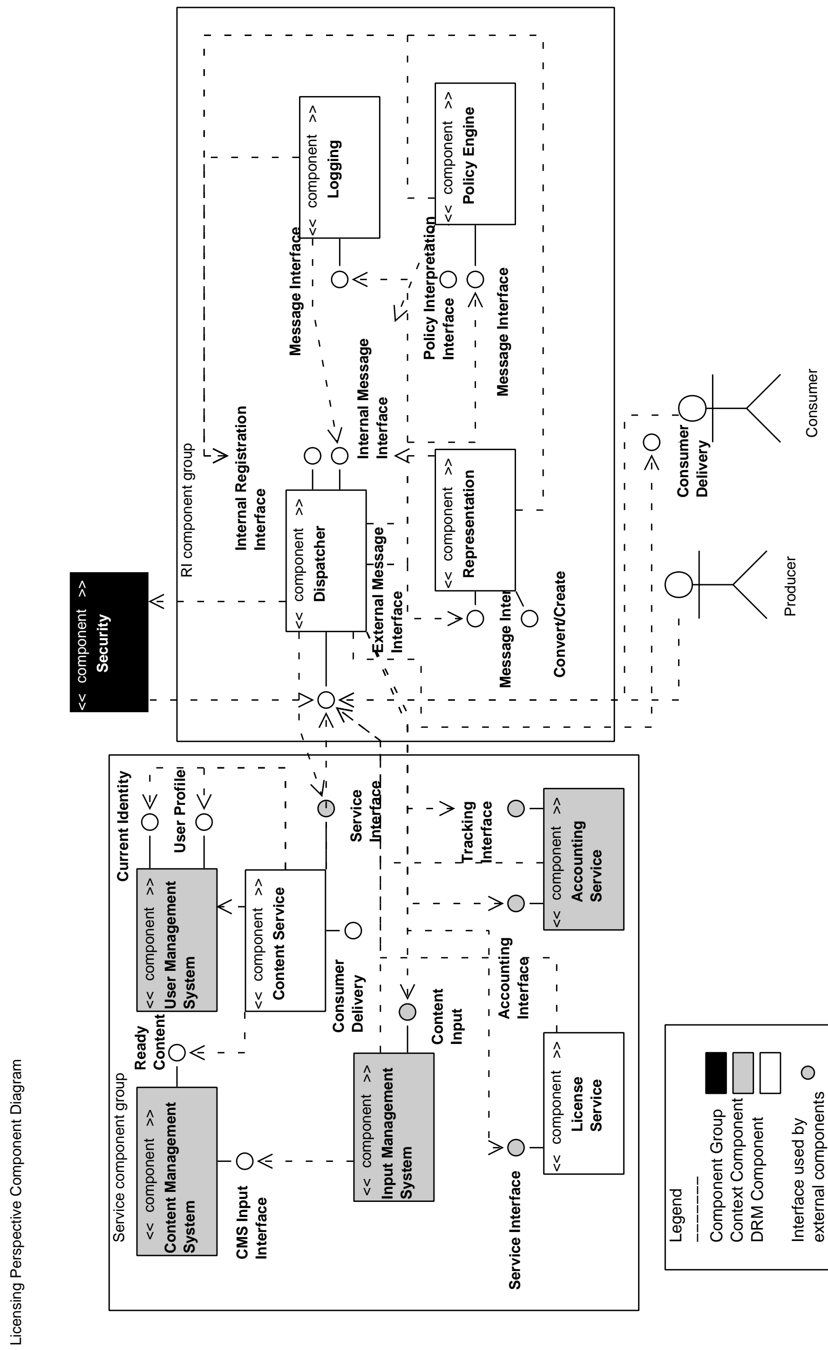


Figure 17: Architecture component diagram of the licensing perspective from publisher side.

5.4.2 Service component group

Only one component, the *License Service*, is added.

License Service **Assumption**: This service offers usage rules, which are stored together with other information in a separated file. It is optional if the *Content Service* interweaves licenses with content.

Licenses issued to the consumer are already bound to the content or can be bound on the fly. The latter has as major advantage that general license models are possible for a whole content range.

This component offers the following interfaces:

- **Consumer Interface**: This interface is used by the consumers (e.g. end users from distributors, distributors from others, etc.) for obtaining, recovering and renewing licenses.
- **License Management Interface**: This interface is used by producers to manage licenses. They can revoke, remove or update licenses. They have also the possibility to bound licenses to content.

5.4.3 RI component group

The *Policy Engine Component* is used to interpret licenses at consumer side. No new components are added.

5.4.4 Scenario

The **Obtain License** scenario is comparable to the **Pull Content** scenario.

5.5 Security perspective

The previously described architecture is capable of license handling, consumer tracking and content consuming. This section focuses on content protection. Different protection schemes are possible, depending on the capabilities of the publishing system and available security mechanisms. We distinguish following schemes:

1. Protection on the fly. The security component group of the online DRM system interweaves a watermark with identification data of the consumer. This is only possible if the publishing system allows content personalization. A major drawback is that this approach generates a lot of computational load at the online DRM system, because content has to be secured for each consumer request. This is supported by Windows Media DRM.

2. Protection at the consumer side. The security component group of the consumer's DRM client protects the content. Main disadvantages are that (i) the DRM client needs possibly a lot of processing power, and (ii) content is send unprotected. A major advantage is that the *Content Service* can be very lightweight.

3. Initial protection by producer tool. First, the producer tool protects content. Next, it submits protected content and protection data or license to the publishing system. Major advantage is that the publishing system can be very lightweight because it does not have to protect content. Main disadvantages are (i) that only non-personalized content protection is possible and (ii) that the producer tool has to contain security services. This protection model is supported by Windows Media DRM.

4. Initial protection at the publishing system's *IMS*. The *IMS* sends content explicitly to the security component group. The main advantages is that (i) that the producer tool can be lightweight and (ii) that content is not protected on the fly. A major drawback is that personalized content is not possible. This protection model is supported by Windows Media DRM.

5. Combination of personalization at the consumer side and one of the first two protection methods. For instance, personalization can be done before the first consumption. A major advantage is that the publishing system can be very lightweight. A main disadvantage is that the publisher has to trust the consumer's DRM Client. This protection model was done by early Fairplay implementations.

5.5.1 Architecture

Content may consist of different parts. Each of these parts can have different licenses associated with it, and can thus be protected in a different way. The *Representation Component* knows how composed content is represented and thus how these parts are glued together and protected. The security component group only knows how to protect a content part, not the composed content. Therefore, the RI component group consults the security layer several times (for each part of the content) in order to construct composed content.

The security component group contains various algorithms and techniques for content securing, such as watermarking, fingerprinting, secure data enveloping, secure communications, certificate management, secure storage, secure timing, secure localization, key exchange, key generation, digital signatures, key management, encrypting, and key storage.

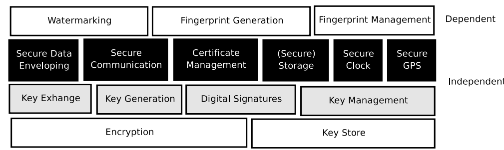


Figure 18: The security architecture consists of two layers: a *type dependent layer*(top layer), which uses functionality of the *type independent layer* (three bottom layers). The type independent layer consists of three layers: (i) a high level layer (black), which requires minimal notions of security techniques to use it, (ii) a complex low level layer (white), which offers basic security algorithms, and (iii) an intermediary layer (gray), which forms the bridge between the high level layer and the low level layer.

Some developers (of DRM systems) want to exactly specify the algorithm and parameters being used, whether others only want to have a notion of encryption. By consequence, we should have at least two levels of interfaces (see figure 18 on page 44): (i) high level interfaces (black) which require no notion of security techniques, but offer an intuitive interface with methods such as "sign a message" or "create a certificate" and take care of creating digital signatures, data encoding, etc; (ii) a more complex layer (white) which offers basic security algorithms such as encryption and hashing; and (iii) an intermediary layer (gray) which uses the lowest complex layer and offers interfaces towards the higher layer with methods such as "create a digital signature", "create a fingerprint" or "create watermark". This intermediary layer makes the implementation of the highest layer less complex.

The type independent layer is placed on top of the type dependent layer because the latter can use functionality of the former. For instance, the *Watermarking Component* uses the interface of the *Secure Data Enveloping Component* to protect the watermark.

It is possible that each security component, despite the same high level interfaces, exists in various versions, developed by different vendors. Therefore, one could apply the microkernel pattern, which allows using these different versions in a uniform way, and makes it possible to replace components. However, secure data enveloping and developing definitely needs to be performant, because the consumer does not want to wait a long time, before he is able to consume his content. Therefore, we advise to use standard encryption libraries (such as cryptlib[9]), because these libraries (i) implement cryptographic functions in an efficient manner, (ii) make it possible to add custom algorithms, (iii) are mostly one monolithic, but fast, service component.

Sometimes, it is necessary to apply certain security functions (within one layer of the layered security architecture) after each other. For instance, when a content

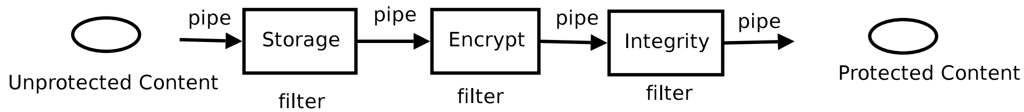


Figure 19: Security architecture: content is passed through different components

part is first retrieved from storage, after which it is encrypted with a symmetric key and integrity check is added. This functionality can be achieved with (i) the pipe-and-filter pattern, or (ii) the broker pattern.

Pipe and filter pattern Each security primitive is encapsulated in a filter component (see figure 19 on page 45). Data is passed through pipes, which connects the filters. The filters can be recombined to allow different protection techniques[5].

Major advantages of this pattern are that (i) it allows flexibility, (ii) and that filter components can be reused.

Main drawbacks of this patterns are that (i) sharing security state information is expensive or inflexible, (ii) it has data transformation overhead, (iii) error handling is difficult.

Broker pattern The components of each layer (high level, intermediate and low level) are organized by the usage of the broker pattern. An *Orchestrator Component*[6] acts as broker and manages the securing process.

Major advantages of applying the broker pattern in this case are (i) that new components can be integrated easy with existing security components, (ii) that new security components can be added very easily (extensability), and (iii) that different configurations of security components are possible (flexibility).

Main drawbacks are (i) that communication messages must be standardized, (ii) that unprotected data is sent in the clear, and (iii) that much communication is needed to protect content.

Solution Table 4 motivates our choice for the the broker pattern. The main argument is that the pipe and filter pattern is not an ideal solution because there is a wide variety of security component compositions.

[5]This is only possible, if the filters know how to add, for instance a signature, to the original data. This can be achieved by configuring the filters with metadata added to the message.

[6]This can be the *Representation Component*.

	Pipe and filter pattern	Broker pattern
Adding new components (extensability)	+	+
Different configurations (flexibility)	+-	+
Integration (flexibility)	+	+
Reusability	+	+
Communication (performance)	-	-
Error handling (availability)	-	?

Table 4: The quality attributes in this table motivate the choice for the Broker pattern.

5.5.2 Component diagram

The architecture component diagram of the security component group is given in the right part of figure 21 on page51.

5.5.3 Security component group

This component group contains a lot of security components, all related to security algorithms:

- *Fingerprinting Component*: type dependent. Used to generate fingerprints [5, 4].
- *Watermarking Component*: type dependent. Used to generate watermarks[7, 10].
- *Secure Data Enveloping Component*: type independent. Used to protect data, calculate hashes, digital signatures, fingerprinting, watermarking, etc. automatically[17]
- *Certificate Management Component*: type independent. Used to manage certificates.[17]
- *Key Exchange Component*: type independent. Used to exchange keys.[17]
- *Digital Signatures Component*: type independent. Used to enforce integrity.[17]
- *Key Generation Component*: type independent. Used to generate keys.[17]

- *Key Management Component*: type independent. Used to manage keys.[17]
- *Key Store Component*: type independent. Used to store keys.[17]
- *Encryption Component*: type independent. Used to encrypt data.[17]
- *(Secure) Data Storage Component*: type independent. Used to store data. This can be a (relational) database, an LDAP system, a filesystem on a hard disk, secure hardware, etc. In case of secure hardware, key management should be done automatically.[17]

Fingerprinting Component The *Fingerprinting Component* is used to generate fingerprints. It should have the possibility to interpret content, because fingerprinting algorithms use certain properties of content (such as beats per minute, frequency) to generate fingerprints. Solutions tackling this problem are:

1. The *Fingerprint Component* can be told that content has certain properties. The major advantage is that in stead of analyzing content itself, this component receives properties used to calculate fingerprints from the *Representation Component*.

2. The *Representation Component* automatically transforms content representation to one that a particular security component understand. For example, an ogg vorbis audio stream can be transformed to e.g. WAV and then fed to an audio fingerprinting algorithm which only understands WAV.

We suggest the use of the first solution, because the responsibilities of the component groups are clearly separated. In the first solution, the *Representation Component* interprets content, and feeds the results to the *Fingerprinting Component*, while in the second solution, the *Fingerprinting Component* is also responsible for interpreting content.

The *Fingerprinting Component* has only one interface:

- **Generate Fingerprint**: this interface is used to calculate a fingerprint.

Watermarking Component The *Watermarking Component* is used to read and add watermarks. This component has the same problem as the *Fingerprinting Component*: it has to interpret content. The same solutions of fingerprinting are applicable, however the second one seems better, because there is no need for a *Watermarking Component*, if the watermarking algorithms are part of the *Representation Component*.

A *Watermarking Component* should have only one interface:

- **Read/Add Watermark**: this interface is used to read watermarked content and to add watermarks to content.

Key Management Component The *Key Management Component* is used to manage keys, which means storing and retrieving keys. This component has one interface:

- **Key Management Interface**: this interface is used to manage keys.

Digital Signature Component *Digital Signature Component* is a component, which is used to create and verify digital signatures. The component has two interfaces:

- **Verification**: this interface is used to verify digital signatures.
- **Creation**: this interface is used to create digital signatures.

Key Generation Component The *Key Generation Component* is used to create and generate symmetric and asymmetric keys. It has only one interface:

- **Generate Key**: this interface is used to generate keys.

Key Store Component The *Key Store Component* is used to store and retrieve keys. It has only two interfaces:

- **Store**: this interface is used to store keys.
- **Retrieve**: this interface is used to retrieve keys.

Encryption Component The *Encryption Component* is used to encrypt data. It has two interfaces:

- **Encrypt**: this interface is used to encrypt data.
- **Decrypt**: this interface is used to decrypt data.

Hash Component The *Hash Component* has the possibility to calculate hashes. This component has two interfaces:

- **Calculate hash**: this interface is used to calculate hashes.
- **Verify hash**: this interface is used to verify hashes.

Secure Storage Component *Secure storage Component* is used to store information in a secure way. This component should handle keys in a normal way. There should be no (big) difference in storing information on a secure or normal disk, database, etc. It has one interface:

- **Store/Retrieve/Delete/Metadata**: this interface is used to do basic file system manipulations.

Secure Clock and Location Component This component is responsible for time and position. It has one interface:

- **Context Awareness**: this interface is responsible for reading time and position.

5.5.4 Scenario

Suppose the consumer wants to consume content which consists of a single content item with an embedded watermark. The *Representation Component* (see figure 20 on page 50) sends its request to the security engine several times. The *Orchestrator* contacts the right security components.

The *Representation Component* sends part of the content together with already obtained keys, and an ordered list with security algorithms to the *Orchestrator* (1), via the *Dispatcher Component* (1.1). In our example, the *Orchestrator* first sends the content part (and the obtained keys) to the *Secure Data Enveloping Component* (1.1.1), which will decrypt it (possible by contacting *Encryption* services, 1.1.1.2) and verify digital signatures (1.1.1.1). After decryption, the *Orchestrator* sends the decrypted content to the *Watermarking Component* (1.1.2), which reads the watermark. Finally, the *Orchestrator* sends the decrypted content and the watermark back to the *Representation Component* (1 and 1.1).

5.6 Putting the components together

The resulting architecture is illustrated in figure 21 on page 51.

5.7 Deployment view

The deployment view (see figure 22 on page 52) presents a mapping of the previously presented components onto network nodes. These nodes represent physical servers and are shown as a 3D rectangle in the deployment view. Figure 22 illustrates a possible deployment view from a DRM publisher perspective.

We can apply different deployment tactics:

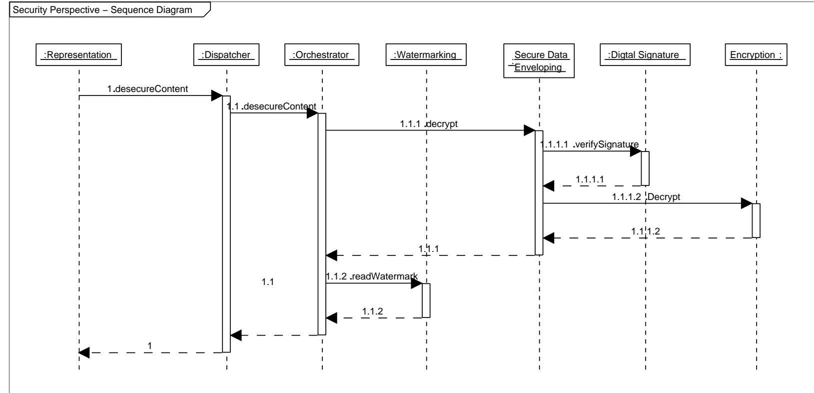


Figure 20: Security perspective: an *Orchestrator* contacts the security components in the right order to protect a content part.

1. **Replication**: This tactic consists of replication services, in order to increase availability, and scalability. It consists of a *Replica Manager* and the actual replica's of the component. This tactic is applied to the components of the RI component group, because they intercept and process almost every inter component message. It should also be applied to other data intensive components such as the *CMS*, the *UMS* and the *Orchestrator*, the *Data Enveloping Component* and the *Watermarking Component*. The replication number depends on the availability one wants to achieve and thus also on expected workload and used system. For instance, one could replicate components of the RI component group ten times and *CMS* and *UMS* two times.

2. **Security Service and RI Deployment** This tactic consists of deploying each security service and each RI component on a separate machine. Each security service is expected to generate a lot of computational load (encryption, etc.). By employing this tactic, one guarantees that these services do not influence each other: overloading one service does not collapse the entire system.

The deployment view from a DRM consumer perspective is totally different. The consumer does only have the necessary components, but all on the same device.

5.7.1 Conclusion

Figure 22 shows only a particular deployment situation. The added value of this deployment view is that it shows the used tactics to support quality attributes such as availability and scalability.

