The NSDL Repository and API

January 9, 2007

Contents

1 Basic Data Model

For the purpose of understanding and using the NSDL Repository API, it is important to understand the basic data model of the repository. This section is intended to provide a brief overview of the model, oriented towards potential users of the API. For a more complete and theoretical discussion of the model, please refer to the JCDL '05 paper *An Information Network Overlay Architecture for the NSDL* on this topic.[1]

The NSDL Data Repository (NDR) is based on the Fedora digital object repository architecture (`http://fedora.info`). As such, it is a collection of digital objects that may contain data such as RDF relationships, XML, images, etc. The NDR provides a content model and API methods for representing and manipulating data in these base digital objects.

The NDR content model defines five classes of objects (Resource, Metadata, Aggregator, MetadataProvider, and Agent), and three types of data that may be present in each object (Properties, Datastreams, and Relationships). Taken as a whole, the objects and their constituent relationships form a graph of interrelated objects. Figure 1 shows the different classes of objects in the NDR and the relationships that form the basis of this graph

[1] Available online at `http://arxiv.org/abs/cs.DL/0501080`

1.1 Object Types

Each class of object in the NDR represents a distinct concept or entity:

Resource objects are the fundamental information units of the NDR, analogous to books, journal articles, videos, etc. in a physical library. They may contain digital content, such as HTML, XML, or binary data, or point to a location that contains it. Resources form the backbone of the NSDL data model, and all other objects in the NDR serve to enhance or otherwise enrich understanding or enhance the discoverability and re-use of Resources.

Metadata objects contain information (e.g. bibliographic) about an NDR Resource or aggregation of resources. Metadata objects contain one or more data streams to support all the representations (formats) of the metadata made available by the NDR. Metadata objects are roughly analogous to items in the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH).[2] When the NSDL harvests metadata from an OAI data provider, for example, each OAI item represented in the harvest corresponds to a Metadata object in the NDR.

Aggregator objects represent a grouping of Resources, Agents, or other Aggregators. These groupings are intended to have a (possibly application-specific) meaning, such as a defining a collection of of resources in the library, someone's favorite physics resources, or a grouping of like-minded people (Agents).

MetadataProvider objects are used to signify a grouping metadata that comes from a particular source. They differ from Aggregators in that a MetadataProviders only define groupings of Metadata (and not other object types such as Resources), and that every Metadata object must be grouped by exactly one MetadataProvider. In this sense, the provenance of any Metadata object in the NDR may be determined by its MetadataProvider.

Agent objects represent an institution, individual, group, or other entity providing information to the NDR. Agent objects indicate the ultimate source of information such as metadata or groupings of resources (Aggregations).

1.2 Object Content

Each object in the NDR may be composed of three types of digital content: properties, datastreams, and relationships. The NDR defines a basic content model that defines specific members and semantics for object content, referred to as the 'basic' or 'core' model.

Properties. Properties are literal values that are associated with a specific NDR object. Internally, properties are stored as relationships in RDF,

[2] See `http://www.openarchives.org/OAI/openarchivesprotocol.html#Item`

where the 'subject' is the object itself, 'predicate' is the property name, and the 'object' is some literal value. In the NDR model, a property is identified by a namespace-qualified property name. All properties defined in the core model (Listed in §A.1) are exposed in the API as members of the namespace `http://ns.nsdl.org/ndr/request_v1.00/`. Properties do not need to be unique unless explicitly defined to be so in a particular content model.

Datastreams. Datastreams are digital content that is stored in or redirected by a specific object. Accessing a datastream of an NDR object may return digital content directly (such as HTML, XML, or binary image data), or return a redirect to some location on the Internet that contains the content. Datastreams are identified by their datastream name, which must be unique in a given object. The datastreams that exist in the basic NDR model are described in §A.2.

Relationships. Relationships indicate that two NDR objects are related in some way. Internally, Relationships are represented in RDF, where the 'subject' is handle NDR object itself, the 'predicate' is the relationship, and the 'object' is the handle of some other NDR object. In the NDR model, a relationship is identified by a namespace-qualified relationship name and an identifier of the related object. All relationships in the basic NDR model are listed in §A.3, and are exposed as members of the namespace
`http://ns.nsdl.org/ndr/request_v1.00/`.

The basic NDR data model may be supplemented by defining additional properties, datastreams, or relationships that may be present in objects, as well as any rules semantics that may apply. All such 'extended' content models must define their additional properties and relationships in a different namespace than is used for the basic model (`http://ns.nsdl.org/ndr/request_v1.00/`). These extensions to the basic NDR model are essentially 'private', and may form the basis for application-specific overlays over the core NDR model.

1.3 Object Identity

All objects in the NDR are identified by a globally-unique CNRI handle.[3] These handles are assigned and managed by the NDR, and are expected to persist indefinitely. Each handle identifies exactly one object, but it may be possible for an object to have multiple handles.

2 Core API

The core API to the NSDL data repository provides basic functionality to create, modify, delete, and read individual objects or components of objects. For

[3] See `http://handle.net`

a complete listing of all API methods supported, as well as more detailed information about the API in general, please refer to the online documentation.[4]

2.1 API Basics

The NDR Application Programming Interface (API) defines REST requests to interact with the repository and its objects. Requests are HTTP requests to a base URL followed by the API method name, an object handle (if applicable), and any additional arguments (if applicable):

```
http://<base.url>/<method>[/handle][?args]
```

Where:

`<base.url>` is the repository base URL

`<method>` is the name of the API method

`[/handle]` is an object's handle

`[?args]` are additional arguments required by the specific API method.

