Implementation Issues in an Open Architectural Framework for Digital Object Services

Carl Lagoze
Cornell University

David Ely
CNRI

June 6, 1995 - Revision 1.3

1 Introduction

This document provides high-level designs for implementing some key aspects of the Kahn/Wilensky Framework for Distributed Digital Object Services (FDDOS)[1]. The descriptions given here assume that the reader has read and understands the Kahn/Wilensky document. This is a working document; as the FDDOS evolves the designs presented here will develop and may possible change.

We focus here on five aspects of the architecture:

1. Negotiation on terms and conditions initiated by requests for stored digital objects.
2. Replication of handle server data and the notion of a primary handle server.
3. The mechanisms for replicating digital objects in multiple repositories and the assertions concerning such replication.
4. The meaning of mutable and immutable states for digital objects and the mechanisms for changing these states.
5. The basic services that the Repository Access Protocol (RAP) needs to support the infrastructure.

Note that, like Kahn/Wilensky our descriptions are of low-level services and are, by necessity, sometimes informal due to the stage of development of the architecture. Throughout the rest of this document the terms FDDOS and Kahn/Wilensky architecture are used interchangeably to refer to the digital object architecture presented in the Kahn/Wilensky document.

2 Object permissions and "Terms and Conditions" negotiation

Each repository in the FDDOS logically consists of a set of *stored digital objects* (SDO) and the services that can be performed on those objects. The *repository access protocol* (RAP) described in appendix A defines the basic services that may be performed on all SDO's. Additional services may be defined as the architecture develops and especially as the type system for digital objects is fully defined; some of these additional services may be content dependent.

[1] A Framework for Distributed Digital Object Services, Robert Kahn, Robert Wilensky.

Conceptually, a repository manages access to each SDO via a *repository access control matrix*. This matrix consists of the set of available services[2] on one dimension (e.g., ACCESS_DO, DELETE_DO) and the set of requesters, or groups of requesters, on the other. The data for each cell of the matrix are the permissions and terms and conditions that apply to this SDO for the respective service by the respective requester or requester group[3]. Terms and conditions are described in more detail later. Figure 1 informally demonstrates this for a hypothetical collection of digitized IEEE publications for which Cornell has negotiated a hypothetical license. Under this license, students and the repository administrator have free access to the digitized collections, alumni have for-fee access (say, $1 per access), and others have no access. Only the repository administrator can perform a re-instantiate operation (REINSTANTIATE_DO).

	service	
Requester Group	ACCESS_DO	REINSTANTIATE_DO
Repository Admin	permissions: all terms and conditions: free	permissions: all terms and conditions: free
Cornell Students	permissions: read terms and conditions: free	permissions: none terms and conditions: n/a
Cornell Alumni	permissions: read terms and conditions: fee $1	permissions: none terms and conditions: n/a
Others	permissions: none terms and conditions: n/a	permissions: none terms and conditions: n/a

Figure 1: Example permissions matrix for repository services

The presence of terms and conditions in the repository access control matrix has three implications.

1. We assume that a formal schema for codifying terms and conditions will be developed. We assume that this codification will allow repository clients and users of these clients to negotiate with repositories over the resolution of terms and conditions. One example of negotiation is a simple acknowledgement from the requester of acceptance of the terms and conditions (e.g., similar to acceptance of a software copyright agreement before breaking the shrink-wrap on a software package). Another example is the authorization of payment (e.g., a credit card number) when terms and conditions require fee payment.

2. The dissemination that a repository returns as a result of an ACCESS_DO request may not be the requested object. Instead the repository may return (for negotiation purposes) the codified terms and conditions [4]. This means that the type *terms-and-conditions* will have to be part of the digital object type hierarchy.

3. Negotiations over terms and conditions implies that repositories will be able to record the

[2]These RAP services and others mentioned throughout this document are completely listed in appendix A

[3]We assume throughout this document that there will be a mechanism for authenticating the identity of parties requesting services from repositories. The authentication mechanism is left undefined at this time.

