Mellon Fedora Technical Specification

(December 2002)

Introduction

1.0 Mellon Fedora System Overview

1.1 **Background of the Project**

In September of 2001, the University of Virginia received a grant of $1,000,000 from the Andrew W. Mellon Foundation to enable the Library to collaborate with Cornell University to build a sophisticated digital object repository system based on the Flexible Extensible Digital Object and Repository Architecture (Fedora). Fedora was originally developed as a research project at Cornell University and was successfully implemented at Virginia in 2000 as a prototype system to provide management and access to a diverse set of digital collections.

The Mellon grant was based on the success of the Virginia prototype and the vision of a new open-source version of Fedora that exploits the latest web technologies. Virginia and Cornell have joined forces to build this robust implementation of the Fedora architecture with a full array of management utilities necessary to support it. A deployment group, representing seven institutions in the US and the UK, will evaluate the system by applying it to testbeds of their own collections. The experiences of the deployment group will be used to fine-tune the software in later phases of the project.

1.2 **Summary of Functionality**

The Mellon Fedora System consists of two fundamental entities: (1) the underlying Fedora *digital object* architecture and (2) the Fedora *repository*. The digital object forms the core of Fedora architecture, providing a framework that enables the aggregation of both content (i.e., data and metadata) and behaviors (i.e., services) that can also be distributed across multiple platforms via a URI. The Fedora repository provides management and access services for these digital objects. Clients interact with the repository through the management and access services. The Fedora system represents a full-featured system that is a foundation upon which interoperable web-based digital libraries can be built.

The goal of the new Mellon Fedora System specification is to create an implementation of Fedora that builds on the designs of the original Cornell reference implementation and Virginia prototype, is highly compatible with the web environment, is built with established standards where possible, and uses freely available technologies. Specifically, the original Fedora model has been reinterpreted using XML and Web services technologies. Our new implementation has the following key features:

- The Fedora repository system is exposed as a Web service and is described using Web Services Description Language (WSDL).

- Digital Object behaviors are implemented as linkages to distributed web services that are expressed using WSDL and implemented via HTTP GET/POST or SOAP bindings.
- Digital objects are encoded and stored as XML using the Metadata Encoding and Transmission Standard (METS).
- Digital objects support versioning to preserve access to former instantiations of both content and services.

The Mellon Fedora System is exposed as two related web services: the Fedora Management service (API-M) and the Fedora Access service (API-A). The service interfaces are expressed in XML using WSDL, as are all auxiliary services included in the architecture. The Fedora Management service defines an open interface for administering the repository, including creating, modifying, and deleting digital objects or components within digital objects. The Fedora Access service defines an open interface for accessing digital objects and the behaviors (i.e., services) associated with them. The open interfaces for the Management and Access APIs enable developers to more easily develop client tools that interact with the repository system or to completely re-implement their own version of the repository system.

This document is divided into four main sections: (1) Introduction, (2) Architecture, (3) Public APIs for Fedora Services, and (4) System Implementation. The Architecture section describes the key architectural components of the Fedora digital object and repository. The Public APIs section describes the open interfaces that define the management and access services. The final section describes a specific implementation of the Mellon Fedora System currently underway by the Fedora development team. The diagram in Section 6.0 depicts the various modules that comprise this implementation of the Fedora system.

Architecture

2.0 Digital Object Architecture

2.1 Digital Object Architecture Summary

The principal entity of the Fedora architecture is the *digital object*. As outlined in Figure 1, a Fedora digital object is comprised of several components including a unique persistent identifier (PID), one or more `disseminators`, system metadata, and one or more `datastreams`. A significant characteristic of the Fedora digital object is how it enables the aggregation of content (i.e., data and metadata) and behaviors (i.e., services). Both content and behaviors can be distributed and referenced via a URI. As depicted in Figure 1, `datastreams` represent content and `disseminators` represent services. A Fedora repository provides both access and management services for digital objects.