Additionally, some requests may require an `inputXML` parameter that contains an XML representation of properties, datastreams, or relationships relevant to the given call.

2.1.1 POST and GET requests

The NDR API follows the convention that any request that modifies data content in the repository must be formulated as an HTTP POST request. A request that only reads information is allowed to use the GET method, but may also use POST if it makes sense technically to do so (e.g. if the parameters request exceeds the size limit for GET request). Currently, PUT requests are not supported.

2.1.2 XML Parameters

Several API methods require properties, datastreams, or relationships of an object as parameters. As such, we define a parameter (http form variable) named `inputXML` to contain a representation of the components of an NDR object. Its basic form is sketched out below:

```
<inputXML xmlns="http://ns.nsdl.org/ndr/request_v1.00/" ...>
  <(objectType)>
    <properties>
      [<ns:command>]
         <(ns:property)>propertyValue</(ns:property)>
              ...
```

[4] Available at `http://ndr.comm.nsdl.org/`

```
      [</ns:command>]
               ...
    </properties>

    <data>
      [<ns:command>]
        <(datastream)>...</(datastream)>
               ...
      [</ns:command>]
               ...
    </data>

    <relationships>
      [<ns:command>]
        <(ns:relationship)>objectHandle</(ns:relationship)>
               ...
      [</ns:command>]
               ...
    </relationships>
  </(objectType)>
</inputXML>
```

where:

`(objectType)` is the type of object (e.g. Agent, MetadataProvider, etc)

`(ns:property)` is a qualified property name

`(datastream` is a datastream name

`(ns:relationship)` is a qualified relationship name

`[ns:command]` is a qualified name representing some command or action to be performed on specific object components.

All input XML must be schema valid according to our basic request schema.[5] Regardless of what schema the input XML declares, the NDR will attempt to validate against the basic request schema. Any request that fails validation will be rejected. If a different schema is declared in the request XML, the request will be validated both against the default request schema and the declared schema. This way, it is possible to have an additional layer of checks on application-specific data, if desired

Unless otherwise stated explicitly, the namespace of any element in the request XML is assumed to be `http://ns.nsdl.org/ndr/request_v1.00/`.

[5] http://ns.nsdl.org/schemas/ndr/request_v1.00.xsd

Commands Properties, relationships, and datastreams present in request XML may be enclosed in command tags that indicate a specified action to be performed on the enclosed components. For example, consider the following fragment of request XML:

```
<properties xmlns:crs="http://example.nsdl.org/ndr/crs#"
            xmlns:oai="http://example.nsdl.org/ndr/oai#">
    <add>
        <crs:collection_id>1234</crs:collection_id>
        <crs:collectionNA>5678</crs:collectionNA>
    </add>
    <oai:setName>NSDL Collections</oai:setName>
</properties>
```

This example contains three properties, two of which are wrapped in the `<add>` command.

The commands defined in the core API are:

`<add>` Adds the selected components to a digital object

`<delete>` Deletes the selected components from a specific digital object

`<replace>` Replaces a specific component in a digital object with with the given value. If that component does not exist, then it is added.

Commands are not always required in the request XML. Certain API requests may define a 'default' command that is applied to every component that is not wrapped in an explicit command. These commands are in the standard `http://ns.nsdl.org/ndr/request_v1.00/` namespace

Example Below is an example of possible inputXML for a `modifyMetadata` API request (see §2.4.2 for an explanation of that specific call). Schema declarations have been omitted for clarity and space, and line numbers have been added for reference in the text:

```
1  <inputXML xmlns=http://ns.nsdl.org/ndr/request_v1.00/">
2      <properties>
3          <uniqueID>oai:crs.nsdl.org:7043</uniqueID>
4      </properties>
5      <data>
6         <delete>
7              <format type="obsolete_format" />
8         </delete>
9      </data>
10     <relationships>
11         <replace>
12             <metadataFor>2200/example_handle</metadataFor>
13         </replace>
```

```
14     </relationships>
15 </inputXML>
```

In line 3, we see a property `uniqueID` that is not wrapped in a command. According to §2.4, the default command for `modify` requests is `<replace>`, so this line indicates to replace the existing `uniqueID` in the digital object with the value
`oai:crs.nsdl.org:7043`. Lines 6-8 indicate that a specific datastream is being deleted (using a syntax that is unique to the `modifyMetadataFormat` request), and lines 11-13 indicate that that the `metadataFor` relationship is being changed to point to the object with handle `2200/example_handle`.

Comprehensive examples of inputXML for every relevant API request may be found in the online documentation at `http://ndr.comm.nsdl.org`.

2.2 Creation

Creation of NDR objects is done by the `add` API methods, which are described below (Please see the Appendix for examples). In general, the request XML is composed of property, data, and relationship sections, the contents of which are given to the newly created digital object.

Commands:

- `<add>` (default)

If desired, any set of components in the request XML may be wrapped in the `<add>` command. Doing so is not required, and is redundant. Use of any command other than `add` will result in an error.