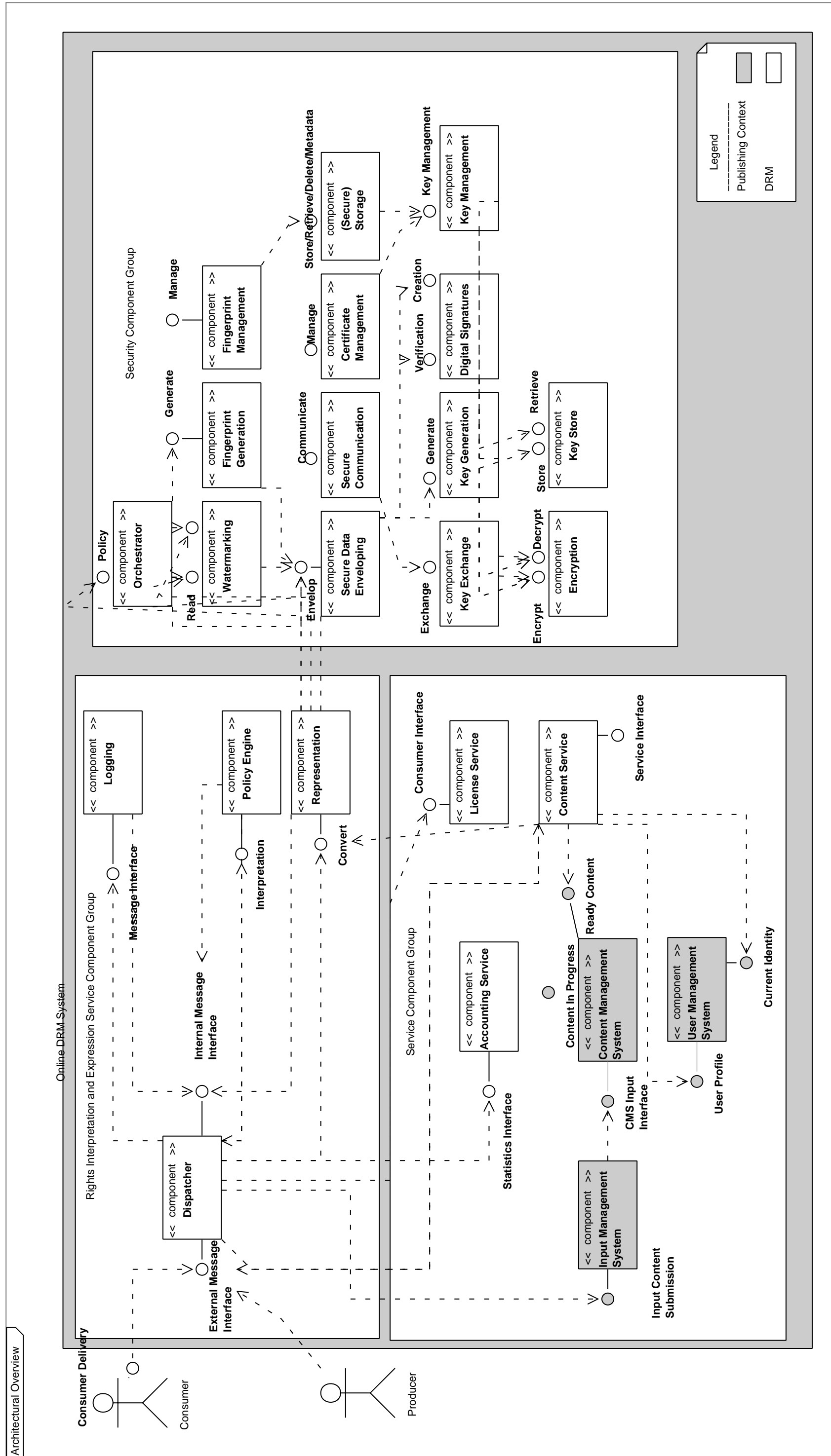


Figure 21: The proposed architecture from publisher side.

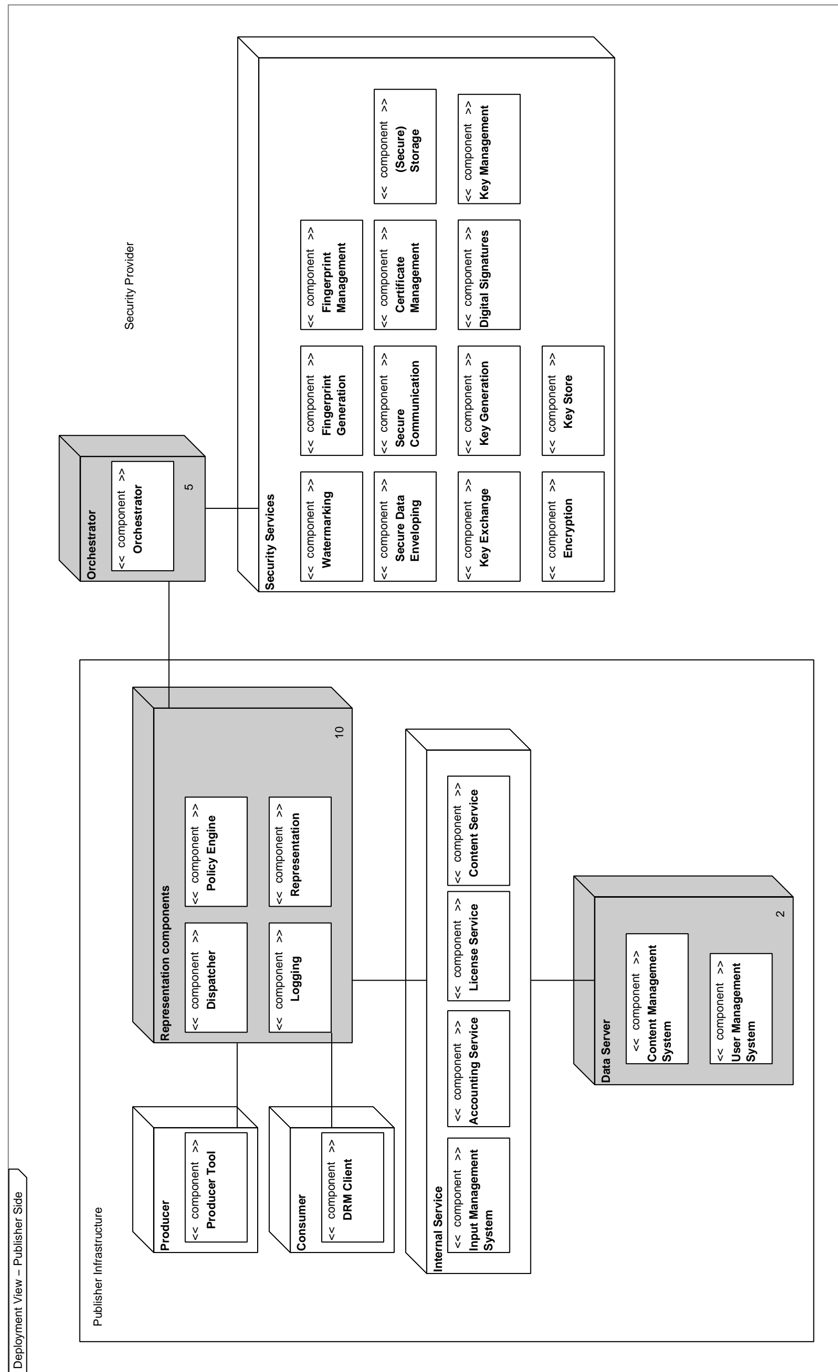


Figure 22: Deployment view from the publisher side. The numbers in the grey boxes represent the number of replica's needed to achieve availability.

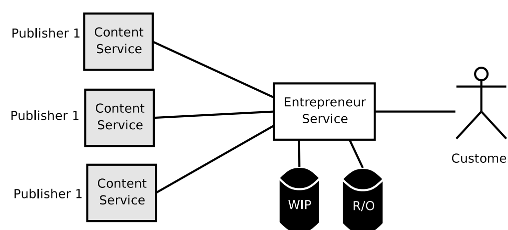


Figure 23: Example solution for integrating multiple content services.

5.8 Application Layer

The application layer uses the services (DRM and context) to offer own solutions. These solutions are mostly:

- **Content oriented**: offering, manipulating, integrating, etc. content from one or more content and *License Service*s.
- **Statistics oriented**: offering, integrating, etc. statistics from one or more *Accounting Service*s.

We work out an example for each solution type.

5.8.1 Content oriented

Suppose someone wants to create a specialized shop where you can buy everything about a particular subject, e.g. Lord of the rings. The entrepreneur does not want to host content itself, but instead redirects customers to publishers offering the requested content. The dealer asks money for the usage of his integration service, while the publisher is paid for content and licenses.

This can be achieved by the usage of the broker pattern. A custom content integration service is plugged-in as service. This service (see figure 23 on page 53) supports an N:1 topology that separate read-only distribution rules and intermediary results. It allows a single interaction from a customer to be distributed to multiple *Content Service*s concurrently.

The solution is divided into a number of logical components:

- The customer represents one or more customers that are interested in interacting with *Content Service*s of other publishers.
- The *Entrepreneur Service* supports message routing, decomposition and recomposition, message enhancement and transformation. Rules for doing this are stored read-only (R/O). This service also uses a work-in-progress data store (WIP) to retain the intermediate results from responses coming back from target applications until all the necessary responses are received.

- The *Content Services* represents *Content Services* in multiple partner organizations.

Obtaining content from the entrepreneur succeeds in several steps. First, a customer contacts the *Entrepreneur Service* (possibly after authenticating) by querying for content. Secondly, the *Entrepreneur Service* concurrently queries several *Content Services* of multiple publishers. Thirdly, the *Entrepreneur Service* waits for answers. What happens next, depends on the business model. For instance, the entrepreneur can resell content or redirect the customer to the right publisher.

If the entrepreneur resells content, its service obtains[7] content and a redistribution license from the publisher who offered the cheapest book. The service repackages the content and gives it to the customer with a corresponding license.

If the entrepreneur redirects the customer, its service combines the search results and presents them to the customer. The customer selects one, and finally is redirected[8] to the publisher who offers the selected content.

5.8.2 Statistics oriented

Suppose someone wants to offer statistics about content produced by some producer, published by more than one publishers. Therefore, he can use a slightly variation of the above solution:

- Replace customer by producer.
- Replace *Content Service* by *Accounting Service*.

6 Evaluation and related work

This section highlights the main advantages of having a detailed architecture, i.e. dealing with quality requirements. It first defines the main quality attributes that are relevant for DRM systems. Secondly, it describes how the proposed architecture supports each of these attributes in form of quality scenarios. The section closes with discussing main conclusions we can draw from this evaluation.

6.1 Evaluation criteria and quality attributes

This section applies an architecture evaluation by usage of quality attributes on recent technologies such as DMP and AXMEDIS.

[7]The Publisher can offer him a subscription based billing model.

[8]The entrepreneur can authenticate the customer before redirection.

A quality attribute is a property of a work product (or software architecture) by which stakeholders can judge its quality. These properties, like for instance interoperability, modifiability, or extendability may have significant impact on the architecture of a system [2]. As such, quality attributes are generally accepted to be crucial attention points during the early phases of system development, i.e. analysis and architecture design.

It should be clear that many quality attributes must be supported when building and deploying DRM systems or content distribution applications: interoperability, modifiability, extendability, usability, testability, availability, security, performance, scalability, etc.

Since many interpretations of these attributes exist, we provide a definition for a selection of them and motivate why it is an important concern for DRM systems:

- **Interoperability.** The ability of two or more (sub)systems to cooperate at runtime. Interoperability of DRM service components is highly important given the complexity and extensiveness of DRM functionality, as well as the rapidly evolving field, which imply that multiple variants of the same service will co-exist and should be able to co-operate.

- **Accountability.** The ease to add accounting and consumer tracking aspects to the system. Since business and payment models will likely be personalized, and since license abusers should be identifiable, tracking of consumer activities is a crucial concern.

- **Modifiability.** The ease with which a software system can accommodate changes to its software. License and payment models will involve various publisher specific policies, which implies that the publishing system's behavior must be modifiable to accommodate varying policies.

- **Extendability.** The ease to add new features to the system. Since the domain of software security and DRM is highly dynamic, DRM systems must embrace change and be open to extend their functionality with new or improved security technologies.

- **Usability for developers.** Programming support and ease of use for developers of the system. In order to prevent security holes in DRM software, it is highly important to guide developers and force them as much as possible in a well-defined software architecture.

- **Testability.** The ease with which software can be made to demonstrate its faults. Testability is crucial for every publicly accessible system, especially if authentication, authorization, and accountability is at stake, as is the case with DRM.

- **Availability.** Availability can be measured as the time that the system is up and running correctly, the time between failures, and the time needed to resume operation after a failure. Business systems should be 'constantly' online, which is highly challenging in a rapidly evolving field which implies frequent (security) updates.

- **Performance.** Performance can be measured as the number of transactions per second (throughput). Performance is important for a DRM system, because consumers do not want to wait longer to consume protected content than to consume unprotected content.

- **Reusability.** Reusability is the degree to which a software module or other work product can be used in more than one computing program or software system. This quality attribute is important for almost every software system, including DRM systems, because it saves a lot of development time, aids interoperability and reliability.

Quality attribute	**Architectural style**	**Advantage/Disadvantage**
Interoperability	service components, broker	reusable service components, message conversion, clearly defined interfaces
Accountability	broker, component groups	(+) broker allows transparent message interception
Modifiability	broker, service components	behavior described in pluggable policies
Extendability	broker, service components	security components registered at broker
Usability for developers	service components, overall architecture	(+) hot spots to insert service components, layered security layer
Testability	component groups	(+) hierarchical views allows incremental test approach
Availability	broker	(-) single point of failure
Performance	broker	(-)
Reusability	components	(+)

Table 5: Overview of main quality attributes, the architectural styles that support them, and the benefits they offer.

Quality attribute	Proposed Architecture	DMP	Axmedis
Interoperability	+	+	+
Accountability	+	-	-
Modifiability	+	+/-	?
Extendability	+	+/-	+/?
Usability	+	-	-/?
Testability	+	-	+/?
Availability	-	-	?
Performance	-	-	?
Reusability	+	+	+

Table 6: Comparison between the proposed architecture and the DMP architecture related to their support for quality attributes.

6.2 Evaluation of the Distrinet DRM architecture

The proposed architecture is evaluated using the criteria sketched in 6.1. A summary is found in table 6 and table 5. The table highlights the key features of the proposed architecture that play a considerable role in achieving these quality attributes: hierarchical component groups, stand-alone service components with well-defined interfaces, and the broker pattern for message delegation and transparent interception.

Interoperability is achieved by identifying service components with documented interfaces and localizing them in an overall architecture.

Accountability is achieved by usage of the broker pattern, which allows transparent message interception.

Modifiability of the system's behavior to application or business specific policies is supported at the level of the architecture by the broker, in which policies can be injected.

Extendability is achieved by enabling to add security components by simply registering them at the broker which transparently maps consumer requests and business policies onto the available set of security algorithms and techniques.

Usability for developers is achieved by the granularity of the overall architecture. Interfaces are clearly defined and explained. Using the security components is easy, because the developer should only have a general knowledge of security principles, not a thoroughly understanding.

Testability is aided by the hierarchical views, which allow an incremental test

approach.

Availability is a weak point, because the broker is a single point of failure.

Performance is a weak point, because the broker intercepts all messages.

Reusability is aided by the right granularity of the components. For instance, someone can easily plug in a service component in the DRM architecture.

6.3 Evaluation of the Digital Media Project architecture

The DMP architecture (see appendix E and [6]) defines users (e.g. consumers, producers, or publishers) as entities that perform so-called primitive functions, which represent underlying DRM services that handle digital content. This section uses the criteria sketched in 6.1 to evaluate DMP. A summary can be found in table 6.

6.3.1 Interoperability

The Digital Media Project (DMP [6]) strongly focuses on interoperability, which is clearly a very important concern for DRM.

First, *DMP* does not want to implement an *interoperable* DRM system, but only a set of standardized tools (interfaces) which won't change throughout time. DMP is interoperable within a whole value chain, because each value chain user uses the same primitive functions. DMP primitive functions can be related (see appendix E.4 on page 128) to our three component groups (and corresponding components) identified in the architecture proposed in this report: the service component group (e.g. the *revokeUser* function), the RI component group (e.g. the *representRightsExpression* function), and the security component group (e.g. the *representKey* function).

However, DRM Tools are not necessary interoperable, because licenses from different third party toolproviders can use a different protection toolpacks.

Finally, the idea of using stable primitive functions and updateable DRM Toolpacks is beneficial for interoperability. If a user does not have the toolpack to consume certain content, that toolpack can be (automatically) obtained.

```
1.DMD
a.Creates dmpxrd:AccessID and dmpxrd:AccessPassword
b.Sends AccessID and AccessPassword to Domain Administrator
```

Figure 24: Extract from DMP's domain management protocol

6.3.2 Accountability

It is hard to achieve accountability in DMP. Even when specific interception functions would be offered, it seems highly difficult to add support for consumer tracking without changing the implementation of primitive functions. This is because primitive functions can call each other directly, which prevents easy and transparent recomposition, and because no architectural infrastructure is available to intercept calls (cfr. the broker pattern).

By consequence, we should modify the conceptual models. Therefore, we suggest the following modifications to the conceptual models: (i) A DRM Tool which allows submitting usage statistics to some other device should be added in the DRM Tool Model. (ii) a new device for usage statistics should be added to the DRM Distribution Model.

6.3.3 Modifiability

DMP offers partial support for modifiability, since it is not always possible to change representation functions, without braking other functions. For instance, the primitive function *Access License as file* extracts the content identifier from content meta-data, which means that this function knows about the representation of the content. This could be solved by grouping data representation functions and separating them from the other functions. This achieves a hierarchical model (cfr. component groups), with clearly separated concerns for each subsystem.

More low level modifiability issues are:

- **DMP assumes all representation primitive functions use keys, hashes and signatures**. What if we want to use another cryptographic primitive?
- **Some protocols are not general enough.** Most domain management protocols use a password based authentication technique. What if we want to use another, more secure?, authentication technique.

6.3.4 Extendability

DMP functionality can easily be extended by adding new primitive functions. However, since primitive functions are very fine-grained, and since there is no blueprint architecture that guides developers in composing an arbitrary set of functionality, application development implies considerable risks for software errors.

6.3.5 Usability for the developer

The usability for the developer is reduced to that of a toolkit. In order to compose a particular DMP tool, which would support a new use case, a developer should combine various very fine-grained primitive functions. Given the complexity of DRM, and the clear need for secure software, this approach implies considerable risks for software errors. It would be better if the developer takes a few components, e.g. license provider device, off the shelf and that he has a working DRM solution in no time.

