

# An Infrastructure for Open-Architecture Digital Libraries

Carl Lagoze and Sandra Payette  
Department of Computer Science  
Cornell University  
{lagoze,payette}@cs.cornell.edu

## 1 Introduction and Definitions

This document describes components of an infrastructure for *open-architecture digital libraries*. By its very nature this is a working document. The development of infrastructure is an evolutionary process, involving the input and negotiation of many parties. While technical considerations play an important role, they are mediated or even driven by social, political, economic, and legal influences. Nevertheless, the process is frequently seeded by strawman proposals, and that is the intent of this document.

For the purpose of this document, we propose the following working definition of a *digital library*. A digital library is a *managed* collection of digital objects (content) and services (functionality) associated with the storage, discovery, retrieval, and preservation of those objects. Management begins with selection of the digital objects in the collection. Objects are selected from a global information space (*e.g.* the set of all published books, or the set of all objects on the Internet), and become constituents of the library collections based on criteria applied by collection managers (which may be human or automated). Management also entails the definition of the services included in the digital library. Some common examples of services are indexing, which allows discovery of objects in the collection; preservation, which assures the longevity of the objects in the collection; and awareness, which alerts users to changes in the collection. By this definition, the World Wide Web, by itself, is NOT a digital library. It represents a set of objects jointed together technically (by the protocol HTTP), but not by any collection management or service management decisions.

The descriptor *open-architecture* means that the total functionality of the digital library architecture is partitioned into a set of well-defined services. Each of these services is accessible via a well-defined *protocol* - a set of service requests - that defines the public interface to this service. Furthermore, each of these service requests is documented with respect to the format of the request, the format of the possible responses and exceptions, and the semantics of the request. A service is instantiated by a *server module* (we will use the shortcut term *server* for the remainder of this document), which implements the set of service requests defined for the service. The actual implementation of a server is opaque and irrelevant from the perspective of *interoperability*, the ability of the service to communicate via its protocol with other services, clients, and agents. In addition, an individual server may be distributed or replicated, the nature of which is also opaque from the perspective of its use.

An open-architecture digital library infrastructure allows the creation of any number of *federated digital library instances*. We define such an instance as the result of aggregating a collection of servers, with their defined protocols acting as the "glue" for the aggregation. The functionality of the digital library instance is a result of the union of the service requests from the aggregated servers. There is no implication that the set of servers in a digital library instance is centrally administered or co-located. The servers may have a high degree of administrative autonomy and may be widely distributed. In fact, incorporation of a server into a digital library instance does not necessarily imply that the server is informed of or needs to acknowledge its participation.

Finally, we note that such an infrastructure, and the resulting federated digital libraries, has unlimited extensibility. New services may be defined and implementations of them introduced into the architecture. As these new services are defined, they may be incorporated into existing or new digital library instances.

## 2 Infrastructure Overview

The remainder of this document describes *core* services in such a digital library infrastructure; their characteristics, publicly-defined interfaces, and interactions with other services. By *core*, we mean the set of services that are necessary to provide basic digital library functionality. Each section of the document describes one of these core services. The service requests for each of those services are summarized in the Appendix.

A brief summary of these services and their interactions is as follows.

- Content in the infrastructure is stored in the form of *digital objects*, which aggregate one or more byte streams, associate content-specific behaviors with the aggregations, and link rights management mechanisms to these behaviors.
- These digital objects are identified by globally-unique persistent names - URNs - that are registered with the *naming service*. An individual name server is able to resolve a URN to the one or more locations of the digital object that is identified by that URN.
- The *repository service* provides the mechanisms for the deposit, storage, and access to digital objects. A digital object is considered *contained* within a repository if the URN of that object resolves to the respective repository (and, thus, access to the object is only available via a service request to that repository).
- The *index service* provides the mechanisms for the discovery of digital objects. Individual index servers index actual or surrogate information on sets of digital objects (that may be distributed across multiple repository servers). Creation of these indexes may be the result of automatic scans of a set of repositories, human entry and intervention, or a mixture of both. Queries submitted to these index servers return result sets that contain the URNs of digital objects that match the query. Clients or agents can then submit these URNs to a name server to access the corresponding object. The index service also provides metadata about the content of its indexed information and the capabilities of its query mechanisms. This metadata is used by other services, such as the collection service described below.
- The *collection service* provides the mechanism for the aggregation of sets of digital objects into meaningful (from some community's perspective) collections. A collection server creates collections by scanning a set of index services, reading their metadata and applying its *collection definition criteria* to define which objects indexed by those index servers are elements of its defined collections. There is no fixed notion of collection definition criteria. One example of a collection definition criterion is *subject*, which may be determined by reading a controlled vocabulary metadata field of objects or derived via some natural language analysis. The elements of a collection defined by a collection server may be indexed by any number of index servers and located in any number of repository servers.
- A *user interface gateway* provides a human-centered entry point to the functionality of the digital library. Each user interface gateway uses the information provided by one or more collection servers to permit searching for and access to objects within those collections. User interface gateways also use information provided by collection servers and index servers to make query routing decisions based on factors such as content, cost, performance, and the like.

Figure 1 illustrates the interactions among some of the services defined above.

Many aspects of the designs described in this document are based on ideas or implementations described elsewhere. The origins of the general service structure lie in the Dienst Architecture [1], which is the technical foundation of the Networked Computer Science Technical Reference Library (NCSTRL) [2]. The naming service description in section 3 is based on the handle system [3]. The design for digital objects and the repository service described in section 4 is derived from the digital library infrastructure work by Kahn and Wilensky [4]. Section 4 extends the design of FEDORA [5]. The description of an index service in section 5 is based on the STARTS protocol design [6]. Some of the concepts in the collection service are derived from [7]. Finally the connectivity region and collection view concepts described in section 6.2 are more fully explained in [8].

This document should not be interpreted as a finished proposal for a ready-to-implement distributed digital library system. In general, the descriptions of services and their interfaces are high-level and conceptual. In many cases, implementation of these ideas for real-world systems will require investigation of a number of core research issues related to reliability, security, query routing, query translation, and other areas of key interest to the digital library research community. Our hope is that this document will provoke that research and be a starting point for future prototypes and production systems.