2.2.1 addResource

Creates a new Resource object in the NDR.

Properties:

- `<identifier type=?>` (required). The `type` attribute is required, and currently can accept the values `URL` and `OTHER`. A type of `OTHER` is used for anything that is not a URL, such as an ISBN number, DOI, call number, etc.

Relationships:

- `<memberOf>` (optional)

All resources have an identifier that is globally unique for each type. If the given identifier type is `URL`, then the URL will be normalized and checked for equivalence to all existing URLs. Otherwise, if the type is `OTHER`, the entire unmodified value will be compared to those already in the repository with type

OTHER. If a match is found, then the addResource call will succeed, but will not actually add a new resource. It will modify the existing, matching resource by adding any new data contained in the addResource call, and return the handle to the existing object.

2.2.2 addMetadata

Creates a new Metadata object. Since metadata inherently describes other objects in the NDR, and must be associated with a MetadataProvider, a valid addMetadata command requires its metadataFor and metadataProvidedBy objects to already exist in the NDR.

Properties:

- <uniqueID> (required). All Metadata objects must have a single identifier that is unique within all Metadata provided by the same MetadataProvider.

Datastreams:

- <format type=?> (required). Contains the metadata content.[6] The value of type is the metadata format (e.g. oai_dc, nsdl_dc). Multiple <format type=?> metadata format types may exist for a single metadata object.

Relationships:

- <metadataFor> (required). The object this Metadata object is describing.
- <metadataProvidedBy> (required). The MetadataProvider associated with this Metadata object.

The behavior of the <format type=?> datastream content is a special case that is unique to Metadata objects. Using <format type="xyz"> in the request XML creates a datastream named format_xyz in the NDR object. Since datastream IDs must be unique within an NDR object, there may exist at most one datastream for each metadata format. All these datastreams are accessed through the <format type=?> construct. For example, nsdl_dc metadata would be included in the <format type="nsdl_dc"> element in the request XML. Upon an add, this would create a datastream named format_nsdl_dc in the NDR object that would contain the metadata content.

2.2.3 addAggregator

Creates a new Aggregator object. An Aggregator must have a relationship with (aggregatorFor) an existing Agent object in the NDR.

Datastreams:

[6] Refer to the online content at http://ndr.comm.nsdl.org/cgi-bin/wiki.pl?addMetadata for more information on datastream content

- `<serviceDescription>`[7] (required). Contains a Dublin Core metadata that describes of the aggregation of items, as well as any additional descriptive information such as contacts, branding, etc.

Relationships:

- `<aggregatorFor>` (Required). A given Aggregator must be `aggregatorFor` exactly one Agent object in the NDR.
- `<associatedWith>` (Optional). Relates an aggregation to a representative resource (perhaps a 'landing page' that provides access to a an application's view of the Aggregator's objects), if one exists.

2.2.4 addMetadataProvider

Creates a new Metadata Provider object.
Datastreams:

- `<serviceDescription>`[7] (required). Contains a description of the nature of this MetadataProvider.

Relationships:

- `<metadataProviderFor>` (Required). A given MetadataProvider must be `metadataProviderFor` exactly one Agent object in the NDR.

2.2.5 addAgent

Creates a new Agent object. *Use of this call may be restricted to certain users/applications.*

Properties:

- `<identifier type=?>` (required). The attribute `type` is required, and currently may be `URL`, `OTHER`, or `HOST`.

Datastreams:

- `<DC>` (required). Authoritative description of the agent.

Like the Resource object, Agents must have a unique identifier for a given type. Agent identifiers, however, may have valid types of { `URL`, `HOST`, `OTHER`}. If an `addAgent` method is called where the identifier matches an existing agent in th NDR with the same identifier and type, that `addAgent` call will *fail* with an error.

[7] See online documentation at `http://ndr.comm.nsdl.org` for details on its content

2.3 Reads

The basic API for reading object content is `get`. Other calls exist outside of the core API that find, fetch, or format content from NDR objects for specialized purposes such as `describe`. For a complete listing, refer to our extended API in §B.

2.3.1 get

The `get` call can perform two functions on a given object:

- List all properties, datastreams, and relationships
- Retrieve the contents of a given datastream

Listing the contents of an NDR object with handle `<objectHandle>` is performed by requesting the URL `http://<base.url>/get/<objectHandle>`, with no additional parameters. Results are returned in the response XML, as described in the online documentation[8]. The data in the response is analogous the request XML in §2.1.2 in that it is composed of three elements: `<properties>`, `<data>`, and `<relationships>`. The key differences between the response and request formats for conveying object data are:

- The default namespace for the response elements is `http://ns.nsdl.org/ndr/response_v1.0.0/`
- All properties and relationships elements are in their native namespace, which is always different from the default `request_v1.0.0/` or `response_v1.0.0/` namespaces[9]
- The `<data>` element contains elements that match the datastream names, but these elements contain only a *URL* of the `get` call that will fetch their true datastream content.

The contents of individual datastreams may be read from the URL `http://<base.url>/get/<objectHandle>/<datastreamName>`. What is returned depends entirely on the content and mime-type of the datastream. Valid responses include text/xml, ascii text, binary data, and redirects to other URLs.

2.3.2 find

The `find` call returns the handles of all objects that match a given criterion. `find` will return every object that contains *all* properties and relationships specified in the `inputXML` parameter.

