Octopus: An Application Independent DRM Toolkit

Gilles Boccon-Gibod, Julien Boeuf, and Jack Lacy

Abstract— Digital Rights Management systems originally arose as an example of a solution to one of the most basic problems associated with secure systems design, the problem of governing access by a credentialed entity to a resource in the context of a set of policies established to restrict or define such access. Most DRMs that are in use today are based on DRM-specific declarative rights expression languages and authorization mechanisms that have strong semantic coupling to the application in which they are employed. A notable exception to this approach can be found in the Octopus system described in this paper. Octopus employs several concepts from the trust management philosophy originally posited by Blaze, et al. in 1996, but perhaps its most striking similarity can be found in its clear separation of authorization mechanism from application semantics.

***Index Terms*—DRM, Trust Management, Octopus, Marlin, MPEG-21 REL, KeyNote, PolicyMaker**

I. Introduction

THIS paper describes a new DRM toolkit called Octopus that is an extension of various elements of trust management and digital rights management work from the mid-1990s. We first provide historical context for this work and then describe the basic philosophy that guided the creation of Octopus. After that we provide a description of the Octopus object model as well as a short description of the application of Octopus to the Marlin DRM system.

II. Historical Context

Digital Rights Management systems originally arose as an example of a solution to one of the most basic problems associated with secure systems design, the problem of governing access by a credentialed entity to a resource in the context of a set of policies established to restrict or define such access. A great deal of work has been done to answer questions about the identity of entities seeking access to protected resources. Public key cryptosystems were established in which public and private keys could be strongly associated with a person and then used to establish secure and authenticated communication with other entities and authenticated access to resources. Identity might be part of the problem but the entity protecting the desired resource must determine whether or not that requestor is authorized or permitted to take a specified action using the resource. Thus, there are three or more independent interactions that must occur. Who is the requestor, what are his or her credentials, and what may he or she do with the resource? The problem can be more involved when policies concerning the environment in which the resource is to be accessed are taken into consideration.

In 1996 Blaze et al. [1] posited that this dependence on identity was insufficient and proposed a new approach to handling the resource access problem. This new approach was coined Trust Management. The fundamental tenet of Trust Management systems as proposed by Blaze et al. was that beyond the traditional notion of certificates that bind names to keys, it must also be possible to bind keys to resource access authorization. In addition, trust management systems must support the identification of principals, the expression of actions on resources, resource governance policies, and principal credentials. Once these are in place, trust management calls for a trusted mechanism for processing requests from credentialed principals to act on a resource in the context of policies governing such access. In the simple case, the response is a Boolean value indicating whether the request to act is granted or not, but it is often useful to respond with conditions of use that obligate the requestor. The application that uses this trust management mechanism is responsible for interpreting the semantics of the response and processing the actual action on the requested resource. This separation of authorization mechanism from application semantics is very powerful in that it promotes the creation of a single, common mechanism that can be used in a variety of applications to process the elements of resource access questions – resource, actions, policies, and principal credentials – in a consistent manner.

Another fundamental element of trust management systems is the provision of a language that can be used to express policies, credentials, and trust relationships. In the original trust management systems described in [1] and [2], this language was a trusted procedural language. Credentials and policies were small programs that were executed in a safe "compliance checking" environment and operated on actions requested by principals on resources. If a chain could be established linking the credentialed principal requesting the action to a policy associated with the resource that grants the action, the unified compliance checking mechanism granted

Manuscript received October 10, 2008.

All three authors are with Intertrust Technologies, Sunnyvale, CA 94085, USA (408-616-1600; fax: 408-616-1626; e-mail: gilles@intertrust.com, jboeuf@intertrust.com, and lacy@intertrust.com).

access.

At about the time that the original trust management work was taking place, early work was beginning on systems that could be used to protect and manage digital music. This work was a direct application of trust management. Content owners seeking to protect their content from unauthorized use needed systems that would enable the persistent governance of content by consumers and other participants in the content distribution value chain. Systems were required that restricted access to a resource (a song) to authorized individuals under specific circumstances or policies. These policies could also be called usage models and user authorization credentials were called licenses. In these systems content would be encrypted using a content encryption key. The key would be bound to a license that indicated to the trust management engine that the holder of the license was authorized to access the content per the terms of the license and the local policy of the application hosting the trust management engine. An early version of these so-called DRM systems explicitly based on trust management is described in [3].