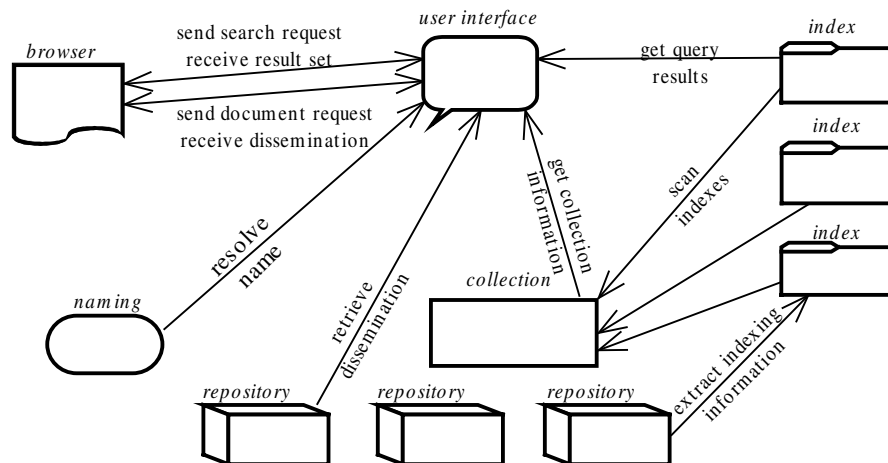


Figure 1 - Infrastructure Service Interactions

### 3 A Naming Service for Objects and Services

Names are essential surrogates for objects and services in the infrastructure. In order for the infrastructure to operate with integrity these name surrogates should be persistent and globally unique. That is, a client, agent, or other service should be able to store a name over time and be reasonably assured that the use of the name will successfully resolve to the *same* object or service. By *same* we do not mean exact equivalence, but *intended* equivalence from the perspective of the owner or manager of the object. As a trivial example, I may change the typeface of this document and correct misspellings, but from my perspective it is still the same document with the same name. (However, I may want to attach a versioning "behavior" to the document, allowing users to access various revisions of the document. The notion of document behaviors is explained in section 4.

Equivalence does not imply location or replica equivalence. A persistent name may at different times resolve to objects in different locations, as long as those objects have the same intentional equivalence as defined above. This is in contrast to the widely used URLs, which are directly tied to location.

A naming service is responsible for the creation, resolution administration of such names. Its functionality is provided by the following service requests.

#### CreateName

Takes as input an instance (location) of an object or service and returns a unique name. The newly created unique name and resolution are stored in the name service.

#### AddResolution

Takes as input an instance (location) of an object or service and an existing unique name. The new resolution is stored with the unique name in the name service.

#### DeleteResolution

Takes as input an instance (location) of an object or service and an existing unique name. The resolution is removed from the entries for the unique name in the name service.

#### ResolveName

Takes as input a unique name and returns the set of instances (locations) for that name.

## 4 A Repository Architecture for the Storage of Digital Objects

A fundamental requirement of an architecture for digital libraries is a reliable and secure means for storage and access to digital content. The repository architecture must be able to: (1) support heterogeneous content types; (2) aggregate mixed, possibly distributed, content into complex objects; and (3) provide mechanisms for access management of digital content.

Extensibility is of paramount importance in the repository architecture. There are already countless forms of content and new ones will inevitably appear. The architecture should seamlessly integrate those new content forms and the mechanisms for disseminating and presenting them. Similarly, the facilities for access management must be inclusive rather than prescriptive - accommodating the variety of existing and new rights management mechanisms rather than attempting to define a globally inclusive mechanism. Both of these extensibility dimensions suggest that the architecture must accommodate the "plug in" of new content handling and rights management mechanisms, rather than solely relying on facilities that are wired into the implementation of specific repositories.

We describe here a digital object repository architecture that meets the goals of extensibility and interoperability. The architecture uses an abstraction, a digital object, as the atomic unit in collections in the digital library infrastructure. This abstraction is based on the assumption that content in a digital library is more than a simple stream of bits. It is a wrapper, aggregating one or more physical bit streams, defining the semantics or behaviors of the aggregated bit streams, and defining the access management mechanisms for those behaviors.

A notable feature of the architecture is its clear separation of structure, interface, and mechanisms. The *structure* of a digital object is the set of bit streams (internal and external) that it aggregates. For example, a digital object may aggregate a set of TIFF bit streams, which is the "content" of a technical report, and an ASCII-encoded bit stream, which is a MARC record describing that report. The *interface* of a digital object is the set of "content types" or behaviors that it exposes to clients. For example, the digital object with the structure described above may expose Dublin Core content and PostScript content to clients. The *mechanisms* of a digital object are the computable objects for translating from the structure to interface. In general, structure and mechanisms are opaque to clients, which only deal with digital objects through their interfaces. Interfaces are uniquely identified by digital objects that disseminate, resulting in a uniquely-named type system in the infrastructure.

The architecture is logically constructed in three layers. Each layer is distinguished by the decreasing opacity of the digital object, and its resulting increasing functionality from the standpoint of clients and agents.

### 4.1 Layer 1 - Management of opaque digital objects

This layer allows the deposit, management, and access of digital objects in repositories. These digital objects, at this layer, are totally opaque - there are no service requests that allow "looking into" the digital object wrapper.

There are two abstractions present at this layer:

#### *DigitalObject*

An opaque wrapper. The only attribute of the wrapper visible at this layer is its unique identity, which is registered with the naming service. A *DigitalObject* effectively does not exist, at the infrastructure level, without such a registered name.

#### *Repository*

An entity that provides for managed access to a set of *contained DigitalObjects*. *Containment* is logically achieved through the name service. That is, if an object named *H* is "contained" in a repository named *R*, then the name service will resolve the name *H* to the repository *R*. A client then can access *H* through an *AccessDigitalObject* service request made to *R* with *H* as an argument.

The service requests for a repository are follows.

### DepositDigitalObject

Takes as input an instance of a *DigitalObject* and returns a unique name for that object. The object is effectively stored "in the repository". This operation entails interaction with the naming service - a **CreateName** operation is invoked with repository name as the location of the object. (Note that the repository name is itself a unique name that at some time was registered with the naming service.) The effect of this **CreateName** operation is that a **ResolveName** request to the name service with the object name returns the repository name. As a result the object is accessible using this name only through its repository context (using **AccessDigitalObject** as defined below).

### AccessDigitalObject

Takes as input a *DigitalObject* name and returns a reference to a *DigitalObject*. This reference than can be used for performing service requests on that *DigitalObject*. These service requests are described in sections 4.2 and 4.3.2.

### MarshallDigitalObject

Takes as input a *DigitalObject* reference and returns a marshalled byte stream representation of that *DigitalObject*.

### ReplicateDigitalObject

Takes as input a *DigitalObject* name and the name of a source repository. The result of the service request is a replica of the *DigitalObject* copied from the source repository into repository that is the target of this request (and the recording of that replica in the name service). The data for the replica is supplied via a **MarshallDigitalObject** request to the source repository and the replica is recorded in the name service via an **AddResolution** request.

### DeleteDigitalObject

Takes as input a *DigitalObject* name. The result is the removal of the *DigitalObject* from the repository and its reference in the name service, via a **DeleteResolution** request.

## 4.2 Layer 2 - Structural manipulation of digital objects

This layer provides abstractions and mechanisms for the construction and decomposition of digital objects. Whereas at layer 1 a digital object is only a named opaque wrapper, this layer exposes the abstractions that are the structure (as opposed to the meaning) of the wrapper.

