Inter-library Service Brokerage in LicenseScript

Cheun Ngen Chong[1], Sandro Etalle[1], Pieter Hartel[1], Rieks Joosten[2], and Geert Kleinhuis[2]

[1] University of Twente, P.O. Box 217, 7500 AE Enschede
{chong,etalle,pieter}@cs.utwente.nl
[2] TNO Telecom Groningen, P.O. Box 15000, 9700 CD Groningen
{H.J.M.Joosten,G.Kleinhuis}@telecom.tno.nl

Abstract. Inter-library loan involves interaction among a dynamic number of digital libraries and users. Therefore, inter-library service management is complex. We need to handle different and conflicting requirements of services from the digital libraries and users. To resolve this problem, we present the concept of a packager who acts as a service broker. We also present an implementation using our Prolog based LicenseScript language.

1 Introduction

Federated digital libraries rely on a complex variety of systems, services and policies that must interwork seamlessly. To illustrate some of the issues, consider:

Example 1. Alice uses her PDA to request a high-resolution video clip from the university library, but a lower-resolution one is available in the city library.

Several questions need to be answered, such as: (1) can Alice request the video clip directly from the city library; (2) can the university library obtain the lower-resolution video clip from the city library on behalf of Alice; (3) can the city library somehow benefit from providing the video clip; (4) can Alice use the video clip on another device; (5) can Alice use the video clip in her own work, etc.

To provide a solution, we present the concept of a packager who acts as a service broker to handle inter-library service management. The packager can provide convenience to users to seek, choose and use a wide variety services available from the libraries, e.g. to find a cheaper item. We present an implementation as part of the *Residential Gateway Environment* (RGE) project [7].

We have derived the complex infrastructure for the service management from a semi-formal high-level description: the "Calculating with Concept" (CC) [5]. The reader may refer to our technical report for more details [2]. We encode all aspects of service brokerage in LicenseScript [1]. LicenseScript is based on Prolog and multiset rewriting and allows one to express *licenses*, i.e. conditions of use on dynamic data. Prolog has the advantage of combining an operational semantics (needed, e.g., in negotiations) with a straightforward declarative reading. Our addition of multiset rewriting to Prolog allows to encode in an elegant and semantically sound way the *state*, and the *state transitions* of a license. The semantics of LicenseScript is given in terms of traces [1].

We demonstrate the practical value of LicenseScript by using it as intelligent messaging middleware for the RGE project. The result is a large distributed software platform which we describe in this paper.

Silva and Delgado [8] suggest an agent-based approach to mediate between libraries and users. We have refined this approach into a full-fledged model for inter-library service management using LicenseScript. Halpern and Weissman [6] propose a policy language based on first-order logic and derive various policies for digital libraries. However, the Halpern and Weissman policy language is incapable of reconciling conflicting policies; LicenseScript provides the hooks for this but the mechanisms have to be programmed specifically.

Our contribution is two-fold: (1) The inter-library service management infrastructure can be specified concisely and prototyped rapidly by using LicenseScript; and (2) The infrastructure supports tracking of resources by using LicenseScript [3] and secure audit logging [4].

Section 2 presents the overall infrastructure of RGE service management and presents an example of inter-library service management by the packager specified in LicenseScript. Section 3 presents a prototype. Finally, section 4 concludes.

2 Service Management

The RGE architecture supports four main roles: the devices (D), the residential gateway (RG), the packager (P) and the service providers (SPs). Service providers are the digital libraries, which provide services to users. The packager behaves as a service broker, being able to manipulate and integrate the services provided by the various SPs. The residential gateway implements the concept of an *authorized domain* [9]. An authorized domain is a network of compliant devices, which ensures that content is only used in the authorized domain. A device is used to render a digital resource obtained from the digital libraries. It is connected (wired/wirelessly) to the RG.

Now, we describe inter-library service management with the packager in LicenseScript. We refine Example 1, as shown in Figure 1. Due to space constraints, we have put our LicenseScript code in the appendix of this paper. We briefly illustrate the steps involved in the process here:

0 Initially, both the digital libraries (`uni_lib` and `city_lib`) and the authorized domains (`cs_rge` and `math_rge`) have established business contracts with the packager (`pack`). We use the LicenseScript object `con` (for digital libraries) and `ser` (for authorized domains) to capture the attributes of the contracts, e.g. expiry date, compensation, etc. Each authorized domain has a LicenseScript object `dom`, which stores a list of compliant devices and users identities.

