INTEGRATION OF REPOSITORIES IN ELEARNING SYSTEMS

José Paulo Leal[1] and Ricardo Queirós[2]

[1]*CRACS/INESC-Porto & DCC/FCUP, University of Porto, Portugal*
zp@dcc.fc.up.pt

[2]*CRACS/INESC-Porto & DI/ESEIG/IPP, Porto, Portugal*
ricardo.queiros@eu.ipp.pt

Keywords: eLearning, Repositories, Learning Objects, LMS, Interoperability.

Abstract: The wide acceptance of digital repositories today in the eLearning field raises several interoperability issues. In this paper we present the interoperability features of a service oriented repository of learning objects called crimsonHex. These features are compliant with the existing standards and we propose extensions to the IMS interoperability recommendation, adding new functions, formalizing message interchange and providing also a REST interface. To validate the proposed extensions and its implementation in crimsonHex we developed a repository plugin for Moodle 2.0 that is expected to be included in the next release of this popular learning management system.

1 INTRODUCTION

In recent years several initiatives to integrate eLearning systems have emerged. The goal of these initiatives, such as specifications and frameworks, is to facilitate the integration between heterogeneous systems. Learning objects (LO) are the corner stone of interoperability in pedagogical eLearning systems, thus the integration of repositories of LOs is particular important in this context.

This paper builds upon previous work (Leal, 2009) on the design and implementation of crimsonHex - a service oriented repository of LOs. The repository provides standard compliant repository services to a broad range of eLearning systems, exposing its functions using two alternative web services flavours. In this paper we highlight the interoperability features of crimsonHex. For sake of standard compliance these features are based on IMS Digital Repositories Interoperability (DRI) specification (DRI, 2003). Our experience in using these recommendations lead us to propose extensions to its set of functions and to the XML binding that currently lacks a formal definition. To evaluate the proposed extensions to the IMS DRI specification and its implementation in the crimsonHex repository, we developed a crimsonHex plugin for the 2.0 release of the popular Moodle LMS. Moodle 2.0 users will be able to download LOs from crimsonHex repositories since this LMS is expected to include the plugin described in this paper in its distribution.

The remainder of this paper is organized as follows: Section 2 traces the evolution of eLearning systems with emphasis on the existing repositories. In the following section we introduce the crimsonHex repository and its application interfaces. Then, we provide basic implementation details of a crimsonHex plugin for Moodle 2.0 using the proposed IMS DRI extensions. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 LEARNING OBJECTS REPOSITORIES

A learning object is a digital, self-contained, reusable unit to support learning (Beck, 2008). A learning object can be as small as a single image or as large as a complete online course and usually

comes in the form of HTML/PDF files, Flash, QuickTime movies and others (Casey, 2007). Usually, they are described with standard metadata, packaged and stored in digital repositories to be easily searchable. The need for this kind of repositories is growing as more educators are eager to (re)use digital educational contents and more of it is available.

A repository of LOs can be defined as a 'system that stores electronic objects and meta-data about those objects' (Holden, 2004). There are several online repositories or collections of LOs worldwide (e.g. MERLOT, Wisc-Online). The Jorum Team made a comprehensive survey (JORUM, 2006) of the existing repositories and noticed that most of these systems do not store actual LOs. They just store metadata describing LOs, including pointers to their locations on the Web, and sometimes these pointers are dangling.

The repositories usually offer several features including upload/download, single/federated search, comment/review and collection management. Despite these features, existent repositories present integration and interoperability issues. For example, the LOs in the previously cited repositories must be manually imported into an LMS. An evaluation engine (EE) cannot query the repository and automatically import the LOs it needs. In summary, most of the current repositories are specialized search engines of LOs and not adequate for interoperating with other eLearning systems, such as an automatic evaluation engine.

Surveys (Holden, 2004) show that users are very concerned with interoperability issues. Some major interoperability efforts (Hatala, 2004) were made in eLearning, such as NSDL, POOL, EduSource and IMS DRI. The IMS DRI specification was created by the IMS Global Learning Consortium (IMS GLC) and provides a functional architecture and reference model for repository interoperability. The IMS DRI provides recommendations for common repository functions, namely the submission, search and download of LOs. It recommends the use of web services to expose the repository functions based on the Simple Object Access Protocol (SOAP, 2007).