[4]This return should be accompanied by an appropriate error code (e.g., "terms and conditions negotiation required") to facilitate client handling of this transaction.

state of RAP requests[5]. At this time we define a single state for transactions, the *completion state*, which may be either be "open" (awaiting negotiation on terms and conditions) or "closed". We leave undefined at this time the meaning of a transaction that remains "open" indefinitely. Other stateful information on transactions may be introduced as the architecture develops. To permit subsequent repository requests to refer to transactions (e.g. in a terms and conditions negotiation), repository transactions will have unique-over-time *transaction identifier*. These will be assigned by the repository and exist as part of a transaction record.

Two simple examples illustrate the the concepts described above. Figure 2 illustrates the communication between a requester and a repository for an object with no terms and conditions. The requester issues an `ACCESS_DO` for the object, the repository records transaction T1 as "closed" in its transaction record, and returns the requested dissemination. Figure 3 illustrates the same com-

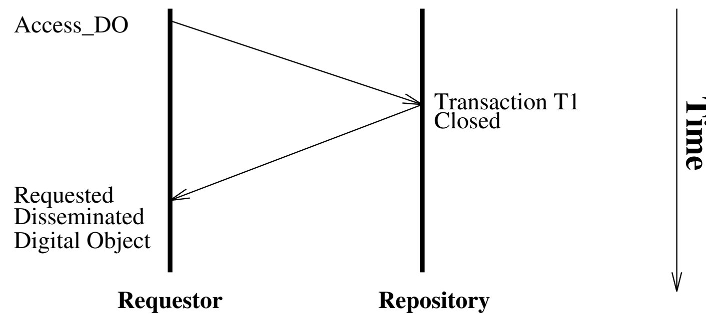

Figure 2: Transaction flow for an object with no terms and conditions

munication for an object with some simple terms and conditions; say, a request for payment. The requester issues an `ACCESS_DO` for the object; in this case the repository returns the codified terms and conditions and records transaction T1 as open in its transaction record. The requester, at some later time, then reissues the `ACCESS_DO`, with parameters that reference transaction T1 and some resolution of the terms and conditions. The repository then returns the requested dissemination and flags T1 as closed.

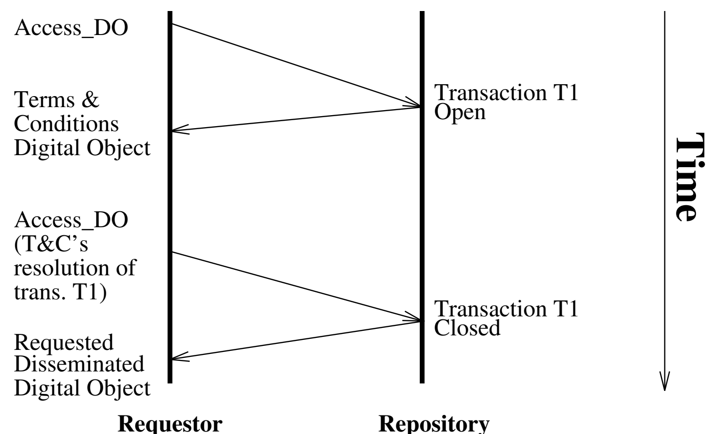

Figure 3: Transaction flow for an object with negotiated terms and conditions

[5]Although RAP is stateful, it is transaction-based rather than connection-based (i.e., there will be no *open-connection* - *close-connection* requests.)

3 Handle server replication

Kahn/Wilensky specifies a handler server infrastructure that is "intended to be a means of universal basic access to objects in the System". Every SDO in the system has a handle; handle servers maintain the mapping from a handle to repositories that store the object. We describe in section 4 the architecture issues related to replicating an object among several repositories.

We expect that in the actual implementation of the architecture there will be handle servers that replicate the mappings of handles to repository locations. The guarantees applying to object replication, mutation, etc. that we describe in the remainder of this document apply to what we call the *primary handle server* (PHS). This is the handle server that is selected by a hashing algorithm given a specific handle. We also use the term *handle server infrastructure* to refer to the set of primary handle servers, and the protocols for registering and locating objects using these servers. The behavior of handle servers outside the handle server infrastructure is not defined.

Finally, throughout the rest of this document we assume that the handle server infrastructure is a trusted component of the FDDOS. The procedures we define are based on the assumption of "good behavior" by the PHS for an object and by the possibility of "badly behaved" repositories (and other architecture components).