1 Alice makes a request from her `pda` to `cs_rge` for the `clip` from `uni_lib`. A LicenseScript object `req` is created, which stores her requirements of the `clip`, e.g. resolution, and other information such as the identity of her `pda`.

2 `cs_rge` checks if Alice's `pda` belongs to the authorized domain. If the check is successful, `cs_rge` forwards `req` to `pack`, after updating some of the data, e.g. to enter the identity of `cs_rge`.

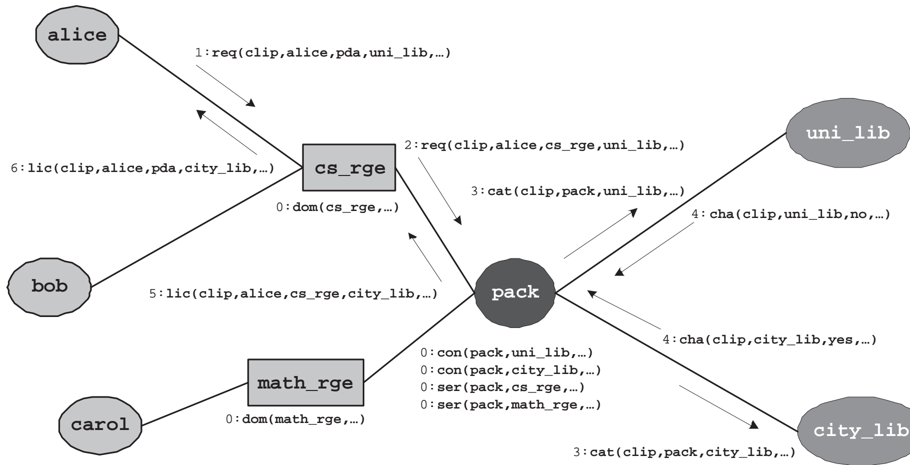

Fig. 1. A service management infrastructure with a packager.

3 pack requests to inspect the catalogue cat of both uni_lib and city_lib for the whereabouts and detailed characteristics cha of clip.

4 pack receives cha of clip, which stores the status and other properties of clip, from uni_lib and city_lib. pack first compares req with cha from uni_lib. If clip is not available or it does not match Alice's requirements (e.g. for quality), it checks with cha from city_lib.

5 pack generates a LicenseScript license lic if the validation of req with one of the cha's succeeds. pack sends lic to cs_rge.

6 cs_rge assigns lic to Alice's pda. Alice can then use her pda to render clip. If compensation is required by city_lib, Alice has to pay before accessing clip.

We have omitted all error reports here to avoid cluttering the presentation. We have also omitted payment and the transmission of the actual clip from city_lib to Alice's pda because it is not the main focus of our paper. We emphasize that, as suggested by the figure, all objects are dynamically generated, including, at step 5, the license.

The role of the broker is central to our infrastructure and the service management it provides. Yet the precise details of matching the content available to the user's requests is fully programmable. For example, different contracts will contain different rules about this matching and all other relevant aspects of service brokerage.

3 Prototype

In the prototype, the Prolog-based components include: the ECL^{i}S^e Prolog inference engine and the LicenseScript Interpreter. In addition, we have a Java user interface and a RMI interface with RGE components, such as the Tomcat Server, MySQL database, JDBC database interface, etc: 50 JSP (Java Server Page) files, 20 SWF (Shockwave Flash) files. The reader is invited to refer to our technical report [2] for more details on our prototype.

4 Conclusions

We present the concept of the *packager*, which acts as a service broker in inter-library service management. We present its implementation in our Prolog based LicenseScript language. To represent complex services in a flexible and efficient manner one needs to employ executable (mobile) code of some kind. Prolog is perfect for this. Services should not only be executable, but should have a clear and concise semantics (after all, they are *licenses*). The close relation between operational and the declarative semantics of Prolog is an invaluable advantage. Prolog is ideal to match requirements, and good at resolving conflicts. Therefore it is a natural platform for service brokerage.