3 CRIMSONHEX REPOSITORY

In this section we introduce the crimsonHex repository and we present its application interface (API) used both internally and externally. Internally the API links the main components of the repository. Externally the API exposes the functions of the repository to third party systems. To promote the integration with other eLearning systems, the API of the repository adheres to the IMS DRI specification. The IMS DRI specifies a set of core functions and an XML binding for these functions. In the definition of API of crimsonHex we needed to create new functions and to extend the XML binding with a Response Specification language. The complete set of functions of the API and the extension to the XML binding are both detailed in this section.

3.1 Architecture

The architecture of the crimsonHex repository relies on an API where the repository exposes a set of functions implemented by a core component that was designed for efficiency and reliability. All other features are relegated to auxiliary components, connected to the central component using this API. Other eLearning systems can be plugged into the repository using also this API. Thus, the architecture of crimsonHex repository is based on **the Core component** that exposes the main features of the repository, both to external services, such as the LMS and the EE, and to internal components - the Web Manager and the Importer. In the remainder we focus on the Core component, more precisely, its API and we introduce a new language for message interchange.

3.2 Applications Interface

The IMS DRI recommends exposing the functions as SOAP web services. Although not explicitly recommended, other web service interfaces may be used, such as the Representational State Transfer (REST) (Fielding, 2000). We chose to expose the repository functions in these two distinct flavours. SOAP web services are usually action oriented, especially when used in Remote Procedure Call (RPC) mode and implemented by an off-the-shelf SOAP engine such as Axis. REST web services are object (resource) oriented and implemented directly over the HTTP protocol, mostly to put and get resources. The reason to provide two distinct web service flavours is to encourage the use of the repository by developers with different interoperability requirements. A system requiring a formal an explicit definition of the API in Web Services Description Language, to use automated tools to create stubs, will select the SOAP flavour. A lightweight system seeking a small memory footprint at the expense of a less formal definition of the API will select the REST flavour. The repository functions exposed by the Core are summarized in Table 1.

Table 1: Core functions of the repository.

Function	SOAP	REST
Reserve	*XML getNextId(URL colectionl)*	*GET URL?nextId > URL*
Submit	XML submit(URL loid, LO lo)	PUT URL < LO
Request	LO retrieve(URL loid)	GET URL > *LO*
Search	XML search(XQuery query)	POST URL < XQUERY > XML GET URL?name1=value1&…> XML
Alert	RSS getUpdates()	GET URL?alert+seconds > RSS
Report	*XML Report(URL loid,Report rp)*	*PUT URL < LOREPORT*
Create	*XML Create(URL collection)*	*PUT URL*
Remove	*XML Remove(URL collection)*	*DELETE URL*
Status	*XML getStatus()*	*GET URL?status > XML*

Each function is associated with the corresponding operations in both SOAP and REST. The lines formatted in italics correspond to the new functions added to the DRI specification, to improve the repository communication with other eLearning systems.

To describe the responses generated by the repository we defined a **Response Specification** as a new XML document type formalized in XML Schema.

The advantage of this approach is to enable client systems to achieve more information from the server and be able to standardize the parsing and validation of the HTTP responses. Figure 1 depicts the elements of the new language and their types.

The schema defines two top level elements: `result` and `rss`. The former will be used by all the functions except the Alert function that returns a feed compliant with the Really Simple Syndication (RSS) 2.0 specification. The `result` element contains the following child components:

- `base-url` attribute, defining a base URL for the relative URLs in the response;
- `request` element, containing the full request URL and an human readable request message;
- `error` element, containing an error message - client systems will search for this element to verify the existence of errors;
- `response` element, describing a successful execution of the function - it's composed by an human readable response message and, for some functions, by a `resources` element that groups a set of resources defined individually in `resource` elements.

