A dynamic OAI Publisher Service in the DRIVER Repository Infrastructure

Michele Artini

Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"
CNR Via G. Moruzzi, 1 - 56124 PISA – Italy
Email: michele.artini@isti.cnr.it

Abstract—OAI-Publishers are software modules enabling harvesting of metadata records describing the digital resources of a Digital Repository. Typically, such modules are manually configured and implemented to export records contained into a local store according to a predefined number of metadata formats. Due to their static nature, these are not apt to the dynamic context of Repository Infrastructures, where a virtual store, made of a dynamic pool of distributed physical stores, may hosts records of unpredictable metadata formats. In this document, we describe the internal architecture of the OAI-Publisher Service built for the DRIVER Repository Infrastructure. The Service is capable of self-configuring its internals so as to adapt to the variable content of the virtual store. Finally, we shall describe some extensions for the creation of dynamic collections and formats.

I. INTRODUCTION

In the Digital Library (DL) context, the term *Repository* generally refers to "containers of digital objects", namely technologies for maintaining a *store* of digital resources; *Fedora* [1], *DSpace* [2] are examples of known Repository technology. OAI-PMH compliant Repositories (*OAI-Repositories*) are Repositories endowed with an *OAI-Publisher* component, which is a software component implementing the OAI-PMH protocol standard interface [3].

By adopting this standard, Repositories enable access to their digital resources as if they were *OAI-Items*, i.e. entities adhering to the OAI-Item Model [4] (See Figure 1). According to the model an OAI-Item represents the existence of a real world *resource* – be it digital or physical – and exposes its description, given in terms of a number of metadata records. Such records conform to specific metadata formats, one of which must be *Dublin Core* [5].

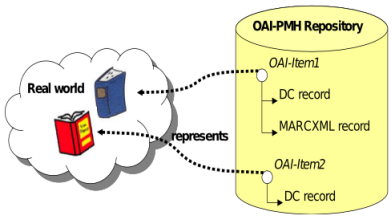

Figura 1: OAI-PMH Model

OAI-Publishers enable bulk-extraction, namely *harvesting*, of OAI-Items w.r.t. to one of the metadata formats exposed by the Repository metadata store. Typically, OAI-Publishers are build around the such store, not expecting changes in the exported metadata formats, and expecting only rare changes in the OAI-Sets to be exported. As such, the technology required for their implementation is quite ad-hoc, entailing only rare maintenance human interventions. å

The DRIVER Repository Infrastructure is a dynamic distributed environment, where several Service-oriented DL applications can coexist. An application is here intended as a dynamic pool of Services *orchestrated* by the *application logic* in order to deliver the functionality expected by the application user community. Service instances can join or leave the infrastructure at any time (*register* or *unregister*, in a peer-to-peer fashion) and be part of one ore more applications. In such dynamic environment applications can share between them content (e.g. metadata records), which they either harvested from OAI-Repositories or natively created into the infrastructure and placed in a highly distributed storage space, supported by special *Store Services*; the “union” of their distributed running instances forms a virtual store, namely the *Information Space*, populated and accessible by all DL applications in the infrastructure.

Currently, the largest DL application running over the DRIVER Infrastructure,[1] is called *DRIVER European Information Space*. The application includes a set of Aggregator Service instances, capable of harvesting records from valid OAI-Publishers of European Digital Repositories. The harvesting activity collects all records from OAI-Repositories in Europe, regardless of the kind of metadata records they yield and the size of their content. When a Repository is willing to join the Information Space, it registers to the infrastructure by providing the information required for its harvesting; i.e. geographical location, URL, metadata formats, and OAI-Sets. The application logic orchestrates (*i*) the Aggregator Service instances in order to harvest all metadata formats available from the Repository; and (*ii*) the Store Service instances in order to allocate the storage space required to host the incoming records.

The DRIVER OAI-Publisher Service must implement an OAI-PMH interface over the virtual store, i.e. the DRIVER Information Space, in order to provide the available records to whoever is willing to access them. Accordingly, it has to be devised to automatically adapt to the potential changes determined by the harvesting of new Repositories, so as to always provide OAI-PMH responses coherent with the actual content of the Information Space. In this document we present how this can be done in the dynamic environment of the DRIVER infrastructure and show how powerful variations of traditional OAI-Publisher technology can be added to this implementation.