4 Digital object replication

The FDDOS provides for the replication of a SDO among multiple repositories. We describe in this section some of the infrastructure guarantees that can be made about replicated objects and the means of ensuring those guarantees.

For each SDO there "must be" one, and only one, repository known as the *repository of record* (ROR). As described below "must be" cannot be strictly enforced, but is enforced by prohibiting certain architecture operations if this rule is broken. The PHS will reference at least the ROR for the object; reference to additional locations is optional. The ROR designation will, by default, be given to the first repository provided by an originator or agent[6] at the time of registration of an object with the handle server infrastructure. An originator may specify another ROR at a later time, but the handle server infrastructure will then remove that designation for the original repository, thus maintaining the "one and only one ROR" rule. Architectural operations will only rely on the PHS's designation of which repository is the repository of record; this information may be recorded within the respective repository, but the use of this information is not defined.

An ROR may be configured to run as a brokerage service. In this case, the ROR is used to negotiate replication rights with other repositories, but the ROR need not directly provide public access to the digital object.

The actual presence of an SDO in a repository[7] will be "eagerly" verified at the time of registration, but will be "lazily" verified after that time. When an originator registers a new location with the handle server infrastructure, the PHS will check that the SDO actually exists at that site (by

[6]Throughout the remainder of this document we use the term originator to refer to the creator of an object (e.g. an author) or the agent (which may be a machine) designated by the originator as having the right to store, replicate, mutate, etc. an SDO.

[7]As defined in Kahn/Wilensky, the notion of the presence of SDO in a repository does not imply physical presence. The SDO may be a logical object. Thus "actual presence" is defined as the ability of the repository to respond to an `ACCESS_DO` for the object (selected by its handle) with a dissemination that is a function of the SDO and the parameters to the request. In addition, in addition the `VERIFY_DO` request, described in appendix A allows a client to request a simple "yes" or "no" answer regarding the "actual presence" of an SDO.

issuing an VERIFY_DO for the object). If this access fails with a "not found" response, the handle server will refuse the registration request.

Over the life of an SDO (which theoretically may be forever), an originator may delete and/or move the object among several repositories. The FDDOS makes no guarantee about the immediacy of location/deletion updates in the PHS. Each PHS will perform a lazy update of its tables by periodically checking the existence of each SDO that it records. We see three possible behaviors by repositories regarding deletions and movement of digital objects:

1. A repository may choose to be "responsible"; it may notify the PHS when an object is deleted or moved to another repository. The PHS will then immediately update its tables in response to the notification (e.g., the location data for that repository would be deleted; the handle itself may remain.)

2. A repository may choose to be "semi-responsible"; that is, it may maintain forwarding information about changes in the location of SDO's. It will then respond to an ACCESS_DO request for the respective object with a dissemination of some type (e.g., X_FORWARD) that provides the new location information[8]. This will allow the PHS to update its tables with the forwarding information.

3. There is no restriction in the FDDOS that repositories must abide by either of the two above behavior's; i.e. a repository may be "uncooperative". In this case, the respective repository will not record forwarding information and will respond to an ACCESS_DO request for a moved object with a "not found" response. In this case, the PHS will remove the reference to that repository.

The procedures that are described above for updating repository dependent information in the PHS may lead to possible violations of the "one and only one" ROR rule. For example, the ROR may be "responsible" and inform the PHS that the object has been deleted. Or, even worse, it may be "irresponsible" or entirely disappear. Finally, the ROR may be the only repository that the PHS knows about, and may no longer provide access to the object. In the case of these violations of the ROR rule, the handle server infrastructure will be responsible for warning the originator that an SDO is in an invalid state and that certain operations and guarantees (some defined in section 5) will be impossible to fulfill.

5 Immutable and mutable digital objects

Stored digital objects can be mutable or immutable. This state is recorded as part of the key-metadata that is part of each object in the repository. In this section we describe the guarantees that can be made concerning the "equality" of replicated digital objects, both immutable and immutable. We also describe the policies and mechanisms for transformations between mutable/immutable states.