6.3.6 Testability

Testability of primitive functions is also related to complexity and importance of securing DRM software. Although each DMP primitive function can be tested individually, special attention should go to incrementally testing functionality, i.e. hierarchically grouping related functionality that can be tested in isolation. Testing a license service in DMP, for instance, would involve a lot of related primitive functions.

Therefore, one could group the primitive functions into groups of related functions. Each group of related functions can be tested individually. This reduces the number of tests.

6.3.7 Availability

Given the tight coupling between primitive functions, it is difficult to maximize availability during updating or reconfiguring a DMP system. A clear separation of functionalities which enables independent changes in different component groups would considerably increase the availability of the system.

6.3.8 Performance

DMP's DRM client can have a negative performance if a lot of toolpacks are required to process content, because they are executed after each other. More delay is added if some toolpacks are not initially available at the DRM client, because they need to contact a toolpack service to obtain the missing toolpacks.

6.3.9 Reusability

DMP uses different primitive functions, which can be combined into tools and toolpacks. Exchange of tools and toolpacks is encouraged by design. Obviously, they are highly reusable.

6.4 Evaluation of the AXMEDIS architecture

The AXMEDIS project has developed a framework which consists of a structured and organized set of software components in the area of cross media production, content sharing and content distribution (see appendix F). Not enough detailed information was available to perform an in-depth analysis of its architecture.

AXMEDIS defines a wide variety of users who can be classified as producers, publishers, and consumers. It defines four major areas of which the *AXMEDIS Player* and the *AXMEDIS Protection and Supervising Area* are the most related to our architecture. The service components of the DRM architecture we have presented can be seen as a subset of the AXMEDIS architecture. The context of our architecture is more elaborated in their *Factory Area*, which includes tools for collecting, producing, composing and formatting content, programming the production process, and producing licenses. However, as far as we know, it does not have an individual security component group, nor architectural support for different content representations. AXMEDIS seems to be partially interoperable, because it has well-defined modules and interfaces.

We use the same evaluation criteria presented in section 6.1. A resulting summary can be found in table 6.

6.4.1 Interoperability

To the best of our knowledge AXMEDIS is interoperable within a value chain as long as every user use the same protection model.

6.4.2 Accountability

The AXMEDIS architecture does not have explicitly support for tracking. Each component should send its tracking information directly...

6.4.3 Modifiability

AXMEDIS is highly modularized. Interfaces for some modules are explained and relations with other modules are declared. Modules such as fingerprinting are algorithm-based and have the possibility to plug in own algorithms. Other modules such as the database area offer a fixed interface for relational databases. It seems impossible to add support for Object Oriented Databases or no user-password authentication techniques. By consequence, these modules are not easy extensible. Modules such as the player and editor have the possibility to plug-in own algorithms. However, documentation is vague about how this is practically done (meaning we do not have access to the interfaces).

6.4.4 Extendability

It is possible to extend some modules with new algorithms (e.g. fingerprinting), and to add new modules.

6.4.5 Usability for the developer

AXMEDIS provides an extensive set of components and an enormous amount of documentation. However, we could not make relevant conclusions concerning usability of the architecture.

6.4.6 Testability

At first sight, it seems possible to test some components independent of each other (e.g. fingerprinting). More detailed information is needed to formulate conclusions.

6.4.7 Availability

We do not have enough information to judge the availability of the AXMEDIS architecture.

6.4.8 Performance

We do not have enough information to judge the performance of the AXMEDIS architecture.

6.4.9 Reusability

Due to the highly modularized approach of AXMEDIS, reusability is supported. Each module, algorithm, etc. can be reused by third party. AXMEDIS already made an implementation of each module. If they publically release the detailed interfaces of each module, other vendors can make new implementations.

The interfaces of the *protection area* are the ones we are interested in. This area is decomposed in several modules such as license manager, domain manager, license verificator, etc. These modules are combined into the Protection Manager Server (our online DRM System), which offers an uniform interface towards clients. This interface is replicated into the DRM support module, which will forward requests to other modules. The interfaces of each module are briefly described. Preconditions and postconditions as exact return values of the methods of the interfaces are absent. Most modules use the database module to store their information in a database. It is unclear whether it is possible to replace the implementation of each module with another one as long as the interface is respected. By consequence, it is unclear whether parts of the System are reusable.

It is unclear how to change DRM related algorithms (in case an unsafe one is found), whether it is easy to change the license representation, etc.

6.5 Conclusion

Table 6 shows an overview of the evaluation on our proposed architecture, DMP's architecture and AXMEDIS architecture.

7 Future work

This section formulates three issues that are open for future work: (i) securing the proposed architecture, (ii) anonymizing consumer tracking, (iii) adapting DRM protected content, (iv) managing domains, and (v) broadcasting personalized content. A possible solution is formulated for the last issue.

7.1 Securing the proposed architecture

We suppose that our DRM architecture is susceptible to various attacks, because we did not take into account security during design. To identify and mitigate various attacks, we can try approaches such as CLASP[18] and STRIDE[12].

7.2 Anonymizing consumer tracking

Consumer tracking produces various privacy sensitive information, which is subject to regulations and legislations in most countries. The publisher is allowed to do more with anonymous consumption data, because they are less subject to regulations and legislations. By consequence, introducing anonymous consumption data in the architecture benefits publishers.

7.3 Adapting DRM protected content

Currently, networks exist that are able to adapt content quality depending on network usage or quality of service (QoS) requirements. This seems impossible with DRM protected content, because content is not modifiable after protection. This is illustrated by the following example. The file size of an audio file as 256 Kb MP3 (high quality) is larger than the same audio file encoded as 128 Kb MP3 (low quality). By consequence, high quality content implies more network traffic than low quality content. In-network processing of DRM protected content, for instance to cope with varying network load, is only possible if the distributed architecture offers support for transforming DRM protected content.

7.4 Managing domains

A domain is a group of users or devices which have the possibility to consume and exchange DRM protected content intended for that domain. Proposed solutions[16] can easily incorporated in our architecture. A *Domain Service*, which is responsible for managing domains, is added to the service component group.

7.5 Broadcasting personalized content

7.5.1 Problem

Broadcasting personalized content introduces problems related to the distributed architecture: (i) badly scalable duplication at the publisher side, (ii) no interaction between publisher and consumer, (iii) inefficient usage of the transportation network.

Personalized content is different for each consumer (group). For instance, consumers may compose their own digital newspaper, meaning that they select and combine articles they are interested in, while the publisher may add several advertisements, which are based on the users interests. DRM systems typically protect this *personal* newspaper after it has been built, and distribute it to the consumer together with a personal license.

By consequence, the personalized data is sent from the sender to each recipient, resulting in inefficient and badly scalable duplication at the sending side. In addition, it is impossible for the consumer to interact with the publisher because broadcasting implies one-way communication. The consumer can thus not (i) subscribe to a particular service, (ii) negotiate a particular license model, or (iii) send statistics about his content usage. Finally, personalized content does not allow caching to improve efficiency of data transportation, because the content is personalized and thus different for every consumer.

7.5.2 Possible solution

The solution requires an up-link, which is necessary to enable two-way communication. Depending on needs, the up-link can be an Internet connection, a postal service, etc. If the up-link is a direct connection (e.g. the Internet), submitting of usage statistics, updating of the content, etc. is also possible. Subscription etc. is always possible.

The basic algorithm is as follows: each content part is protected individually by the publisher and broadcasted to everyone. He constructs a single package with the different protected content parts. A license is constructed for each consumer and contains the keys for each of the content parts he is interested in. It contains an unique worldreadable, but unmodifiable identifier, to let the consuming device

know that particular license is intended for him. The license is also encrypted with a personal key (e.g. obtained via a webservice, device specific, etc.) More formally: $\{contentpart1\}_{K1}, \ldots \{contentpartx\}_{Kx} \rightarrow_{package}$ content
Optional: $\{content\}_K$
$\{license, K\}KP$

The packaged content is broadcasted to everyone. If the client does not have the possibility to contact the publisher directly (via e.g. the Internet), licenses of each consumer (probably on a different broadcasting channel) are broadcasted at regular intervals. The license is thus *not* interweaved with the content. After all, the consumer may start consuming during the middle of a content broadcast. But if the client has the option to communicate with the server, the license is only transmitted once.

Main disadvantages are that (i) everyone has the possibility to consume content if the key is leaked, because the content is encrypted by only one key, and (ii) remarkable delays before consumption are possible, because the licenses, and thus the keys, could be missed.

Possible solutions are that (i) this key should be replaced at regular intervals during a broadcast, (e.g. every 10 minutes) meaning the licenses should also be updated, and (ii) remarkable delays are avoided by caching all possible licenses.

Main advantages are (i) bandwith savings and, (ii) caching.

7.5.3 Extensions

Broadcast encryption The key used to encrypt the content or license is based on a broadcast encryption technique. Each consumer device receives key material so it has the possibility to compute keys for each of the members of its group (except itself - exclusion key). If the exclusion keys are used to encrypt content, only the devices/persons from which the exclusion key has not been used, have the possibility to consume content.

Major disadvantages are (i) a new exclusion key must be distributed across the existing members everytime a new device registers to consume content, and (ii) content has to be re-encrypted everytime a new device registers to consume content,

The main advantages is that broadcast encryption is ideal for groups with a stable userbase (e.g. videostreams of security cameras, videoconferencing). If they all use the same license, one can use broadcast encryption for licensing techniques.

8 Conclusion

This report has presented a detailed DRM architecture that provides an answer to three main challenges in DRM: availability of a robust and common core architecture, which can easily be extended, and which supports several architectural drivers (including interoperability, modifiability, and extendability). The architecture represents three component groups consisting core services, representation and interpretation services, and security services. The report has incrementally presented the architecture from four key perspectives, starting from the core (content handling), and then extending it with consumer tracking, licensing, and content securing. By way of evaluation, the report has assessed to what extent quality attributes are supported by the architecture.

The presented DRM architecture is sufficiently detailed to enable the creation and management of DRM systems and content distribution applications. To the best of our knowledge, it is more sophisticated than related work published so far.

Acknowledgements

Part of the research described in this report has been conducted in the context of the e-paper project (http://epaper.ibbt.be/) and funded by the Interdisciplinary institute for BroadBand Technology (IBBT). This project is being carried out in collaboration with a consortium of the following companies: Philips, De Tijd, Belgacom, Hypervision and I-Merge.

A Detailed use cases

Each use case is presented in a compact format that illustrates every step as follows: Sending Component → Message → (Receiving Component):Interface. In other words, every step identifies the component that sends a message, the message itself, the component that receives the message and the interface of this component where the message arrives. *Comments are in italic.*

A.1 Content handling perspective

A.1.1 Consumer

Pull content This consumer executes this use case if he wants to obtain content.

1. Consumer → getContent → RI component group
2. RI component group → getContent → (Content Service):Service Interface

   The next step is executed in case of non-anonymous content consumption.

3. Content Service → obtainIdentity → (User Management System):CurrentIdentity
4. Content Service → getContent → (Content Management System):Ready Content
5. Content Service → convert → RI component group
6. Content Service → protect → RI component group
7. RI component group → protect → Security component group
8. Content Service → send content → RI component group
9. RI component group → send content → (Consumer):Consumer Delivery

Search content in the catalog The consumer executes this use case if he wants to search content in the catalog.

1. Consumer → search → RI component group
2. RI component group → search → (Content Service):Service Interface

Obtain content information The consumer executes this use case if he wants to obtain metadata about content.

1. Consumer → get content information → RI component group
2. RI component group → get content information → (Content Service):Service Interface

Register user The consumer executes this use case if he wants to register him with the publishing system.

1. Consumer → register → RI component group
2. RI component group → register → (Consumer Service):User Subscription
3. Consumer Service → register → (User Management System):User Subscription

Register device The consumer executes this use case if he wants to register his device for obtaining content.

1. Consumer → registerDevice → RI component group
2. RI component group → registerDevice → (Consumer Service):User Subscription
3. Consumer Service → registerDevice → (User Management System):User Subscription

Consume content The consumer (actual player) executes this use case if he wants to consume content.

Assumption: the user hands over content

1. Consumer → consume content → (Dispatcher Component):External Message Interface
2. Consumer → consume content → (Dispatcher Component):External Message Interface
3. Dispatcher Component → getContext → (Context Component):Context Component Interface
4. Dispatcher Component → deprotect → (Representation):Convert

A.1.2 Producer

Submit content The producer executes this use case if he wants to submit content to the online publishing system.

Content can be automatically converted by the *Representation Component* of the RI component group.

1. Producer → submit content → RI component group
2. RI component group → (Input Management System):Content Input
3. Input Management System → input content → (Content Management System):CMS Input Interface

A.1.3 Publisher

Identify source of abuse The publisher executes this use case if he wants to identify the user who has spread content on mass distribution networks.

1. Publisher → identify → RI component group
2. RI component group → identify → (Identification Service):Identification

Revoke user We made abstraction of security techniques. By consequence, revoking a user (which includes authentication/authorization) is out of scope for this report.

Revoke device We made abstraction of security techniques. By consequence, revoking a device (which includes authentication/authorization) is out of scope for this report.

A.2 Tracking perspective

A.2.1 Consumer

Pull content This consumer executes this use case if he wants to obtain content.

1. Consumer → getContent → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → getContent → (Content Service):Service Interface

The next step is executed in case of non-anonymous content consumption.

4. Content Service → obtainIdentity → (User Management System):CurrentIdentity
5. Content Service → getContent → (Content Management System):Ready Content
6. Content Service → convert → (Dispatcher Component):External Message Interface
7. Dispatcher Component → convert → (Representation):Message Interface
8. Content Service → protect → (Dispatcher Component):External Message Interface
9. Dispatcher Component → protect → Security component group
10. Content Service → send content → (Dispatcher Component):External Message Interface
11. Dispatcher Component → submitMessage → (Logging):Message Interface
12. Dispatcher Component → send content → (Consumer):Consumer Delivery

Search content in the catalog The consumer executes this use case if he wants to search content in the catalog.

1. Consumer → search → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → search → (Content Service):Service Interface

Obtain content information

1. (Consumer) → get content information → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → get content information → (Content Service):Service Interface

Register user The consumer executes this use case if he wants to register him with the publishing system.

1. (Consumer) → register → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → register → (Consumer Service):User Subscription
4. Consumer Service → register → (User Management System):User Subscription

Register device The consumer executes this use case if he wants to register his device for obtaining content.

1. (Consumer) → registerDevice → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → registerDevice → (Consumer Service):User Subscription
4. Consumer Service → registerDevice → (User Management System):User Subscription

List obtained content The consumer executes this use case if he wants to obtain a list of obtained content.

1. Consumer → list obtained content → (Dispatcher Component):External Message Interface
2. Dispatcher Component → list obtained content → (Accounting Service):Accounting Interface

List obtained licenses The consumer executes this use case if he wants to obtain a list of obtained licenses.

1. (Consumer) → list obtained licenses → (Dispatcher Component):External Message Interface
2. Dispatcher Component → list obtained licenses → (Accounting Service):Accounting Interface

Consume content The consumer executes this use case if he wants to consume content.

Assumption: the user hands over content

Assumption: submitMessageging is done before decryption

1. Consumer → consume content → (Dispatcher Component):External Message Interface
2. Dispatcher Component → getContext → (Context Component):Context Component Interface
3. Dispatcher Component → submitMessage → (Logging Component):Message Interface
4. Dispatcher Component → deprotect → (Representation):Convert

A.3 Producer

Obtain statistics The producer executes this use case if he wants to obtain statistics about his content.

1. Producer → getStatisticalInformation → (Dispatcher Component):External Message Interface

   Optional step

2. Dispatcher Component → submitMessage → (Logging Component):Message Interface
3. Dispatcher → obtains statistics → (Accounting Service):Accounting Interface

A.4 Publisher

obtain statistics The publisher executes this use case if he wants to obtain statistics about content.

1. Producer → getStatisticalInformation → (Dispatcher Component):External Message Interface

   Optional step

2. Dispatcher Component → submitMessage → (Logging Component):Message Interface

3. Dispatcher → obtains statistics → (Accounting Service):Accounting Interface

Identify source of abuse The publisher executes this use case if he wants to identify the user who has spread content on mass distribution networks.

1. Publisher → identify → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging Component):Message Interface
3. Dispatcher Component → identify → (Identification Service):Identification

A.5 Licensing perspective

A.5.1 Consumer

Obtain license We have two variants of the obtain license use case. In the first variant, the license is already coupled with the content. The second variant first couples a general license model with the content. Both use cases are from a publisher point of view.

Assumption: The license has already the right representation

Assumption: The license is already coupled with the content

1. Consumer → get license → (Dispatcher Component):External Message Interface
2. Dispatcher Component → get license → (License Service):Service Interface
3. License Service → deliver license → (Dispatcher Component):External Message Interface
4. Dispatcher Component → deliver license → (Consumer):Consumer Delivery

Assumption: The license is not coupled with the content.

1. Consumer → get license → (Dispatcher Component):External Message Interface
2. Dispatcher Component → get license → (License Service):Service Interface
3. License Service → adapt representation → (Dispatcher Component):External Message Interface

this message contains an identifier of the content wherefore the license is requested and some other information (e.g. whether the content service is anonymous or not, etc...)

4. Dispatcher Component → obtain Context Component → (Context Component):Context Interface

5. ***Assumption***: *Content Service stores key material*

nously content issuing

6. Context Component → getIdentity → (UMS):Identity Interface

7. Context Component → getProtectionMaterial → (Content Management System):Ready Content

   The next step ensures that the license contains the protection material of the content, which was requested by the Context Component.

8. Dispatcher Component → adapt representation (Representation):Convert/Create

9. License Service → deliver license → (Dispatcher Component):External Message Interface

10. Dispatcher Component → deliver license → (Consumer):Consumer Delivery

Obtain license information The consumer executes this use case if he wans to obtain information about a license. This information includes restrictions in a readable form.

1. (Consumer) → get license information → (Dispatcher Component):External Message Interface

2. Dispatcher Component → submitMessage → (Logging):Message Interface

3. Dispatcher Component → get license information → (License Service):Service Interface

Update license The consumer executes this use case if he wants to update his license. For instance, if it was expired.

1. (Consumer) → update license → (Dispatcher Component):External Message Interface

2. Dispatcher Component → submitMessage → (Logging):Message Interface

3. Dispatcher Component → update license → (License Service):Service Interface

Negotiate about license The consumer executes this license if he wants to negotiate about the license restrictions.

1. (Consumer) → update license → (Dispatcher Component):External Message Interface

2. Dispatcher Component → submitMessage → (Logging):Message Interface

3. Dispatcher Component → update license → (License Service):Service Interface

Recover license The consumer executes this use case if he wants to recover a lost license.