The abstractions at this layer are as follows:

#### *DataStream*

A stream of bytes. The sequence or encoding of the stream may conform to that defined by some standard or application (for example, PostScript, TIFF, or Microsoft Word). The encoding of a stream is determine by its *type*. IETF MIME types are one way of naming these type encodings.

#### *Dissemination*

A byte stream that is produced by any of the service requests defined for a *DigitalObect*, exclusive of exceptions or error responses. These service requests are defined below and in section 4.3.2. The exact nature of the stream of bytes is dependent on the particular service request. The sequence or encoding of the stream may conform to that defined by some standard or application, in the same manner as a *DataStream*. From a rights management point of view, a *Dissemination* should be considered a view of the content or information within the *DigitalObect* and may be therefore subject to access controls or terms and conditions.

#### *Disseminator*

A package of service requests that is associated with a *DigitalObject*. Every digital object, regardless of content or composition, has one set of service requests associated with it known as the *Primitive Disseminator*. These service requests are defined below. Layer 3 defines mechanisms for adding additional *Disseminators*, specific to the content of the *DigitalObject*.

#### *AccessManager*

A rights enforcement mechanism that is associated with a *Disseminator*. The effect of this association is that the set of service requests defined by *Disseminator* are processed under control of the *AccessManager*. Thus, input to them and output from them is subject to any constraints or manipulations imposed by the *AccessManager*.

Structural manipulation of these abstractions and their aggregation within an individual digital object will in general occur only during the construction and modification of digital objects. In fact, we expect that the *AccessManager* associated with the Primitive Disseminator will prohibit structural manipulation by general clients. Clients accessing digital objects will in general be concerned with content or behavior specific actions - those which are defined at layer 3.

The service requests pertinent to structural access, defined by the *Primitive Disseminator* are as follows.

#### ListDataStreams

Returns a sequence of references to the *DataStreams* within the *DigitalObject*. In effect, this request exposes the number of *DataStreams* within the respective *DigitalObject* and the stream of bytes within each of those streams. While this service request effectively disseminates the "content" within a *DigitalObject*, it provides no information on the meaning of the content in the context of the *DigitalObject* (for example, whether a particular stream of bytes is "data" or "metadata" for the *DigitalObject*).

#### ListDisseminators

Returns a sequence of references to the *Disseminators* associated with a *DigitalObject*. Each of these references may then be used for direct manipulation of the respective *Disseminator*.

#### ListAccessManagers

Returns a sequence of references to the *AccessManagers* associated with a *DigitalObject*. Each of these references may then be used for direct manipulation of the respective *AccessManager*.

### 4.3 Layer 3 - Content-dependent manipulation of digital objects

This layer provides mechanisms to support content specificity in *DigitalObjects*. From an operational standpoint (and from the client's point-of-view), content specificity is determined by the set of behaviors (service requests) associated with a *DigitalObject*. For example, a *DigitalObject* is a book if it *acts* like a book - there is an operation `nextPage` or an operation `nextChapter`. Similarly, a *DigitalObject* with operations like `nextArticle` or `nextIssue` is effectively a journal. Each of these sets of service requests - those associated with book behavior or journal behavior - is a *content type*.

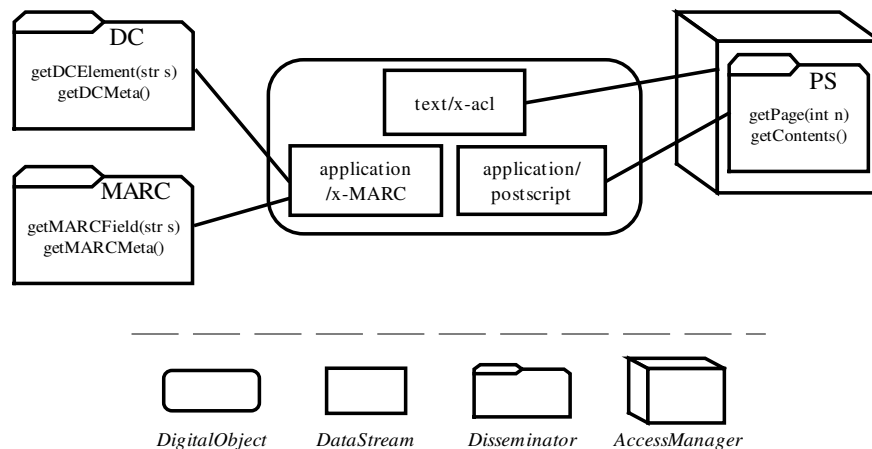
A single *DigitalObject* may have multiple *content types*. For example, one *DigitalObject* may have a Dublin Core content type and a Technical Report content type, meaning that it supports the service requests associated with that content type.

Two *DigitalObjects* with entirely different structural compositions may have *content-type equivalence*; effectively having the same sets of service requests associated with them. A simple example of this is the Dublin Core content-type, which formally may be represented by the service requests `getDCElement` and `getDCRecord`. One *DigitalObject* with this content-type may have the Dublin Core description literally stored as a *DataStream*. Another may provide it through a mechanism that translates, via some cross-walk rules, a MARC record in a *DataStream* into its Dublin Core equivalent. From the client perspective, this structural differentiation between the two *DigitalObjects* and the mechanisms for disseminating the content-types should be opaque.

The association of content-types with *DigitalObjects* is accomplished via the *Disseminators* introduced at layer 2. To review, a *DigitalObject* encapsulates a set of byte streams known as *DataStreams*. *Disseminators* are packages of service requests that are associated with a *DigitalObject*. The *Primitive Disseminator* is logically associated with every *DigitalObject* - it endows the *DigitalObject* with level 2 structural behavior. Additional *Disseminators* can be associated with the *DigitalObject* to endow it with content-type behaviors. Finally, *AccessManagers* can be associated with *Disseminator* to provide rights management for sets of service requests.

**Figure 2** illustrates the use of these abstractions to create a *DigitalObject* with content-specific behaviors. The illustrated *DigitalObject* aggregates three *DataStreams*: an ASCII MARC bibliographic record, a PostScript encoding of content, and an ASCII access control list. The *DigitalObject* has three *Disseminators* associated with it.

1. One labeled DC that extracts Dublin Core elements from the MARC data using "crosswalk" rules between the two descriptive formats. This *Disseminator* has two service requests, `getDCElement`, which extracts individual Dublin Core elements, and `getDCMeta`, which extracts the entire Dublin Core element set.
2. One labeled MARC that extracts MARC record elements. This *Disseminator* has two service requests, `getMARCElement`, which extracts individual MARC fields, and `getMARCMeta`, which extracts the entire MARC record.
3. One labeled PS that provides PostScript specific behavior. This *Disseminator* has two service requests, `getPage`, which extracts individual pages from the document, and `getContents`, which extracts the PostScript content. This *Disseminator* is associated with an *AccessManager*, which manages access to the service requests in PS using the access control list data in the *DataStream* labeled `text/x-acl`.