A `resource` element contains an identification of the collection absolute path (attribute `idCol`) and an identification of the LO itself (attribute `idLo`).

In the remainder of this section we enumerate the Core functions of the repository, describing both the request and response data. For sake of simplicity we illustrate the requests using the REST interface since these can be used as command lines in a Linux system shell.

The **Register/Reserve** function requests a unique ID from the repository. We separated this function from Submit/Store in order to allow the inclusion of the ID in the meta-data of the LO itself. This ID is an URL that must be used for submitting or retrieving an LO. The producer may use this URL as an ID with the guarantee of its uniqueness and with the advantage of being a network location from where the LO can be downloaded.

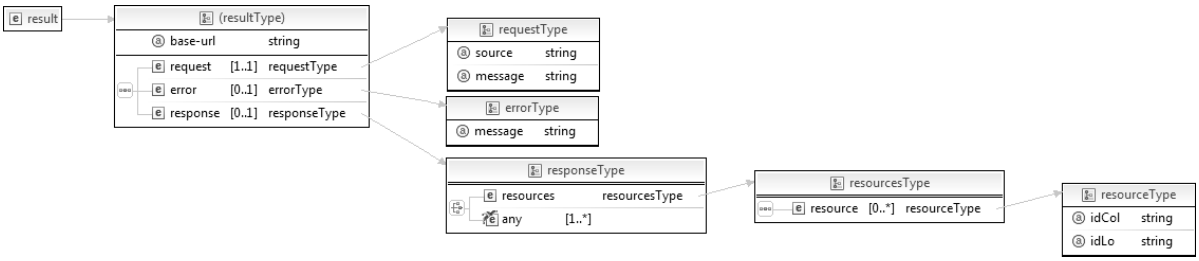

Figure 1: Response specification schema.

This action is performed by sending a GET HTTP request to the server, as in the next example.

```
GET http://server/ch/lo?nextId > URL
```

The HTTP response includes an XML file complying with the Response Specification and containing all the details of the response generated by the Core. Nevertheless, in this particular function and for convenience of programmers using REST, the HTTP *Location* header contains the URL returned by the server.

```
Location: http://server/ch/lo/3
```

The **Submit/Store** function uploads an LO to a repository and makes it available for future access. This operation receives as argument an IMS CP compliant file and an URL generated by the Reserve function. This operation validates the LO conformity to the IMS Package Conformance and stores the LO in the internal database. To send the LO to the server we could use, in the REST flavour, the PUT or the POST HTTP methods. An example using the POST syntax is the following.

```
POST http://server/ch/lo/3 < LO
```

The repository responds with submission status data compliant with the Response Specification.

The **Search/Expose** function enables the eLearning systems to query the repository using the XQuery language, as recommended by the IMS DRI. This approach gives more flexibility to the client systems to perform any queries supported by the repository's data. To write queries in XQuery the programmers of the client systems need to know the repository's database schema. These queries are based on both the LO manifest and its usage reports, and can combine the two document types. The client developer needs also to know that the database is structured in collections. A collection is a kind of a folder containing several resources and sub-folders. From the XQuery point of view the database is a collection of manifest files. For each manifest file there is a nested collection containing the usage reports. As an example of a simple search, suppose you want to find all the titles of LOs in the root collection whose author is Manzoor. The XQuery file would contain the data.

```
declare namespace imsmd = “http://...”;
for                  $p    in    //imsmd:lom
where   contains($p//imsmd:author,’Manzoor’)
return  $p//imsmd:title//text()
```

After creating the XQuery file you can use the following POST request.

```
POST http://server/ch/lo < XQUERY
```

Alternatively, you can use a GET request with the searched fields and respective values as part of the URL query string, as in the following example.

```
GET http://server/ch/lo?author=Manzoor
```