A. Outline

In Section II we shall first describe the notion of Digital Library applications built on top of the DRIVER Repository Infrastructure, while in Section III we describe how the virtual DRIVER Information Space is maintained by the infrastructure. In Section IV we explain how self-configurable and powerful OAI-Publisher Services can be devised to operate over such Information Space. Finally, in Section VI we summarize the work done so far in such direction and underline possible future avenues.

II. DRIVER REPOSITORY INFRASTRUCTURE

Nowadays some DL user communities have changed their requirements, by posing no constraints on time, space, and functionalities. DL applications range from temporary work spaces to elaborate collaborative environments (e.g. DL endowed with peer-review mechanisms), and very large

[1] Largest in terms of resources involved, currently more than 350’000 records.

distributed-federated object stores, whose functionality may change very frequently in time. In order to face this new requirements we envisage the shift from DL applications based on standalone Repository technology to a loosely coupled Service-oriented architecture, which offers greater reusability and helps to reduce development and maintenance costs.

The aim of Repository Infrastructures is that of enabling the construction and maintenance of DL applications with higher orders of sustainability. Infrastructures exploit the Service-oriented architecture in order to provide distribution and sharing of content and functionality, i.e. Services offering efficient content storage/access and business logic. Organizations building their DL applications on top of an Infrastructure can profit from reusing the Services offered by other Organizations and add (therefore themselves share) their own Services when those available are not enough for their needs.
A real running Repository Infrastructure has been implemented in the DRIVER project [6]. The project has been funded by the European Commission with the goal to contribute in implementing the future *knowledge infrastructure* of the European Research Area. The project concluded in November 2007 and has focused on (*i*) the production and maintenance of a Repository Infrastructure and (*ii*) the construction of the first DL application on top of it. The application, named *European Information Space*, aims at harvesting *Open Access* content from all European OAI-Repositories into the Repository Infrastructure so as to make it accessible as a whole to world-wide academics and students trough advanced user functionalities. Other DL applications have been subsequently built, exploiting the Repository Infrastructure populated by the European Information Space application.

The DRIVER Infrastructure consists of three layers of running Service instances: Enabling, Data, and Application Layers.

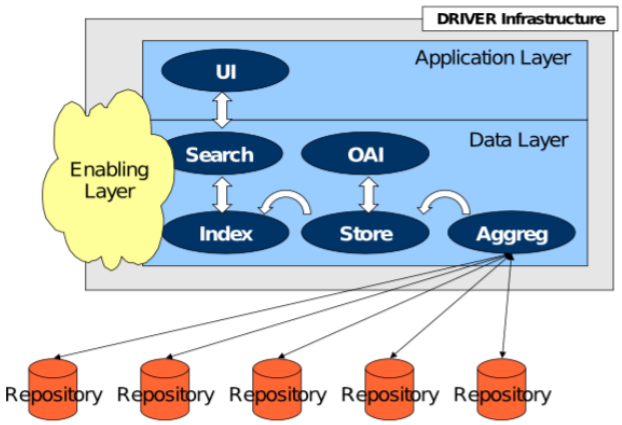

Figura 2: Driver infrastructure

A. Enabling Layer

The layer includes instances of Information Service, Manager Service, and Authentication and Authorization Service. These accomplish core tasks typical of "volatile" infrastructures, such as registering, unregistering, authenticating, authorizing Service instances residing at different nodes on the Internet. Differently from similar SOA-oriented architectures, an application is not here intended as a specific set of dedicated and interconnected Service instances. Apart from User Interfaces, applications can use any of the Service instances which are available in a certain moment in time and whose functionality is required to accomplish a certain operation. To this aim, the layer provides *resource discovery* and *orchestration logic* mechanisms. The former enable discovery of Service instances satisfying certain functional condition; searches are based on the Service *profile* information provided at Service instance registration time. The latter perform application activities by applying discovering techniques. For example, user running queries entail a number of orchestration logic activities. The query can be sent by the UI to any of the Search Service instances available (part of the Data Layer). To this aim, the logic discovers the "best" instance available (e.g. the less overloaded, the IP-closest), and sends the query to it. The same holds for the Search Service instance, which needs to discover the Index Service instances capable and available to run the query, decide which to use (query optimization), and then send them the query. In such scenario, for example, Index Service instances may be added/removed at any time, without the need of reconfiguring the available Search Services.

B. Data Layer

The layer builds on top of the Enabling Layer and includes instances of Store Service, Index Service, Search Service, Collection Service, Aggregation Service, and OAI-Publisher Service. As such, the Data Layer can be considered an application with a specific orchestration logic.