Once DRM systems started to proliferate, various participants involved in the content distribution value chain – content providers, service providers, and device manufacturers – began to realize that in order to have interoperability as well as content usage model consistency across differing DRMs, the language and terminology used in creating content licenses and policies were prime targets for standardization. The Moving Picture Experts Group (MPEG) had started work on a consistent framework to support content distribution value chain participants in their quest to take full advantage of network-based content delivery. The group responsible for this work, called MPEG-21 [4], seemed a logical entity to take on the work of standardizing a language that could be used to describe any kind of content usage model and policy as well as the complex kinds of relationships that can arise in real content delivery ecosystems.

MPEG-21 created its Rights Expression Language (REL) based on a language called XrML [5]. In addition to the language MPEG-21 created a Rights Data Dictionary (RDD) [6] that defines the terms necessary for description of rights. MPEG-21 REL is a declarative language. Licenses expressed in MPEG-21 REL are automatically processed by DRM systems responsible for governing access to content. The scope of MPEG-21 REL did not include the specification of the mechanism for enforcing rights specified using REL.

The general idea behind MPEG-21 REL was that once defined, it (and the application-specific RDD) would gradually replace other DRM-specific rights expression paradigms – including those used by trust management based DRMs. As would be expected of a language that is designed to be capable of describing a wide variety of rights across a complex set of participants, the full MPEG-21 REL is quite large and complex and very flexible in its expressivity. In many cases consumer devices need not be capable of parsing the entire REL since their typical interactions involve very simple scenarios, such as access to simple content rendering, timed rendering (start and stop dates), and limited copying/sharing functionality. MPEG recognized this and to some extent they have dealt with the issue by creating REL profiles for smaller, resource-constrained devices or providing direct interoperability with the rights expression languages employed by specific DRMs. See for example the REL profile for OMA described in [7]. MPEG-21 REL is very powerful in business-to-business applications that involve intermediaries associated with content distribution such as models that require management of content usage frequency or that involve tracking content usage across various services.

Most DRMs that are in use today are based on DRM-specific declarative rights expression languages and authorization mechanisms that have strong semantic coupling to the application in which they are employed.

The Octopus system is a notable exception to this approach. Octopus employs several concepts from the trust management philosophy originally posited by Blaze in 1996. Its greatest similarity can be found in its clear separation of authorization mechanism from application semantics. But Octopus uses an interpreted, imperative, Turing-complete language for governance, and defines a set of abstract rights objects that can be used to naturally define rights in terms of the relationships among the entities that those objects can represent. We describe Octopus in detail, first providing an introduction to its philosophical goals, then an overview of its architecture and some powerful extensions. Last, we describe the Marlin DRM system [http://www.marlin-community.com] that is built on top of Octopus.

III. OCTOPUS

A. Introduction to Octopus

Octopus defines a set of objects, used to represent different elements and entities that comprise a DRM system (such as content, encryption keys, usage rules, users, etc…), ways of expressing static and dynamic relationship between those objects, and ways to use those objects in computations. These computations are what constitute, in the Octopus system, the notion of evaluating the usage rules that can be associated with content.

Along with those basic elements, Octopus defines ways of using cryptographic primitives, such as symmetric ciphers, public key systems and digital signature algorithms in order to provide secure binding between objects when necessary, distribute content keys to the points of content consumption, and tie the object system with authentication schemes and trust models.

When used in a complete ecosystem, including file and stream formats, file and stream encryption, network protocols and application interfaces, Octopus can be used to create complete DRM systems, such as Marlin. A typical Octopus client is shown in figure 1. Note in this figure that Octopus is not itself responsible for crypto functionality, media rendering, or general content services. Octopus is primarily focused on application-independent content governance.