**Figure 2 - A *DigitalObject* with *Disseminators* and *AccessManagers***

Layer 3 provides the mechanisms for storing, registering, and disseminating these content-specific behaviors (*Disseminators*) and access management mechanisms (*AccessManagers*). Since, both *Disseminators* and *AccessManagers* are themselves "content", they themselves are stored and disseminated by *DigitalObjects*. This has two implications. Like all *DigitalObjects*, those that disseminate *Disseminators* and *AccessManagers* are uniquely named. This endows both content type and access management schemes with unique names. In addition, since new *DigitalObjects* can be created and stored in the infrastructure, the content type and access management schemes are infinitely extensible.

The abstractions for storage and dissemination of *Disseminators* and *AccessManagers* are listed below.

#### *DisseminatorSignature*

A description (signature) of a set of service requests specific to a particular content type. In effect, these can be considered definitions of content types in that each defines the set of service requests, and their syntax, specific to that content type. For example, a signature `getPage(n)`, `getChapter(n)` might define the type *book*. A *DisseminatorSignature* is available as a dissemination of a *DigitalObject*. Therefore its identity is the unique name of that respective *DigitalObject*. Effectively, this provides a unique content-type naming scheme for the infrastructure.

#### *DisseminatorServlet*

A mechanism for executing the service requests defined by a particular *DisseminatorSignature*, identified by its unique name as defined above. Each *DisseminatorServlet* has a specific *AttachmentSpec*, which defines the types of *DataStreams* that must be present in a *DigitalObject* to provide necessary data for the execution of this

*DisseminatorServlet*. Note that a single *DisseminatorSignature* may be implemented by multiple *DisseminatorServlets* - there are many possible implementations of any particular signature. A *DisseminatorServlet* is available as a dissemination of a *DigitalObject*.

#### *AccessManagerServlet*

A mechanism for rights management. Each of these defines the set of *DisseminatorSignatures* for which it can provide rights management. Note that certain *AccessManagerServlets* may be not *Disseminator*-specific - for example, an access control list *AccessManagerServlet* may be appropriate for any *Disseminator*. Each *AccessManagerServlet* has a specific *AttachmentSpec*, which defines the types of *DataStreams* that must be present in a *DigitalObject* to provide necessary data for the execution of this *AccessManagerServlet*. For example, an access control list *AccessManagerServlet* will require a *DataStream* that provides the access control list specific to the *DigitalObject*.

### 4.3.1 Constructing Digital Objects

The use of these abstractions and those defined at level 2 in an individual *DigitalObject* is described by the following steps.

1. *DataStreams* are created from sequences of bytes.
2. The *DataStreams* are aggregated within a *DigitalObject*. At this point, the *DigitalObject* has only "level 2" behavior - the only set of service requests available from it are those defined by the *PrimitiveDisseminator*.
3. A creator of a *DigitalObject* selects *DisseminatorServlets* to endow the object with desired behaviors. There must be *DataStreams* in the *DigitalObject* that conform to the *AttachmentSpec* of each *DisseminatorServlet*. These *DisseminatorServlets* are linked to the *DigitalObject* by creating a *Disseminator*. This *Disseminator* references the *DisseminatorServlet* and identifies the *DataStreams* in the *DigitalObject* that fulfill its *AttachmentSpec*. (Note that a conforming *DataStream* may be also be referenced indirectly; that is as the result of a dissemination of another *DigitalObject*.)
4. A creator of a *DigitalObject* selects *AccessManagerServlets* to endow its *Disseminators* with desired rights management functionality. A selected *AccessManagerServlet* must be compatible with a *Disseminator* associated with the *DigitalObject*. In addition, there must be *DataStreams* in the *DigitalObject* that conform to the *AttachmentSpec* of each *AccessManagerServlet*. These *AccessManagerServlets* are linked to the *Disseminators* by creating an *AccessManager*. This *AccessManager* references the *AccessManagerServlet* and identifies the *DataStreams* in the *DigitalObject* that fulfill its *AttachmentSpec*.

### 4.3.2 Content-Specific Access to Digital Objects

From the client's point-of-view, all this complexity is hidden under two simple service requests that can be submitted to *DigitalObjects*.

#### ListDisseminations

This returns a sequence of names that uniquely identify the content types (named by their *DisseminatorSignatures*) available from this *DigitalObject*. For example, a *DigitalObject* might return  $H_1$  indicating that it can disseminate Dublin Core description,  $H_2$  indicating that can disseminate a MARC record, and  $H_3$  indicating that can disseminate content in the form of a journal. The *DisseminatorSignature* referenced by each of these names defines the service requests appropriate for each of these content types - for example, `getDCElement` for the Dublin Core description content type.

#### GetDissemination

This request takes as input a dissemination service request and returns the stream of bytes produced by that request. In essence this is an encapsulated service request. The `GetDissemination` request is defined for all *DigitalObjects*, and is thus content-independent.

The input to the request is a service request that is specific to the content types (returned by the *ListDisseminations* request) of the individual *DigitalObject*.

Figure 3 illustrates the relationships among the abstractions and service requests described above. In the figure dotted lines indicate the association of a *Disseminator* or *AccessManager* with a *DataStream*. Heavy arrows indicate service requests and returns. The remaining solid arrows indicate a reference to another *DigitalObject* using its unique name. The *DigitalObject* named  $H_{DO}$  has a single *Disseminator*, which is "protected" by an *AccessManager*. The *Disseminator* internally references the *DigitalObject* labeled  $H_{PS1}$ , which disseminates a *DisseminatorServlet* - a mechanism for producing PostScript type disseminations. This *DisseminatorServlet*, in turn, internally references the *DigitalObject* labeled  $H_{PS}$  - the *DisseminatorSignature* ("content-type") of the *DisseminatorServlets* that reference it. ( $H_{PS}$  may be referenced by other *DisseminatorServlets* since a single signature may have multiple mechanisms.) The effect is that client submission of the service request *ListDisseminations* to  $H_{DO}$  returns the single "content-type"  $H_{PS}$ . Finally, note that the *AccessManager* internally references the *DigitalObject*  $H_{ACL}$ , which disseminates an *AccessManagerServlet* - a mechanism for performing access-control list "type" rights enforcement.