Queries using the GET method are convenient for simple cases but for complex queries the programmer must resort to the use of XQuery and the POST method. In both approaches the result is a valid XML document such as the following.

```
<result base-url="http://server/ch/lo/">
 <request
  source="http://server/ch/lo/"
  message="Querying repository" />
 <response message="3 LOs found...">
   <resources>
      <resource idCol="" idLo="5">
        Hashmat the Brave Warrior
      </resource>
      <resource idCol="" idLo="123">
        Summation of Four Primes
      </resource>
      <resource idCol="graphs/" idLo="2">
        InCircle
      </resource>
   </resources>
 </response>
</result>
```

The **Report/Store** function associates a usage report to an existing LO. This function is invoked by the LMS to submit a final report, summarizing the use of an LO by a single student. This report includes both general data on the student's attempt to solve the programming exercise (e.g. data, number of evaluations, success) and particular data on the student’s characteristics (e.g. gender, age, instructional level). With this data, the LMS will be able to dynamically generate presentation orders based on previous uses of LO, instead of fixed presentation orders. This function is an extension of the IMS DRI.

The **Alert/Expose** function notifies users of changes in the state of the repository using a RSS feed. With this option a user can have up-to-date information through a feed reader. Next, we present an example of a GET HTTP request.

```
GET http://server/ch/lo?alert+seconds > RSS
```

The repository responds with an RSS document.

The **Create** function adds new collections to the repository. To invoke this function in the REST interface the programmer must use the PUT request method of HTTP. The only parameter is the URL of the collection.

```
PUT http://server/ch/lo/newCol
```

The following is an example of the repository response to a create function.

```
<result base-url="http://server/ch/lo/" ...>
  <request
    source="http://server/ch/lo/newCol"
    message="Creating new collection" />
  <response message="Collection created">
    <resource idCol="newCol" idLo=""/>
  </response>
</result>
```

The **Remove** function removes an existent collection or learning object. This function uses the DELETE request method of HTTP. The only parameter is an URL identifying the collection or LO, as in the following example.

```
DELETE http://server/ch/lo/123
```

The following is an example of the repository response to a remove function.

```
<result base-url="http://server/ch/lo/" ...>
  <request
    source="http://server/ch/lo/123"
    message="Deleting a LO" />
  <response message="LO deleted">
    <resource idCol="" idLo="123" />
  </response>
</result>
```

The **Status** function returns a general status of the repository, including versions of the components, their capabilities and statistics. This function uses the GET request method of HTTP, as in the following example.

```
GET http://server/ch/lo?status
```

The repository responds with status data compliant with the Response Schema Specification.

4 INTEGRATION WITH MOODLE

To validate the interoperability features of the crimsonHex repository we integrated it with Moodle, arguably the most popular LMS nowadays. In this section we present the new APIs for Moodle 2.0 plugins and we provide basic implementation details of a plugin for crimsonHex repositories.
The development of this plugin was straightforward. In terms of programming effort we spent half a day to produce approximately 100 new lines of code. This quick and simple integration benefited from the new interoperability features of the repository.

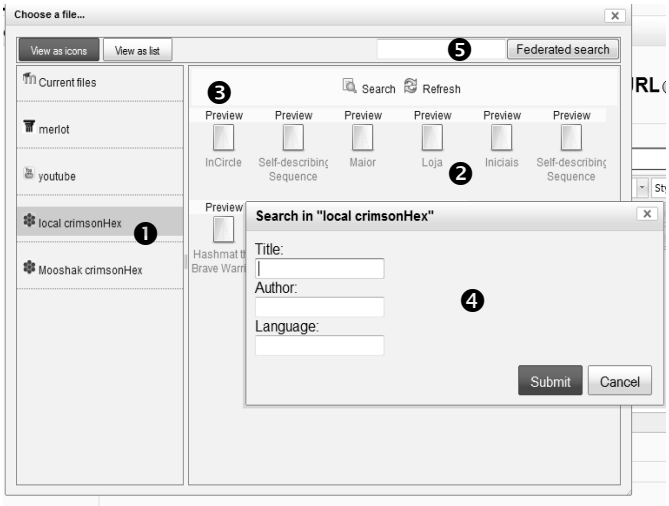

Figure 2: crimsonHex plugin interface.