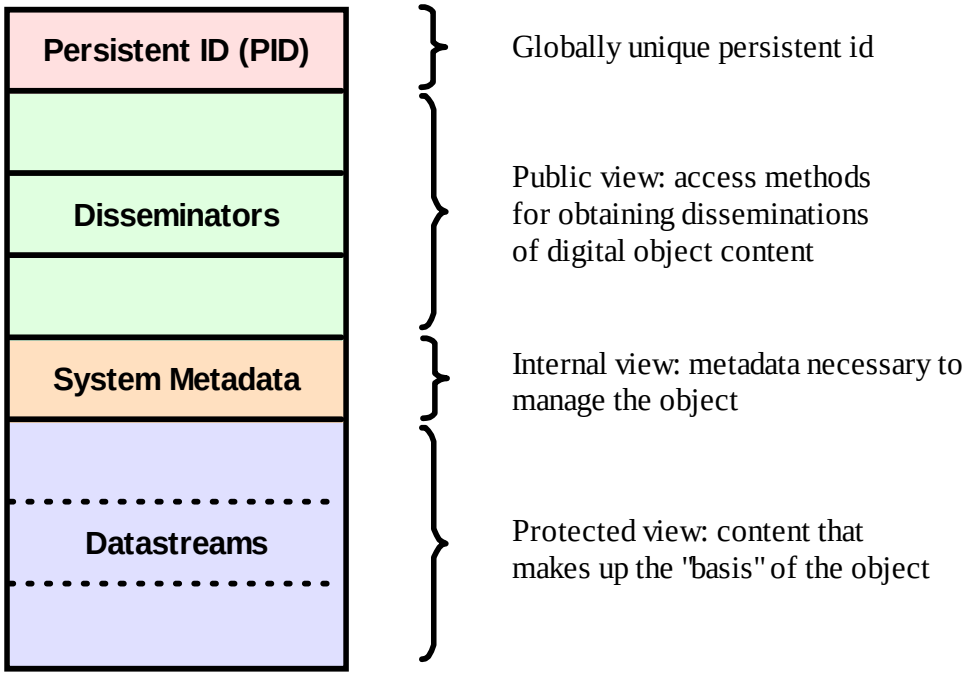

Figure 1. Fedora Digital Object

From an access perspective, the architecture fulfills two basic functions: (1) it exposes both generic and extensible behaviors for digital objects (i.e., as sets of method definitions), and (2) it performs *disseminations* of content in response to a client's request for one of these methods. A dissemination is defined as a stream of data that manifests a view of the digital object's content.

`Disseminators` are used to provide public access to digital objects in an interoperable and extensible manner. Each `disseminator` defines a set of methods

through which the object's `datastreams` can be accessed. For example, there are simple `disseminators` that define methods for obtaining different renditions of images. There are more complex `disseminators` that define methods for interacting with complex digital creations such as multi-media course packages (e.g., GetSyllabus, GetLectureVideo). Finally, there are `disseminators` that define methods for transforming content (e.g., translating a text between different languages or formats). A `disseminator` is said to subscribe to a `behavior definition`, which is an abstract service definition consisting of a set of methods for presenting or transforming the content of a digital object. A `disseminator` uses a `behavior mechanism`, which is an external service implementation of the methods to which the `disseminator` subscribes. A `disseminator` also defines the binding relationships between a `behavior mechanism` and `datastreams` in the object.

2.2 Digital Object Components

2.2.1 **Persistent Identifier (*PID*) –** A unique, persistent identifier for the digital object. A PID must be guaranteed unique across all Fedora repositories to prevent identity clashes when multiple repositories are running in parallel, or when repositories are federated to form distributed digital libraries across multiple institutions.

2.2.2 **System Metadata –** The System Metadata for a digital object is the metadata that must be recorded with every digital object to facilitate the management of that object. System metadata is distinct from other metadata that is stored in the digital object as content. System metadata is the metadata that is *required* by the Fedora repository architecture. All other metadata (e.g., descriptive metadata, technical metadata) is considered optional from the architectural standpoint, and is treated as a `datastream` in a digital object. The following elements are defined as System Metadata for a digital object:

2.2.2.1 **Digital Object Label (*doLabel*) –** a descriptive label for the digital object appropriate for presentation to humans.

2.2.2.2 **Content Model Type Identifier (*contentModelID*) –** a unique identifier that signifies the particular Content Model upon which the object is built (e.g., the UVa standard image content model, or the UVa TEI book content model).

2.2.2.3 **Digital Object Created DateTime (*createdDT*) –** the date and time that the digital object was originally created.

2.2.2.4 **Digital Object Last Modified DateTime (*modDT*) –** the date and time of the most recent change to the digital object. Note that if a digital object is marked as "deleted" in the record status element (see section 7.3), then the Last Modified DateTime essentially constitutes the date of deletion.

2.2.2.5 **Digital Object Status (*status*) –** a flag that indicates the current state of an object. See Section 7.3 for a list of valid object and object component state values.

2.2.2.6 **Digital Object Component Audit Trail** – The System Metadata records transaction records for all changes made to the digital object. The audit trail records are expressed in XML in accordance with the Fedora Audit Trail Schema (fedoraAudit.xsd). An audit trail record describes an action, date, responsible agent, process used, and justification for action.

- **Internal Identifier (*auditTrailID*) –** an identifier for the audit trail section of the object. This is an internal component identifier that is unique within the object, and not considered a public identifier.
- **Transaction Record** ()
  - **Internal Identifier (*auditRecordID*) –** an identifier for the audit trail record within the audit trail section of the object. This is an internal component identifier that is unique within the object, and not considered a public identifier.
  - **Record (*fedoraAudit:record*) –** every change to either a `datastream` or a `disseminator` requires an audit record to be inserted into the audit trail section of the digital object. The `datastream` or `disseminator` to which the record pertains must have a link to the audit record.