There is an *is-equal* method to determine if two objects are the same. We assume that the *is-equal* method for the primitive type *digital object* will return true if and only if the key metadata and the bit stream are respectively, in both cases, equal. Kahn/Wilensky initially specifies a minimal key-metadata that consists of a handle; thus, two SDO's can be equal only if their handles are equal. As additional information is added to the key-metadata, the definition of key-metadata equality

[8]This return should be accompanied by an appropriate error code (e.g., "object has been moved") to facilitate client handling of this transaction.

will have to evolve (but handle equality will always be necessary). Similarly, the "is-equal" method may be defined as something other than bit stream equality for sub-types of digital object as they are defined. Equality between two SDO's is guaranteed by the replicate operation (`REPLICATE_DO`) that is defined in appendix A.

The instances of an immutable SDO that are replicated across several repositories and referenced by the PHS are guaranteed by the infrastructure to be equal[9] . This guarantee is enforced through the following procedure for object deposit and replication. An originator of a digital object uses a `DEPOSIT_DO` request to deposit the first instance of the object in a repository. The originator then registers the handle for the object and the corresponding address for the repository with the handle server infrastructure[10]. Since this is the first registration of this handle, the PHS marks the repository as the ROR. This procedure is illustrated in figure 4; the originator deposits a digital object with handle H1 in repository R1 then registers with the handler server infrastructure that

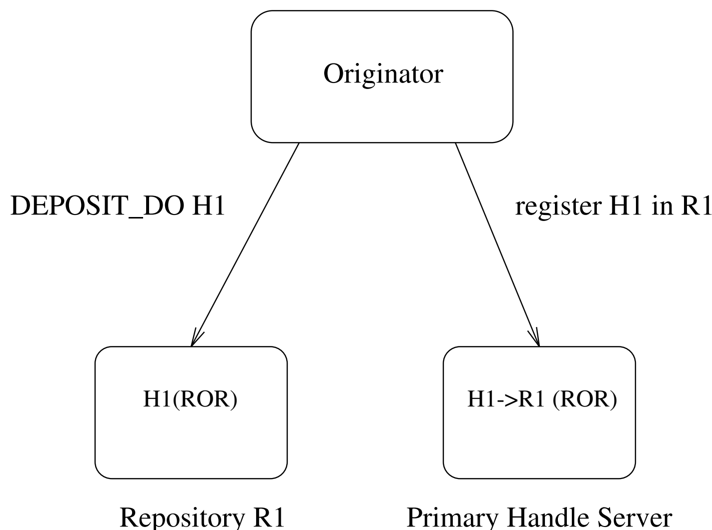

Figure 4: Initial deposit and registration of a digital object

H1 is located in R1. The designated PHS then records this location and flags it as the ROR. Any further attempts to register a new location with the handle server infrastructure by any other party but the ROR for the object will be rejected. This does not mean that an originator may not issue a `DEPOSIT_DO` to another repository, just that the registration of the additional location (with the handle server infrastructure) will be rejected[11] (since there is no guarantee that the object placed in the other repository is identical to the original SDO).

The only prescribed method for replication is via a `REPLICATE_DO` request to the ROR, which is defined as follows. An originator or requesting repository issues a `REPLICATE_DO` to an ROR[12]. The ROR issues a `DEPOSIT_DO` to the *destination repository* (specified in the replicate request)

---

[9] Note that we say "guaranteed by the infrastructure". In other words, both the RAP services and the operations on the handle server infrastructure are defined to enforce this guarantee. We can not guarantee that a malicious repository doesn't willy-nilly disobey RAP rules and modify an instance of a SDO, or that other types of aberrant behavior will not occur.

[10] As defined in appendix A, some repositories may support a `DEPOSIT_DO` operation where the repository itself does the PHS registration.

[11] The alternative to this would be to specify that the registration of a new location makes that the ROR, and eliminates the reference to the original ROR (since, again, we can not trust that the two instances are equal).

[12] This implies that repositories may be clients to other repositories.

with an exact instance of the SDO. The ROR then registers the new location with the handle server infrastructure. The PHS will then check if the registration is coming from the ROR and will accept it if that is true. Note that it is possible to issue a `REPLICATE_DO` to a non-ROR repository, but the handle infrastructure will reject the registration by the source repository of the instance in the destination repository. This procedure is illustrated in figure 5; the originator issues a `REPLICATE_DO` to repository R1 for the SDO with handle H1, with destination repository R2. R1 issues a `DEPOSIT_DO` with an instance of the object to R2 and then registers the new location with the handle server infrastructure. The PHS then has two references for H1; R1, which is still the ROR, and R2.