References

1. C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, W. Jonker, and Y. W. Law. LicenseScript: A novel digital rights language and its semantics. In K. Ng, C. Busch, and P. Nesi, editors, *3rd International Conference on Web Delivering of Music (WEDELMUSIC)*, pages 122–129, Los Alamitos, California, United States, September 2003. IEEE Computer Society Press.
2. C. N. Chong, S. Etalle, P. H. Hartel, R. Joosten, and G. Kleinhuis. Service brokerage with prolog. Technical Report TR-CTIT-04-14, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, February 2004.
3. C. N. Chong, Y. W. Law, S. Etalle, and P. H. Hartel. Approximating fair use in LicenseScript. In T. M. T. Sembok, H. B. Zaman, H. Chen, S. R. Urs, and S. H. Myaeng, editors, *6th International Conference of Asian Digital Libraries (ICADL'2003)*, volume 2911 of *LNCS*, pages 432–443. Springer-Verlag, December 2003.
4. C. N. Chong, R. van Buuren, P. H. Hartel, and G. Kleinhuis. Security attribute based digital rights management (SABDRM). In F. Boavida, E. Monteiro, and J. Orvalho, editors, *Joint Int. Workshop on Interactive Distributed Multimedia Systems/Protocols for Multimedia Systems (IDMS/PROMS)*, volume 2515 of *LNCS*, pages 339–352. Springer-Verlag, November 2002.
5. R. M. Dijkman, L. F. Pires, and S. M. M. Joosten. Calculating with Concepts: a technique for the development of business process support. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Proceedings of the UML 2001 Workshop on Practical UML-Based Rigorous Development Methods*, volume 7 of *Lecture Notes in Informatics*, pages 87–98. GI-Edition, October 2001.
6. J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *Proceedings of 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 187–201. IEEE Computer Society Press, July 2003.
7. R. Joosten, J-W. Knobbe, P. Lenoir, H. Schaafsma, and G. Kleinhuis. Specifications for the rge security architecture. Technical Report Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands, August 2003.
8. A. Silva and J. Delgado. Agentspace as a framework to support interlibrary cooperation. In *Proeedings of 5th International Conference Crimea 1998*, page To appear, 1998.
9. S.A.F.A. van den Heuvel, W. Jonker, F.L.A.J. Kamperman, and P.J. Lenoir. Secure content management in authorised domains. In *The World's Electronic Media Event IBC 2002*, pages 467–474, September 2002.

Appendix: LicenseScript code

LicenseScript is based on logic programming and multiset rewriting. The basic construct is the license, which has the following form:

$$\mathrm{object_name}(Content, Clauses, Bindings)$$

Here $\mathrm{object_name}$ is the name of the LicenseScript object; $Content$ is the unique identifier of the associated content; $Clauses$ is a list of Prolog clauses that decide if the operations requested are allowed or forbidden; and $Bindings$ is a list of attributes that carry the status of the LicenseScript object.

A clause has the following form:

```
head :- body_1,body_2,...,body_n.
```

Here `head` is the head of the clause (i.e., the name and parameters of the clause), and the conjunction of `body_1,...,body_n` is the body of the clause. Authorization rules, obligations, conditions and mutability are captured by the clauses.

LicenseScript licenses are notionally gathered in a multiset on which multiset rewrite rules operate. These capture aspects of communication and updates. The rewrite rules take the following form:

$$\begin{aligned} rule_name(arguments) \quad : \quad & multiset_1 \longrightarrow multiset_2 \\ & \Longleftarrow conditions \end{aligned}$$

Here $rule_name$ is the name for the rule; $arguments$ are the arguments for this rule; $multiset_1$ and $multiset_2$ refer to the multiset of before and after the execution of the rule, respectively; and $conditions$ must be satisfied for the rule to apply. The conditions invoke queries over the clauses.

To describe inter-library service management with the packager in LicenseScript, we refine Example 1. The description follows closely that of section 2 (we suggest that the reader skips step 0 on a first reading):

0 Initially, a number of digital libraries have established business contracts with the packager to perform service brokerage. For instance, the LicenseScript contract `con`, which captures the contract between `uni_lib` and `pack`, can be written as follows:

```
con(pack,
[(cancomply(Bcon1,Bcon2,Breq1,Breq2,Bcha1,Bcha2,Blic) :-
    get_value(Bcon1,allowable_domains,Domains),
    get_value(Breq1,domain,D),is_member(D,Domains),
    get_value(Breq1,requirements,Reqs),
    get_value(Bcha1,properties,Chas),
    validate_requirements(Reqs,Chas),
    set_value(Bcon1,paid,true,Blic),
    set_value(Bcha1,availability,false,Bcha2)),
 (cancompensate(Bcon1,Bcon2,Breq1,Breq2,Blic,Com):-
    get_value(Breq1,digital_library,DL1),
```

```
    get_value(Bcon1,digital_library,DL2),DL1 =/= DL2,
    get_value(Bcon1,compensation,C),
    set_value(Bcon1,compensation,C,Blic),
    set_value(Bcon1,paid,false,Blic),
 (canseek(Bcon1,Bcon2) :-
    true)]
[digital_library=uni_lib,compensation=0.1,
 currency=euro,paid=false,
 allowable_domains=[cs_rge,math_rge]])
```