2.2.3 **Datastreams –** a `datastream` is the component of a digital object that represents digital *content*. In other words, `datastreams` represent the digital stuff that is the essence of the digital object (e.g., digital images, encoded texts, audio recordings). All forms of metadata, except system metadata, are also treated as content, and are therefore represented as `datastreams` in a digital object. All `datastreams` have the potential to be disseminated from a digital object. A `datastream` can reference any type of content, and that content can be stored either locally or remotely to the repository system. All `datastreams` have the following attributes:

- **Internal Identifier (*datastreamID*)** – the identifier for a `datastream` within a digital object. This identifier is common to all versions of a particular `datastream`. This is an internal component identifier that is unique within the object, and not considered a public identifier.
- **Version Identifier (*datastreamVersionID*)** – the identifier for a particular version of a `datastream` within a digital object.
- **Datastream Label (*dsLabel*) –** a human readable label describing the `datastream`.

- **MIME (*dsMIME*)** – the MIME type of the content byte stream
- **Created Date (*dsCreateDT*) –** the date and time the `datastream` component was created in the digital object.
- **Size (dsSize) –** size of content byte stream in bytes
- **Datastream Control Group Type (dsControlGroupType)** – the control group type of the content datastream. There are three possible control group types: Referenced External Content (E), Repository-Managed Content (M), or Implementor-Defined XML Metadata (X).
- **Content (dsContent)** – the actual byte streams that the `datastream` component represents. A `datastream` may point to content that is stored outside the repository system, or it may point to content that is under the custodianship of the repository. Some content may be XML-encoded metadata that is tightly bound to a digital object. Fedora defines three types of `datastreams` to accommodate these three content scenarios:

2.2.3.1 **Referenced External Content –** A form of `datastream` that points to content that is outside the custodianship of the repository. It is used to reference remote content that is outside the container of the digital object. An acceptable pointer for referencing content is a URL. External Referenced Content `datastreams` can point to any type of content (e.g., images, text documents, video, executables, etc.). It should be noted that metadata can also be stored as a Referenced External Content `datastream`. In cases where metadata is not available in an XML format, or when it is desirable to keep metadata stored in separate files, that metadata can be stored by reference. The fact that the `datastream` content is actually metadata is opaque from the repository perspective.

2.2.3.2 **Repository-Managed Content –** a form of `datastream` that references content that is under the custodianship of the repository. Repository-Managed Content `datastreams` can refer to any type of content (e.g., images, text documents, video, executables, etc.). It should be noted that in cases where metadata is not available in an XML format, or when it is desirable to keep metadata stored in separate files, that metadata can be stored in the repository using a Repository-Managed Content `datastream` component. The fact that the `datastream` content is actually metadata is opaque from the repository perspective.

2.2.3.3 **Implementer-Defined XML Metadata –** a form of `datastream` that is stored "inline" as part of the XML-encoded digital object. Fedora digital objects are stored as XML files, therefore, it is possible to store XML-encoded metadata directly inside a digital object. These special `datastreams` are intrinsically bound to the digital object (i.e., they are NOT remote references (i.e., URLs) pointing to content

outside of the digital object container). By definition Implementer-Defined XML Metadata `datastreams` contain well-structured XML with an XML namespace. A side benefit of this type of `datastream` is that metadata can be directly indexed when a digital object XML file is indexed (i.e., via XML indexing software that may be configured with Fedora). It should be noted, however, that from an access perspective, Implementer-Defined XML Metadata `datastreams` are treated the same as any other `datastream`. This means that they can be disseminated from the digital object (i.e., the metadata gets treated like other content in the digital object).

2.2.4 **Disseminators –** A `disseminator` is the component in a digital object that is used to associate behaviors (i.e., services) with the object. Each `disseminator` names a `behavior definition` and a `behavior mechanism`. The `behavior definition` is a formal definition of a set of methods. This can be thought of as similar to an abstract interface. A `behavior mechanism` is an executable that implements the behaviors. `Behavior definitions` are stored in `behavior definition` objects, which are themselves Fedora digital objects with unique PIDs. `Behavior mechanisms` are stored in `behavior mechanism` objects, which are themselves Fedora digital objects with unique PIDs. A `disseminator` is associated with `behavior definition` object and `behavior mechanism` objects by naming the PID for each.

2.2.4.1 **Internal Identifier (*disseminatorID*)** – the identifier for a `disseminator` within a digital object. This is an internal component identifier that is unique within the object, and not considered a public identifier.

2.2.4.2 **Disseminator Type Identifier (*bdefPID*) –** The unique identifier (PID) of the `behavior definition` object to which the `disseminator` subscribes. The PID can serve as a type identifier for the `disseminator`, instead of establishing a separate type taxonomy. Ultimately, there may be a registry of official Fedora `behavior definitions` keyed by their PIDs. Also, since a digital object must not have multiple `disseminators` of the same `disseminator` Type Identifier, the `disseminator` Type Identifier can be considered unique within the digital object, and it qualifies as a unique public identifier for a `disseminator`. Note that recording the PID of the `behavior definition` object as the `disseminator` Type Identifier is redundant since it is also recorded in the `behavior definition` part of the `disseminator` (see below).