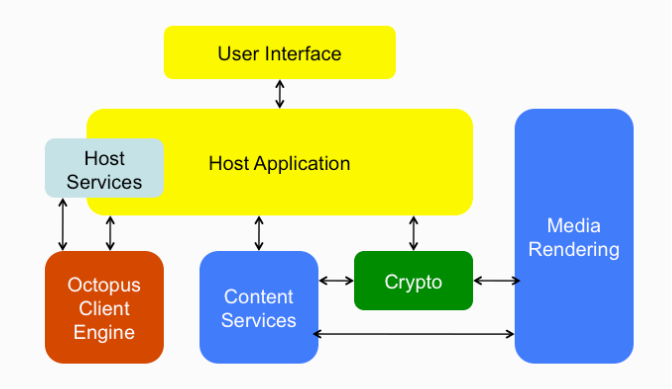

Figure 1: A Typical Octopus Client

B. *Octopus Architecture*

At a high level, Octopus facilitates the association of simple-to-complex usage rules with digital content. In order to guarantee that the content can only be used in accordance with its associated usage rules, the content is encrypted. This ensures that only entities that have access to the content key are able to use the content. By limiting the distribution of content keys only to entities that implement and comply with the way Octopus specifies the evaluation of usage rules, we can assure that the usage of the encrypted content will always be subject to the usage rules associated with that content. An important concept in Octopus is that access to content implies access to the content key, but access to the content key does not imply the right to use the content[1].

The association of usage rules with content is done using sets of objects and cryptographic elements, which together are called a license. The rules themselves are expressed as a set of executable routines encoded as byte code for a simple virtual machine called Plankton, which is described in section C.1. To evaluate whether a certain action on the content (such as 'Play' or 'Print') can be granted, Octopus executes one or more byte code routines, providing all the input necessary for the action, the current environment (time of day, characteristics of the application the is processing the content, etc…), a persistent state store, and information about a number of entities and their relationships. The result of the execution signals that the action is either granted or denied. The information about various entities and their relationship is a key concept of Octopus. It is what allows the richness of expression of the usage rules without requiring Octopus to have a pre-defined set of semantics. These entities and their relationships are represented by objects called Nodes and Links.

1) *Octopus Objects*

All Octopus objects share common basic traits: they can have an ID, a list of attributes, and a list of extensions. All objects that are referenced by other objects have a unique ID. IDs are simply URIs (Universal Resource Identifiers), and the convention in Octopus is that those URIs are URNs (Universal Resource Names). Attributes are typed values. Attributes can be named or unnamed. The name of a named attribute is a simple string or URI. The value of an attribute is of a simple type (string, integer and bytes) or a compound type (list and array). Attributes of type 'list' contain an unordered list of named attributes. Attributes of type 'array' contain an ordered array of unnamed attributes.

An object's 'attributes' field is an (possibly empty) unordered collection of named attributes.

Extensions are elements that can be added to objects to carry optional or mandatory extra data. Extensions are typed, have unique IDs, and can be internal or external. Internal extensions are contained in the object they extend. External extensions are not contained in the object they extend. They appear independently of the object, and have a 'subject' field that contains the ID of the object they extend. The ID of an external extension must be globally unique.

Licenses include five types of objects: Content, ContentKey, Protector, Controller and Control objects. Figure 2 depicts the relationships among these objects.

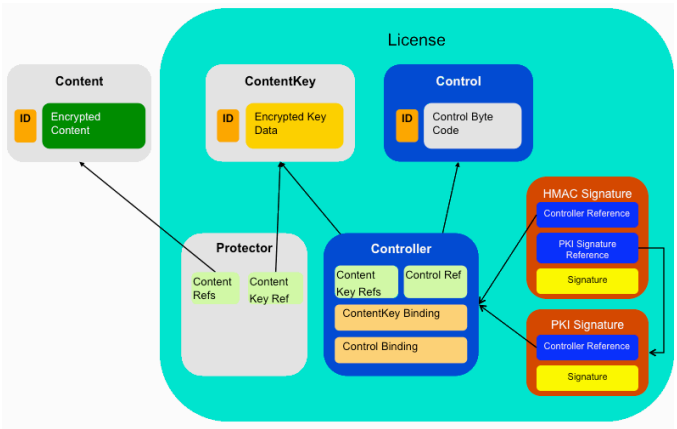

Figure 2: Octopus License Structure

a) *Content Objects*

A Content object represents an encrypted content item. In this context, the only important part of the Content object is a globally unique content identifier. The representation of the data for the Content object depends on the format used for the content. It can be, for example, a metadata box in an ISO-MP4 file [8].