The beta version of Moodle 2.0 is due in February 2010 and will include support for different types of repositories. Several API are already available to enable the development of plugins by third parties, including:

File API for managing internal repositories;

Repository API for browsing and retrieving files from external repositories;

Portfolio API for exporting Moodle content to external repositories.

We chose the Repository API for testing the integration features of the crimsonHex repository in Moodle. The goal of this particular API is to support the development of plugins to import content from external repositories. The Repository API is organized in two parts: Administration, for administrators to configure their repositories, and; File picker, for teachers to interact with the available repositories. Each with its own graphical user interface (GUI). In Figure 2 we present the file picker GUI of the crimsonHex plugin. On the left panel are listed the available repositories as defined by the administrator. Two crimsonHex repository instances are marked with label 1. Label 2 marks the default listing of the selected repository. Pressing the "Preview" link marked with 3 presents a preview of the respective LO. Pressing the "Search" link pops-up a simple search form, marked as 4 in Figure 2. For federated search in all available crimsonHex repositories is used the text box marked as 5.

5 CONCLUSIONS AND FUTURE WORK

In this paper we present the interoperability features of crimsonHex - a repository of learning objects. These features were designed based on the IMS Digital Repository Interoperability and we propose several extensions to this specification. These extensions include new functions and a formal definition of a response specification for the complete function set. To evaluate the proposed extensions we implemented a plugin for 2.0 release of Moodle that uses the new interoperability features of crimsonHex.

The main contributions of this work are the proposed extensions to the IMS DRI specification, the improved interoperability features and a plugin to be included in the Moodle 2.0 distribution. The improved interoperability of crimsonHex is expected to support the development of new eLearning tools requiring greater interoperability with repositories. The repository plugin will facilitate the use of crimsonHex by Moodle users. In its current status crimsonHex is available for test and download at the site of the project (crimsonHex, 2009).

Adding authoring features to the crimsonHex is the next step in this research. Creating LOs with metadata of good quality is a challenge since the typical author of eLearning content usually lacks the knowledge of metadata standards. This is also an interoperability issue since the LMS is where eLearning content is tested and used in first place but repositories are the appropriate place to promote content reuse as LOs. We plan to continue using Moodle's repository APIs for that purpose, in particular the Portfolio API. A plugin using this API will enable the content author to upload learning content to crimsonHex and create a new LO with the essential metadata. Them, using the authoring features of crimsonHex, the content author will be assisted in refining the LO metadata.

REFERENCES

Leal, J.P., Queirós, R., 2009. CrimsonHex: a Service Oriented Repository of Specialised Learning Objects. In: *ICEIS 2009: 11th International Conference on Enterprise Information Systems, Milan*.

IMS DRI - IMS Digital Repositories Interoperability, 2003. Core Functions Information Model, URL: http://www.imsglobal.org/digitalrepositories.

Beck, Robert J., 2008. What Are Learning Objects? *Learning Objects, Center for International Education, University of Wisconsin-Milwaukee*.

Casey, J., McAlpine, M., 2007. Writing and Using Reusable Educational Materials - A Beginners Guide. *CETIS Educational Content Special Interest Group*. URL: http://zope.cetis.ac.uk/educational-content/.

Holden, C., 2004. What We Mean When We Say "Repositories" User Expectations of Repository Systems. In: *Academic ADL Co-Lab*.

JORUM team, 2006. E-Learning Repository Systems Research Watch. *Technical report*.

Hatala, M., Richards, G., Eap, T., Willms, J., 2004. The EduSource Communication Language: Implementing Open Network for Learning Repositories and Services. In: *ACM symposium on Applied computing*.

SOAP (Simple Object Access Protocol), Version 1.2, 2007. Part 0: Primer, 2nd edition. URL: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

Fielding, R., 2000. Architectural Styles and the Design of Network-based Software Architectures, *Phd dissertation*. URL: http://www.ics.uci.edu/~fielding /pubs/dissertation/rest_arch_style.htm.

crimsonHex – project web site, 2009. URL: http://www.dcc.fc.up.pt/crimsonHex.