2.2.4.3 **Disseminator Label (*dissLabel*) –** a human readable label that describes the purpose of the `disseminator`.

2.2.4.4 **Created Date (*dissCreateDT*) –** the date and time that the `disseminator` was created in the repository.

2.2.4.5 **Datastream Binding Map (*fedoraBindMap*) –** A `disseminator` must store information that provides a mapping of `datastreams` (content) to a particular `behavior mechanisms` that processes those `datastreams` in producing disseminations. (In Fedora, `behavior mechanisms` typically operate on `datastreams` in the object, ultimately producing disseminations by performing some function on those `datastreams`.) The Datastream Binding Map conforms to a set of rules specified by a Datastream Binding Specification found within a `behavior mechanism` object (see Section 2.3). Thus, each `disseminator` stores a Datastream Binding Map which is a mapping of `datastream` identifiers to abstract binding keys that are defined by a particular `behavior mechanism`. In essence, this identifies particular `datastreams` as the input data to a mechanism. For example, if a mechanism implements a set of methods (e.g., GetThumbNail, GetFullImage) to provide disseminations of images, then the mechanism that implements these methods only knows how to deal with specific types of image `datastreams` (e.g., the mechanism may only know how to process a MRSid-encoded image `datastream`). The `behavior mechanism` object that stores this mechanism must publicize the fact that it only processes MRSid and also specify a binding key that can be used to identify an appropriate `datastream` in the object. When a `disseminator` names a `behavior mechanism`, it also records an association between the mechanism's binding key and the qualifying `datastream` in the object. The attributes of a Datastream Binding Map are:

- **Internal Identifier (*dsBinderMapID*) –** an unique internal identifier for the Datastream Binding Map.
- **Datastream Binding Set –** one or more *associations* of a `datastream` to a data input role that is defined by a Binding Specification in a `behavior mechanism` object.
    - **Datastream Binding Key (*dsBindKey*) –** a label that identifies a role that a `datastream` plays in the context of a particular `behavior mechanism`. The mechanism uses this key to identify the `datastream` at runtime. There can be many different binding keys
    - **Datastream Identifier (*datastreamID*) –** the internal identifier of a `datastream` in the digital object that is to be associated with the particular binding key
    - **Datastream Binding Sequence (*dsBindSeq*) –** the order in which to present or process multiple `datastreams` with the same binding key

2.2.4.6 **Behavior Definition –** the definition of the `behavior definition` which the `disseminator` represents. A `behavior definition` (and its methods) are assigned to the `disseminator` *by reference* to the PID of a particular `behavior definition` object. The Fedora Repository System resolves all references at runtime. See Section 2.3 for more details.

- **Internal Identifier (optional) –** an internal identifier for the `behavior definition` component of a `disseminator`
- **Locator Type (*bdefLocType*) –** default is URN (which is the format of a PID for a `behavior definition` object)
- **Location (*bdefPID*) –** the PID for a `behavior definition` object

2.2.4.7 **Behavior Mechanism –** the definition of the `behavior mechanism` that the `disseminator` uses as the implementation of the chosen `behavior definition`. (We note that there are many possible mechanisms which can implement the same `behavior definition`.) A `behavior mechanism` (and its method implementations) are assigned to the `disseminator` *by reference* to the PID of a particular `behavior mechanism` object. The Fedora Repository System resolves all references at runtime. See Section 2.3 for more details.

- **Internal Identifier (optional) –** an internal identifier for the `behavior mechanism` component of a `disseminator`
- **Locator Type (*bmechLocType*) –** default is URN (which is the format of a PID for a `behavior mechanism` object)
- **Location (*bMechPID*) –** the PID for a `behavior mechanism` object

2.3 Special Digital Objects

There are three distinct types of Fedora digital objects: (1) *data objects*, (2) *behavior definition objects*, and (3) *behavior mechanism objects*. As their name implies, data objects are Fedora digital objects that represent content (i.e., data and metadata) and a set of associated behaviors or services that can be applied to that content. Data objects comprise the bulk of a repository. The other two types of objects are also Fedora digital objects, but play a special role in the architecture.

A `behavior definition` object is a special type of Fedora digital object that models a set of abstract `behavior definitions`. These `behavior definitions` consist of a set of methods for transforming or presenting the contents of a digital object. A `behavior mechanism` object is a special type of Fedora digital object that models a set of external behavior implementations for those behaviors defined by the `behavior definition` object. The `behavior mechanism` object also contains binding relationships between a `behavior mechanism` and `datastreams` in the data object. The abstract `behavior definitions` and binding information are expressed using a WSDL definition.