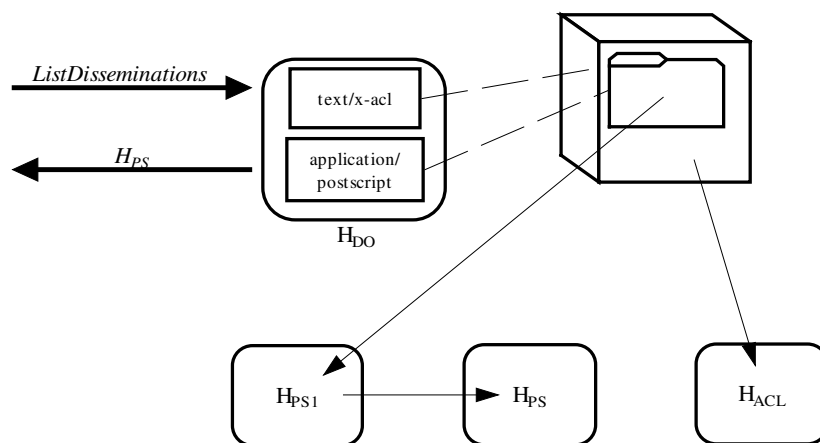


Figure 3 - Relationship of *Disseminators*, *DisseminatorServlets*, and *DisseminatorSignatures*

## 5 An Index Service for Resource Discovery

Patrons of a library expect to be able to easily find and discover objects in the collection. We define an *Index Service* as the component of the digital library infrastructure that facilitates such *resource discovery*.

An Index Server collects information about digital objects. This information may be in the form of surrogates, such as the familiar cataloging records in the traditional library, or it may be full disseminations of the objects, such as that used by full-text search engines. The information that is collected is organized into structured indexes that allow search engines to respond to queries on the information with precision, recall, and efficiency. The response to a query is a "result set" - where each member of the set is generally a surrogate for a digital object that "matches" the query. The minimal form of surrogate is the unique identifier for the object that, when resolved by the name service, allows access to the digital object.

We are intentionally informal about the nature of this information collection and indexing process. The method may vary across a broad spectrum with the following interesting data points:

- *Automatic Indexing* - An index server visits a set of repository servers and extracts and automatically indexes information from the digital objects contained in those repositories. This is similar in nature to the "web crawling" technique used by existing web search engines.
- *Manual Indexing* - In the style of traditional libraries, professional catalogers create ordered surrogates for digital objects and index that information.
- *Human-Mediated Indexing* - A hybrid of the two end points on the spectrum.

Resource discovery is a highly complex task with a broad variety of requirements and solutions. These requirements vary according to community needs, the types of resources, the needs (or role) of the client or person, current hardware capabilities, and a host of other factors. Needless to say, there is no single technical solution to the resource discovery problem. At best, the infrastructure needs to accommodate a spectrum of solutions ranging from minimal-cost general solutions to high cost solutions that have functionality specific to the communities willing to make the investment in them.

The web search engines are the state-of-the-art at the low-cost and general end of the spectrum. They require little if no investment of effort from information providers and provide a fairly good rate of return in functionality for that investment. We expect that the centralized web-crawling technology that typifies these search engines will continue to exist and will improve over time as better information retrieval techniques develop.

At the other end of the spectrum - high cost but high functionality - Z39.50 has proven to be quite successful within its specialized community. From the library community's perspective, transaction-oriented searches with result sets that persist over time are essential, and the cost and complexity of a protocol that permits this is deemed worthwhile.

## 5.1 STARTS Protocol for Meta-Searching

Our goal in this paper is to propose a protocol that targets the middle of the cost/functionality spectrum - one that provides support for reasonably rich queries, facilitates distributed searching (which we believe is an essential aspect of the infrastructure), and that is not onerous in terms of complexity. The STARTS protocol, developed by the Stanford Digital Library Project with collaboration from Cornell Digital Library Research Group, matches these requirements. There is no presumption that STARTS handles all resource discovery needs, but it does provide a reasonable foundation for a federated resource discovery infrastructure. Federated resource discovery has the following broad components:

1. Choosing the index server(s) at which to evaluate a query.
2. Evaluating the query at the index servers.
3. Merging the results from these index servers.

The remainder of this section describes the functional components of STARTS that address these needs.

### 5.1.1 Query Language

STARTS defines a general query language interface. A STARTS query has two parts. A *filter expression* is Boolean and defines the documents that qualify for an answer to the query. A *ranking expression* defines how the documents that match the filter expression should be ranked. For example, consider a query with the filter expression:

```
((author "Garcia Molina") and (title "databases"))
```

and ranking expression:

```
list ((body-of-text "distributed") (body-of-text "databases"))
```

Documents that match this query have *Garcia Molina* as one of the authors and the word *databases* in their title. The result set is then ranked according to how well the full text within them matches the words "distributed" and "databases".

Like Z39.50, STARTS uses the notion of an *attribute set*, which is the set of query fields (*e.g.*, author, title, etc.) that can be used in a query. The protocol is extensible in that additional attribute sets can be defined, in addition to the *Basic-1* attribute set defined by the protocol.

Finally, the query protocol defines a set of parameters settings that can be submitted with a query, such as whether stop words should be dropped, what fields should be returned with each member of the result set, and the like.

## 5.1.2 Rank Merging

STARTS defines a result set reporting format that includes information for algorithmic result set merging. For each document in the result set, STARTS returns:

- The unnormalized score of the document for the query.
- Statistics about each query term in the ranking expression including 1) the number of times that the query term appears in the document, 2) the weight of the query term in the document as assigned by the search engine, and 3) the number of documents in the source that contain the term.
- The size of the document.
- The number of tokens in the document.

## 5.1.3 Server and Source Metadata

STARTS divides an index server into a set of *sources*, each of which is a logical partitioning of the documents indexed at the server. There is no presumption that the partitioning is based on any global considerations. STARTS makes available a set of metadata attributes for index servers and their contained sources. These metadata attributes are divided into three broad categories:

1. *Source Characteristics* - These attributes reveal the functional characteristics of a source including what attribute sets it supports, what modifiers, its stop word list and the like.
2. *Source Content Summary* - These attributes export data about the contents of a source. This data can be used by another server for content based query routing. Attributes in this category include a list of words that appear in the source, statistics for each word, and document statistics.
3. *Index Server Metadata* - This metadata is mainly used to list the set of sources in the index server.

Like the query syntax, the source metadata capabilities of STARTS are extensible through the creation of and use of extended attribute sets. Section 6 of this document, which describes the *collection service*, describes the use of the basic and extended STARTS metadata in the digital library infrastructure.

## 5.2 Index Service Interactions

Using the STARTS protocol as a basis, the following service requests are defined for the index service.

### SubmitQuery

Takes as input a query (as defined above) and returns a set of surrogates for the digital objects that match the criteria in the query. These surrogates should include the URN of the digital object, to allow subsequent access to the disseminations of the digital object.

### GetSourceMetaData

Takes as input a source name and returns metadata about that source. The nature of the metadata depends on the metadata attribute sets supported by the source and the index server.

### GetSourceContentSummary

Takes as input a source name and returns a summary of the content in that source, as defined above.

### GetServerMetaData

Returns metadata about the index server. The nature of the metadata depends on the metadata attribute sets supported by the index server.

## 6 A Collection Service for Semantic Aggregation of Objects

A distinguishing aspect of a library (digital or otherwise) is the definition and management of *collections*. These collections play an important role in the usability of the information space. We can easily see the utility of collections by examining an information space without collection distinctions - the World Wide

Web. One of the key problems with resource discovery in the web is that it applies general resource discovery methods over an information space with vastly divergent content. The application of well developed aids for information retrieval such as thesauri, stemming algorithms, and stop word elimination is virtually impossible in such a mixed domain environment. Division of the information space into collections allows the application of collection-specific methods to improve discovery and access within those collections.