Commands:

[8] `http://ndr.comm.nsdl.org/cgi-bin/wiki.pl?APIBasics`. The entire response schema can be retrieved from `http://ns.nsdl.org/schemas/ndr/response_v1.00.xsd`

[9] For core NSDL properties and relationships, their native namespace is currently `http://ns.nsdl.org/api/relationships#`

- `<match>` (Default)

Since `<match>` is the only valid command for the `find` operation, its use is not required. All properties or relationships contained within a `<match>` command (or not wrapped in any command) will be used to find matching objects.

Properties and Relationships

- Any property or relationship is accepted by `find`, though no particular property or relationship is required.

Matching object returned by `find` will contain every property and relationship value specified in the `inputXML`. Thus, if one imagines each specified property or relationship value as a search term, there is an implicit 'and' between every value. Currently, disjunctions or other logical operations cannot be used in a `find` command.

2.4 Updates

Modification of NDR objects is done with the `modify` API methods described below.

Commands:

- `<add>` (optional). Indicates that specified components are to be added to the given object.
- `<replace>` (default, optional). Indicates that specified components are to be replaced with the given value, or added if they do not exist.
- `<delete>` (optional). Indicates that specified components are to be deleted from the given object.

Properties, relationships, or datastreams may be individually added, deleted or replaced by wrapping them in a `<add>`, `<delete>`, or `<replace>` element. If no command is specified, then the default behavior is `<replace>`. For `<delete>` and `<replace>` commands, each element in the request XML is matched to a property or relationship in the NDR by matching namespace-qualified tag names. Datastreams are matched solely by tag name.

2.4.1 modifyResource

Currently, resources are treated as immutable objects in the sense that once they are created, they may not be modified. Therefore, the `modifyResource` command is not available for general use.

2.4.2 modifyMetadata

Modifies a Metadata object.

Properties:

- `<uniqueID>`. The only valid command for `uniqueID` is `<replace>`, as it is a required singleton property.

Datastreams:

- `<format type=?>`. Valid commands are `<add>`, `<delete>`, and `<replace>`, with the following caveats:
  - At least one metadata format datastream must exist in the object at all times
  - At most one datastream in a specific format may exist at any time.

  For `<delete>` and `<replace>` of the `<format type=?>` datastreams, the exact datastream will be matched using the `type` attribute. As explained in §2.2.2, a `<format type="xyz">` element will be matched to a datastream named `format_xyz`.

Relationships:

- `<metadataFor>`.
- `<metadataProvidedBy>`.

2.4.3 modifyAggregator

Modifies an aggregator object.

Datastreams:

- `<serviceDescription>`. The only valid command for `serviceDescription` is `<replace>`, as it is a required, singleton datastream.

Relationships:

- `<aggregatorFor>`. The only valid command for `aggregatorFor` is `<replace>`, as it is a required, singleton relationship.
- `<associatedWith>`. There may be at most one associatedWith relationship.

2.4.4 modifyMetadataProvider

Modifies a MetadataProvider object.

Datastreams

- `<serviceDescription>`. The only valid command for `serviceDescription` is `<replace>`, as it is a required, singleton datastream.

Relationships

- `metadataProviderFor`. The only valid command for `metadataProviderFor` is `<replace>`, as it is a required, singleton relationship.

2.4.5 modifyAgent

Modifies an Agent object.

Properties

- `<identifier type=?>`. The only valid command for `<identifier type = ?>` is `<replace>`, as it is a required, singleton property. Additionally, agent identifiers must be globally unique for each given `type`, so modifying the identifier to an existing value will throw an error.

Datastreams

- `<DC>`. The only valid command for `<DC>` is `<replace>`, as it is a required, singleton datastream.

2.5 Deletes

Deletes to NDR objects do not actually remove the object from the NDR, but instead mark it as deleted. Since NDR objects are involved in relationships to others, we must take care when deleting not to violate the following rule:

> If the 'subject' of a relationship is active, then the 'object' of the relationship must also be active.

Stated another way, an 'active' object may not contain a relationship that points to a deleted object. For example, It is possible to delete a MetadataProvider object only if there are no active Metadata objects with `metadataProvidedBy` relationships pointing to it. The relationship `metadataProviderFor` between the MetadataProvider and Agent is not relevant, since the MetadataProvider is the 'subject' of that relationship. Since `metadataProvidedBy` is a required relationship for Metadata objects, the only way to delete a MetadataProvider object is to delete all related Metadata objects first.

As another example, an Aggregator may be deleted only when there are no `memberOf` relationships pointing to it from resources or other aggregations.

Since `memberOf` is not a required relationship, an Aggregator may be deleted after modifying its member resources to delete all `memberOf` relationships that point to it.

2.5.1 delete

Marks an object as deleted. Does not require any parameters other than the object handle in the request URL. Since the `delete` request affects repository state, it must be submitted with the POST method.