In the example in Figure 2, a digital object has a Watermarker `disseminator` that can dynamically apply a watermark to an image. The `disseminator` has two notable attributes: a `behavior definition` identifier and a `behavior mechanism` identifier. These identifiers are actually *persistent identifiers to other Fedora digital objects*. These are special digital objects that are *surrogates* for external services, for example, a service for obtaining images at different resolutions. A `behavior definition` object contains a special `datastream` whose content is a WSDL definition of abstract methods for watermarked images (e.g., *getImage*). A `behavior mechanism` object contains a special `datastream` that is a WSDL definition describing the run-time bindings to an external service for these methods (operations). Service bindings can be via HTTP GET/POST or SOAP. See Appendix B and C for examples of `behavior definition` and `behavior mechanism` objects.

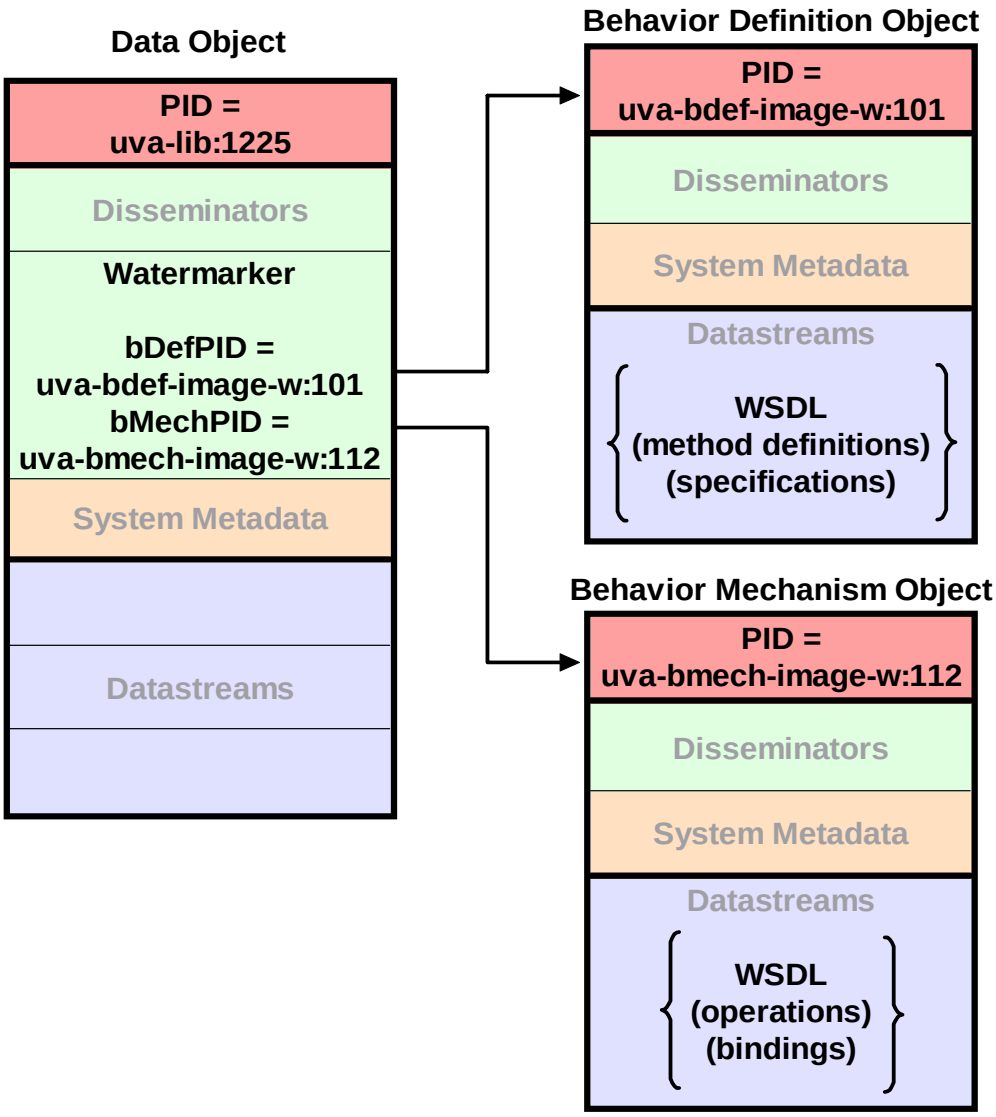

Figure 2. Behavior Definition and Behavior Mechanism Objects

2.4 XML Encoding of Digital Objects

The internal structure of each Fedora digital object is represented as XML encoded text. The XML schema used to encode the Fedora digital object model is an extension of the Metadata Transmission and Encoding Standard (METS, see http://www.loc.gov/mets), a Digital Library Federation initiative focused on developing an XML format for encoding metadata necessary to manage digital library objects within a repository and to facilitate exchange of such objects among repositories. The decision to use METS is based on several factors: (1) METS is expressed using the XML Schema language which enables the expression of data types and constraints, (2) METS is freely available from the METS website, (3) METS represents a standard maintained by the Network Development and MARC

Standards Office of the Library of Congress, and (4) the METS schema provides the majority of the functionality required to encode Fedora digital objects.