b) *ContentKey Objects*

A ContentKey object represents a unique encryption key, and associates an ID with it. The purpose of the ID is to enable Protector objects and Controller objects to make references to ContentKey objects. The actual key data encapsulated in the ContentKey object is itself encrypted so that it can only be read by the recipients that are authorized to decrypt the content. It is possible for a ContentKey object to contain more than one expression of the encrypted content key, which allows for several ways of making the content key available to the entity that will process this object.

[1] As with any application of cryptography, functions that access cryptographic keys must be designed carefully and must employ tamper resistance techniques.

The ContentKey object specifies which cryptosystem(s) was (were) used to encrypt the key data. The cryptosystem used to protect the content key data is called the Key Distribution System. Different Key Distribution Systems can be used. The default Key Distribution System for Octopus is the Scuba Key Distribution System that is part of the Octopus Specification (see section f below).

c) Protector Objects

A Protector object contains the information that makes it possible to find out which key was used to encrypt the data of Content objects. The Protector object contains one or more IDs that are references to Content objects, and exactly one ID that is a reference to the ContentKey object that represents the key that was used to encrypt the data. If the Protector points to more than one Content object, all those Content objects represent data that has been encrypted using the same encryption algorithm and the same key. Unless the cryptosystem used allows a safe way of using the same key for different data items, it is not recommended that a Protector object point to more than one Content object.

d) Control Objects

A Control object represents the usage rules for one or more content items, or the validity rules for a Link object (defined below). It contains the information that allows Octopus to make decisions regarding whether certain actions on the content should be permitted when requested by the application interacting with the content, or whether a Link object is valid. Those rules are encoded in the Control object as one or more Plankton byte code routines. The Control object also has a unique ID so that a Controller object can reference it. Control objects must be signed, so that Octopus can verify that the control byte code is valid and trusted before it is used to make any decision. The signature of a Control is either direct or indirect. A direct signature is a signature of a Control object itself. An indirect signature exists when a Control is contained in a Link object or when a signed Controller object has a secure reference to a Control (in this case, the ControlRef field of the signed Controller object contains a digest of the Control object).

Other byte code routines in a Control object can be used to compute and return a description of the usage rules represented by the Control.

e) Controller Objects

A Controller object contains the information that allows Octopus to find out which Control governs the use of one or more keys represented by ContentKey objects. It contains information that binds it to the ContentKey objects and the Control object that it references. Controller objects must be signed, so that the validity of the binding between the ContentKey and the Control object that governs it, as well as the validity of the binding between the ContentKey ID and the actual key data, can be established. The signature of the Controller object can be a public key signature or a symmetric key signature, or a combination of both. Also, when the digest of the Control object referenced by the Controller object is included in the Controller object, the validity of the Control object can be derived without having to separately verify the signature of the Control object.

f) Node Objects

A Node object represents an entity in the system. The Node object's Attributes define certain aspects of what the Node objects represent, such as the role or identity represented by the Node object in the context of a specific deployment. Nodes are used to represent diverse types of entities, such as users, playback devices, subscriptions, groups, capabilities, etc… As far as Octopus is concerned, all Nodes look the same. They are simple data structures with no special meaning. Octopus does not have built-in semantics. As such, it does not define special treatments for nodes based on what they represent.

g) Link Objects

A Link object represents a relationship between two Node objects. A Link is a signed assertion that there exists a directed edge in the graph whose vertices are the Node objects. For a given set of nodes and links, we say that there is a path between a node X and a node Y if there exists a directed path between the node X vertex and the node Y vertex in the graph. When there is a path between node X and node Y, we say that node Y is reachable from node X. Those assertions represented by Link objects are used to express which nodes are reachable from other nodes. The controls that govern Content objects can check, before they allow an action to be performed, that certain nodes are reachable from the node associated with the entity performing the action. For example, if node D represents a device that wants to perform the 'Play' action on a Content object, a control that governs this Content object can test if a certain node U representing a certain user is reachable from node D. To determine if node U is reachable, Octopus will check if there exists a set of Link objects that can establish a path between node D and node U.