Here, `get_value(W,X,Y)` and `set_value(W,X,Y,Z)` are primitives to get (respectively set) the value `Y` associated with `X` in binding list `W`; binding `allowable_domains` stores a list of authorized domains that are allowed to render the resources; `digital_library` stores the identity of the university (`uni_lib`) to which this request is made; and `validate_requirements(·)` is a function to perform requirements validation, which will be explained later.
The clauses `cancomply`, `cancompensate` and `canseek` are Prolog clauses to determine if the operation performed by a user is allowed or forbidden: `cancomply` checks whether the user's authorized domain is authorized to access the resource, and validates the requirements of the user with the characteristics of the resource (in our case, `paid` indicates whether the compensation is made, and `availability` indicates whether the resource is available); `cancompensate` determines whether compensation is required by the digital library, it sets the binding `compensation` in `Blic` (of `lic`) to the value of `compensation` in `Bcon` (of `con`); `canseek` allows `pack` to ask for updated information of the digital resource from the digital libraries, which will be explained later.
Similarly, different residential gateways environments (RGE) have established service contracts with the packager for service brokerage management. For instance, a simple form of the LicenseScript service contract `ser` between `pack` and `cs_rge` can be written as follows:

```
ser(pack,
[canrequestser(Device,B1,B2) :-
    get_value(B1,domain,D),D==Device],
[domain=cs_rge])
```

Here, `canrequestser(·)` is a clause that determines whether `Device` is allowed to make a request to `pack`.
Initially, each residential gateway (RG) has a LicenseScript domain list `dom`, which stores a list of compliant device identities `devices` and user identities `users`:

```
dom(cs_rge,
[canrequestdom(Subject,Device,B1,B2) :-
    get_value(B1,devices,Ds),
    get_value(B1,users,Us),
    is_member(Device,Ds),is_member(Subject,Us)],
[devices=[pda,computer],users=[alice,bob]])
```

Here, `canrequestdom(·)` is a clause that determines whether `Subject` is allowed to make a request to `cs_rge` using `Device`.

1 Alice uses her pda to send a request to cs_rge to ask for clip from uni_lib. The LicenseScript object req captures the necessary data (for brevity, here we omit the details of creating req by clauses and rules):

```
req(clip,
[...],
[requestor=alice,digital_library=uni_lib,device=pda,
 requirements=[availability=true,resolution=100]])
```

Here, binding requestor stores Alice's identity; digital_library stores the library identity from which Alice asks for the clip. There are two requirements stored in the list requirements, namely availability indicates the clip must be available; and resolution denotes the minimum required resolution of the clip.

2 cs_rge checks if pda belongs to the authorized domain. If the check is successful, cs_rge forwards the request to the packager pack after updating the binding device (compare to req from step 1):

```
req(clip,
[...],
[requestor=alice,digital_library=uni_lib,device=cs_rge,
 requirements=[availability=true,resolution=100]])
```

3 pack sends LicenseScript object cat to uni_lib and city_lib, respectively requesting the updated characteristics of clip by executing the rule seek (with canseek as shown in step 0):

```
seek(Object) :
    con(pack,Ccon,Bcon1) ->
    con(pack,Ccon,Bcon1),
    cat(Object,Ccon,Bcon2)
 <= Ccon |- canseek(Bcon1,Bcon2)
```

The LicenseScript object cat for uni_lib is as follows:

```
cat(clip,
[...],
[packager=pack,digital_library=uni_lib])
```

4 pack validates Alice's request req with cha of clip from uni_lib.
We use a LicenseScript object cha to capture characteristics, i.e., current status (e.g. availability etc.) and other properties (e.g. resolution, number of pages, etc.) of a digital library resource. The packager receives cha from uni_lib and city_lib. For instance, cha of clip from uni_lib is:

```
cha(clip,
[...],
[digital_library=uni_lib,borrower=bob,
 properties=[availability=no,resolution=500]])
```