1. (Consumer) → recover license → (Dispatcher Component):External Message Interface

2. Dispatcher Component → submitMessage → (Logging):Message Interface

3. Dispatcher Component → recover license → (License Service):Service Interface

4. License Service → getLicense → (Content Management System):Ready Content

   Optional step(License Service) → hasObtained → (Accounting Service):Accounting Interface

Consume content The consumer executes this use case if he wants to consume content.

   Assumption: the user hands over content and license

   Assumption: representation of license is understandable by the Policy Engine Component

   Assumption: submitMessageging is done before decryption

1. Consumer → consume content → (Dispatcher Component):External Message Interface

2. Dispatcher Component → getContext → (Context Component):Context Component Interface
3. Dispatcher Component → interpretPolicy → (Policy Engine Component):Interpretation

   License ok

4. Dispatcher Component → submitMessage → (Logging Component):Message Interface
5. Dispatcher Component → deprotect → (Representation):Convert

A.5.2 Producer

Upload license The producer executes this use case if he wants to submit a license to the online DRM system.

1. Producer → submitInput → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitInput → (License Service):Management Interface

Create license The producer executes this use case if he wants to create a license.

1. Producer → create → (Dispatcher Component):External Message Interface
2. Dispatcher Component → create → (Representation):Create/Adapt

Cancel license The producer executes this use case if he wants to cancel a license.

1. Producer → cancel license → (Dispatcher Component):External Message Interface
2. Dispatcher Component → cancel license → (License Service):License Management Interface

Update license The producer executes this use case if he wants to update a license.

1. Producer → update license → (Dispatcher Component):External Message Interface
2. Dispatcher Component → update license → (License Service):License Management Interface

A.6 Security perspective

A.6.1 Consumer

Pull content Protection at obtaining, non transparent

1. DRM Client → getContent → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → getContent → (Content Service):Service Interface

   The next step is executed in case of non-anonymous content consumption.

4. Content Service → obtainIdentity → (User Management System):CurrentIdentity
5. Content Service → getContent → (Content Management System):Ready Content

   Optional conversion step

6. Content Service → convert → (Dispatcher Component):External Message Interface
7. Dispatcher Component → convert → (Representation Component):Message Interface
8. Content Service → protect → (Dispatcher Component):External Message Interface
9. Dispatcher Component → protect content → (Representation Component)
10. Representation Component → protect → (Dispatcher Component):Internal Message Interface

11. Dispatcher Component → protect → (Orchestrator):Policy
12. Orchestrator → watermark → (Watermarking Component):Add
13. Orchestrator → encrypt → (Secure Data Enveloping Component):Envelop
14. Content Service → send content → (Dispatcher Component):External Message Interface
15. Dispatcher Component → submitMessage → (Logging):Message Interface
16. Dispatcher Component → send content → (Consumer):Consumer Delivery

Protection at obtaining, transparent

1. DRM Client → getContent → (Dispatcher Component):External Message Interface
2. Dispatcher Component → submitMessage → (Logging):Message Interface
3. Dispatcher Component → getContent → (Content Service):Service Interface

   The next step is executed in case of non-anonymous content consumption.

4. Content Service → obtainIdentity → (User Management System):CurrentIdentity
5. Content Service → getContent → (Content Management System):Ready Content
6. Content Service → send content → (Dispatcher Component):External Message Interface
7. Dispatcher Component → protect content → (Representation Component)
8. Representation Component → protect → (Dispatcher Component):Internal Message Interface
9. Dispatcher Component → protect → (Orchestrator):Policy
10. Orchestrator → watermark → (Watermarking Component):Add
11. Orchestrator → encrypt → (Secure Data Enveloping Component):Envelop

    Optional submitMessageging step

12. Dispatcher Component → submitMessage → (Logging):Message Interface
13. Dispatcher → send content → (Consumer):Consumer Delivery

Consume content

Assumption: the user hands over content and license

Assumption: representation of license is understandable by the Policy Engine Component

Assumption: submitMessageging is done before decryption

1. Consumer → consume content → (Dispatcher Component):External Message Interface
2. Dispatcher Component → getContext → (Context Component):Context Component Interface
3. Dispatcher Component → interpretPolicy → (Policy Engine Component):Interpretation

   License ok

4. Dispatcher Component → submitMessage → (Logging Component):Message Interface
5. Representation Component → protect → (Dispatcher Component):Internal Message Interface
6. Dispatcher Component → deprotect → (Representation):Convert
7. Dispatcher Component → deprotect → (Orchestrator):Policy
8. Orchestrator → watermark → (Watermarking Component):Add
9. Orchestrator → encrypt → (Secure Data Enveloping Component):Envelop

A.6.2 Producer

Submit content Protection at submitting, transparent

1. Producer → submit content → (Dispatcher Component):External Message Interface
2. Dispatcher Component → protect → (Representation):Create/convert

   Following steps depend on used security techniques

3. Representation Component → protect → (Dispatcher):Internal Message Interface

4. Dispatcher → protect → (Orchestrator):Policy
5. Orchestrator → watermark → (Watermarking Component):Add
6. Orchestrator → encrypt → (Secure Data Enveloping Component):Envelop

   Following step puts content and license protection data into the CMS

7. Dispatcher → submit content and protection data → (IMS):Input Interface

   Following step puts content and license protection data into the CMS

8. Input Management System → submitContent → (Content Management Service):CMS Input Interface

Protection at submitting, not transparent

1. Producer → submit content → (Dispatcher Component):External Message Interface
2. Dispatcher → submit content → (IMS):Input Interface
3. IMS → protect → (Dispatcher Component):External Message Interface
4. Dispatcher Component → protect → (Representation):Create/convert

   Following steps depend on used security techniques

5. Representation Component → protect → (Dispatcher):Internal Message Interface
6. Dispatcher → protect → (Orchestrator):Policy
7. Orchestrator → watermark → (Watermarking Component):Add
8. Orchestrator → encrypt → (Secure Data Enveloping Component):Envelop

   Following step puts content and license protection data into the CMS

9. Dispatcher → submit content and protection data → (IMS):Input Interface

   Following step puts content and license protection data into the CMS

10. Input Management System → input content → (Content Management Service):CMS Input Interface

Protect content Protection at producer side, explicitly

1. Producer → protect → (Dispatcher Component):External Message Interface
2. Dispatcher Component → Representation Component):Create/convert

   Following steps depend on used security techniques

3. Representation Component → protect → (Dispatcher):Internal Message Interface
4. Dispatcher → protect → (Orchestrator):Policy
5. Orchestrator → watermark → (Watermarking Component):Add
6. Orchestrator → encrypt → (Secure Data Enveloping Component):Envelop

B Interfaces of the service component group components

B.1 Content Service

The *Content Service* is used by producers to spread content to consumers. It has the following interfaces:

- **Consumer Delivery:** This interface is used to trigger the release of content. It is a notification that certain content was added. In a push-case, the *Content Service* fetches the content and send it to all consumers that are subscribed to this *Content Service*.
- **Service Interface:** This interface is offered to the consumer for browsing through content, obtaining content, etc.

B.1.1 Consumer Delivery

This interface is used to trigger the release of content. It has the following methods:

- *deliver(String contentId)*: this method is used to notify certain content was added.

deliver(String contentId) This method is used to notify certain content was added. **Parameters**

- *contentId*: the id of the content that was added.

Return Value

- None.

Preconditions

- There exist content with an id equal to *contentId*.

Postconditions

- The content with id equal to *contentId* is send to the interested consumers.

B.1.2 Service Interface

This interface is offered to the consumer for browsing through content, obtaining content, etc. It has the following methods:

- *getContent(contentId: String): Content*: this method returns the content with given id.
- *search(query:String):List*: This method is used to obtain a list of ids of the content which matches the given criteria (query).
- *getContentInformation(contentId:String):Metadata*: This method is used to obtain metadata of content.

getContent(contentId: String): Content This method returns the content with given id.

Parameters

- *contentId*: the id from the content to be obtained.

Return Value

- the content with id equals to *contentId* is returned.

Preconditions

- There exists a content with contentid equal to *contentId*.
- optional The user who obtains content is known by the system.

Postconditions

- The user who requested the content has a copy of the content with contentid equal to *contentId*.

search(query:String):List This method is used to obtain a list of ids of the content which matches the given criteria (query). This method can also be used for obtaining a list of all content.

Parameters

- *query*: the search criteria of content we are looking for.

Return Value

- If no content matches, an empty list is returned.

- A list with the ids of the content which matches the given criteria.

Preconditions

- the query is valid
- optional The user who performs the request is known by the system.

Postconditions

- The user who requested the list has a list of *contentids* from which the metadata of the corresponding content matches the search criteria (query).

getContentInformation(contentId:String):Metadata This method is used to obtain metadata of content.

Parameters

- *contentId*: the id of the content where one wants the information from.

Return Value

- Information about the content with contentid equals to *contentId*.

Preconditions

- There exist content with id equal to *contentId*.
- optional The user who performs the request is known by the system.

Postconditions

- The user who requested the content information of content with contentid equals to *contentId*, has the metainformation of that content.

B.2 Input Management System

The *Input Management System* (*Import Service*) is the publishing component that handles incoming content, such as books, music, and videos, that is uploaded to the system by the producer. After initial processing, the publisher is notified and content is submitted to the *Content Management System*.

This component offers the following interfaces:

- **Content Input:** This interface is used by producers to send in their content.

B.2.1 Content Input

This interface is used by producers to send in their content. It has the following methods:

- *submitInput(Content content):String*: This method submits content to the *Input Management System* and returns the id of the submitted content.

submitInput(Content content):String This method submits content to the *Input Management System* and returns the id of the submitted content. Examples of content can be text, pictures, or other types of media. A first amount of metadata can already be added to the content at the moment of uploading.

Parameters

- *content*: the content that is uploaded.

Return Value

- the id of the submitted content.

Preconditions

- This producer has the right to upload content.

Postconditions

- The content is uploaded to the system.
- The content will be sent to the *Content Management System*.

B.3 Content Management System

All content, such as books, music, movies and corresponding licenses, are stored and organized centrally in the *Content Management System*. The *Content Service* retrieves content from the *Content Management System* and makes it available to the consumer.

This component offers the following interfaces:

- **Ready Content:** Once the publisher approve content, this interface can be used to retrieve this content in an easy way.
- **CMS Input Interface:** New content is passed to the *Content Management System* via this interface.
- **Content In Progress:** The publisher can manipulate content via this interface. For instance, he indicates that content is ready for publication.

B.3.1 Ready Content

This interface can be used to retrieve approved content in an easy way. It has the following methods:

- *getContent(String contentId):content*: This method retrieves content from the *Content Management System*.

getContent(String contentId):content This method retrieves content from the *Content Management System*.

Parameters

- *contentId*: the id of the content to be retrieved.

Return Value

- The content with id equal to *contentId*.

Preconditions

- The *Content Management System* has content with id equal to *contentId*.

Postconditions

- The system which requested content has received content with id equal to *contentId*.

B.3.2 CMS Input Interface

New content is passed to the *Content Management System* via this interface. It has the following methods:

- *submitContent(Content content):String*: This method adds content to the *Content Management System* and returns the id of the added content.

submitContent(Content content):String This method adds content to the *Content Management System* and returns the id of the added content.

Parameters

- *content*: the content to be added.

Return Value

- The id (contentId) of the added content.

Preconditions

- None.

Postconditions

- The content is added to the system.
- The added content has id equal to *contentId*.

B.3.3 Content In Progress

The publisher can manipulate content via this interface. For instance, he indicates that content is ready for publication. This interface is not worked out, because it is not important for this version of our DRM architecture.

B.4 User Management System

The *User Management System* is the component that handles all information about all of the publishing system users, which are the consumers, the publishers and the producers. This information contains amongst others contact information, such as the name and address. This component is also knows the users who currently make use of the system. It offers the following interfaces:

- **Current Identity:** This interface is used for obtaining the identity of the user that currently makes use of the system.
- **User Profile:** This interface is used for reading and changing the profile of a user.

B.4.1 Current Identity

This interface is used for obtaining the identity of the user that currently makes use of the system. It has the following methods:

- *obtainIdentity(String:id):Identity*: This method returns the identity of the user current accessing the system (e.g. by SSO).

obtainIdentity(String:id):Identity This method returns the identity of the user current accessing the system (e.g. by SSO).

Parameters

- *id*: the id of the the identity to be obtained.

Return Value

- Identity: the current identity

Preconditions

- There exist a user with sessionid id.

Postconditions

- The component applying this method knows which user is currently using his interfaces.

B.4.2 User Profile

This interface is used for reading and changing the profile of a user.

getUserInformation(String:userId):Identity This methos returns information of user with given id.

Parameters

- *userId*: the id of the user from which we want more information.

Return Value

- The extra information.

Preconditions

- The component/user requesting this information is allowed to do so.

Postconditions

- The component/user has extra information about the user with id equals to userId.

B.5 Identification Service

The *Identification Service* is the component that identifies users, devices, groups, and content. This service can identify for instance the person who has spread certain content. This component offers one interface:

- **Identification**: Content, e.g. found on mass distribution networks, is passed via this interface.

B.5.1 Identification

Content, e.g. found on mass distribution networks, is passed via this interface. It has the following methods:

- *revealIdentity(content:Content):Identity*: This method reveals the identity of the user who spread the given content.
- *identifyContent(content:Content):String*: This method identifies content, e.g. if you want to know information about unknown (unidentified) content.

revealIdentity(content:Content):Identity This method reveals the identity of the user who spread the given content.

Parameters

- *content*: the content.

Return Value

- the identity of the user who spread the content (if any).

Preconditions

- The user who wants to reveal the identity is known by the system.
- The user is authorized to reveal the identity.

Postconditions

- The identity of the identity who spread the content is revealed.

identifyContent(content:Content):String This method identifies content, e.g. if you want to know what content something is.

Parameters

- *content*: the content to be identified.

Return Value

- The id of the content (if known).

Preconditions

- The user who wants to identify content is known by the system.
- The user is authorized to identify content.

Postconditions

- The user knows what content it is.

B.6 Payment Service

The *Payment Service* (Billing System) is a publishing component that keeps information about all payment methods and all *External Financial Institutions*. It also keeps track of pending payments: these are payments which are requested from the *External Financial Institution*, but not yet confirmed. It is also responsible forwarding the payment notifications from the *External Financial Institutions* to the interested parties. It offers thefollowing interfaces:

- **Billing Interface:** This interface is used to request payments from users. As discussed before, payment can be synchronous or asynchronous. The *Payment Service* internally knows which *External Financial Institution* to address in order to finish the payment correctly.
- **Payment Notification:** This interface is used by the *External Financial Institutions* to notify the publisher when a payment transaction has been completed successfully.

The interfaces of this component are not worked out, because it is out of scope for this report.

B.7 License Service

The licenses issued to the consumer are already coupled to the content or can be coupled on the fly. The latter has as advantage that some general license models are possible for a whole content range.

This component offers the following interfaces:

- **Consumer Interface**: This interface is used by the consumers (e.g. end users from distributors, distributors from others, etc.) for obtaining, recovering and renewing licenses.
- **License Management Interface**: This interface is used by producers to manage licenses. He can revoke, remove or update licenses. He has also the possibility to couple licenses and content. Other services can potentially submit new licenses.

B.7.1 Consumer Interface

This interface is used by the consumers (e.g. end users from distributors, distributors from others, etc.) for obtaining, recovering and renewing licenses. It has the following methods:

- *getLicense(licenseId:String,params:Params):License*: this method is used by the consumer to obtain a license.
- *getLicenseTypes(contentId:String):List*: is used by the consumer to get the different licensetypes for certain content.
- *search(query:String):List*: This method returns a list with the ids of the licenses which match the query (e.g. all licenses for content with a certain id valid after a certain period).
- *getLicenseInformation(licenseId:String):String*: This method returns a textual description of a license.
- *recoverLicense(userId:String,licenseId:String):License*: This method generates a new license for the user who lost the original license.
- *updateLicense(userId:String,licenseId:String,newLicense:License):License*: This method updates an expired license.

getLicense(licenseId:String,params:Params):License This method is used by the consumer to obtain a license.

Parameters

- *licenseId*: the id of the license to be obtained
- *params*: optional parameters

Return Value

- The license with id equal to *licenseId*.

Preconditions

- There exist a license with id equal to *licenseId*.
- optional The user who wants to obtain the license is known by the system.
- optional The user is authorized to obtain the license.

Postconditions

- The users has a license with id equal to *licenseId*.

getLicenseTypes(contentId:String):List This method is used by the consumer to get the different licensetypes for certain content.

Parameters

- *contentId*: the id of the content

Return Value

- A list with the ids of the available licenses for certain content.

Preconditions

- There exist content with id equal to *contentid*.
- There are licenses associated with the content.
- optional The user who wants to obtain a list is known by the system.
- optional The user is authorized to obtain a list.

Postconditions

- The user has a list of licenses which are valid for the content with given id.

search(query:String):List This method returns a list with the ids of the licenses which match the query (e.g. all licenses for content with a certain id valid after a certain period).

Parameters

- *query*: the search query

Return Value

- A list with the ids of the licenses which match the given criteria.

Preconditions

- The query is valid.
- optional The user who wants to search is known by the system.
- The user is authorized to search.

Postconditions

- The user has a list of ids of licenses which match the given criteria.

getLicenseInformation(licenseId:String):String This method returns a textual description of a license.

Parameters

- *licenseId*: the id of the license.

Return Value

- a textual description of the license.

Preconditions

- there exists a license with id equal to *licenseId*.
- optional The user who wants to obtain the license is known by the system.
- The user is authorized to obtain license information.

Postconditions

- The user has a textual description of the license with id equal to *licenseId*.

recoverLicense(userId:String,licenseId:String):License This method generates a new license for the user who lost the original license.

Parameters

- *userId*: the id of the user wherefor the license should be recovered.

Return Value

- A license.

Preconditions

- There exists a license with id equal to *licenseId*.
- The user has previously obtained the license with id equal to *licenseid*.
- The user is known by the system.
- The user is authorized to recover licenses.

Postconditions

- The user has obtained a license.

updateLicense(userId:String,licenseId:String,newLicense:License):License
This method updates the license for the consumer.

Parameters

- *userId*: the user who wants to update the license (if known by the system, this should be left out).
- *licenseId*: the id of the license which must be updated.
- *newlicense*: the proposal (optional).

Return Value

- the updated license.

Preconditions

- There exists a license with id equal to *licenseid*.
- The user who wants to update the license known by the system.
- The user is authorized to update the license.

Postconditions

- The user received a new license with given id

B.7.2 License Management Interface

This interface is used by producers to manage licenses. He can revoke, remove or update licenses. He has also the possibility to couple licenses and content. Other services can potentially submit new licenses. It has the following methods:

- *addLicense(license:License):boolean*: This method is used to add a license model (contract).
- *removeLicense(licenseId:String):boolean*: This method is used to remove an existing license.
- *updateLicense(String oldLicenseId,License newlicense):boolean*: This method is used to update a license.

addLicense(license:License):boolean This method is used to add a license model (contract).