Management of the collection begins with *selection* of the objects to be included in the collection. Objects are selected from a global information space (*e.g.*, the set of all published books, or the set of all objects on the Internet), and become constituents of library collections based on criteria applied by selectors or collection managers. In the traditional library environment, selection criteria are one of the principle aspects that distinguish individual libraries. Such criteria make a library a good "music library" or a good "agricultural library".

The digital library infrastructure described in this document provides selection capabilities at multiple levels of the infrastructure. Creators of digital objects select the content they are interested in making available. Repository managers may adopt policies that implicitly select the digital objects that can be deposited into the repository. These policies may be motivated by legal considerations (no pornographic or libelous content), quality judgements (only objects created by certain parties), or any other criteria. Administrators of index servers select the digital objects that are indexed in that server. For example, one index server may index all the digital objects in a selected set of repositories, another may index digital objects that fit certain criteria in one repository, while another may be the result of human indexing of hand-selected materials.

The *collection service*, described in this section, provides an additional level of selection and, therefore, collection, management. Its main role is to federate sets of services, and as a result sets of digital objects, into meaningful and uniformly functional (by some criteria) units. This role can be divided into two sub-roles:

1. Define the semantics of the collections and facilitate resource discovery within those collections.
2. Provide the mechanisms that allow global distribution of the collection.

These roles are described in the sub-sections that follow.

## 6.1 Collection Definition

The definition of what is actually "contained" in digital library collections can be ambiguous. For instance, in the traditional library model, some librarians argue that physical containment of objects (*e.g.*, in stacks) is the primary criterion for inclusion in the collection. This notion of physical control breaks down in the networked environment of digital libraries where both overt and implicit linkages can be made between objects that reside in different physical locations. For example, if object A is included in a collection, are objects B, C, and D that are linked to object A also included in the collection? If so, are all objects transitively linked to object A via other objects also included? The answer to these questions has important implications in the areas such as legal responsibility and public service.

While there are, undoubtedly, multiple perspectives on the definition of digital library collections, in this document we will adopt the following working definition. An object is "in" a digital library's collection if it can be directly *discovered* using the resource discovery tools defined and implemented by the respective digital library.

The collection service exploits this definition by dynamically partitioning the information space into collections and defining how objects in those collections can be found. Each collection server is configured with one or more collection definition criteria. A criterion is a mechanism for dynamically deriving distinct collections from a set of digital objects. For example a simple criterion might use a subject metadata field in a simple metadata format (*e.g.* Dublin Core) to derive the subject or domain based collections (*e.g.*, computer science, physics). The nature of these subject-based collections will change as the digital objects in the information space change - the appearance of new subjects will add new collections.

The nature of a collection criterion is arbitrary - obviously if a collection server is automated then its criteria must be computable and derivable from the digital objects that will be constituents of collections. Through

the application of imaginative techniques such as natural language processing, one can imagine some very interesting collection definition criteria. For example, there might a collection server that divides a set of digital objects into age-appropriate collections (*e.g.*, for children under 10, teenagers, college students, and adults) based on the type of writing used in the digital objects.

A collection service operates by reading the source metadata (as defined in section 5.1.3) of one or more index servers and applying its collection criteria. This source metadata provides statistics on the tokens that are indexed by the index server, their frequencies, and their field-specific locations (*e.g.*, author, abstract, title, etc.). Using this information, a collection server can determine if the an index server has indexed documents the are appropriate for its collection criteria and can modify the collection criteria according to the contents of the indexes it scans. For example, a collection server that develops subject based collections might create a new subject category when the number of a certain term (*e.g.*, "digital libraries") appears with high frequency in its index server source scanning.

The following service requests return information derived from this process. These service requests are mainly for use the by a user interface gateway as described in section 7.

#### GetCollections

Returns the set of collections derived by this collection server. For example, a collection server that produces simple subject-based collections might produce the following information: (subject (computer science) (chemistry)(geology)). A user interface gateway can then use this information to present to the user a subject menu from which they can choose the target of their query.

#### GetCollectionIndexers

Returns for a collection, the set of index servers at which digital objects are indexed that fit into this collection. A user interface gateway might then use this information for query routing.

#### GetCollectionFilter

Returns for a collection and index server pair, the filter expression that is appropriate for that index server to limit searches to that collection. A user interface gateway might then use this information to limit the search at a specific indexer to the collection chosen by a user.

## 6.2 Global Distribution - Connectivity Regions and Collection Views

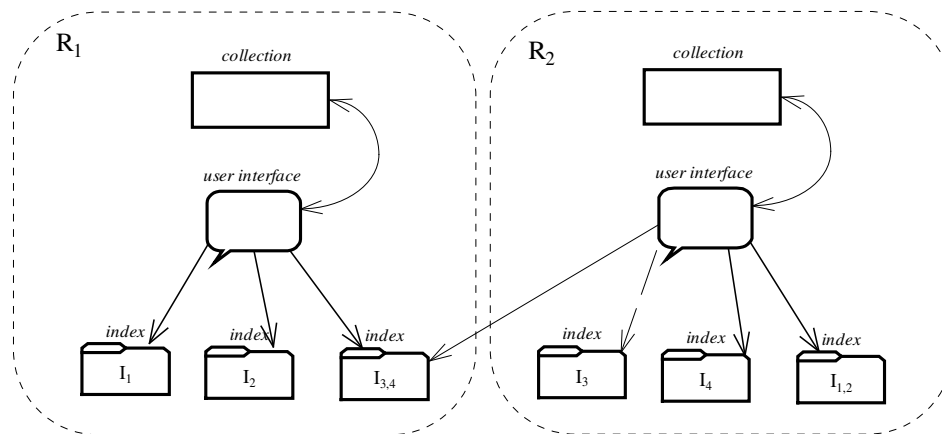
As described in section 6.1 , the collection service defines the set of index servers that should be used to locate items in a collection - known as the *collection resource set*. In an infrastructure with global distribution and replication of index servers, performance and reliability considerations should affect the composition of collection resource sets.

As is well known, global connectivity varies dramatically. In fact, the latency times between nodes can differ by several orders of magnitude. In addition, the patterns of connectivity are not necessarily geographically related. Points that are coincident in physical space may be "distant" in network space, as measured by reliability and speed of the connection. This disparity between geographic and electronic "proximity" often corresponds to patterns of telecommunication development over the past fifty years.

We model the patterns of global connectivity through the notion of a *connectivity region*. A connectivity region is defined as a group of nodes on the network that among them have good connectivity, relative to nodes outside of the region. In the absence of network or server failures, a collection viewed from the perspective of a specific network node should be restricted to those index servers in the same connectivity region. In case of a failure, an alternative indexing server should be chosen either in the same region or in another region with which there is good connectivity.