There are certain functions required by the Fedora repository to manage digital objects like the state of an object that currently have no representation in the METS schema. To support these features we have extended the METS schema by adding additional attributes on certain elements to accommodate the additional functionality. The table in Section 7.2 shows the translation of the major Fedora digital object components to their equivalent METS entities.

See Appendix A for a complete example of a Fedora data object encoded using the METS schema. Examples of METS encoded `behavior definition` and `behavior mechanism` objects are found in Appendix B and C.

3.0 Repository Architecture

3.1 **Repository Architecture Summary**

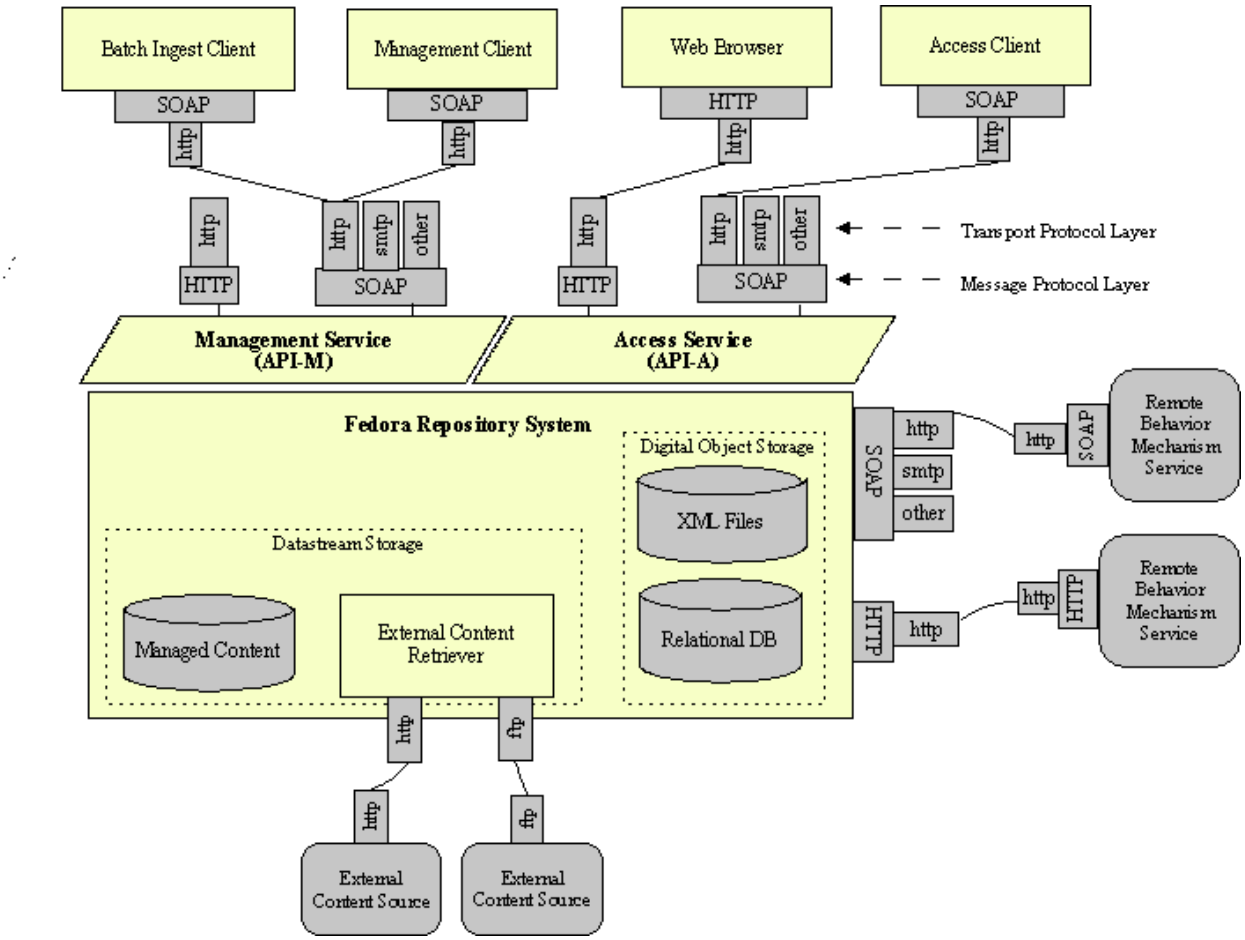

Figure 3. Macro-Level System Diagram

The Fedora Repository architecture provides access and management services for digital objects. The Repository architecture is built on Web services technology. A Web service can be defined as a distributed application that runs over the internet. Web services are typically configured to use HTTP as a transport protocol for sending messages between different parts of the distributed application. The use of XML is a key feature of such applications, serving as a standard for encoding structured messages that are sent to and from the distributed applications. The Web Services Description Language (WSDL) is an XML format for describing services as a set of abstract operations that are realized as a set of *endpoints* that receive and respond to structured messages. Each endpoint communicates over a specific network protocol and uses a specific message format.