Just as with Node objects, Octopus does not fix the semantics of the relationship represented by the object. The meaning of the link between two nodes depends on what the nodes represent. For example, one could create a Link object between a Node object representing a video player D and a Node object representing a user U. That link would mean: 'the video player D belongs to user U'. This, in turn, makes it possible to write a Control object with a byte code routine that would check whether there is a valid path between node D and node U; such a byte code routine would implement the usage rule: "this content can be played on any device that belongs to user U".

All Links must be signed. They are a form of certificate. Octopus verifies Link objects before using them to decide the existence of paths in the node graph.

A Link object may contain a Control object that will be used to constrain the validity of the link. This is very powerful, as it allows the relationships represented by the Link object to be dynamic. Whenever Octopus evaluates a path

made of a sequence of one or more Link objects, it evaluates the validity of each link, including the execution of the byte code in any Control that may be embedded in some of those links. As with any other Control, the byte code routine has access to a set of inputs, state, and execution environment. For example, to represent the concept that a user U is a member of a group G until a certain date Y, but not after, one would create a Link object from node U to node G, representing the membership, and would embed in the Link object a Control with a validity byte code routine that would check, when executed, if the current date were past Y. If it were, it would return that the Link object is no longer valid.

An example of the way that links make content accessible to users is shown in Figure 3. Here we see that while the objects in Octopus are abstract, very natural meanings can be applied to them. Thus, in Figure 3, we can see that Anna's phone can play content item 3 because it is part of a subscription that Joe subscribed to, and Joe's account is part of the Smith Family domain, and Anna's phone is part of that domain. The fact that the links can be distributed independently of one another makes the system very powerful. Externally applied relationship semantics allow very natural interpretations of rights that approximate real-world expectations.

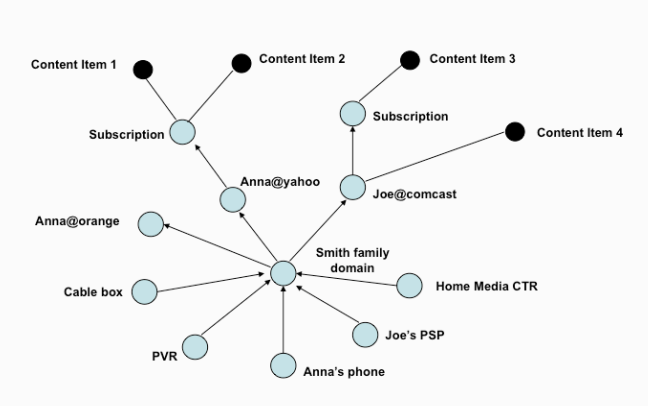

Figure 3: Octopus Link Relationships

h) Scuba

Scuba[2] is the cryptographic scheme used by Octopus to leverage the existence of the Node and Link topology in order to facilitate the distribution of content keys to consumption endpoints.

A typical usage rule will require, as one of its conditions, that a certain node U can be reached from the node D representing the device on which the rule is being evaluated. Scuba uses the set of Link objects that creates a path from D to U as a way to enable the device D to compute the content key for the content governed by the rule. For this, Scuba assigns to each Node object a set of 'sharing' keys (one symmetric and one private/public pair), and to each Link object from node X to node Y an extension that carries the private and/or secret part of node Y's 'sharing' keys encrypted with the public and/or secret 'sharing' key(s) of node X. Finally, the content key is encrypted with the secret and/or public key of the target node U. By following all the links from D to U the device D can compute all the private and/or secret sharing keys of all the nodes on the path, including the target node U; it can therefore compute the content key. The execution of the usage rule then either grants or denies the use of that content key. Figure 4 illustrates these concepts.