3 Authentication and Security

The security policy of the NDR exists to enforce three general goals:

- Verify the identity of a user of the NDR
- Ensure a user can only modify, add, or delete objects when he/she/it has permission to do so
- Ensure that some types of content (e.g. specific properties, datastreams, or relationships) may only be added, modified, or deleted by certain users

To that end, we have implemented an infrastructure that meets these goals by

- Associating each NDR user with a single 'Agent' digital object
- Establishing the identity and authenticity of the user as an NSDL Agent using digital signatures on all API requests
- Defining a base set of rules that use the graph structure to determine if a particular user may modify a particular object.
- Creating a mechanism that accepts or rejects an API request given the user's identity and a set of applicable rules

3.1 Users and Agents

A user of the NDR, as far as our security model is concerned, is the identity of the Agent object that represents a person or application that issues API requests to the NDR, henceforth known as the 'agent'. Stated another way, the NDR is accessed by 'users'. A user is be an individual, institution, or application that is capable of issuing NSDL API requests. Every such user must have a representative Agent object in the NDR. The contents and identity of this Agent object are used for authentication and authorization of API requests.

An API requests may identify the agent that is issuing the call by including its handle (i.e. the handle of the corresponding Agent object) in the http header.

3.2 Authentication

The current NDR authentication protocol (1.0) uses signed HTTP headers and public key cryptography to verify the identity of the agent and the veracity of the contents of the API request (if desired). The process is sketched out below and explained in detail in the following sections:

Client Side:

- Generate HTTP headers that contain the required content
  - Include a `content-md5` or `x-nsdl-md5` header item with an md5 hash of message content, if desired
- Generate a "canonical header" string given this HTTP header and request URL
- Generate an SHA1 hash of this canonical header string, and sign it with a private RSA key
- Insert a base64-encoded version of this signed hash into the HTTP header, along with the agent's identity (handle).

Server (NDR) Side:

- Generate a "canonical" string from the HTTP header and request, and an associated SHA1 hash
- Decrypt the signed SHA1 hash with the Agent's public key
- Compare the calculated SHA1 hash value of the canonical header with the signed SHA1 hash.
- Compare md5 hashes of the content with `content-md5` or `x-nsdl-md5`, if provided

3.2.1 The Canonical Header

The canonical header is a string composed of elements from an HTTP header and a request URL. The actual process is explained in detail in the appendix, §C. Essentially, there is a set of required information that must be present in a canonical header, such as a date and the request URL. Given an acceptable header, the canonicalization process is able to deterministically generate a string that may be used for signing the request on the client side and verifying signatures on the server-side.

3.2.2 Header Signature

Given a canonical header, an agent's private key, and an agent's identity (i.e. its handle), an HTTP header may be signed by:

- Calculate an SHA1 hash of the canonical header
- Encrypt the canonical header's SHA1 hash with the Agent's private RSA key
- Base64 encode the signed SHA1 hash
- Include the encrypted hash and agent handle in the header by inserting the following line into the HTTP header:

  `x-nsdl-auth: <protocol> <agent handle>:<encrypted SHA1 hash>`

  where:

  `<protocol>` is the authorization protocol

  `<agent handle>` is a handle to the Agent's object in the NDR

  `<encrypted SHA1 hash>` is the base64 encoded signed hash of the canonical header.

  Currently, there is only one accepted value for `<protocol>`, namely `nsdl-1.0`

Example Suppose a client application is communicating a request to the NDR `POST http://repository.nsdl.org/api/addAggregator` as an agent identified by the handle "2200/myHandle". The http client may produce the following http header:

```
accept: text/xml,application/xml,text/plain
accept-encoding: gzip,deflate
accept-charset: utf-8
keep-alive: 300
connection: keep-alive
```

In this example, none of the default content is relevant to the canonical header, so required content must be added. According to the given spec, at minimum an `x-nsdl-date` element must be added to the header. With this, we may generate the following canonical header to sign:

```
POST http://repository.nsdl.org/api/addAggregator


x-nsdl-date: Tue, 04 Jun 2005 04:21:05 -0400
```

...where each blank line is a single newline character. This canonical header would then be hashed and signed, giving a base64 encoded signature of `:tIi5Ve4KbELf+Ji2ZSjy2AC1E6Y=`. The resulting signed header sent to the NDR, then, would look like:

```
accept: text/xml,application/xml,text/plain
accept-encoding: gzip,deflate
accept-charset: utf-8
keep-alive: 300
connection: keep-alive
x-nsdl-date: Tue, 04 Jun 2005 04:21:05 -0400
x-nsdl-auth: nsdl-1.0 2200/myHandle:tIi5Ve4KbELf+Ji2ZSjy2AC1E6Y=
```

3.2.3 Public Keys

The 1.0 authentication protocol uses RSA asymmetric key cryptography to sign the HTTP headers. Each agent has one key pair. The private key is kept secret by the user or user application, and is used to sign API headers. The public key is stored as a Datastream named `Public Key` in the Agent object in the NDR. When a signed API request is sent to the repository, this datastream is fetched from the object indicated by the agent's handle.

Currently[10], the public key is stored as an inline XML datastream that conforms to the W3C XML digital signature recommendation[11].

3.3 Rules for Authorized Actions

This section describes the most basic set of rules that are applied to every API request to determine if a given agent is authorized to add, delete, or modify (or read) a particular object or component of an object. This set of rules is expected to grow and perhaps change as the NDR model matures. Please note that there are application or context-specific rules in place in the NDR that are not part of this basic set. These rules are documented elsewhere.