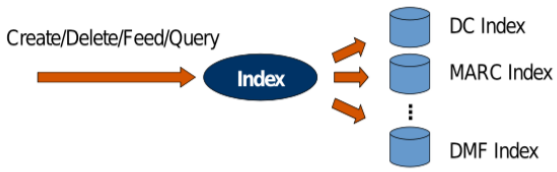

Figura 3: Index Factory Services

Store Services and Index Services instances, also called *factory services*, can return new empty stores and indexes to the consuming applications. Such entities are called *Data Structures* and are registered to the infrastructure with a descriptive profile. In practice, there is a distinction between a Store Service and the Store Data Structures (DS) it manages: consuming Services may ask a Store Service instance to generate a new Store DS for their use, as well as to put or retrieve some metadata records from the Store DS. Similarly, Index Service instances can be asked to create Index DSs, feed them with metadata records, or query them to search for records. The same Index and Store Service instances can therefore serve different DL applications, as well as the DS created in the context of one application can be reused in the context of another; i.e. sharing of Store DSs and Index DSs. This is the case for Search Service instances, which resolve queries by discovering the Index DSs capable of answering the query in the shortest amount of time. After identifying the candidate Index DSs, they need to interact with the responsible Index Services in order to send the query, merge the results, and send the answer to the consumer, e.g. the User Interface.

Aggregator Service instances can harvest records from OAI-Repositories, clean and enrich the records according to special rewriting rules (to be provided by an administrator), and store them into a given Store DS. Aggregator Services can also map records of an input metadata format into another possibly proprietary metadata format, thus generating records native to the specific DL application involved.

The Data Layer orchestration logic is in charge of maintaining the Information Space consistent and robust. It therefore reacts to operations such as Repository registering and unregistering to and from the infrastructure. For example, the registration of a Repository tells the infrastructure what metadata formats are to be harvested from the Repository. The logic assigns the Repository to a discovered Aggregation Service instance (currently it chooses the one in the same geographical region of the Repository), together with information on which Store DSs should be used to save the harvested records. In the case of need, the logic also creates the Store DSs required to host the incoming records: it first discovers the Store Service instances available to do so, creates the Store DSs, then informs, i.e. dynamically configures, the Aggregator Service involved on where to divert the harvested records. Furthermore, any new record should be inserted into at least one Index DS, in order to be reachable through searches. The logic ensures this, by sending the incoming records to the appropriate Index DS and creating new Index DSs when necessary. Finally, the logic ensures robustness by managing replication of Store and Index DSs and space optimization by removing obsolete DSs from the infrastructure.

C. Application Layer

The largest application running into the layer is the European Information Space application. It builds on top of the Data Layer and Enabling Layer by adding new Service typologies. Among these: User Profiling Services, Community Services, Recommendation Services, and User Interfaces. Its administrators operate a number of Aggregator Service instances to aggregate Open Access content of Repositories from countries including the Netherlands, Germany, France, Belgium, Italy, Ireland, Poland, Spain, and UK. Currently, the application has registered and harvested from 64 OAI-Repositories, reaching the threshold of about 300.000 Open Access OAI-Items from all over Europe.

III. DRIVER INFORMATION SPACE

In principle, one Index DS and one Store DS must contain records of the same metadata format, but not necessarily originating from the same Repository. Such a choice depends on the orchestration

logic, which can apply different storage and indexing policies exploiting the factory Service instances at hand. In particular, each Store and Index DS specifies in its registration profile the kind of metadata format it supports. More specifically, the infrastructure envisages special system Data Structures, called *MDFormat DSs*, created whenever a new metadata format is introduced into the system; this event can be implicit, when harvesting from a Repository records matching a format that was never harvested before, or explicit, when an Aggregator Service is used to generate records of an DL application-specific metadata format. In both cases, a new MDFormat DS is created and its profile registered to the system, specifying the name of the format, the relative XML schema, and other relevant information, not to be explained here.