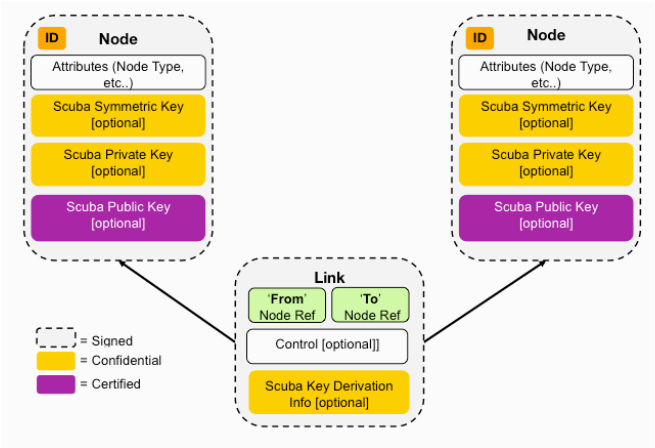

Figure 4: Octopus Nodes and Links

C. Octopus Rules Execution Engine

At the heart of Octopus is its rules execution engine. Unlike traditional DRM systems, Octopus does not use a rights expression language (REL) to represent usage rules. Instead of a declarative approach, it uses a semantic-free procedural approach. All rules in Octopus, be they content usage rules or the rules that govern the validity of Link objects, are represented by executable programs, expressed as sequence of byte codes for a small virtual machine called Plankton (described in the next section).

To evaluate a rule, Octopus prepares the input parameters for the rule, such as what type of action on content is desired, and what the details of the action are; it then executes the byte code for the routine that corresponds to the rule being evaluated. The routine, as it executes, has access to its environment, which includes information such as the current date and time, information about Node and Link objects that represent entities and their relationships, as well as a persistent state storage (Seashell: see below) for rules with side effects. The routine then returns its answer. When evaluating a content usage rule, the answer specifies whether the requested action on the content is granted or denied. If it is granted, a set of obligations can be also returned to specify how the content can be used; for example, when playing a video, the 'Play' action may be granted with the obligation that HDCP video-output protection be engaged on the link to the screen. If the action is denied, the answer can contain details about why the action is denied (for example, a data structure indicating what preconditions were not met, such as an end date that may be in the past).

[2] For more information on Scuba, Plankton, Seashell, and Octopus, see http://www.marlin-community.com.

1) Plankton

Plankton is the virtual machine that is used to execute the control programs representing rules. It is not a general-purpose virtual machine like the Java virtual machine, but rather a very simplified one that is dedicated to the task of executing simple routines that take simple input and produce a simple output, in a predictable way. Plankton routines are represented by byte codes that encode instructions for a stack-based execution engine. The instructions can perform basic computations, read and write in a memory buffer, perform tests and program flow control (jumps and subroutines).

Also, Plankton routines can read and write data from and to their execution environment through a set of specified "system calls" that bridge the virtual machine to external sources of information. For instance, routines can make calls to get the current date and time, check the Node and Link graph topology, read arbitrary attribute information about the device or application hosting the virtual machine, etc…

2) SeaShell

A number of usage rule types require some type of side effect, such as storing a value that can be read at a later time.

For this purpose, Octopus provides a persistent store where Plankton routines can read and write arbitrary data objects or various types (strings, integers, lists, …). This persistent store is called SeaShell. SeaShell objects, in addition to their data, have metadata fields, such as an expiration date and an ownership information, which allows Octopus to manage the lifetime of the data as well as an access policy: Plankton routines can only access, by default, data that has the same owner as the author of the Control object from which the byte code was loaded.