Parameters

- *license*: the license that must be added

Return Value

- True if the license can be added, false otherwise.

Preconditions

- The license is valid.
- The user who wants to submit the license is known by the system.
- The user is authorized to submit the license.

Postconditions

- The system has processed the license.

removeLicense(licenseId:String):boolean This method is used to remove an existing license.

Parameters

- *licenseId*: the id of the contract to be removed.

Return Value

- True if the license can be removed, false otherwise.

Preconditions

- There exists a contract with id equal to *licenseId*.
- The user who wants to remove the license is known by the system.
- The user is authorized to remove the license.

Postconditions

- The system removed the license with id equal to *licenseId*.

updateLicense(String oldLicenseId,License newlicense):boolean This method is used to update a license.

Parameters

- *oldLicenseId*: the id of the license to be updated.
- newlicense: the new license.

Return Value

- True if the contract was updated, false otherwise.

Preconditions

- There exists a license with id equal to *licenseId*.
- *newlicense* is a valid license.
- The user who wants to update the contract is known by the system.
- The user is authorized to update the license.

Postconditions

- The system has updated the license with id equal to *oldLicenseId*.

B.8 Accounting Service

The *Accounting Service* is used by publishers and producers to collect information about (actions executed on) content. It has the following interfaces:

- **Tracking Interface** This interface is used by the logging components at the RI-layer to submit statistical data.
- **Accounting Interface** This interface is used by other components to generate reports, obtain statistical data for billing, etc.

B.8.1 Tracking Interface

This interface is used by the logging components at the RI-layer to submit statistical data. It has the following methods:

- *submitUsageData(params:Params):boolean*: This method is used to submit usage information.

submitUsageData(params:Params):boolean This method is used to submit usage information.

Parameters

- *params*: the usage data to be stored.

Return Value

- true if the usage data is successfully stored, false otherwise.

Preconditions

- The user who wants to submit usage data is known by the system.
- The user is authorized to submit usage data.

Postconditions

- The system has stored usage data.

B.8.2 Accounting Interface

This interface is used by other components to generate reports, obtain statistical data for billing, etc. It has the following methods:

- *getStatisticalInformation(logId:String):Params*: This method returns detailled statistical information.
- *search(query:String):List*: This method returns a list of entryids which matches the given criteria.
- *listBoughtContent(userid:String):List*: This method returns the ids of the content a user has bought.
- *listObtainedLicenses(userid:String):List*: This method returns the ids of the licenses a user has obtained.

getStatisticalInformation(logId:String):Params This method returns detailled statistical information.

Parameters

- *logId*: the id of the entry to be returned.

Return Value

- the statistical information.

Preconditions

- There exists an entry with id equal to *logId*.
- The user who wants to retrieve the information is known by the system.
- The user is authorized to retrieve information.

Postconditions

- The user has obtained statistical information.

search(query:String):List This method returns a list of entryids which matches the given criteria.

Parameters

- *query*: the query.

Return Value

- a list of ids of entries which matches the given criteria.

Preconditions

- There query is valid.
- The user who wants to retrieve the list is known by the system.
- The user is authorized to retrieve the list.

Postconditions

- The user has obtained a list of ids of entries which matches the given criteria.

listBoughtContent(userid:String):List This method returns the ids of the content a user has bought. This is a combination of search and getStatisticalInformation.

listObtainedLicenses(userid:String):List This method returns the ids of the licenses a user has obtained. This is a combination of search and getStatisticalInformation.

C Interfaces of the RI component group components

C.1 Dispatcher Component

This component dispatches the message to the right component group. It has the following interfaces:

- **Internal Message Interface**: this interface is used by components of the RI component group to send messages to each other.
- **External Message Interface**: this interface is used by other component groups to send messages to each other.

C.1.1 Internal Message Interface

this interface is used by components of the RI component group to send messages to each other. It contains the following methods:

- *register(String identity)*: This method is used to register a component with a certain identity.
- *setInterceptionPolicy(Policy policy)*: This method is used by the registered components to specify what, how and when they want to intercept messages.
- *unregister()*: This method is used by the components to undo a registration.
- *submitMessage(Message message)*: This method is used to submit a message to the *Dispatcher Component*.

register(String identity) This method is used by the internal RI components to register themselves. **Parameters**

- *identity*: the identity used to register yourself. (This should be generic, a certificate can be used also).

Return Value

- None.

Preconditions

- None.

Postconditions

- The requesting component has registered itself by the *Dispatcher Component*.

setInterceptionPolicy(Policy policy) This method is used by internal RI components to to specify which, how and when they want to intercept messages. **Parameters**

- *policy*: the policy which specifies when, what and how intercepted messages should be send to the components which registers.

Return Value

- None.

Preconditions

- The registering component has already registered.

Postconditions

- The interception policy is set.

unregister() This method is used by the internal RI components to undo a registration. **Parameters**

- None.

Return Value

- None.

Preconditions

- The component which want to undo registration has registered itself.

Postconditions

- The component is not registered anymore.

submitMessage(Message message) This method is used by the internal RI components to send a message. **Parameters**

- *message*: the message to be send.

Return Value

- None.

Preconditions

- The component which want send a message is registered.

Postconditions

- The message is send to its destination.

C.1.2 External Message Interface

This interface is used by external components to send and receive data. They do not have to register themselves. It contains only one method:

- *submitMessage(Message message)*: This method is used by the external components to send a message.

submitMessage(Message message) Parameters

- *message*: the message to be send.

Return Value

- None.

Preconditions

- The component which want send a message is registered.

Postconditions

- The message is send to its destination.

C.2 Logging Component

C.2.1 Message Interface

This interface is used by other components to send messages to. It contains only one method:

- *submitMessage(Message message)*: This method is used by other components to send messages to the *Logging Component*. The *Logging Component* extracts the needed data out of the message.

submitMessage(byte[] message) Parameters

- *message*: the message to be send.

Return Value

- None.

Preconditions

- The component submitting a message is the *Dispatcher Component*.

Postconditions

- The message is logged by the *Logging Component*.

C.3 Policy Engine Component

C.3.1 Policy Interpretation Interface

This interface contains the following methods:

- *interpretPolicy(Context context, Policy policy):boolean*: This method is used to interpret a policy.

interpretPolicy(Context context, Policy policy):boolean Parameters

- *context*: the context needed to interpret a policy (including subject and action).
- *policy*: the policy which needed to be interpret.

Return Value

- True if the *Policy Engine Component* subject is allowed to do an action, false otherwise

Preconditions

- None.

Postconditions

- The *Policy Engine Component* has given a true/false answer.

C.3.2 Message Interface

This interface is used by other components to send messages to. It contains only one method:

- *submitMessage(Message message)*: This method is used by other components to send messages to the *Logging Component*. The *Logging Component* extracts the needed data out of the message.

submitMessage(Message message) Parameters

- *message*: the message to be send.

Return Value

- None.

Preconditions

- The component submitting a message is the *Dispatcher Component*.

Postconditions

- The message is logged by the *Logging Component*.

C.4 Representation Component

C.4.1 Convert/Create

This interface contains the following methods:

- *protect(byte[] data):byte[]*: This method is used to protect data.
- *unprotect(byte[] data):byte[]*: This method is used to unprotect data.
- *convert(byte[] data, String representation):byte[])*: This method is used to convert a representation to another.
- *create*: This method is used to create a representation.
- *adapt*: This method is used to adapt a representation.

These methods are not worked out in full detail.

C.4.2 Representation Policy

This interface is used to add a possible representation to the *Representation Component*. It contains the following methods:

- *addPolicy(Policy policy)*: This method is used to add a representation policy (e.g. MPEG-21 representation).
- *removePolicy(String policy)*: This method is used to remove a policy.

These methods are not worked out in full detail.

C.4.3 Message Interface

This interface is used by other components to send messages to. It contains only one method:

- *submitMessage(Message message)*: This method is used by other components to send messages to the *Logging Component*. The *Logging Component* extracts the needed data out of the message.

submitMessage(Message message) **Parameters**

- *message*: the message to be send.

Return Value

- None.

Preconditions

- The component submitting a message is the *Dispatcher Component*.

Postconditions

- The message is logged by the *Logging Component*.

C.5 Context Component

C.5.1 Message Interface

This interface is used by other components to obtain some context. This component tries to figure this out (e.g. by contacting other components).

It has the folling methods:

- *getContext(String query):Context*: This method obtains the context specified in the query.

getContext(String query):Context This method obtains the context specified in the query.

Parameters

- *query*: which context do we need?

Return Value

- Context.

Preconditions

- None.

Postconditions

- The requested context is retrieved (if possible).

D Interfaces of the security component group components

These interfaces are minimal required. Most of them are worked out in detail.

D.1 Fingerprinting Component

The *Fingerprinting Component* has only one interface.

- **Generate Fingerprint**, which is used to calculate a fingerprint.

D.1.1 Generate Fingerprint

This interface is used to generate a fingerprint. It has the following methods:

- *void update(content:byte[]*: This method adds content to the *Fingerprinting Component*.
- *fingerprint():byte[]*: This method fingerprints the content added by the previous method.

void update(content:byte[]) This method adds content to the *Fingerprinting Component*.

Parameters

- *content*: the content to be added. Bytes, because the *Fingerprinting Component* interprets it.

Return Value

- None.

Preconditions

- The **Fingerprinting Component** is able to handle content.

Postconditions

- None.

fingerprint():byte[] This method fingerprints the content added by the previous method. **Parameters**

- None.

Return Value

- A fingerprint for the previously added content.

Preconditions

- The **Fingerprinting Component** has already received the content.

Postconditions

- The component/user of this method has received the fingerprint of the previously submitted content.

D.2 Watermarking Component

This component makes it possible to embed watermarks into and to read watermarks from content. It has the following interfaces:

- **Read Watermark**: this interface is used to read watermarks from content.
- **Add Watermark**: this interface is used to add watermarks to content.

D.2.1 Read Watermark

This interface is used to read watermarks from content. It has the following methods:

- *read():byte[]*: This method reads the watermark.
- *contains():boolean*: This method verifies whether the media contains the watermark(s).

read():byte[] This method reads the watermark. **Parameters**

- None.

Return Value

- The watermark of the previously added content.

Preconditions

- The **Watermarking Component** has already received the content.
- The content contains a watermark.

Postconditions

- The component/user of this method has received the watermark of the previously submitted content.

contains():boolean This method verifies whether the media contains the watermark(s). **Parameters**

- None.

Return Value

- true if the previously added content contains a watermark.

Preconditions

- The **Watermarking Component** has already received the content.

Postconditions

- The component/user of this method has received the result.

D.2.2 Add Watermark

This interface is used to add watermarks to content. It contains the following methods:

- *add():byte[]*: This method returns the content with embedded watermark(s).
- *init(params:Param)*: This method initialized the watermarking algorithm with essentials parameters.
- *updateContent(byte[] content)*: This method adds (a part of the) content to the *Watermarking Component*.
- *updateWatermark(byte[] watermark)*: This method adds a (part of the) watermark to the *Watermarking Component*.

add():byte[] This method returns the content with embedded watermark(s). Parameters are taken from the init-method.

Parameters

- None.

Return Value

- Previously added content interweaved with previously added watermark.

Preconditions

- The **Watermarking Component** has already received the content.
- The **Watermarking Component** has already received the watermark.

Postconditions

- The component/user of this method has received the result.

init(params:Param) This method initialized the watermarking algorithm with essentials parameters.

Parameters

- *param*: the initialization parameters of the watermarking algorithm.

Return Value

- None.

Preconditions

- None.

Postconditions

- The *Watermarking Component* is initialized.

updateContent(byte[] content) This method adds (a part of the) content to the *Watermarking Component*. One could use this method repeatedly to send content to the *Watermarking Component*. **Parameters**

- *content*: the content to be added.

Return Value

- None.

Preconditions

- The *Watermarking Component* is initialized.

Postconditions

- The *Watermarking Component* has the content, which can be manipulated afterwards.

updateWatermark(byte[] watermark) This method adds a (part of the) watermark to the *Watermarking Component*. **Parameters**

- *watermark*: some bytes of the watermark.

Return Value

- None.

Preconditions

- The *Watermarking Component* is initialized.

Postconditions

- The **Watermarking Component** has the content, which can be manipulated afterwards (e.g. interweave with content).

D.3 Certificate Management Component

This component is used to manage certificates. It has the following interfaces:

- **Certificate Management Interface**: this interface is used by components to store and retrieve certificates from a Certificate Authority (CA).

D.3.1 Certificate Management Interface

This interface is used by components to store and retrieve certificates from a Certificate Authority (CA). It has the following methods:

- *submitCertificate(cert:Certificate)*: This method submits a certificate to a CA.

- *recieveCertificate(param:Params):Certificate*: This method receives a certificate from a CA.

submitCertificate(cert:Certificate) This method submits a certificate to a CA. **Parameters**

- *cert*: the certificate that is submitted to the certificate authority.

Return Value

- None.

Preconditions

- *cert* is a valid certificate

Postconditions

- The certificate is submitted to the CA.

receiveCertificate(param:Params):Certificate This method receives a certificate from a CA. **Parameters**

- *param*: the needed parameters to receive a certificate.

Return Value

- The requested certificate

Preconditions

- None.

Postconditions

- The requester has received the requested certificate.

D.4 Key Management Component

Key Management Component is used to manage keys, which means storing and retrieving keys. This component has one interface, namely **Key Management Interface**.

D.4.1 Key Management Interface

This interface is used to store and retrieve keys from key stores. It has the following methods:

- *init(param:Params)*: This method initializes this component.
- *createKey(param:Params)*: This method creates a key pair.
- *storeKey(param:Params)*: This method stores a key(pair).
- *transformKey(param:Params):Key*: This method transforms a key(pair) from one representation to another.

init(param:Params) This method initializes key management. **Parameters**

- *param*: the parameters used to initialize the usage of the *Key Management Component*.

Return Value

- None.

Preconditions

- None.

Postconditions

- This component has been initialized for usage.

createKey(param:Params) This method creates a key(pair). **Parameters**

- *param*: the parameters needed to create a key

Return Value

- a key (for which the given parameters are used).

Preconditions

- the component has been initialized.

Postconditions

- the requester has received a key which has been generated by this component.

storeKey(param:Params) This method stores a key(pair). **Parameters**

- *param*: the parameters needed to store a keypair.

Return Value

- None.

Preconditions

- This module has been initialized.

Postconditions

- the key has been stored.

transformKey(param:Params):Key This method transforms a key(pair) from one representation to another. **Parameters**

- *param*: the parameters needed to transform a key.

Return Value

- the transformed key

Preconditions

- This component is initialized.

Postconditions

- None.

D.5 Digital Signature Component

This component is used to generate and verify digital signatures. It has only one interface:

- **Interface**: This interface is used to generate and verify digital signatures.

D.5.1 Interface

This interface is used to generate and verify digital signatures. It has the following methods:

- *init(params:Param*: This method is used to initialize the component.
- *update(data:byte[])*: This method is used to add the data to be signed or verified.
- *update(data:byte[])*: This method is used to add the data to be signed or verified.
- *sign():byte[]*: This method generates a digital signature.
- *verify():boolean*: This method verifies the digital signature.

init(params:Param This method is used to initialize the component. **Parameters**

- *params*: the parameters needed to initialize this component.

Return Value

- None.

Preconditions

- None.

Postconditions

- This component is initialized for usage.

update(data:byte[]) This method is used to add the data to be signed or verified. **Parameters**

- *data*: the data to be signed or verified.

Return Value

- None.

Preconditions

- The **Digital Signature Component** is initialized.

Postconditions

- The data is uploaded.

sign():byte[] This method generates a digital signature. **Parameters**

- None.

Return Value

- The signature.

Preconditions

- The **Digital Signature Component** has been initialized and received data to be signed via the update method.

Postconditions

- The requester has received a signature of previously added data.

verify():boolean This method verifies the digital signature. **Parameters**

- None.

Return Value

- True if the signature was correct, false otherwise.

Preconditions

- The **Digital Signature Component** has been initialized and received data to be verified via the update method.

Postconditions

- None.

D.6 Key store

D.6.1 Store/Load

Keys can be loaded and stored from various devices, databases, etc. The minimal interface is:

- *open(params:Params)*: This method initializes the key store.
- *close(params:Params)*: This method stores the key-entries.
- *addKey(key:Key):String*: This method adds the key to the database and returns keyid.
- *getKey(keyid:String):Key* This method retrieves the key from the database.
- *removeKey(keyid:String):boolean*: This method removes the key from the database.

D.7 Key generation

D.7.1 Generate

The minimal required methods to generate keys and keypairs are:

- *initialize(params:Params)*: This method is used to set the parameters for the key generation algorithm.
- *generateKey()*: This method generates a key.
- *generateKeyPair()*: This method generates a key pair.

D.8 Encryption

D.8.1 Encryption/Decryption

Encryption is used to protect data in a secure way. The minimal number of required methods is:

- *init(params:Params)*: This method adds the key (and other parameters).
- *update(data:byte[])*: This method adds data to the algorithm.
- *crypt():byte[]*: This method obtains the decrypted or encrypted data.

D.9 Hash

D.9.1 Hash

The minimal number of methods for hashes is:

- *update(data:byte[])*: This method gives the data to be hashed to the component.
- *digest():byte[]*: This method computes and returns the hash.

D.10 Secure Storage

D.10.1 Store/Retrieve/Delete

The minimal number of methods for secure storage is:

- *open(param:Params)*: This method accesses the device. Parameters such as key information can be given.
- *close(param:Params)*: This method terminates the access to the device.
- *read(param:Params)*: This method obtains information.
- *write(param:Params)*: This method writes information.

E DMP architecture

E.1 Context

The Digital Media Project[6] (DMP) is a non-profit association registered in Switzerland. Its aim is to develop the fundamentals of standardized and interoperable DRM for digital media. The vision of DMP is to make an architecture suitable for all the players on the value-chain and satisfactory for end-users. The key for achieving this goal is in standardizing DRM technology. This section mentions the DMP users and value chain, describes the general high level architecture, and gives the advantages and disadvantages of the DMP project.

E.2 Users

DMP introduces a value chain, which is a group of interacting users, connecting (and including) creators to end users with the purpose of delivering content to the end user.

A *creator* makes a work that is performed by the *performer*. This is used by the *producer* to make a product, which is published by the *publisher*. The resulting product is distributed by *content and service providers* (distributors) to the *end-user* and other users in the value chain.

The **creator** (Author, Composer, . . .) creates or adapts a work. In the book publishing industry, a creator typically gives his copyrights to a publishing company, in return for a percentage of royalties produced by the content. The creator(s) should have the possibility to submit his created content and possibly the assigned copyrights to the publishing company. The creator wants to submit a contract to the publishing company. He should also have the possibility to retrieve statistical information about his content. In the music industry, the creator should also have the possibility to negotiate with the instantiator.