The records harvested from the OAI-Repositories become part of a virtual Information Space (i.e. the union of all Store DSs), which virtually hosts so-called *DRIVER Objects*. One DRIVER object groups all records relative to one OAI-Item, plus extra records possibly needed by the DL application responsible of harvesting. For example, in the case of the European Information Space application, DRIVER Objects contain the original metadata records, plus a further record in *DRIVER Metadata Format* (DMF). DMF records are intended as metadata descriptions of the OAI-Item as a whole, thus include Dublin Core fields (resource description), fields containing keywords extracted from the full-text (resource synthesis), provenance-related fields (resource origin, e.g. Repository name, institution, geographical location, etc) and others. DMF records are uniform in content, meaning that values of their fields are derived from the records of the relative OAI-Item but chosen from well-defined DMF field domains.

As specified above, harvested records are kept in Store DSs dedicated to a given metadata format. Accordingly, records of the same OAI-Item, virtually belonging to the same DRIVER Object, are contained into different Store DSs. In DRIVER, records are uniquely identified in the Information Space with a triple: OAI-Item identifier (unique by definition in the context of the originating Repository), Repository identifier (unique and created by the infrastructure at Repository registration time), and Store DS identifier (unique and created by the infrastructure at DS registration time). Accordingly, DRIVER Objects are virtually identified by the pair Repository identifier and OAI-Item identifier.

Access Services enable random access to one metadata record, given its record identifier *(OAIid, RepId, StoreId)*. The Service discovers the location of the Store Service in charge of the Store DS identified by *StoreId* and fires it a request for retrieving the given record.

IV. DRIVER OAI-PUBLISHER SERVICE

The DRIVER OAI-Publisher Service is designed to support an OAI-PMH interfaces over the dynamic pool of Services and DSs available to the infrastructure. The Service exploits the information available into the Service and DS profiles of the infrastructure to self-configure its internals and accomplish its tasks with not need of human administration and maintenance. In particular, DRIVER OAI-Items are uniquely identified by the DRIVER Object identifier and contain all metadata records available to the DRIVER Object; OAI-Sets are intended as the Repositories harvested so far. The OAI-PMH methods are implemented as follows:

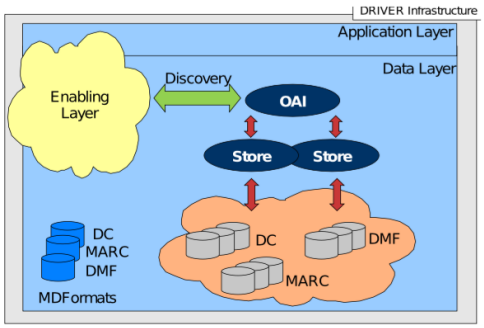

Figura 4: OAI Publisher

- **Identify:** the method returns the description of the "DRIVER Information Space Virtual Repository";
- **ListMetadataFormats:** the method discovers all MDFormat DSs registered to the infrastructure and constructs the appropriate response;
- **GetRecord:** the method expects an OAI-Item identifier and returns the relative record; to this aim, the Service discovers the first available Access Service and sends it a random access request; it then returns the record to the caller;
- **ListSets:** the method answers with the list of Repositories harvested so far;
- **ListRecords:** the method expects a metadata format and, optionally, a from-to condition on dates and/or an OAI-Set; it returns a list of records with relative resumption token, according to

the standard specification. Two different implementations of the method are possible. The first exploits the Data Layer as it is, by discovering all Store DSs containing records of the given format and originating from a specified Repository, that is located in a given OAI-Set; given the Store DSs, the method sends the relative Store Services a request for retrieving the records, merges the results, and delivers to the caller. The second exploits instead special Index DSs, created on purpose to efficiently answer such calls; the Data Layer orchestration logic maintains one Index DSs for each format in the system, enabling record searches by record identifier, by date range, and by Repository of origin. In this scenario, the OAI-Publisher Service searches for the appropriate Index DSs and efficiently answers the call; note that the method GetRecord could exploit such Index DS too.

- **ListIdentifiers:** the same as ListRecords, but returns only the record identifiers.

A. OAI-Publisher extensions

DRIVER aims at experimenting two extensions of the OAI-Publisher Service. User and application consumers will be able to configure from the outside the OAI-Publisher Service, in order to define special OAI-Sets or enable record harvesting according to different metadata formats. More specifically, the extensions regard:

1) Dynamic format configuration: the Service accepts format-to-format mappings (rewriting rules), let's assume from format A onto format B. This operation adds to the MDFormat DS of B the dependency from A and registers a new MDFormat DS for B in the Enabling Layer, if B does not exist. The method ListRecords, when called for records of format B, will return all records in Store DSs of format B, plus all records from Store DSs of format A, after applying the mapping. This feature can therefore be used to permit consumers to harvest formats not present in the Information Space or to enlarge the number of records matching an existing format and will use the same techniques used by the Aggregator Services for cleaning and enriching the harvested metadata records. The rewriting rules are composed using a transformation language specifically tailored for metadata records. The OAI-Publisher will exploit this tool in order to provide the transparent mapping between formats, when possible.