There are three general types of rules that can apply to a particular API operation:

1. Rules that govern adding, deleting, or modifying properties
2. Rules that govern adding, deleting, or modifying datastreams
3. Rules that govern adding, deleting, or modifying relationships

The first two (properties and datastreams) are similar, and can both be thought of as 'content' rules, while rules for modifying relationships slightly differ. The key difference lies in the fact that a relationship may physically exist as part of a single object, yet represent a concept found in another. Perhaps the most relevant example is aggregation membership. An aggregation is conceptually thought to 'contain' items, so one might want to 'add a resource to an aggregation'. To do so, one must add a 'memberOf' relationship to the resource that points to the aggregation. By physically modifying the resource, you are

[10] Perhaps this will change at some point to allow the more ubiquitous plaintext PEM wrapped PKCS#8 format

[11] `http://www.w3.org/TR/xmldsig-core/#sec-KeyInfo`

logically modifying the contents of the aggregation. Because relationships inherently involve two objects, rules that apply to adding or deleting relationships may have to involve multiple objects.

3.3.1 Authorized Agents

The Security subsystem defines a relationship `http://ns.nsdl.org/ndr/auth#authorizedToChange` that may exist between a MetadataProvider or Aggregator and an Agent *or aggregation of agents*. An agent that is related to an Aggregator or MetadataProvider through that relationship is known as an "Authorized Agent". An authorized agent may perform the following actions on an object it is authorized to change through that relationship:

- Add or remove a 'memberOf' or 'metadataProvidedBy' relationship to a Resource or MetadataProvider object that points to that object
- Add, delete, or modify a property of that object
- Delete the authorizedToChange relationship to itself
- Add an authorizedToChange relationship pointing to another Agent *or aggregation*
- Delete that object

3.3.2 Trusted Applications

A trusted application is an agent that is a member of the 'Trusted Applications' aggregation. A trusted application has the ability to add relationships to any agent in the NSDL repository. Specifically, a trusted application may create a MetadataProvider or Aggregator, and have its metadataProviderFor or aggregatorFor relationship point to any agent. Additionally, a trusted application may add an authorizedToChange relationship to any aggregation or MetadataProvider, pointing to any agent *or aggregation*.

3.3.3 Content Rules

If the agent is authorized according to table 1, it may modify any property in the `http://ns.nsdl.org/ndr/request_v1.0.0/` namespace or create, modify, or delete any non-reserved datastreams in the basic content model[12]

As an example from the table, a Metadata object may have datastreams or properties modified by its metadataProvider's authorized agent, Aggregators or MetadataProviders may be modified by their authorized agents, and the only one who may modify an agent is itself.

[12]Unlike properties and relationships, Datsstreams are not stored as RDF content. Instead, they are Fedora datastreams or disseminations. As such, there is really no native concept of namespace for them. They just have names. Consequently, a certain number of names are reserved datastreams

Object	Authorization
Resource	—
Metadata	$A_{m \to mp}$
MetadataProvider	A_{mp}
Aggregator	A_{agg}
Agent	A

Table 1: Authorization table for relationships, where $A_{m \to mp}$ is the "Authorized Agent of the Metadata's MetadataProvider", A_{mp} is the "Authorized Agent of the Metadataprovider", A_{agg} is the "authorized agent of the Aggregator", and A is "the Agent itself"

Currently, there are no content rules that apply to resources, and resources are considered "immutable" once created. We intend for this to change, but have not yet incorporated any rules regarding Resources into our framework, therefore rendering them immutable by default.

3.3.4 Relationship Rules

The basic relationship rules are shown in table 2, which apply to the relationships in the basic content model. These rules determine which agents are authorized to create, delete, or modify the corresponding relationships

		Object				
		Resource	Metadata	M.P	Aggregator	Agent
Subject	Resource	—	—	—	A_{agg}	—
	Metadata	$A_{m \to mp}$	—	$A_{m \to mp}$	$A_{m \to mp}$	$A_{m \to mp}$
	M.P.	—	—	—	—	A, TA
	Aggregator	A_{agg}	—	—	A_{agg}	A, TA
	Agent	—	—	—	A_{agg}	—
		Authorization				

Table 2: Authorization matrix for relationships, where $A_{m \to mp}$ is the "Authorized Agent of the metadata's metadataProvider," A_{agg} is the "authorized agent of the aggregator," A is "the Agent itself," TA is "a Trusted Application"

As an example, suppose an API request adds a 'memberOf' relationship between a Resource and an Aggregator. According to the table, the only agent that may do so is the Aggregator's authorized agent. Therefore, the rule's logic would be "A 'memberOf' relationship between a Resource and an Aggregator is only valid if the agent making the API request is the Aggregator's authorized agent."

3.3.5 Application of Rules

The basic authorization rules described in §3.3.3 and §3.3.4 are combined with any others that may apply[13] into a common pool. The rule application mechanism tries to match all individual components (properties, relationships, and datastreams) in the API request one or more rules in the pool. Currently, we employ a rather strict policy in matching and applying the rules, namely:

- If a component matches any rules, it must comply with each one, evaluated in the context of the agent making the request
- If there is a component to which no rule applies, the API request is not authorized

[13]The system was designed to accommodate the dynamic loading of rules on a per-namespace basis, but only the minimum required for loading the basic rules, and rules relating to NDSL applications (e.g. CRS, OAI) are implemented.