This suggests that collection servers should be distributed in a manner that corresponds to connectivity regions. For a given collection or collections, each regional collection server could provide a specific *collection view* - the collection resource set tailored for that connectivity region. Figure 4 illustrates a

simple example of such a connectivity-oriented topology. In this figure there are two regions, labeled  $R_1$  and  $R_2$ . Each region is associated with a collection server, which provides collection information tailored to the connection characteristics of that region. In addition a user interface server and three index servers are logically within their respective regions, because of their association with the regional collection server. In the example, the indexed data in the collection is divided into four partitions, and the subscript(s) on each of the indexing servers indicates the partition(s) indexed at the index server. For example, index server  $I_1$  holds indexing information in partition 1 and index server  $I_{3,4}$  holds indexing information in partitions 3 and 4. As illustrated, data from the collection server permits each user interface server to route queries to the index server(s) that are "in" their respective region. As also illustrated, the collection server also provides information necessary for query re-routing in case of server failure. For example, the failure for index server  $I_3$  in  $R_2$  results in re-routing of a query to an index server outside the region -  $I_{3,4}$  in  $R_1$ .



**Figure 4 - Regional Collection Servers and Index Servers**

Since collection servers are distributed in this manner, they could also serve as distributed *metadata repositories* for the index servers defined to be part of the collection. We have already described in section 5.1.3 the means for providing access to extensible metadata sets from sources and index servers. One example of such metadata might be performance or cost characteristics; for example, what the average load of the index server is, how much it charges, and other similar attributes. Regional collection servers might act as a collection point for this metadata, making it quickly accessible to other servers and user interface gateways in the connectivity region.

## **7 A User Interface Gateways to Digital Libraries**

Machines and agents interact with digital library services using the service requests defined for those services. Humans interact with digital library services through *user interface gateways* that conflate a federation of services and objects into a usable digital library. We purposefully do not use the word "service" in association with the user interface gateway, since interactions with it are not protocol driven, but human driven.

The goal of logically separating user interface gateways from all other services is to allow the creation of user interfaces that are tailored towards specific communities. The infrastructure should support the notion that a set of collections and set of services may be aggregated with an entirely different look-and-feel. For example, one user interface gateway to a collection of medical references (say *Medline*) might be tailored for the general public, employing help screens, thesauri, and information space visualization techniques that assume no knowledge of technical terms. The same collection might be also accessible from an entirely different user interface gateway designed for the professional practitioner.

A user interface gateway may provide access to any number of collections, information about which is derived from any number of collection servers. In line with the notion of *selection* described at the beginning of section 6, a user interface gateway represents the final level of selection in the infrastructure.

A manager of a user interface gateway *selects* the collection servers to use and the collection choices to present to the users of that gateway.

User interface gateways play a number of important roles. They allow users to browse and submit queries to one or more collections. They combine and present the results returned by these queries in a manner useful to users. They provide the facilities and environments for the display of disseminations of digital objects. The two subsections that follow describe some notable behaviors of user interface gateways that involve interactions with other infrastructure services.

## 7.1 Collection-specific behavior

A user interface gateway extracts information from one or more collection services using the service requests defined for that service. In the simplest model, a user interface gateway uses this information to present a list of possible collections to the user and allows that user to select one or more collections as the target for queries. For example, if a collection service provides information on subject-oriented collections, a user might be able to select "computer science" and "physics" as target collections and then submit a search such as *author="Einstein"* to those collections.

More complex user interface models may use innovative HCI techniques to tailor user actions for specific collections. For example, the selection of a computer science collection might make available to the user a keyword list and browsable thesaurus that aids in the formulation of queries specific to this collection.

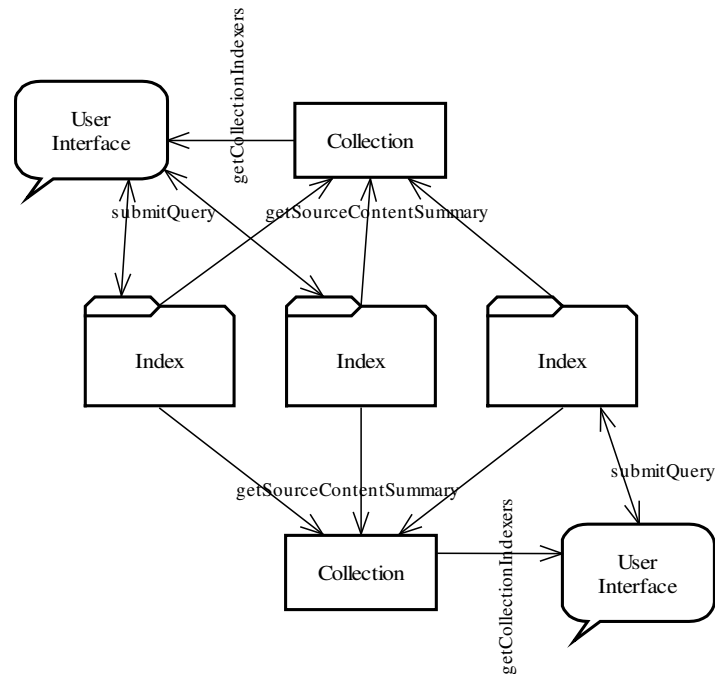
## 7.2 Query Routing

As described in section 5.1.3, the index service defines service requests that return meta-information about the index server. This meta-information can be content or capability information, as defined in the basic STARTS attribute set, or, through the use of additional attribute sets, any other type of information (such as performance characteristics).

User interface gateways use this information and information from the collection service as the basis for query routing. The goal of query routing is avoid inefficient broadcast distributed searches. Content routing is generally based on content; for example, which index server(s) are likely to return non-empty results sets from queries about "computer science". However, user interfaces gateways may offer a number of other query routing capabilities, each requiring a different type of metadata from index servers and collection servers.

- *Performance-based* - that directs queries towards index servers most likely to respond in the quickest time.
- *Cost-based* - that directs queries towards index servers that charge the least to provide the information.
- *Freshness-based* - that directs queries towards index servers that have been most recently updated.
- *Quality-based* -that directs queries towards index servers with the "best information", however that may be measured.

One interesting challenge of research is how to allow users to choose among these criteria and then the algorithms for optimization among the criteria.



**Figure 5 - Interaction of User Interface Gateways, Collection Servers, and Index Servers**

Figure 5 give a simple illustration of the interaction of user interface gateways with collection servers and index servers for content-based query routing. Collection servers use the `getSourceContentSummary` request to derive collection composition and location. User interface gateways use the `getCollectionIndexers` (and other service requests) to "learn" about these collection characteristics and locations from the collection servers. They eventually issue `submitQuery` requests to index servers - the routing of those requests is determined by the nature of the query and the collection information they have derived from the collection servers.