2) Dynamic OAI-Set configuration: the Service accepts user defined *Collections* [7], i.e. named and fielded queries over a given format, and builds and maintains the corresponding OAI-Sets; harvesting w.r.t. such OAI-Sets, restricts the attention to the records matching the Collection query at the time of harvesting; the OAI-PMH listSets call to this Service returns the names of the Repositories, plus the names of the Collections submitted so far.

B. Incremental harvesting

The OAI-PMH protocol specifies a mechanism for continuous synchronization called "incremental harvesting". Consumers of OAI-Items (harvesters) are able to retrieve only recent additions to the source repository by specifing a date range. Modifications of already existing metadata records are gracefully propagated as each modification updates the date stamp associated with each OAI-Item, while the OAI-Item identifier is preserved. However, deleted record handling proves to be a problematic area.

According to the OAI-PMH standard, OAI-Publishers may implement three levels of support for deleted records by announcing it in the OAI-PMH Identify response:

- **no:** the repository does not maintain information about deletions. A repository that indicates this level of support must not reveal a deleted status in any response.
- **persistent:** the repository maintains information about deletions with no time limit. A repository that indicates this level of support must persistently keep track of the full history of deletions and consistently reveal the status of a deleted record over time.
- **transient:** the repository does not guarantee that a list of deletions is maintained persistently or consistently. A repository that indicates this level of support may reveal a deleted status for records.

If the source OAI-Publisher supports deleted record handling, the records provided by the OAI-PMH ListRecords method may be tagged with a deleted status label, informing the OAI harvester that it should remove the record from its store.

If an OAI-Publisher does not support deleted record handling, already harvested records may refer to unexisting items. Harvesters aware of that fact may decide to periodically refresh the whole repository from the source, in order to minimize the probability of stale data accumulation. The DRIVER harvester can be configured to perform a mixed method of full and incremental harvesting strategies, according to performance related decisions specific to each single source repository.

When the DRIVER harvester doing a full refresh delivers the records to a store, the Store Service[2] may compute the difference between the current content and the incoming records, in order to reconstruct the information relative to the deleted records. However, since such operation may be impractical for large data sets, the system can be configured to avoid even trying this last resort strategy. In this case the OAI-Publisher exposing the infrastructure's virtual store contents needs to able to keep track of the inability to implement the full level of deleted record handling support, and to announce the real level of compliance through the OAI-PMH Identify response of the exported view.

The level of deleted record support is thus highly dependent upon the dynamic nature of the virtual store and the subset of the records to be exported. Moreover, since the announced deleted record support level must be shared by all the OAI-Sets exported by the same OAI-PMH entry point, multiple DRIVER OAI-Publisher instances can be configured to provide a partition of the virtual store.

A similar situation may arise with the different level of date granularity support.

V. REFERENCE IMPLEMENTATION

Our current implementation of the OAI-Publisher service is based on the DRIVER Testbed infrastructure accessible at *http://testbed.driver.research-infrastructures.eu*.[3]

[2] depending on the specific Store Service implementation.

[3] Currently an alpha release, to be released definitively in June 2008.

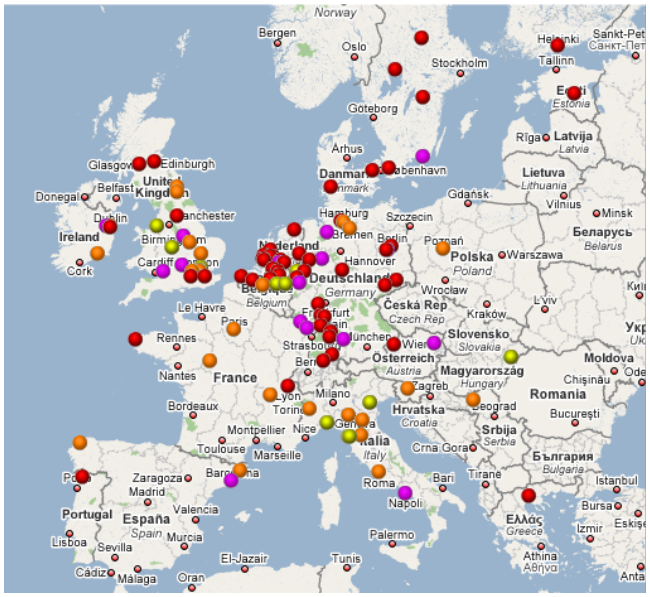

Figura 5: DRIVER Infrastructure repositories

The DRIVER Infrastructure contains data from a 117 European repositories (See Figure 5), containing more than 350’000 items describing textual documents in over 20 languages authored by more than 200’000 authors.