A Fedora Repository is exposed as two related Web services, the Fedora Management service and the Fedora Access service. As depicted in the Macro System Diagram (Fig. 3), these services are the public entry points into the Repository (i.e., from a client perspective). The core repository system resides beneath the web service layer and is the internal implementation of the publicly exposed web services. As seen in the diagram, the internal repository system also uses Web services technology to facilitate backend communication with supporting services that are distributed on the internet, including other Fedora repositories. Clients interact with the Fedora Repository via the public web services.

3.2 Fedora Management Service

The Fedora Management service defines an open interface for administering the repository, including creating, modifying, and deleting digital objects, or components within digital objects. The Management service interacts with the underlying repository system to read content from and write content to the digital object and `datastream` storage areas. The Management service exposes a set of operations that enable a client to view and manipulate digital objects from an abstract perspective, meaning that a client does not need to know anything about underlying storage formats, storage media, or storage management schemes for objects. Also, the underlying repository system handles the details of storing `datastream` content within the repository, as well as mediating connectivity for `datastreams` that reference external content.

3.3 Fedora Access Service

The Fedora Access service defines an open interface for accessing digital objects. The access operations include methods to do reflection on a digital object (i.e., to discover the kinds of disseminations that are available on the object), and to request disseminations.

The major function of the Fedora Access service is to fulfill a client's request for dissemination. To support disseminations, the underlying repository system must evaluate the behavior associations specified in a digital object, and figure out how to dispatch a service request to a supporting service with which the digital object associates. The supporting service may be internal to the repository system, or it may

be an external web services that the repository must call upon. The underlying repository system facilitates all external service bindings on behalf of the client, simply returning a dissemination result via the Access service layer.

3.4 Client Connectivity

Clients interact with Fedora via the publicly exposed Access and Management services. Note that a client can be a web browser, a web application with embedded Fedora service requests, or any custom client application that is Fedora-aware. Public interface definitions for the Fedora Access and Management services are published in XML using WSDL. Each service definition provides a description of the operations available for the particular service, including information on how to connect to the service to invoke operations. The Macro System Diagram (Fig. 3) depicts various connectivity scenarios for clients interacting with the Fedora Access and Management services. In brief, these connectivity scenarios are:

- Clients make service requests that are encoded in the HTTP message protocol format (GET/PUT) and transmitted to Fedora over the HTTP transport protocol
- Clients make service requests that are encoded using the Simple Object Access Protocol (SOAP) and transmitted to Fedora over the HTTP transport protocol
- Clients make service requests that are encoded using SOAP and transmitted to Fedora over an alternative transport protocol.

Public APIs for Fedora Services

4.0 Management Service (API-M)

4.1 **API-M Definition** - the WSDL description of API-M can be obtained at http://www.fedora.info/documents/Fedora-API-M.wsdl.

4.1.1 Object Management Methods

4.1.1.1 Object Creation - Object creation occurs when either 1) an object is built up, component by component, via API-M requests, or 2) a new digital object is created in the repository by accepting an xml document, encoded in the METS format.

4.1.1.1.1 **CreateObject()** – creates a new, empty digital object in the repository. The object's initial state will be N. The repository will generate and return a new PID for the object resulting from this request. The PID will have the namespace of the repository.

Parameters: none

Returns: xsd:string - the PID of the newly created object.

Example:
CreateObject();

4.1.1.1.2 **IngestObject(xsd:base64Binary METSXML)** – creates a new digital object in the repository, given the data in the provided METS document. The object's initial state will be N. If the METS document does not specify a PID in the OBJID attribute of the root element, the repository will generate and return a new PID for the object resulting from this request. That PID will have the namespace of the repository. If the METS document specifies a PID, it will be assigned to the digital object provided that 1) it conforms to the Fedora PID Syntax, 2) it does not specify the namespace of the repository, and 3) it does not collide with an existing PID of an object in the repository.

Parameters:
a. METSXML: the digital object in METS format.

Returns: xsd:string - the PID of the newly created object.

Example:
IngestObject(metsDocument);

4.1.1.2 **GetObjectXML(xsd:string PID) –** provides the XML portion of the entire METS-encoded digital object for external use (viewing, editing, moving to another repository). XML metadata datastreams will be included inline, content datastreams will not be included, and external datastreams will be referenced by url.

Parameters:
a. PID: the PID of the object.

Returns: xsd:base64Binary – the digital object in METS format.