The **instantiator** (Performers) performs a work. The instantiator should have the possibility to submit his created content and possibly the assigned copyrights to the publishing company. The instantiator wants to submit a contract to the publishing company ans should have the possibility to retrieve statistical information about his content. In the music industry, the instantiator should have the possibility to negotiate with the producer.

The **producer** produces content. This entity has the economically and technically responsibility to produce e.g. sound recordings. producers want to easily compose a contract. Both content and contract must be submitted to the online DRM system (e.g. publisher, distributor, content provider, etc.). After some time, they may want to update the contract or maybe even cancel it, i.e. stop the distribution of the content. Content producers want to obtain statistics about

the usage (by other business users or by end users) of their produced content. Producers wants to get paid for their produced content.

The **content provider** provides content or content with a license to another actor (e.g. end user). The content provider obtains content from other content providers, content integrators, producers, ... and offers it to end users, distributors, content providers, ... The content provider wants to have statistics about the content he distributed. The provider want to give their customers the possibility to search for content. He also wants to obtain up to date information about content that can be obtained from all content providers. They also want to give producers, integrators, ... the possibility to store content with them.

The **service provider** provides content or content with a license to another actor (e.g. the end user).

The **end user (consumer)** buys or rents a work or rights on it. End users want to consume protected content in a user-friendly way. They want to be able to browse the content catalog of the service provider where the content at stake can be obtained. Since consumers also need a license, they must be able to select a license type and view the usage rules , in a human readable format, associated with it. Generally, consumers first have to pay, one way or another; different business models should be possible (e.g. subscription, pay-per-license, or pay-per-use). When time-based licenses expire, it must be possible to update them, which may also require some financial transaction. Consumers also want to browse their obtained licenses locally and view the usage rules in a human readable format. Finally, consumers want to (must?) consume the protected content, according to the usage rules associated with the corresponding license. In order to fetch licenses (and sometimes also protected content), consumers need to authenticate to the license service and the content service (online DRM System).

Other roles include *registration authority* which ensures trust between devices.

E.3 Architecture

The DMP architecture defines users (e.g. consumers, producers, or publishers) as entities that perform so-called primitive functions, which represent the underlying DRM services that handle digital content (e.g. *Identify data* or *Authenticate user*) The set of standardized DRM tools based on the primitive functions is a toolkit called the Interoperable DRM Platform (IDP).

Different value chains could be build from interoperable functions, so that the service is interoperable. These functions are presented in the following subsections:

E.3.1 DMP Functions

Identification

- **Identify content**. The means by which the identity of content can be uniquely and unambiguously determined.
- **Identify device**. The means to identify devices employed in a particular instance of use.
- **Identify user**. The means to identify the user in a particular instance of use.
- **Identify domain**. The means by which the identity of a domain (groupings of users and/or devices) can be uniquely and unambiguously determined.
- **Identify DRM tool**. The means to identify executable code that implements one or more DRM functions on a device.
- **Identify Footprint**. The means to identify devices within primary broadcast distribution areas.
- **Identify Class of User**. The means to identify devices belonging to a particular class of users.
- **Identify jurisdiction**. The means to identify devices within a particular jurisdiction.

Content Service

- **Package as file**. The function of processing content for the purpose of delivering it between devices as file.
- **Package as broadcast**. The function of processing content for the purpose of delivering it between devices and broadcast.
- **Package as streaming**. The function of processing content for the purpose of delivering it between devices as stream.
- **Revoke content**. The function by which a user ceases to recognize a content item.
- **Revoke content element**. The function by which a user ceases to recognize a content element.

License Service

Access Service

- **Authenticate device**. The function of proving the identity of a device to another device, user or domain.
- **Authenticate user**. The function of proving the identity of a user to a device, another user or domain.
- **Authenticate domain**. The function of proving the identity of a domain to a device, a user or another domain.
- **Revoke device**. The function by which a user ceases to recognize a device as a device.
- **Revoke domain**. The function by which a user ceases to recognize a domain.
- **Revoke user**. The function by which a user ceases to recognize a device as a user.

Billing

- **Pay**. The function of providing use, user, device and governed content information to a payment system external to an environment.

Domain Management

- **Manage domain**. The function of managing a set of devices such that only those devices can use the content licensed to the domain.

Other

- **Certify content**. Assuring that a given content item is conformant to the DMP specifications.
- **Certify device**. Assuring that the claim by a device to be a DMP device is guaranteed.
- **Certify user**. Assuring that the claim by a device to represent a DMP user is guaranteed.
- **Test conformance of rights expressions**. Checking that a rights expression is interpreted and provides the output as intended by the originator of the rights expression.

- **Test conformance of enforcing rights expressions**. Checking that the functions corresponding to the output are executed as intended.
- **Test conformance of tamper resistance**. Defining the levels of tamper resistance and the methods to be used when an implementation is put under test for tamper resistance to determine such levels.

DRM Client Consuming

- **Render**. The function of generating human-perceivable signal.
- **Store**. The function by which device A delivers content to device B for the purpose of retaining it in device B for possible use at a different point in time.
- **Copy**. The function by which device A stores content in device B, preserving the original content in device A.
- **Move**. The function by which device A stores content in device B, deleting the original content in device A.
- **Backup**. The function by which a device can copy a content item to and from a device where the content item is not for use, e.g. for the purpose of later restoring the content item.
- **Export**. The function by which a device makes available a content item for use by a non DMP DRM System.
- **Access**. The function by which device A obtains content from device B so that device A can execute functions.
- **Restore**. The function by which a device can copy a content item to and from a device where the content item is not for use.
- **Import**. The function by which a device accesses a piece of content governed by a non DMP-DRM System.
- **Update**. The function by which a content item may replaced by a new content item.
- **Verify content**. The function by which a device detects corruption or loss of part of the content.
- **Verify device**. The function by which a device detects corruption of part of the software of a device.

producer Tool

- **Manage Use Data Confidentiality**. The means to allow user A to negotiate the way user B will utilize acquired user and use data of user A.

Presentation and Enforcement

- **Represent content**. The syntax of the information used to describe the organization and association of content and content elements in a form that can be processed by a device.
- **Represent identifier**. The syntax of the information used to describe an identifier in a way that can be processed by a device.
- **Represent resource**. The syntax of the information used to describe the format of a resource in a way that can be processed by a device.
- **Represent metadata**. The syntax of the information used to describe features and attributes associated with content and content elements or devices in a way that can be processed by a device.
- **Represent DRM information**. The syntax of the information used to describe governance of content.
- **Represent DRM Tool Body**. The syntax of the specific information associated with DRM Tool embodiments when these are sent to a device for being instantiated with the purpose of enabling access to governed content where those DRM tools are signalled.
- **Represent DRM Tool**. The syntax of the information used to describe DRM tools.
- **Represent rights expression**. The syntax of the information used to describe rights so that it can be processed by a device.
- **Represent rights data**. The means to represent the semantics of functions that relate to rights.
- **Represent device information**. The syntax of the information used to describe the capabilities of a device to perform functions.
- **Represent domain information**. The syntax of the information used to describe the attributes of a domain.

- **Represent use data**. The syntax of the information used to describe the elements making up one or more instance of use of content, device or user.
- **Represent resource format**. The syntax of the information used to describe the format of resources.
- **Assign identifier**. The function performed by a user when assigning an identifier to content, content element, device, domain and user.
- **Assign metadata**. The function performed by a user when describing
- **Authenticate DRM Tool**. The function of proving the identity of a DRM Tool to a device.
- **Authenticate content**. The function of proving the identity of a content item to a device.

Security Functions

- **Encrypt/decrypt**. Methods used to hide portions or totality of content elements.
- **Represent key**. The syntax of the information used to describe keys and associated parameters.

E.3.2 DMP Distribution and consumption Model

Distribution The typical distribution model (see figure 25 on page 125) includes: content providers providing content to service providers, service providers providing services to end-users, license providers providing licenses to end-users and DRM Tool providers providing DRM tools to end-users.

Several steps are accomplished to distribute a work. First, the producer creates content and chooses a corresponding license. Both entities needs to be uniquely identifiable. Therefore, he contacts the content/license identification device, located at a registration agency. Secondly, content and license are submitted to a content provider device and a license provider device respectively. Thirdly, the content/service provider device protects the content by using several toolpacks.

Content is delivered via several delivery mechanisms: file, broadcast and stream. After end users obtain content, either via a pull or push mechanism, they request licenses via license or service providers. Next, they request the missing DRM tools needed to consume the content. Finally, they consume the obtained content.

If order to consume content with a media player without a broadcast or Internet-connection (PAV-device), the consumer must connect the player with a device with such connection (PAV External Device). The connect device transfers content to this PAV-device.

Consumption The DMP consumption model assumes that SAV Devices can receive content that may contain a bundled license and the DRM tools that are required to use the content. While content can be obtained via a number of delivery mechanisms a file (DCF), a broadcast (DCB) or a stream (DCS) tools and license can only be obtained via a two-way delivery system.

Once the SAV has obtained content and necessary licenses and DRM tools (either because the latter were bundled within the content or they were individually accessed), it can play, store or adapt this content. The SAV can store, move or copy this content or an adaptation of this content to another SAV or PAV.

The devices (both PAVs and SAVs) in an environment (e.g. a home) can belong to a domain, which is managed by a special device called Domain Manager.

Each of these users can combine the following devices to satisfy their needs:

- **License identification device**. After a creator has finished making a license, he requests an unique identifier for it.
- **Content identification device**. After a creator has finished making a DCI, he requests an unique identifier for it.
- **DRM tool identification device**. After a tool creator has finished making a tool, he requests an unique identifier for it.
- **License provider device**. Licenses are stored on a license provider device and accessed by a content consumption device in case licenses are not bundled within content.
- **Content creation device**. A creator creates content with this device.
- **DRM Tool provider device**. DRM tools are stored on a DRM Tool provider device and accessed by a content consumption device, in case DRM tools are bundled within the content.
- **Content provider device**. This device gets content from the content creation device and provides it to the content consumption device.
- **Stationary end user device (SAV)**. This device obtains content and licenses in order to consume content (e.g. adapt, play, move, copy, etc.).

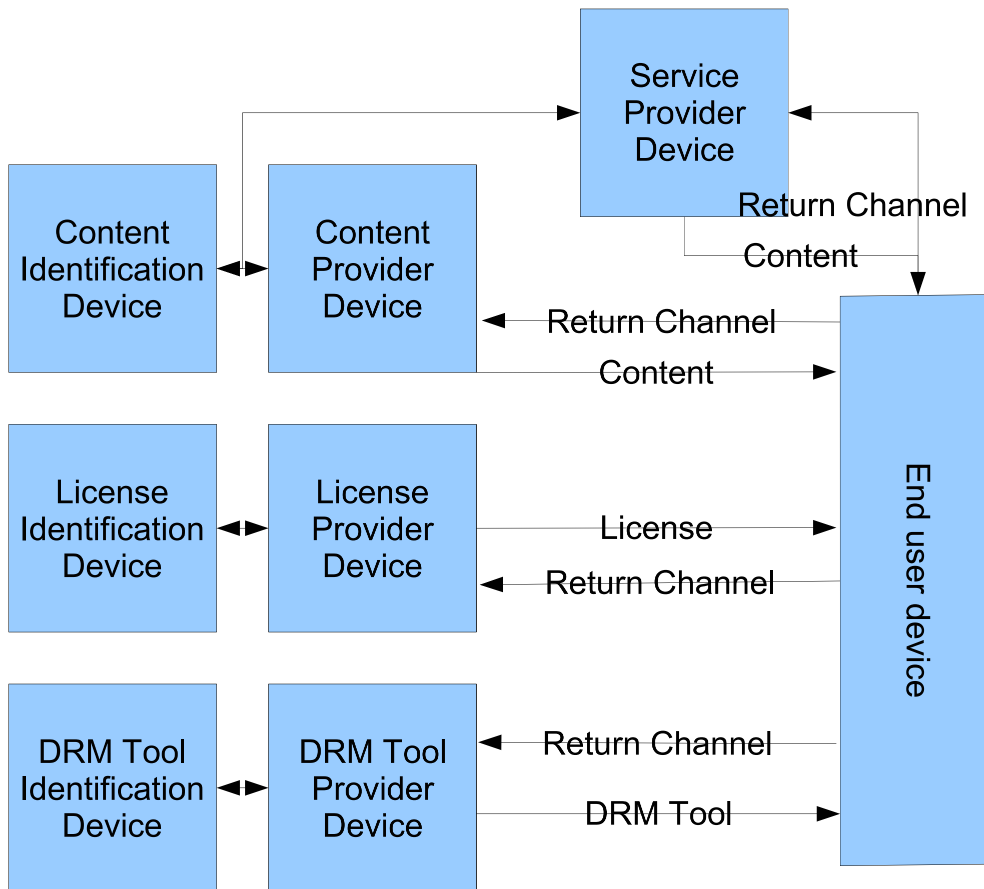


Figure 25: DMP Distribution model

Figure 26: The DMP Distribution Model is a conceptual model explaining how content, licenses and DRM tools are distributed. Identification Devices generate unique identifiers for content, licenses and DRM tools. Provider devices provide licenses, content and DRM tools and distribute them to end-user devices or service provider devices.

- **Portable end user device (PAV)**. This device obtains content and licenses from PAVs and consumes the content afterwards.
- **Domain management device**. This device manages domains and the list of devices and users belonging to this domain.
- **Domain identification device**. This device assigns globally unique identifiers to domain management devices.
- **PAV eXternal Device**. This device does not have network or broadcast connections to access content or licenses. However, it can be connected to an external device (PXD), e.g. SAV, that in turn is connected to a network or broadcast channel.
- **Device Identification Device**. This device identifies other devices.

E.3.3 DMP Tool Model

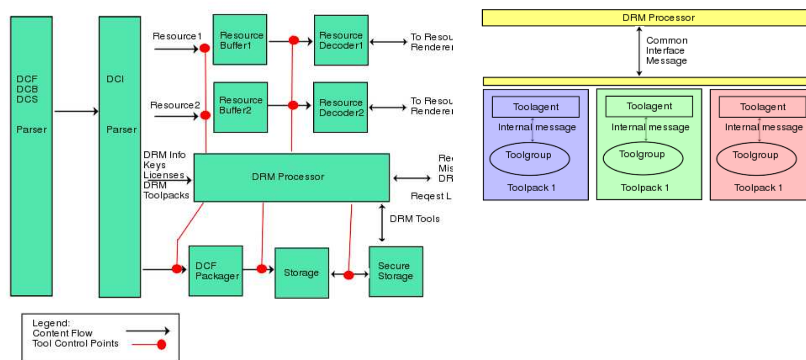
A tool or toolpack contains DMP functions and is used to protect or consume content. The working of the DRM Tool model is illustrated in figure 27. The model consists of:

1. **Entity Representation**. This module parses representations of content (file - DCF, broadcast - DCB, streaming - DCS). The resulting DCI is parsed into metadata, license information and content.
2. **Decoding Pipelines**. These modules buffer and decode content.
3. **DRM Processor**. This module executes DRM-related functions in a trusted way.

The rest of this section explains entity representation, decoding pipelines and the working of the DRM processor.

First of all, DCI is the representation of the content (resources, metadata, DRM information, DRM tools or tool packs, licenses, keys). This representation is not suitable for delivering content over a variety of specific delivery mechanisms: DCF (delivery as file), DCB (MPEG-2 Transport stream) and a real-time transport protocol (DCS). The packaging function described earlier supports conversion from DCI to the transport mechanisms or the other way around.

Secondly, resources and metadata are passed to the appropriate *Decoding Pipelines* while DRM information, DRM tools or tool packs, licenses and keys are passed to the *DRM Processor*, which is a module in a device that executes DRM-related functions in a trusted way. If the device does not have already the required DRM tools (e.g. bundled within content), the *DRM Processor* will access



(a) Handling of DRM tools in a device (b) Toolpack interoperability

Figure 27: In order to consume content, consumer devices parse content representations and metadata. This DRM metadata is interpreted by the DRM processor and the content is send to a decoding pipeline. The DRM processor manipulates content (e.g. decoding) by applying several toolpacks at several points in the pipeline.

the missing DRM tools from the DRM tool provider device. The DRM tool is placed in secure storage once used for later use.

E.4 Mapping onto proposed architecture

First, this section maps the different DMP users onto our consumer, distributor and publisher. Secondly, the distributed view of DMP and Distrinet are compared. Thirdly, this section takes a look at the components of the DMP architecture and those of the Distrinet architecture. Fourth the interfaces of those two are compared.

E.4.1 Users

Most of the DMP users can be mapped on our three DRM users, as illustrated in table 7 and figure 28.

A creator can be seen as a DRM consumer and a DRM producer. He has the possibility to *adapt* existing work (DRM Consumer), or create a new one from scratch (DRM Producer). An instantiator is a DRM consumer, because he *performs* a work and a DRM producer because he produces his *performance*. A producer *produces* someones work. By consequence he is a DRM consumer. He is also a DRM producer, because the *produced work* can be submitted to a publisher. The content and service providers are DRM consumers and DRM publishers, because they *distribute content* produced by someone else. The end user is a DRM consumer, because he consumes content.

Table 7: Relation between DRM roles and value chain users

	Consumer	**Producer**	**Publisher**
Creator	adapt	work	
Instantiator	performs	performance	
Producer	produces	produced work	
Content Provider	distributes		content
Service Provider	distributes		content
End User	consumes		

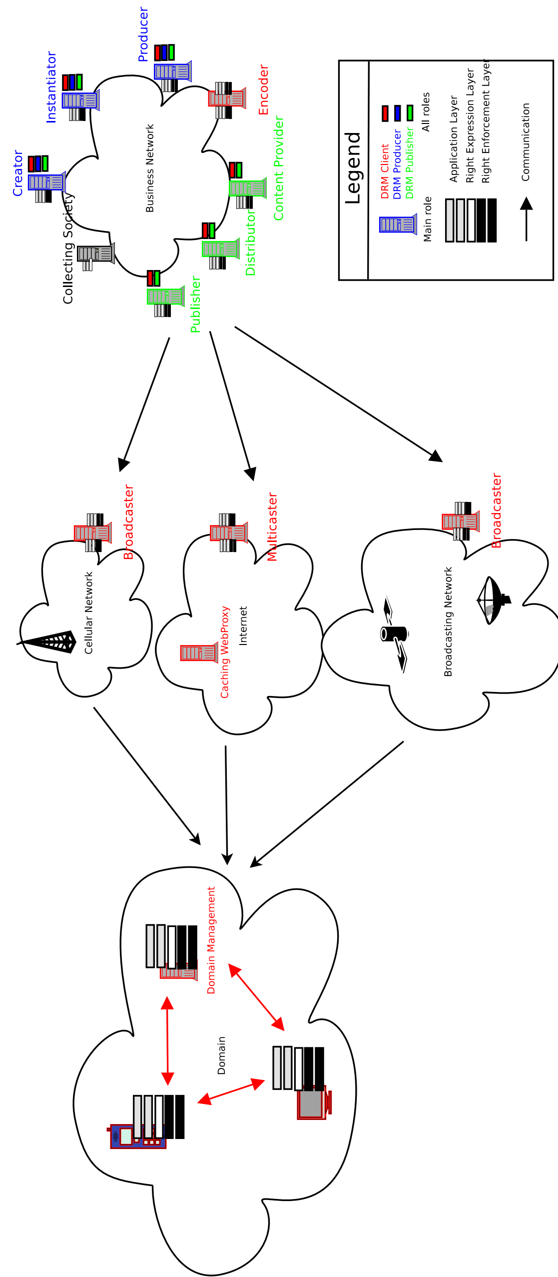


Figure 28: Distributed view: all parties

E.4.2 Distributed View

DMP does not really has a distributed view. Each device is a piece of software or hardware and can combined into one device.