The sizes of source repositories vary between very small ones and relatively large ones (85’000 items), with an average size of around 8’000 items.

For each repository we keep the originally harvested metadata format, enriched with provenance-related informations, along with native *DRIVER Metadata Format* (DMF) records, obtained from the original metadata.

All the metadata is exported through several redundant OAI-Publisher instances which expose each repository as single OAI-Set of the whole DRIVER Information Space.

VI. CONCLUSIONS AND FUTURE ISSUES

In this work we presented OAI-Publisher technology as needed in the context of highly dynamic Repository Infrastructures. In particular, the case of the DRIVER Infrastructure OAI-Publisher Service was presented. Currently the Infrastructure offers an OAI-Publisher Service capable to self-adapt its configuration so as to return responses consistent with the dynamic and continuous status changes of

the DRIVER Information Space. Future steps will be those of extending the OAI-Publisher with dynamic formats and OAI-Set configuration interfaces as described above.

The DRIVER Project has been re-financed in the 7th FP so as to extend the Information Space with *Compound Objects*. Compound objects are sets of digital objects related by relationships representing the nature of their association; for example, a "Research Package Object" might consist in a set of publications, e.g. PDF, DOC files, interconnected by relationships named "refersTo" and a set of raw data files linked to such publications through relationships named "experimentalData". Compund object graphs will be exposed both through the OAI-ORE protocol [8], and through the OAI-PMH protocol using MPEG-21 DIDL payload [9].

ACKNOWLEDGMENT

This work is partially supported by the Information Society Technologies (IST) Program of the European Commission as part of the DRIVER (Project no. IST-034047).

REFERENCES

[1] C. Lagoze, S. Payette, E. Shin, and C. Wilper, "Fedora: An Architecture for Complex Objects and their Relationships," *Journal of Digital Libraries, Special Issue on Complex Objects*, 2005.

[2] R. Tansley, M. Bass, and M. Smith, "DSpace as an Open Archival Information System: Current Status and Future Directions," in *Research and Advanced Technology for Digital Libraries, 7th European Conference, ECDL 2003, Trondheim, Norway, August 17-22, 2003, Proceedings*, ser. Lecture Notes in Computer Science, T. Koch and I. Sølvberg, Eds. Springer-Verlag, 2003, pp. 446–460.

[3] C. Lagoze and H. Van de Sompel, "The open archives initiative: building a low-barrier interoperability framework," in *Proceedings of the first ACM/IEEE-CSJointConferenceonDigitalLibraries*. ACMPress,2001, pp. 54–62.

[4] Carl Lagoze and Herbert Van de Sompel,“TheOAIProtocolforMetadata Harvesting,” http://www.openarchives.org/OAI/openarchivesprotocol. html.

[5] “Dublin Core Metadata Initiative”, http://dublincore.org.

[6] “DRIVER Digital Repository Infrastructure Vision for European Research,” http://www-driver-repository.eu.

[7] L. Candela, D. Castelli, and P. Pagano, “A Service for Supporting Virtual Views of Large Heterogeneous Digital Libraries,” in *7th European Conference on Research and Advanced Technology for Digital Libraries, ECDL 2003*, ser. Lecture Notes in Computer Science, T. Koch and I. Sølvberg, Eds. Trondheim, Norway: Springer-Verlag, August 2003, pp. 362–373.

[8] Carl Lagoze and Herbert Van de Sompel, “Object Reuse and Exchange (ORE),” http://www.openarchives.org/ore/.

[9] H. V. de Sompel, M. L. Nelson, C. Lagoze, and S. Warner, “Resource harvesting within the oai-pmh framework,” in *D-Lib Magazine, Volume 10 Number 12, ISSN 1082-9873*, December 2004. [Online]. Available: http://www.dlib.org/dlib/december04/vandesompel/12vandesompel.html