A Core Object Components

While any object may contain properties, datastreams, or relationships not detailed here, we define a base set that comprises our basic data model. Below is a listing of all objects and all components, required and not, that are part of the basic data model.

A.1 Properties

status (All objects) Indicates whether an object is 'Active' or 'Deleted'.

created (All objects) Contains the date and time the object was created in the NDR.

modified (All objects) Contains the date time the object was last modified in the NDR.

objectType (All objects) Indicates the objects class. The current classes of object are, of course, Resource, Metadata, Aggregator, MetadataProvider, and Agent.

identifier (Resource, Agent) Uniquely identifies an agent or resource. There are currently three types of identifiers: URL, host, and 'other'. Most resources are identified by the URL that points to where their online content may be found, while many Agents are identified by a 'host'. Examples of 'other' identifiers include an ISBN number, DOI handle, or ISSN.

uniqueID (Metadata) Uniquely identifies a Metadata object for each MetadataProvider. Has no other particular meaning in the basic data model other than its uniqueness.

A.2 Datastreams

content (Resource) Contains the digital content of a Resource. Examples of resource content include HTML web pages, PDF documents, XML files, images, and videos.

format_xyz (Metadata) The Metadata object may contain different formats of metadata that serve to describe the same object. For a given metadata format 'xyz', there exists a datastream named 'format_xyz' that contains the metadata content.

format_xyz_info (Metadata) Contains information a particular unit of metadata.

serviceDescription (Aggregator, MetadataProvider) May contain Dublin core metadata, branding, and contact information for a given Aggregator or MetadataProvider. The function of serviceDescriptions are to explain the nature of a particular aggregation or source of metadata.

DC (Agent) Contains Dublin core metadata that serves as the authoritative description of a particular agent.

A.3 Relationships

memberOf (Subject:{Resource, Aggregator, Agent}; Object: Aggregator) Indicates that the given object is a member of an aggregation. A Resource, Aggregator, or Agent may have any number of memberOf relationships.

metadataFor (Subject: Metadata; Object: {Resource, Aggregator, Agent}) Indicates that a Metadata object contains metadata about another. A Metadata object must have at least one metadataFor relationship.

metadataProvidedBy (Subject: Metadata; Object: MetadataProvider) Indicates that a given Metadata object came from a particular source. A Metadata object must have exactly one metadataprovidedBy relationship.

aggregatorFor (Subject: Aggregator; Object:Agent) Indicates that a particular aggregation was created by or belongs to a specific Agent. An Aggregator object must have exactly one aggregatorFor relationship.

associatedWith (Subject: Aggregator; Object: Resource) Indicates that a particular resource is representative of a given aggregation. In other words, the contents of the Resource are in some way indicitive of the nature of the contents of the Aggregation.

metadataproviderFor (Subject: MetadataProvider; Object:Agent) Indicates that a particular source of Metadata is attributed to an Agent.

B Extended API

§2 detailed the requests present in the core API. These requests comprise a minimal set of operations that allows full read and write access to all NDR objects. In this section, we describe all other API requests that exist outside this minimal core, but are useful for information discovery or visualization. This list is expected to grow and change as the NDR use cases mature. More detailed documentation and examples may be found in the online documentation at http://ndr.comm.nsdl.org

B.1 countMembers

Counts the number of Active members of an Aggregator, or the number of Metadata items provided by a metadataprovider. This request is executed by simply viewing the contents of `http://<base.url>/countMembers/<handle>` with no parameters. The count is returned in the responseXML.
See: `http://ndr.comm.nsdl.org/cgi-bin/wiki.pl?countMembers`

B.2 describe

Provides information about the content and relationships of an NDR object. It differs from `get` in response format and content, as it is intended to provide a human or machine readable summary of an object, with added data from related objects and relationships. As an example, the `describe` page for a resource contains metadata gleaned from all related Metadata objects.

Usage is `http://repository.nsdl.org/describe/<handle>[?view=html]`, where the `view` parameter may be used to produce a human-readable description of the object's contents in HTML.
See: `http://ndr.comm.nsdl.org/cgi-bin/wiki.pl?describe`

B.3 findMetadata

Finds a Metadata object given a Resource identifier or MetadataProvider handle.

The base URL for the `findMetadata` request is, of course, `http://<base.url>/findMetadata/<handle>`. The usage depends upon what information is given:

Given a Resource URL *xxx***:** This request may be executed by appending a parameter `?URL=`*xyz* to the request base URL

Given a MetadataProvider Handle *xyz***:** Append the parameter `MetadataProvider=`*xyz* to the request URL

Given a Resource Identifier (Not necessarily a URL): Submit a request XML parameter containing a `<identifier type=?>` property.

If a matching Metadata item is found, its handle will be returned in the request XML.

B.4 findResource

Finds a resource given an identifier or handle, and returns the results of a `describe` call on that resource.

Usage:

Given a Resource URL*xyz* It is possible to formulate a request by appending `?URL=`*xyz* to the request URL.

Given an identifier *i* **and type** *t* It is possible to formulate a request by appending `?identifier=`*i*`&type=`*t* to the URL. Alternatively, one may submit inputXML containing the identifier and type in the `<identifier type=?>` property.