## 8 References

- [1] Lagoze, Carl, Erin Shaw, James R. Davis, and Dean B. Krafft, *Dienst: Implementation Reference Manual*, Cornell Computer Science Technical Report TR95-1514, <http://cs-tr.cs.cornell.edu:80/Dienst/UI/2.0/Describe/ncstrl.cornell/TR95-1514>.
- [2] Davis, James R and Carl Lagoze, *The Networked Computer Science Technical Reports Library*, Cornell Computer Science Technical Report TR96-1595, <http://cs-tr.cs.cornell.edu:80/Dienst/UI/2.0/Describe/ncstrl.cornell/TR96-1595>.
- [3] The CNRI Handle System, <http://www.handle.net>
- [4] Kahn, Robert H. and Robert Wilensky, *A Framework for Distributed Digital Object Services*, Corporation for National Research Initiatives, <http://www.cnri.reston.va.us/cstr/arch/k-w.html>.
- [5] Daniel, Ron Jr., Carl Lagoze, and Sandra Payette, *A Metadata Architecture for Digital Libraries*, Advances in Digital Libraries 1998, Santa Barbara, April 1998.
- [6] Gravano, Luis, Kevin Chang, Hector Garcia-Molina, Carl Lagoze, and Andreas Paepcke, *STARTS: Stanford Protocol Proposal for Internet Retrieval and Search*, Stanford Computer Science Technical Report CS-TR-97-1580, <http://elib.stanford.edu:80/Dienst/UI/2.0/Describe/stanford.cs/CS-TR-97-1580>.
- [7] Baldonado, Michelle, Chen-Chuan K. Chang, Luis Gravano, and Andreas Paepcke, *The Stanford Digital Library Metadata Architecture*, <http://www.diglib.stanford.edu/cgi-bin/WP/get/SIDL-WP-1996-0051>.

[8] Lagoze, Carl, David Fielding, and Sandra Payette, *Making Global Digital Libraries Work: Collection Services, Connectivity Regions, and Collection Views*, ACM Digital Libraries 1998, Pittsburgh, June 1998.

## APPENDIX - Service Request Summary

This appendix lists the service requests described earlier in the document.

### Name Service

#### CreateName

Takes as input an instance (location) of an object or service and returns a unique name. The newly created unique name and resolution are stored in the name service.

#### AddResolution

Takes as input an instance (location) of an object or service and an existing unique name. The new resolution is stored with the unique name in the name service.

#### DeleteResolution

Takes as input an instance (location) of an object or service and an existing unique name. The resolution is removed from the entries for the unique name in the name service.

#### ResolveName

Takes as input a unique name and returns the set of instances (locations) for that name.

### Repository Service

#### DepositDigitalObject

Takes as input an instance of a *DigitalObject* and returns a unique name for that object. The object is effectively stored "in the repository". This operation entails interaction with the naming service - a **CreateName** operation is invoked with repository name as the location of the object. (Note that the repository name is itself a unique name that at some time was registered with the naming service.) The effect of this **CreateName** operation is that a **ResolveName** request to the name service with the object name returns the repository name. As a result the object is accessible using this name only through its repository context (using **AccessDigitalObject** as defined below).

#### AccessDigitalObject

Takes as input a *DigitalObject* name and returns a reference to a *DigitalObject*. This reference then can be used for performing service requests on that *DigitalObject*. These service requests are described in sections 4.2 and 4.3.2.

#### MarshallDigitalObject

Takes as input a *DigitalObject* reference and returns a marshalled byte stream representation of that *DigitalObject*.

#### ReplicateDigitalObject

Takes as input a *DigitalObject* name and the name of a source repository. The result of the service request is a replica of the *DigitalObject* copied from the source repository into repository that is the target of this request (and the recording of that replica in the name service). The data for the replica is supplied via a **MarshallDigitalObject** request to the source repository and the replica is recorded in the name service via an **AddResolution** request.

#### DeleteDigitalObject

Takes as input a *DigitalObject* name. The result is the removal of the *DigitalObject* from the repository and its reference in the name service, via a **DeleteResolution** request.

### Digital Objects

#### ListDataStreams

Returns a sequence of references to the *DataStreams* within the *DigitalObject*. In effect, this

request exposes the number of *DataStreams* within the respective *DigitalObject* and the stream of bytes within each of those streams. While this service request effectively disseminates the "content" within a *DigitalObject*, it provides no information on the meaning of the content in the context of the *DigitalObject* (for example, whether a particular stream of bytes is "data" or "metadata" for the *DigitalObject*).

#### ListDisseminators

Returns a sequence of references to the *Disseminators* associated with a *DigitalObject*. Each of these references may then be used for direct manipulation of the respective *Disseminator*.

#### ListAccessManagers

Returns a sequence of references to the *AccessManagers* associated with a *DigitalObject*. Each of these references may then be used for direct manipulation of the respective *AccessManager*.

#### ListDisseminations

This returns a sequence of names that uniquely identify the content types (named by their *DisseminatorSignatures*) available from this *DigitalObject*. For example, a *DigitalObject* might return  $H_1$  indicating that it can disseminate Dublin Core description,  $H_2$  indicating that can disseminate a MARC record, and  $H_3$  indicating that can disseminate content in the form of a journal. The *DisseminatorSignature* referenced by each of these names defines the service requests appropriate for each of these content types - for example, `getDCElement` for the Dublin Core description content type.

#### GetDissemination

This request takes as input a dissemination service request and returns the stream of bytes produced by that request. In essence this is an encapsulated service request. The `GetDissemination` request is defined for all *DigitalObjects*, and is thus content-independent. The input to the request is a service request that is specific to the content types (returned by the *ListDisseminations* request) of the individual *DigitalObject*.

## Index Service

#### SubmitQuery

Takes as input a query (as defined above) and returns a set of surrogates for the digital objects that match the criteria in the query. These surrogates should include the URN of the digital object, to allow subsequent access to the disseminations of the digital object.

#### GetSourceMetaData

Takes as input a source name and returns metadata about that source. The nature of the metadata depends on the metadata attribute sets supported by the source and the index server.

#### GetSourceContentSummar

Takes as input a source name and returns a summary of the content in that source, as defined above.

#### GetServerMetaData

Returns metadata about the index server. The nature of the metadata depends on the metadata attribute sets supported by the index server.

## Collection Service

#### GetCollections

Returns the set of collections derived by this collection server. For example, a collection server that produces simple subject-based collections might produce the following information: (subject (computer science) (chemistry)(geology)). A user interface gateway can then use this information to present to the user a subject menu from which they can choose the target of their query.

#### GetCollectionIndexers

Returns for a collection, the set of index servers at which digital objects are indexed that fit into this collection. A user interface gateway might then use this information for query routing.

#### GetCollectionFilter

Returns for a collection and index server pair, the filter expression that is appropriate for that index server to limit searches to that collection. A user interface gateway might then use this information to limit the search at a specific indexer to the collection chosen by a user.