E.4.3 Components

The mapping between our services and the devices of DMP is explained in this paragraph. Table 8 gives a summary.

When a producer has created a work, and wants to add a license, he contacts the *License Identification Device* to generate an unique identification number. This functionality can be done by the *Representation Component*.

The *Content Identification Device* generates a global unique identification number in order to uniquely identify content. The representation of this global unique number is handled by the *Representation Component*.

A *DRM Tool Identification Device* generates a global unique identification number for DRM tools. This functionality is not available in our layered architecture, but can be placed in the *Representation Component*, if it was.

The *License Provider Device* provides licenses to the consumer. Our *License Service* provides the same functionality. The representation of this license is handled by the *Representation Component*.

The *Content Creation Device* is used by the producer to create content. The *Producer Tool* (its *Representation Component*) offers this functionality. The representation of content is handled by the *Representation Component*.

The *DRM Tool Provider* device is not available in our architecture.

The *Content Provider Device* provides content to the users of the DMP system. This functionality is provided by our *Content Service*.

The *End User Device* and the *DRM Client* (the RI and security component group) allows the consumer to consume content.

The *Domain Management Device* allows a user (consumer or publisher) to manage domains. This is not available in our architecture yet.

The *Domain Identification Device* and *Device Identification Device* are used to identify respectively domains and devices. This functionality is not offered by our architecture yet.

DMP has the possibility to update DRM software (as a tool pack). This can be achieved by adding a representation policy (*Representation Component* and the needed security services).

DMP uses certification and registration authorities to achieve trust between devices. Our architecture uses external certification and registration authorities to enable trust between the different parties.

DMP	Our Architecture
License Identification Device	Representation Component
Content Identification Device	Representation Component
DRM Tool Identification Device	-
License Provider Device	License Service
Content Creation Device	Representation Component
DRM Tool Provider Device	-
Content Provider Device	Content Service
End User Device	DRM Client
Domain Management Device	-
Domain Identification Device	-
Device Identification Device	-

Table 8: DMP Devices vs. our services

Our security component group handles the secure storage of DRM information transparently. The DMP model uses also secure storage for licenses, keys and DRM toolpacks.

E.4.4 Interfaces

The primitive functions can also be mapped onto interfaces of our proposed architecture (see table 9). This was partially done in section E.3.1 on page 118. The functions itself are grouped by the proposed services and correspond partially with the proposed interfaces.

Function	Remarks	Group	Component	Method
identify content	fingerprint, uid, doi	RI	Representation	-
identify device	HW profile, software profile, fingerprint	RI	Representation	-
identify user	certificates, information from OS	RI	Representation	-
identify domain	certificates, fingerprint, HW profile, software profile	RI	Representation	-
identify DRM tool	fingerprint, certificate, SW/HW profile	RI	Representation	-
identify territory	e.g. secure gps	RI	Representation	-
assign identifier	information + protocol?	RI	Representation:Adapt	adapt
assign metadata	information + protocol	RI	Representation:Adapt	adapt
revoke content	blacklist content	Service	License Service:Management	revoke
revoke device	blacklist device	Service	UMS:User Profile	revoke
revoke domain	revoke devices	Service	UMS:User Profile	revoke
revoke user	blacklist user	Service	UMS:User Profile	revoke

revoke DRM Tool	-	Service	-	-
represent content	encoding	RI	Representation	create
represent identifier	encoding	RI	Representation	create
represent resource	encoding	RI	Representation	create
represent metadata	encoding	RI	Representation	create
represent DRM information	encoding	RI	Representation	create
represent DRM Tool Body	encoding	RI	Representation	-
represent rights expression	encoding	RI	Representation	create
represent rights data	encoding	RI	Representation	create
represent key body	encoding	Security	KeyGeneration	-
represent key	encoding	Security	KeyGeneration	-
represent device information	encoding	RI	Representation	create
represent domain information	encoding	RI	Representation	create
represent use data	encoding	Service	Tracking Service	-
represent resource format	encoding	RI	Representation	create
authenticate content	protocol	Service	Identification Service	identify
authenticate device	protocol	Service	UMS:CurrentIdentity	authenticate
authenticate user	protocol	Service	UMS:CurrentIdentity	authenticate
authenticate DRM tool	protocol	Service	-	-
authenticate domain	protocol	Service	UMS:CurrentIdentity	authenticate
verify content	algorithm	RI	Policy Engine/Representation	interpret

verify device	algorithm	RI	Policy Engine	interpret
verify DRM tool	algorithm	RI	Policy Engine	interpret
verify metadata	algorithm	RI	Policy Engine/Representation	interpret
certify content	protocol	Service	-	-
certify user	protocol	Service	-	-
package as file	algorithm	RI	Representation:Convert	convert
package as broadcast	algorithm	RI	Representation:Convert	convert
package as streaming	algorithm	RI	Representation:Convert	convert
deliver render, store, restore, copy, move, backup	algorithm	RI	Policy Engine:Interpret	interpret
deliver access	protocol	RI	-	-
deliver import	protocol	RI	Policy Engine	-
manage domain	protocol	Service	Domain Service	-

Table 9: Relation between DMP components and interfaces and our proposed components and interfaces.

F AXMEDIS architecture

F.1 Context

AXMEDIS is a large project partially founded by the European Commission and includes about 20 partners among which Fraunhofer Institute for Computer Graphics, HP, Tiscali, University of Florence, University of Leeds, University Pompeo Fabra, etc. The Consortium creates several demonstrators, to bring different types of content via different distribution channels.

Main objectives of the AXMEDIS project are (i) reducing costs of multimedia production by accelerating the production process with artificial intelligence techniques, (ii) reducing costs of multimedia distribution at the B2B-level (business to business) and the B2C-level, (iii) searching for and integrating objects and components, and (iv) managing and monitoring distribution through the usage of DRM.

AXMEDIS developed a framework which consists of a structured and organized set of software components in the area of cross media production, content sharing, content distribution towards multi-channel. These tools can be used to setup and built a set of full system applications based on DRM for:

- content production and distribution to end-users via different channels including mobile, satellite data broadcast, interactive TV, personal computer, kiosks, PDA and others.
- content integration, protection and event reporting services.
- massive automated plants for content processing and distribution.
- content production and B2B distribution and its harmonization with B2C (Business to Consumer).
- etc.

The AXMEDIS content can contain any type of cross media content, from simple multimedia files to games and software components. AXMEDIS also developed tools covering the whole value chain, supporting valuable standards for DRM, content production, ...

This section describes the AXMEDIS value chain users, the AXMEDIS framework and AXMEDIS architecture in a high level. More technical information can be found on their website[1].

F.2 Users

AXMEDIS calls their users actors, which are explained in the listing below:

- **Aggregator** provides procuring, packaging, presenting, cataloging, archiving, indexing and promoting services typically to retailers.
- **Author** creates a work that is copyrightable, whether it be artistic, literary, dramatic, or musical.
- **Certificate authority** issues digital certificates used to create digital signatures and public-private key pair.
- **Clearing house organization** collects value expressions from other users to distribute to right holders for the purchase of use rights over a given instance of content.
- **Collective management society organization** provides collective representation to its members, e.g. authors, performers, publishers etc. It collects license fees on behalf of composers and publishers for certain types of usages of copyright-protected material and works.
- **Conformance certification provider** provides conformance, robustness and encoding rules.
- **Connectivity provider** provides point-to-point or point-to-multipoint connectivity between users.
- **Content provider** provides content or content with license to another actor (e.g. an end user).
- **Creator** can ask for an AXMEDIS Object ID from the AXMEDIS OID Generator of the AXCS.
- **Customer** buys or rents a work or rights on it. A customer can also be a performer.
- **Device manufacturer** manufactures or assembles hardware and/or software components to make devices.
- **Distributor** possesses the right, acquired by sale, license or other transfer, to control the disposition and transfer of content to end users.
- **End-user or final user** legally acquires content for personal use. He is the last user in a value chain.

- **Engraver** typesets the manuscript of the composer. He edits the digital sheet music by copying it from the composers manuscript, using a music editor (Finale, Sibelius, Wedelmusic)., or from a print out of an old music sheet from which the symbolic code is not available.
- **Financial service provider** provides the infrastructure for financial transactions, and accept deposits and channel the money into lending activities.
- **Marketer** provides promotional, sale enhancement, brand enhancement and merchandising services.
- **Mediation service provider** provides mediator/agent services to broker closed information such as actor identity.
- **Metadata service provider** recognizes, assigns, delivers and processes structured metadata.
- **Music author/song writer** is the original creator/writer of lyrics of a music composition.
- **Network service provider** provides IP (or equivalent) services and typically various other services above it, e.g. guarantee of quality of service.
- **Performer** uses works to make content with a production process. As defined in the WIPO Performances and Phonograms Treaty with respect to rights granted: performers are actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret, or otherwise perform literary or artistic works or expressions of folklore.
- **Phonographic producer** undertakes the technical and economical responsibility necessary for the making of sounds recordings.
- **Platform service provider** provides services on (parts of) the technology infrastructure of a value chain.
- **Producer** produces content.
- **Public Authority** provides rules relating to the use of content and taxation on transactions related to content.
- **Publisher** makes content available to the public. In many cases, the publisher is responsible for publishing, marketing, distribution of content, protecting at the same time his IPR during the legal time or the time indicated in the contract. An organization/company that makes music works public

and share the author and composer rights. The copyrights must be assigned to the music publisher by the authors/composer in written contract, in return for a percentage of royalties produced by the content.

- **Reseller** possesses the right, acquired by sale, license or other transfer, to control the disposition and transfer of content from end users to different end users.
- **Resolution service provider** provides the service of mapping disparate sets of metadata.
- **Retailer** distributes, sells or licenses content to an end user
- **Rights Holder** has rights on a piece of content.
- **Security provider** provides technologies and services related to security technologies and all levels of relevant computer and network security solutions.
- **Sub-publisher** is a foreign company/entity retained by the original publisher of a work to exploit that work in the foreign companys geographic territory.
- **Syndicator** manages and provides content to retailers using a variety of purchase options.
- **Technology licensing provider** provides device manufacturers and platform providers with a license to utilize patented technology to make devices and platforms.
- **User** intends to use a copyrighted work (owned by others).

F.3 Architecture

The AXMEDIS architecture supports the whole value chain, from production to distribution, trying to prevent the leakage of unprotected content within a value chain. The project provides a set of tools which allows secure P2P B2B distribution of content and reporting back information related to the exploitation of rights along the value chain. The business area can be decomposed in production and distribution (see figure 29). Several production actors (blue) are involved, among which content producers, content owners, content providers, content integrators and collecting societies (e.g. SABAM). The distribution actors (red) are

the internet distributor, the mobile distributor (mobile phone), the media distributor (DVD's, CD's, . . .), the broadcasters (TV, . . .) and the kiosk distributors (PDA's, . . .).

A more detailled version of the architecture is give in figure 30. The diagram includes the most important areas of the AXMEDIS framework and architecture.

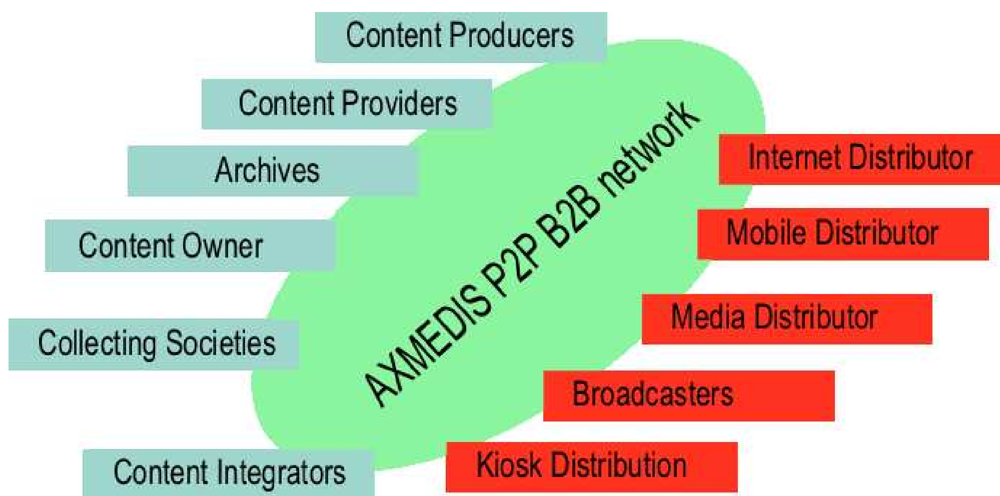


Figure 29: AXMEDIS Business to Business area with distributors (red) and producers (blue).

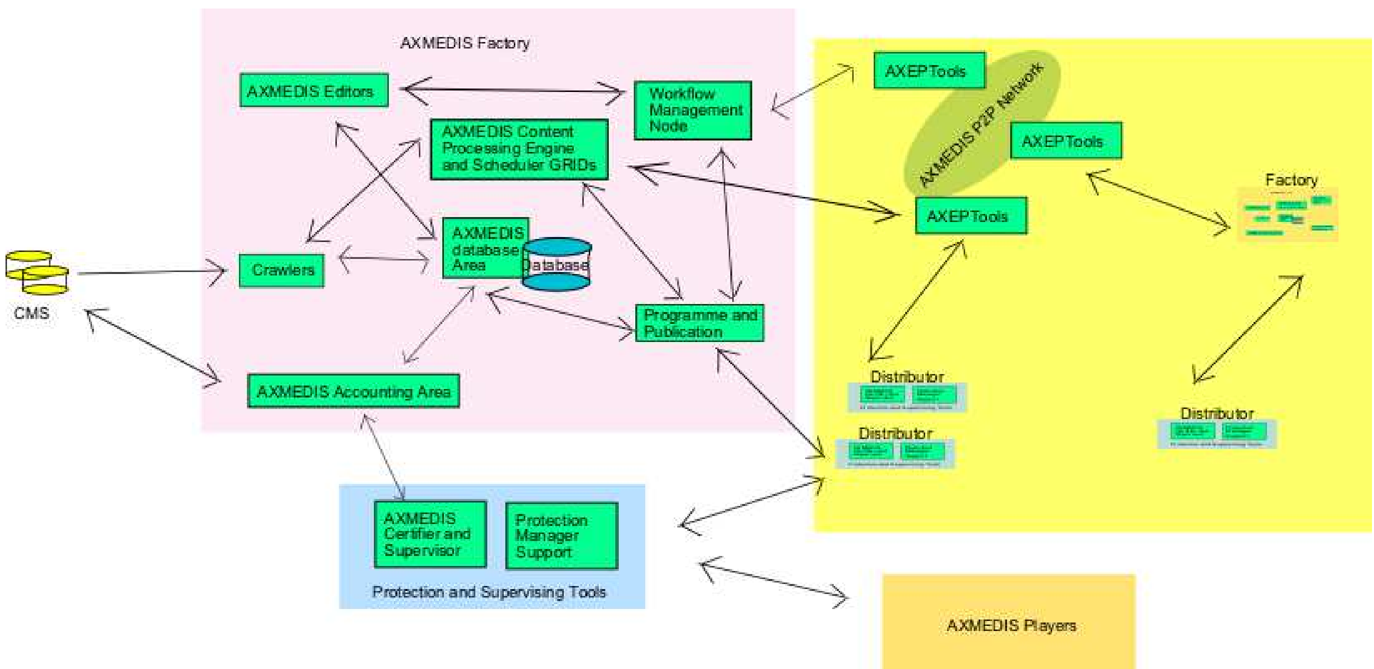


Figure 30: AXMEDIS general architecture consists of four areas. The factory (pink) contains tools which aid the production process. The distribution area (yellow) contains tools for automating B2B content distribution and acquisition. The protection and supervising area (blue) contains tools for registering, certificating users, authenticating devices and tools, monitoring activities on content, processing licenses, etc. The players area (orange) contains tools for content consumption (including distribution with a P2P-tool) on several platforms.

The major AXMEDIS areas are:

- **AXMEDIS factory area** for automating production. This area includes tools for collecting content from legacy CMS's, for producing content, programming en scheduling the production process, editing and processing metadata, composing and formatting content, collecting statistical information about content usage, producing licenses, etc. The factory should satisfy the needs of large and small producers, publishing companies and distributors.

- **AXMEDIS content distribution area** for automating content publication and acquisition in the business area. This area includes tools for scheduling content distribution and publication towards end distribution servers, automatically acquiring/updating content from business partners, etc. Several factories are connected to each other by the secure AXMEDIS P2P Tool (AXEPTools).

- **AXMEDIS protection and supervising tools area** for registering users, certificating users, authenticating devices and tools, monitoring all the activities on AXMEDIS content on players and tools, processing licenses, managing black lists, and collecting and reporting information about content usage, etc.

- **AXMEDIS player** for content consumption. This area includes tools for content consumption on several different platforms, for distributing and sharing content between final users, etc.

F.3.1 AXMEDIS factory

The AXMEDIS factory can be built upon different AXMEDIS tools. The tools can be selected on the basis of the needs of the producer. It is thus possible to set up a large set of possible configurations. The most important tools are:

Local CMS are directly connected to AXMEDIS. These content management systems (CMS) contain digital content, corresponding metadata, resources, files, license information, ... The access to these CMSs is performed by the Focusseek Crawler of the AXMEDIS Crawling Area.

AXMEDIS Crawling Area includes a set of plug-ins which allows interaction among many proprietary CMSs and the AXMEDIS System to migrate sources towards the AXMEDIS Factory.