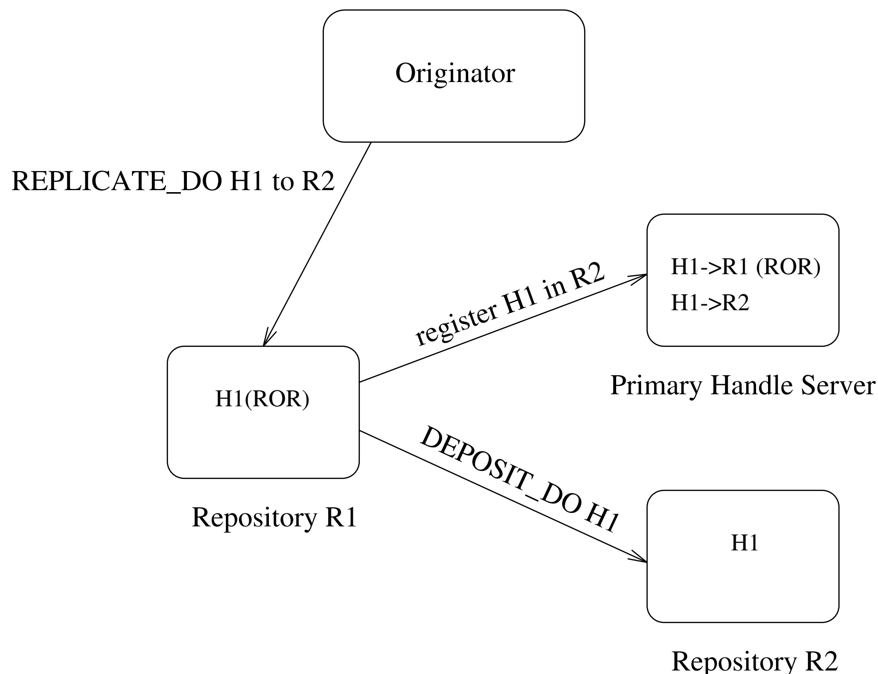

Figure 5: Replication of a digital object

The instances of a mutable SDO that is replicated across several repositories and referenced by the PHS are **not** guaranteed to be equal. Enforcement of equality would require well-behaved repositories and handle servers that enforced a strict transaction commitment protocol upon mutation of an instance of a SDO. The FDDOS, on the other hand, will need to contend with possibly badly behaved repositories and a network with partitions that will not be resolved in any scheduled manner. The originator can make an effort to enforce equality by mutating the object (via a `MUTATE_DO` request) at the ROR and then propagate the change via a `REPLICATE_DO` request to the other repositories on which the object is resident. There is the possibility, however, that the originator fails to propagate the change, the s/he makes different changes to different repositories, or that even a well-intentioned originator/agent tries to propagate a change but is prevented by a partition in the network. Given the possibility of inconsistent instances of a mutable SDO spread across the network, a legal framework that holds the originator responsible only for the instance in the ROR may have to evolve.

While it is not part of the architecture, it is interesting to comment on the market model that is made possible by these assumptions about mutable digital objects. We may see repository vendors position themselves in either of two possible classes; 1) highly reliable operations with high rental

and access fees, and 2) less reliable repositories with correspondingly low fees. Because of the importance of the ROR for mutable SDO's, originators will likely use the high-priced repository as their repository of record. Accessers may be willing to accept a non-guaranteed instance at a lower price from a non-ROR repository (especially for mutable objects where changes are not "mission critical"). An accesser will probably choose to pay for the high-priced repository when "guaranteed latest instance" is essential.