Given a resource handle *xyz* The request must be formulated by appending `?handle=`*xyz* to the request URL

B.5 findAgent

Finds an Agent object given a request XML parameter containing an identifier as a `<identifier type=?>` property. Returns response XML containing the handle of the matching Agent

B.6 listMembers

Lists the handles of all active members of an Aggregator or Metadata provided by a MetadataProvider. Much like `countMembers`, this request has no parameters, and is executed by simply viewing
`http://<base.url>/listMembers/<handle>`.

The response XML contains the handles of all members in no specific order. Be warned: some aggregations or collections of metadata are very large, and will result in a correspondingly large response from the repository.

C Header Canonicalization

The canonical header is a string composed of elements from an HTTP header and a request URL. Given a header and a request URL, the canonical header string is formed from the following, in order:

1. The method (e.g.. PUT, POST, GET) followed by a space, then the given URL, terminated with a newline.
2. The exact contents of the `Date` header element (if present), followed by a newline. If there is no `Date` element, place a newline character in its place.
3. The `content-md5` element, if defined. If not in header, then place a newline in its place.
4. All header elements that begin with `x-nsdl`, except for `x-nsdl-auth`, ordered alphabetically in ASCII order.

In addition to the above, there are some additional constraints on canonical header content:

- If there is no `Date` element, there must be an `x-nsdl-date` element. in the canonical header.
- The date (derived from either `Date` or `x-nsdl-date`) must be no more than 5 minutes away from the current NDR time.
- `x-nsdl-date` must be in standard rfc-822 format
- If both `Date` and `x-nsdl-date` are provided, the value from `x-nsdl-date` will be compared with the NDR system clock.
- All content in the canonical header must be UTF-8 encoded.

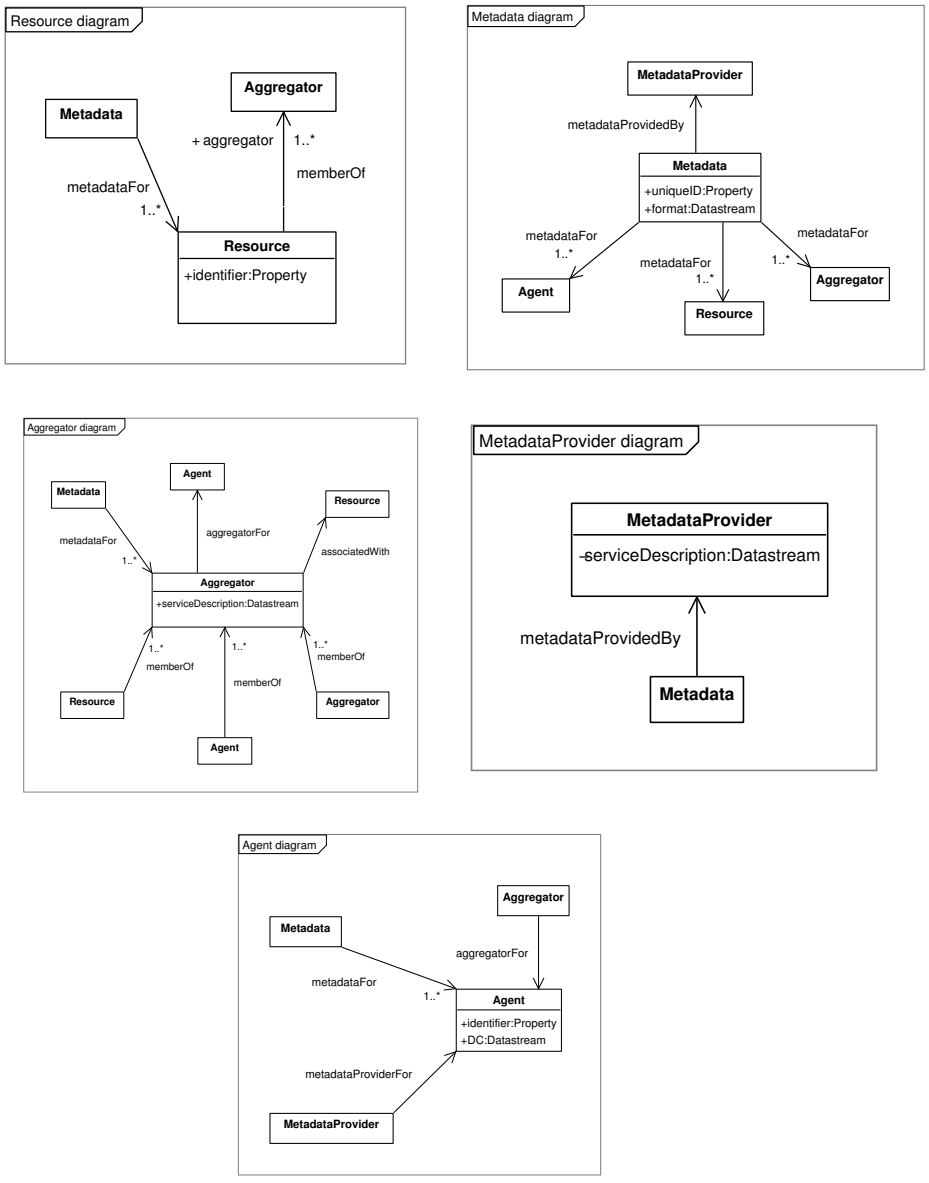

Figure 1: The five classes of objects in the NDR, along with properties, datastreams, and relationships defined in the basic model