AXMEDIS Database Area includes the databasemodel, supports the storage and access to AXMEDIS content via a large set of metadata. The database also includes produced licenses, history of the performed actions on the content, contracts, potentially available rights. The database area is based on AXMEDIS database, AXMEDIS Database manager, and a AXMEDIS Query Support tool. The user may perform queries to search for content (AXMEDIS objects) located in the local CMS, the local AXMEDIS Database and those located in other factories via the AXEPTools (see later).

AXMEDIS Content Production Area contains processing procedures and algorithms to manipulate and produce AXMEDIS objects, metadata, any digital resource such as documents, video files, audio files, images, etc.

AXMEDIS Editor is the authoring tool for producing AXMEDIS Objects. It contains all the tools and modules to manipulate and create AXMEDIS objects and related information. Some tools are (i) a set of plug-ins to allow integration of the editor with other editing and viewing information, (ii) a set of plug-ins to use algorithms for content processing, (iii) a set of internal viewers and players for digital resources such as documents, images, videos, . . . , (iv) metadata editors and viewers, (v) DRM viewers and editors to create and verify licenses, (vi) protection information viewers and editors to specify, apply and browse protection aspects and, (vii) interfaces with workflow applications, such as OpenFlow and Biztalk.

AXMEDIS Workflow Management Area includes support tools and plug-ins to allow interfacing the whole factory to workflow tools such as Open Flow and biztalk. It is thus possible to define workflow policies and managing inter-factory workflow policies.

AXMEDIS Accounting Area includes a set of tools which allow final producers or distributors to collect information about what has been done on their objects. It also can be used to interfere with the administrative side of the CMS to prepare billing etc.

AXMEDIS Programme and Publication Area includes components which allow interconnection among AXMEDIS networks and databases to the distribution channels for producing programs to public content on the distribution channel. It allows the management of request for content production and adaptation on demand.

F.3.2 AXMEDIS Distribution Area

The AXMEDIS tools for distribution area allow automating content publication and acquisition in the business area. It consists of several tools:

AXEPTool (AXMEDIS P2P for B2B distribution) allows searching for content among business partners connected to the AXMEDIS Network, automatically publish or acquire content to or from the business partners, etc. The corresponding rights and metadata of the content are used to start the negotiation of content acquisition. Each transaction and trial is monitored by the AXEPTool and by the AXMEDIS Certifier and Supervisor. The content and metadata are certified.

Distributors represent any kind of content distribution services: Internet, satellite, broadcast, mobile, . . . Distributors may use some tools of the AXMEDIS Factory for e.g. retrieving statistical information or setting up a legal P2P service, etc.

F.3.3 AXMEDIS Protection and Supervising Tools

The AXMEDIS tools for protection and supervising provide support for registering users, certificating users, authenticating devices and tools, monitoring all the activities on AXMEDIS content on players and tools, processing licenses, managing black lists, and collecting and reporting information about content usage, etc. The tools are:

AXMEDIS Certifier and Supervisor (AXCS) is responsible for supervising the certification process, registration of user and tools, generation and manipulation of object identifiers, collecting object metadata, registration and management of object usage. Each distributor could contain his own object AXMEDIS Certifier and Supervisor. The AXCS does not contain detailed data about users: only a user identifier and its related data about object usage is known. It consists of the following tools (i) AXMEDIS OID Generator is responsible of object ID generation and will elaborate all new-id requests coming from the client or automatic engines, (ii) Fingerprint Extractors Area includes access to a set of algorithms capable of estimating fingerprints of a large variety of possible resource/content type for extracting a large amount of futures.

AXMEDIS Protection Manager Support are the tools for managing DRM. This tool collects licenses, and has the duty of processing chains of licenses on the basis of the requests received from the AXMEDIS players and other tools. It is also capable of managing domains.

AXMEDIS Protection Tool Engine allows protection objects in a Systematic manner.

F.3.4 AXMEDIS Players

AXMEDIS Players includes tools for content consumption on several different platforms, for distributing and sharing content between final users, etc.It is basically a reduces version of the AXMEDIS Editor only capable to read and show AXMEDIS objects. It also includes plug-in versions of this tools which allows content consumption in other players.

F.4 Framework

The framework is a set of information and tools that is the basis of the above mentioned applications and solutions. It contains all the necessary tools to manage content modeling, storage, load and save, protection and processing from content production to distribution over different channels (see figure 31 on page 145).

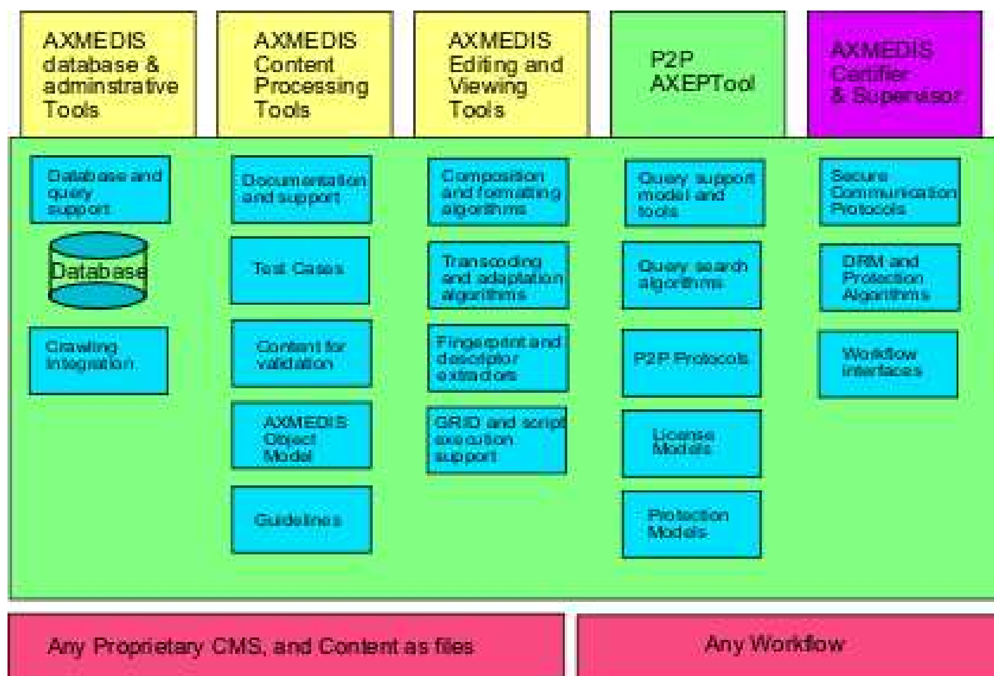


Figure 31: The AXMEDIS framework contains all the necessary tools to manage content modeling, storage, load and save, protection and processing from content production to distribution over different channels

The most relevant parts are (i) a common model for content representation,(ii) a repository administrator tool for administering proprietary CMSs, (iii) access methods to local databases, (iv) AXEPTool and its P2P engine (v) composition tools and algorithms, (vi) formatting tools and algorithms, (vii) transcoding and digital item adaptation algorithms, (viii) test cases and content validation, (ix) format for query and results, (x) distribution algorithms.

F.5 Mapping onto proposed architecture

The first subsection compares the finegrained AXMEDIS users to our three roles. The next section compares the distributed view. The following section explores the components and the last section compares the interfaces.

F.5.1 Users

AXMEDIS	Producer	Publisher	Consumer	Other
Aggregator		packaging	indexing	
Author	creates work			
Certificate authority			Certificate authority	
Clearing house organization		statistics		
Collective management society		statistics		
Conformance certification provider			-	
Connectivity provider				Connectivity provider
Content provider		publishes		
Creator	creates work		adaptation	
Customer			consumes work	
Device manufacturer				devices
Distributor		distributes	distributes content	
End-user			consumes content	
Engraver	typesetting		engraves contant	
Financial service provider				Financial service provider
Marketeer			marketees	marketees
Mediation service provider				-
Meta data service provider	Metadata			
Music author/song writer	writes songs			
Network service provider				Network service provider

Performer	performance		performs content	
Phonographic producer	phonographs			
Platform service provider				Platform service provider
Producer	Producer			
Public authority		statistics		
Publisher		publisher		
Reseller		sells content	resells content	
Resolution service provider				-
Retailer		sells content		
Rights holder	Producer			
Security provider				-
Sub-publisher		publisher		
Syndicator		publishes content		
Technology licensing provider				-
User			consumes content	

Table 10: AXMEDIS actors mapped onto our roles

F.5.2 Distributed view

AXMEDIS does not give an overall distributed view. As most components are webservices, it should no problem to install each webservice on a different machine.

F.5.3 Components

AXMEDIS has the intention to develop a framework for a whole value chain and is thus very broad. Our architecture incorporates the Protection and Supervising Tools, Accounting Area and the player of AXMEDIS. We do not have a counterpart for some components of AXMEDIS (such as Synchronizer and Collector), because AXMEDIS tried to solve the distributed nature of a B2B value chain and we do not.

AXMEDIS	AXMEDIS Description	component group	component
Player	consumes content	Service	DRM Client
AXMEDIS Registration	registers user	Service	UMS
AXMEDIS Certification and Verification	verifies used tools are ok	RI	-
AXMEDIS Supervisor	tracks all user actions	RI	Logging Component
AXMEDIS Reporting	gives usage information	Service	Accounting Service
AXMEDIS Statistical Analysis Tool	gives usage statistics	Service	Accounting Service
AXCS Synchronizer	synchronizes usage information	Service	-
AXCS Database Interface	interface towards database	Service	-
AXCS Manager User Interface	administrative tasks	Service	-
AXMEDIS Software Tool Offline registration	submit 3rd party tools for certification	Service	-
AXMEDIS OID Generator	generates global unique ID for content, etc.	RI	-
Global Object			

List Web Interface	gives a list of all available content	Service	Content Service
AXMEDIS Registration of AXCS	registration of AXCS	Service	-
AXCS Collector	retrieves data from AXCS synchronizer and update database	Application layer	-
Protection Manager Support Server	protect content, consume content	Application Layer	Online DRM System
Protection Manager Support	content creation and consumption	Application Layer	Online DRM System
Protection Manager Support Home	allows content consumption	Application Layer	DRM Client
Protection Manager Support Domain Factory	allows creation and consumption of content	Application Layer	Producer Tool
Protection Manager Support Client	allows content consumption	Application Layer	DRM Client
License Manager	manages licenses	Service	License Service
License Generator	generates license from presentation	RI	Representation Component
License Verificator	verifies licenses	RI	Policy Engine

Content Consumption Status Manager	manages offline tracking	Service	Tracking Service
Domain Manager	keeps track of users domain	Service	-
Domain Registration Manager	allows registration	Service	-
Domain Registration Client	does registration	Service	-
AXMEDIS Certifier and Supervisor	RI	-	
Authorization Support	supports authorization	Service	UMS
Secure Cache Manager	stores info securely	Security	Secure Storage
Secure Communication Support	Security	Secure Communication	
Rights Expression Translator	translates licenses from one REL to another	RI	Representation Component
Key Generator Support	generates protection keys	Security	Key Generator
Protection Info Manager	deals with protection information for end-users (keys)	**RI and Security**	Representation Component and Key Manager

Protection Tool Engine	protects content	RI and Security	Representation Component and security components
Core Accounting Manager and Reporting Tools	overview of content consumption	Service	Accounting Service

Table 11: Relation between AXMEDIS components and Distrinet components

F.5.4 Interfaces

Most interfaces of the digital rights management part of AXMEDIS can also be mapped on our interfaces. Table 12 gives for each module described above our corresponding interfaces (if any).

AXMEDIS	(Distrinet component):Interface	Distrinet method
AXMEDIS Registration		
registration	(UMS):User Profile	register
AXMEDIS Certification and Verification		
verifyUse	-	-
certify	-	-
verify	-	-
AXMEDIS Supervisor		
storeUserLicenseRequest	(Logging Component):Logging	submitMessage
storeUserToolDeviceCertification	(Logging Component):Logging	submitMessage
storeUserAuthorisationAndRequestedOperation	(Logging Component):Logging	submitMessage
storeActionLog	(Logging Component):Logging	submitMessage
storeListActionLog	(Logging Component):Logging	submitMessage
storeHistoryIsNotConsitstent	(Logging Component):Logging	submitMessage
getActionLogs	(Accounting Service):Accounting	getStatisticalInformation
getAndStoreObjectId	-	-
getLastFingerprint	-	-
updateProtectionInfo	Representation Component	-
getProtectionInfo	Representation Component	-
Reporting		
acceptRequest	(Accounting Service):Accounting	getStatisticalInformation

	and (UMS):CurrentIdentity	isAuthorized
AXMEDIS Statistical Analysis Tool		
acceptRequest	(Accounting Service):Accounting and (UMS):CurrentIdentity	getStatisticalInformation isAuthorized
AXCS Synchronizer		
not applicable		
AXCS Database Interface		
not applicable		
AXCS Manager User Interface		
No interfaces defined		
AXMEDIS Software Tool Offline Registration		
not applicable		
AXMEDIS OID Generator		
registration	Representation Component and (UMS):CurrentIdentity	- isAuthorized
Global Object List Webservice		
acceptRequest	(Content Service):Service Interface	search

	and (License Service):Service Interface and (UMS):CurrentIdentity	search isAuthorized
AXMEDIS Registration of AXCS		
Not applicable		
AXCS Collector		
Not applicable		
Protection Manager Support Server, Client, . . . and DRM Support		
See the interfaces of the services below		
Authorization Support		
authorise	(UMS):CurrentIdentity	isAuthorized
Secure Communication Support		
sendData	Secure Communication	-
recieveData	Secure Communication	-
sessionClode	Secure Communication	-
Content Consumption Status		
insertActionLog	(Logging Component):Logging	submitUsageData
retrieveActionLog	(Accounting Component):Accounting	getStatisticalInformation

deleteActionLog	(Accounting Component):Tracking	clearStatisticalLogs
clearCacheContent	-	-
getLastActionLog	(Accounting Component):Accounting	getStatisticalInformation
License Manager		
retrieveLicenseModel	(License Service):Service Interface	getLicenseTypes
retrieveLicense	(License Service):Service Interface	getLicense
updateLicenseModel	(License Service):Service Interface	updateLicense
deleteLicenseModel	(License Service):Management Interface	removeLicense
deleteLicense	(License Service):Management Interface	removeLicense
revokeLicense	(License Service):Management Interface	removeLicense
revokeAddLicense	(License Service):Management Interface	removeLicense and addLicense
storeLicense	(License Service):Management Interface	addLicense
storeLicenseModel	(License Service):Management Interface	addLicense and addKeyData
LicenseVerificator		
verifyLicense	(Policy Engine):Interpretation	interpret
verifyCreatedLicense	(Policy Engine):Interpretation	interpret
verifyTemporalLicense	(Policy Engine):Interpretation	interpret
verifyPAR	(Policy Engine):Interpretation	-
checkPARRules	(Policy Engine):Interpretation	-
Protection Info Manager		

getProtectionInfo	(Representation Component):Create	-
setProtectionInfo	(Representation Component):Create	-
Key Generator		
generateKey	(Key Generation):Generate	generate
Domain Manager and Domain Registration Client		
registrationRequest	Domain Service	registrationRequest
unRegistrationRequest	Domain Service	unRegistrationRequest
createDomain	Domain Service	createDomain
deleteDomain	Domain Service	deleteDomain
updateDomain	Domain Service	updateDomain
searchDomain	Domain Service	searchDomain
Secure Cache Manager		
security layer secure storage interface should be independant of stored information as opposed to the secure cache manager.		
License Generator		
generateFinalLicense	(Representation Component):Create	create
generateDrmLicense	(Representation Component):Create	create
adaptDRMRules	(Representation Component):Adapt	adapt

adaptPar	-	-
Rights Expression Translator generateTranslation	(Representation Component):Convert	generateTranslation

Table 12: Mapping between interfaces of the proposed architecture and interfaces of the digital rights management part of AXMEDIS.

References

[1] Automating Production of Cross Media Contentfor Multi-channel Distribution (AXMEDIS), 2006. http://www.axmedis.org/. 135

[2] Software Architecture Glossary, May 2006. http://www.sei.cmu.edu/architecture/glossary.html. 55

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. 17

[4] D. Boneh and J. Shaw. Collusion-secure fingerprinting for digital data, 1995. 46

[5] P. Cano, E. Batlle, T. Kalker, and J. Haitsma. A review of algorithms for audio fingerprinting, 2002. 46

[6] L. Chiariglione. Digital Media Project (DMP), 2006. http://www.dmpf.org. 59, 117

[7] I. Cox, J. Kilian, T. Leighton, and T. Shamoon. Secure spread spectrum watermarking for multimedia, 1995. 46

[8] J. S. Erickson. Fair use, drm, and trusted computing. *Commun. ACM*, 46(4):34–39, 2003. 7

[9] P. Gutmann. Cryptlib Security Toolkit, Sept. 2005. ftp://ftp.franken.de/pub/crypt/cryptlib/manual.pdf. 44

[10] F. Hartung and F. Ramme. Digital rights management and watermarking of multimedia content for m-commerce applications. In *IEEE Communications*, pages 78–84, 2000. 46

[11] P. A. Jamkhedkar and G. L. Heileman. DRM as a layered system. In *DRM '04: Proceedings of the 4th ACM workshop on Digital Rights Management (DRM 2004)*, pages 11–21, New York, NY, USA, 2004. ACM Press. 10, 12

[12] J. Meier, A. Mackman, M. Dunner, S. Vasireddy, R. Escamilla, and A. Murukan. Threat modeling. Technical report, Microsoft Corporation, Jun 2003. 64

[13] S. Michiels, K. Buyens, K. Verslype, W. Joosen, and B. De Decker. DRM interoperability and reusability through a generic software. *INDICARE Monitor - About Consumer and User Issues of Digital Rights*, 2(11):16–20, Jan. 2006. 7

[14] S. Michiels, W. Joosen, E. Truyen, and K. Verslype. Digital rights management - a survey of existing technologies. Technical report, K.U.Leuven, Dept. of Computer Science, Leuven, Belgium, Nov. 2005. 7, 25

[15] S. Michiels, K. Verslype, W. Joosen, and B. D. Decker. Towards a Software Architecture for DRM. In *In Proceedings of 5th ACM Workshop on Digital Rights Management (DRM2005)*, Nov. 2005. 7

[16] B. C. Popescu, B. Crispo, A. S. Tanenbaum, and F. L. Kamperman. A drm security architecture for home networks. In *DRM '04: Proceedings of the 4th ACM workshop on Digital rights management*, pages 1–10, New York, NY, USA, 2004. ACM Press. 65

[17] B. Schneier. *Applied Cryptography – Protocols, Algorithms and Source Code in C*. Wiley, New York, NY, USA, second edition, 1997. 46, 47

[18] S. Software. Clasp - comprehensive lightweight application security process version 1.2. Technical report, Secure Software, 2004-2006. 64