As described above, key-metadata for an SDO records whether it is mutable or immutable. This state is initially determined by parameters of the `DEPOSIT_DO` request that deposits the digital object in the repository. The rules for replication described above guarantee that the state is the same for all instances of the object. Since there is no guarantee that all the instances of a mutable SDO are all the same, mutating an SDO in a repository from mutable to immutable is not allowed. We define the operation `REINSTANTIATE_DO`, which allows an originator to generate a new instance of an SDO, which is an immutable version of a mutable object. Given a source handle and a destination handle, this operation creates an exact instance of an SDO (except for the new handle) in the target repository. A parameter specifies whether the new SDO is immutable or mutable. The following example illustrates this procedure. Assume a handle H1 that references three replications of an SDO in repository R1 (the ROR), R2, and R3. A `REINSTANTIATE_DO` directed to R1 for object H1 with destination handle H2 and the parameter *immutable* will create the new immutable object H2 in R1. The originator may then register H2 with the handle server infrastructure. The originator may then use `REINSTANTIATE_DO` as defined previously to replicate H2 to R2 and R3. Finally, if the originator wants H1 to reference the immutable object, s/he may change H1 in the PHS to point only to H2, effectively making all references to H1 dereference to the immutable SDO's referenced by H2. The handle server infrastructure should expressly **prohibit** the inverse operation; changing the reference of a handle to an immutable SDO to a handle for a mutable SDO. The meaning of immutable from the user's point of view should be that the current instance of the object is exactly as deposited by the originator.[13]

[13]This implies that handles are not re-usable. For example, an originator may not register an object with a handle, delete the object, and then register another object with that handle.

A Repository access protocol services

In this appendix we list the defined operations in the repository access protocol (RAP). Some of these operations have already been described earlier in this document. These are by necessity informal descriptions; more formal definitions will be possible only after the schema for digital objects is developed. In addition, we leave undefined the means of physically encoding RAP requests and responses.

`ACCESS_DO`

- *description*: request a dissemination of a stored digital object from a repository.
- *inputs*:
  1. the handle of the SDO requested.
  2. a method - each sub-type of digital object will have a set of methods defined. For example, the method *execute* might be defined for an element (within an SDO) of type program. The primitive method called *data* is defined for the primitive type digital object. This method returns the data for the SDO (parameters in the request may affect the data; e.g., encrypt).
  3. parameters - parameters are defined for each method. For example, the parameter *page=n* might be defined for the method *data* and type "set of scanned images".
- *outputs*: Possible outputs are:
  1. the disseminated digital object that is a function of the SDO, the method, and the parameters in the request.
  2. a "not authorized" error for an unauthorized client.[14]
  3. an "invalid parameters or method" error.
  4. a "not found" error if the SDO with the handle is not in the repository.
  5. an "object has been moved" error, usually with an `X_FORWARD` object, as defined earlier.
  6. a "terms and conditions negotiation required" error and a terms and conditions object, as defined earlier.

`VERIFY_DO`

- *description*: verify that a digital object is in a repository.
- *inputs*:
  1. the handle of the SDO requested.
- *outputs*: Possible outputs are:
  1. an "object found" response.
  2. a "not authorized" error for an unauthorized client.
  3. a "not found" error if the SDO with the handle is not in the repository.
  4. an "object has been moved" error and an `X_FORWARD` object, as defined earlier.

[14] Authorization is controlled by the repository access control matrix described in section 2. Note that a "not authorized" error to this, and other, RAP requests does not necessarily imply that the requested SDO is in the repository. We can imagine situations where information about the presence or absence of a specific SDO should only be provided to authorized clients.

`ACCESS_META`

- *description*: request an element of the metadata that exists for an SDO in a repository. Note that Kahn/Wilensky leaves the notion of metadata undeveloped at this time (except for the handle in key-metadata). We assume that metadata will be formally defined by a schema for the type system that descends from the primitive type digital object. This schema will define metadata for the primitive type digital object (e.g. type, handle, date of origination, terms and conditions) and additional metadata for each sub-type of digital object (e.g. number of pages for a set of scanned TIFF images).
- *inputs*:
  1. the handle of the SDO for which metadata is requested.
  2. the metadata element requested; valid metadata elements are type dependent, primitive metadata (e.g. type, handle, date of origination, terms and conditions) are always valid.
  3. parameters - the valid parameters are defined by the type schema.
- *outputs*: Possible outputs are:
  1. the requested metadata element.
  2. a "not authorized" response for an unauthorized client.
  3. an "invalid parameters or metadata element" error.
  4. a "not found" response if the SDO with the handle is not in the repository.