Here, the binding properties stores a list of properties of clip, i.e., availability and resolution.
The packager validates the requirements for step 2 using a parametric approach, in which the list of requirements to be complied with is compared to the characteristics by the function validate_requirements:

```
validate_requirements([],[]).
validate_requirements([[Req_name|Req_value]|Reqs],Chas) :-
    get_value(Chas,Req_name,Cha_value),
    check_requirement(Req_value,Cha_value),
    validate_requirements(Reqs,Chas).
```

Here, `check_requirement(R,C)` is a function to check the requirement value `R` and the characteristic value `C`. We simply use (in)equalities to compare two values in `check_requirements(R,C)`:

```
check_requirements(Req_value,Cha_value) :-
    Req_value == Cha_value.
```

We can define a more complex and flexible requirement validation policy by using more elaborate data structures for the requirements and characteristics, and a matching `check_requirements` rule.
To validate and permit Alice's request, the packager executes multiset rewrite rule `permit`:

```
permit(Object) :
    con(pack,Ccon,Bcon1),
    req(Object,Creq,Breq1),
    cha(Object,Ccha1,Bcha1) ->
    con(pack,Ccon,Bcon2),
    cha(Object,Ccha2,Bcha2),
    lic(Object,Clic,Blic)
 <= Ccon |- cancomply(Bcon1,Bcon2,Breq1,Breq2,Bcha1,Bcha2,Blic),
    Ccon |- cancompensate(Breq1,Breq2,Bcon1,Bcon2,Blic)
```

Here, `Ccon |- cancomply(·)` and `Ccon |- cancompensate(·)` are the two conditions of the rule `permit`.
In our example, the validation fails because `clip` is not available at `uni_lib`, therefore the packager validates `req` with `cha` from `city_lib`. As the `clip` is available at `city_lib`, a LicenseScript `lic` is generated for Alice to use the `clip`:

```
lic(clip,
[(canassign(B1,B2,Subject,Device1,Device2) :-
    get_value(B1,requestor,S),S==Subject,
    get_value(B1,device,D),D==Device1,
    set_value(B1,device,Device2,B2)),
 (canview(B1,B2,Subject,Device) :-
    get_value(B1,device,D),D==Device,
    get_value(B1,requestor,S),S==Subject,
    get_value(B1,paid,P),P=true)),
 (canpay(B1,B2) :-
    get_value(B1,compensation,C),
    get_value(B1,digital_library,DL),
    pays(C,DL),set_value(B1,paid,true,B2))],
[requestor=alice,paid=false,digital_library=city_lib,
 device=cs_rge,compensation=0.1])
```

Here, there are three clauses, namely `canassign`, which can be invoked to decide if `Subject` can assign the license from `Device1` to `Device2`; `canview`, which determines if `Subject` allows to view `clip` on `Device`; and `canpay`, which is executed to make the payment. `pays(X,Y)` is a primitive to perform payment of money `X` to entity `Y`. When the compensation is made, the binding `paid` is set to `true`.

5 The packager then sends the license `lic` to `cs_rge`. `cs_rge` can then assign this license to Alice's `pda` by executing the following rule:

```
assign(Object,Subject,D1,D2) :
    lic(Object,Clauses,B1)->
    lic(Object,Clauses,B1),
    lic(Object,Clauses,B2)
 <= Clauses |- canassign(B1,B2,Subject,D1,D2)
```

6 Alice can view the `clip` by executing the multiset rewrite rule `view` (i.e., clicking a "View" button on her PDA):

```
view(Object,Subject,Device) :
    lic(Object,Clauses,Bindings1) ->
    lic(Object,Clauses,Bindings2)
 <= Clauses |- canview(Bindings1,Bindings2,Subject,Device)
```

However, to view the `clip`, Alice is required to pay the compensation to `city_lib` by executing multiset rewrite rule `pay`:

```
pay(Object) :
    lic(Object,Clauses,Bindings1) ->
    lic(Object,Clauses,Bindings2)
 <= Clauses |- canpay(Bindings1,Bindings2)
```

Here, the consequence of executing rule `pay`, the binding `paid` is set to `true`, which indicates that Alice is now allowed to view `clip`.