A typical example of a usage rule with side effects is one where a user can purchase the right to view a video for 24 hours, where the 24-hour window starts when the user starts playing the content. When asked for permission to play, the control routine checks whether it has already run by querying the SeaShell store for the end date for the requested content ID. If that data object exists, the routine compares its value to the current date and grants the permission to play. If that object does not exist, the routine gets the current time, adds 24 hours, and stores the result in a SeaShell data object so that it can be retrieved the next time the user tries to play the video.

D. Octopus and Marlin

Marlin is a complete DRM system specification. The Marlin architecture specifies technologies for building copy protection and digital rights management (DRM) into consumer devices and services in a manner that is friendly to end users. Marlin technology allows users to acquire content through multiple distribution channels and to access it on any device that is part of their home domain. In order to accomplish this, Marlin defines both client capabilities and a service architecture so that the capabilities of consumer electronics devices can be powerfully enhanced by services provided over both local and wide-area networks.

Marlin uses Octopus as its core DRM engine for enforcing the content usage rules. In addition to Octopus, Marlin specifies a set of web-service protocols to perform tasks such as acquiring various Octopus objects. These include protocols for acquiring a license for a given piece of content and acquiring Node and Link objects to implement domain models and subscriptions.

Also, Marlin defines a complete unified trust model so that the web service protocols can be authenticated and secure, and that the Octopus objects can be trusted by compliant implementations.

IV. Conclusion

Octopus was designed to provide a comprehensive and flexible approach to the governance of digital resources distributed on open networks. Its use for the Marlin DRM system illustrates how to use the completely abstract object model and governance language to enable usage models that are based on the natural relationships among entities that are familiar to consumers. While Octopus is rich and extensible, the separation of governance semantics from executable controls has two main consequences:

- Client implementations are very simple, consisting mainly of a simple virtual machine and implementations of cryptographic processes, and an object database;
- Much of the complexity inherent in the semantics of complex business and governance models that are enforced by Octopus can be relegated to other parts of the system, outside of the client environment where they may be more easily maintained.

In addition, even after a rights management system has been deployed, new system level semantics can be defined that can radically extend the scope of rights from a user's point of view. The reader may appreciate this by considering the graph in Figure 3) and the effect of deploying new nodes and links, including nodes that have completely new meanings (such as new types of domains).

Finally, we note that the Octopus system has been implemented on a number of different client environments, including Windows, OS-X, Linux, and the Symbian OS running on a variety of different processors. Our experience with Octopus shows that compact implementations on smart cards and SIMs should be straightforward.

Acknowledgments

The authors would like to thank Dave Maher for his insights and helpful review of this paper.

References

[1] M. Blaze, J. Feigenbaum, J. Lacy. Decentralized Trust Management. Proceedings of the 17th IEEE Symp. on Security and Privacy. pp 164-173. IEEE Computer Society, 1996.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, A. Keromytis. The KeyNote Trust Management System, Version 2. RFC-2704. IETF, September 1999.

[3] J. Lacy, J. Snyder and D. Maher. Music on the Internet and the Intellectual Property Protection Problem. In *Proc. of the International Symposium on Industrial Electronics*, pages SS77-83. IEEE Press, 1997.

[4] J. Bormans, K. Hill. MPEG-21 Overview v.5. http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm.

[5] ISO/IEC 21000-5:2004, "Information Technology: Multimedia Framework, Part 5: Rights Expression Language," Int'l Organization for Standardization (ISO) and Int'l Electrotechnical Commission (IEC), 2004.

[6] ISO/IEC 21000-6:2004, "Information Technology: Multimedia Framework, Part 6: Rights Data Dictionary," Int'l Organization for Standardization (ISO) and Int'l Electrotechnical Commission (IEC), 2004.

[7] Delgado, J., Prados, J., and Rodriguez, E. 2005. Profiles for Interoperability between MPEG-21 REL and OMA DRM. In *Proceedings of the Seventh IEEE international Conference on E-Commerce Technology* (July 19 - 22, 2005). CEC. IEEE Computer Society, Washington, DC, 518-521. DOI= http://dx.doi.org/10.1109/ICECT.2005.69.

[8] Coding of audio-visual objects – Part 12: ISO base media file format, ISO/IEC 14996-12.