`MUTATE_DO`

- *description*: modify the data of an SDO in a repository. Refer to section 5 for more information on mutation of objects.
- *inputs*:
  1. the handle of the SDO to be modified.
  2. parameters - the valid parameters are defined by the type schema.
- *outputs*: Possible outputs are:
  1. a "successful" response.
  2. a "not authorized" response for an unauthorized client.
  3. an "object is not mutable" response.
  4. an "invalid parameters" error.
  5. a "not found" response if the SDO with the handle is not in the repository.

`MUTATE_META`

- *description*: modify the metadata of an SDO in a repository. Refer to section 5 for more information on mutation of objects.
- *inputs*:
  1. the handle of the SDO to be modified.
  2. parameters - the valid parameters are defined by the type schema.
- *outputs*: Possible outputs are:
  1. a "successful" response.

2. a "not authorized" response for an unauthorized client.
3. an "object is not mutable" response.
4. an "invalid parameters" error.
5. a "not found" response if the SDO with the handle is not in the repository.

`DEPOSIT_DO`

- *description*: place a digital object in a repository.
- *inputs*: Kahn/Wilensky defines three possible formats:
  1. inputs are type-data, a handle, a other metadata (e.g., mutable/immutable), in which case the repository stores the object and registers it with the handle server infrastructure. Other metadata may be nil if a mutable digital object with the given handle already exists in the repository.
  2. inputs are a digital object and other metadata (e.g., mutable/immutable), in which case the repository simply stores the digital object, resulting in a stored digital object.
  3. inputs are typed data, in which case the repository requests a handle from the handle server infrastructure, stores the object and registers it with the handle server infrastructure.
- *outputs*: Possible outputs are:
  1. a "successful" response.
  2. an "input format not accepted by this repository" if the service implied by the input format is not provided by the repository.
  3. a "terms and conditions negotiation required" error and a terms and conditions object, as defined earlier.
  4. a "not authorized" error for an unauthorized client.

`REPLICATE_DO`

- *description*: request to cause an identical instance of an SDO to be stored in another repository and register that instance with the handle server infrastructure. As described in section 5 and illustrated in figure 5, the repository that receives this request performs two operations. First it issues a `DEPOSIT_DO` to the repository specified as the destination and then issues a registration request with the handle server infrastructure. The handle server infrastructure will reject the registration if the repository issuing the registration request is not the ROR. The mechanisms for authenticating these requests are left undefined at this time.
- *inputs*:
  1. the handle of the SDO to replicate.
  2. the destination repository.
- *outputs*: Possible outputs are:
  1. a "successful" response.
  2. a "not authorized" response for an unauthorized client.
  3. a "not found" response if the SDO with the handle is not in the repository.
  4. an "invalid destination" response.

5. a “repository is not ROR” response if the repository that receives the request is not the repository of record for the SDO.
6. one of the responses that are specified for `DEPOSIT_DO`.

`REINSTANTIATE_DO`

- *description*: from a specified SDO, produce an instance of an SDO in the repository, with a same type-data and a different handle. This request is fully described in section 5.
- *inputs*:
  1. the handle of the orignal SDO.
  2. the handle of the new SDO.
  3. an indication of whether the new SDO should be mutable or immutable.
- *outputs*: Possible outputs are:
  1. a “successful” response.
  2. a “not authorized” response for an unauthorized client.
  3. a “not found” response if the SDO with the handle is not in the repository.

`DELETE_DO`

- *description*: remove a stored digital object from a repository and delete the reference in the handle server infrastructure.
- *inputs*:
  1. the handle of the object to be deleted.
- *outputs*: Possible outputs are:
  1. a “successful” response.
  2. an “item not found” response.
  3. a “removed from ROR” response, when the repository from which the object is deleted is the repository of record. In this case it is the responsibility of the originator to designate a new ROR with the handle server infrastructure.
  4. a “no replications of object response, when the deletion is of the only instance of an object.
  5. a “not authorized” error for an unauthorized client.

`ACCESS_REF`

- *description*: Return references to one or more servers that are known to index the contents of this repository. Usage of this information (i.e., the RI protocol) is left undefined at this time.
- *inputs*: none
- *outputs*: the list of RI servers, or an ”initial” response from one of the servers.

Acknowledgements

This work was supported in part by the Advanced Research Projects Agency under Grant No. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). Its content does not necessarily reflect the position or the policy of the Government or CNRI, and no official endorsement should be inferred.