Example:
GetObjectXML("uva-edu:123");

4.1.1.3 **ExportObject(xsd:string PID) –** provides the entire METS-encoded digital object for external use (viewing, editing, moving to another repository). XML metadata datastreams will be included inline, content datastreams will be included inline (base64-encoded), and external datastreams will be referenced by url.

Parameters:
a. PID: the PID of the object.

Returns: xsd:base64Binary – the digital object in METS format.

Examples:
ExportObject("uva-edu:123");

4.1.1.4 **WithdrawObject(xsd:string PID) –** withdraws an object, meaning it is made inactive, and inaccessible to everyone except the repository administrator.

Parameters:
a. PID: the PID of the object.

Returns: nothing

Examples:
WithdrawObject("uva-edu:123");

4.1.1.5 **DeleteObject(xsd:string PID) –** deletes an object in the repository, meaning that the object is marked as deleted, but not physically removed from the repository.

Parameters:

a. PID: the PID of the object.

Returns: nothing

Examples:
DeleteObject("uva-edu:123");

4.1.1.6 **PurgeObject(xsd:string PID) –** physically removes an object from the repository.

Parameters:
b. PID: the PID of the object.

Returns: nothing

Examples:
PurgeObject("uva-edu:123");

4.1.1.7 **ObtainLock(xsd:string PID) –** locks an object for writing. Upon obtaining a lock on an object, a user may execute write methods on the object. Objects are unlocked via the ReleaseLock method (see below).

Parameters:
a. PID: the PID of the object.

Returns: nothing

Example:
ObtainLock("uva-edu:123");

4.1.1.8 **ReleaseLock(xsd:string PID, xsd:string logMessage) –** unlocks an object for writing.

Parameters:
a. PID: the PID of the object.
b. logMessage: a log message for the change.

Returns: nothing

Example:
ReleaseLock("uva-edu:123");

4.1.1.9 **GetLockingUser(xsd:string PID) –** tells who has a write lock on an object.

Parameters:
a. PID: the PID of the object.

Returns: xsd:string – the ID of the user.

Example:
GetLockingUser("uva-edu:123");

4.1.1.10 **GetObjectState(xsd:string PID) – reports the state of the object, indicating among other things whether the object is active or deleted.**

Parameters:
b. PID: the PID of the object.

Returns: xsd:string – the code for the object state.

Example: GetObjectState("uva-edu:123");

4.1.1.11 **ListObjectPIDs(xsd:string state) –** provides an enumeration of the PIDs of the digital objects stored in the repository.

Parameters:
a. **state:** the state of objects to list. This can be 'A', 'N', 'W', 'D', or null. If null, all objects will be listed regardless of state.

Returns: fedora:ArrayOfString – the PIDs

Examples:
ListObjectPIDs('A');

4.1.2 Component Management Methods

These methods are used to create and maintain objects within the repository sub-system. The repository subsystem exposes this interface and when clients use it, there is assurance that the repository sub-system is taking full control of the individual operations that can be performed on an object. This is in contrast to what happens when the IngestObject method is used, where the repository accepts an XML-encoded file that represents a digital object. In this case, the XML editing has taken place outside the context of the repository. Although the repository can run XML validation on the file before accepting it, there is a limited amount of control that the repository can take over how the object is encoded. The Advanced Object Management methods are used to create objects when there is no offline XML editing available, or it is not desirable for other reasons. Once an object has been stored in the repository, the Advanced Object Management methods offer a trusted means of maintaining the object over time. When clients use these methods to create or maintain an object, the repository can have more control over the encoding, versioning, and change tracking. These methods enable the repository to maintain tight control over objects. Instead of offline clients modifying the

digital object XML files, the repository makes these changes in response to client requests.

4.1.2.1 **AddDatastreamExternal(xsd:string PID, xsd:string dsLabel, xsd:string dsLocation) –** creates a `datastream` of the form Referenced External Content in the digital object. The mime type is determined by running an HTTP GET request against the provided url, and taking the value of the "Content-Type" response header.

Parameters:

a. **PID:** the PID of the object.
b. **dsLabel:** a human readable label describing the `datastream`.
c. **dsLocation:** a valid URL that resolves to content intended to serve as a `datastream` in the digital object.

Returns: xsd:string – the ID of the `datastream`

Examples:

AddDatastreamExternal("uva-edu:123", "imagexyz", "http://foo.edu/img/xyz.gif");
AddDatastreamExternal("uva-edu:456", "imageabc", "http://foo.edu/img/abc.gif");

4.1.2.2 **AddDatastreamManagedContent(xsd:string PID, xsd:string dsLabel, xsd:string MIMEType, xsd:base64Binary dsContent) –** Creates a datastream of the form Repository-Managed Content. The repository copies the contents of the datastream (i.e., the parameter named dsContent) to a location under the control of the repository.

Parameters:

a. **PID:** the PID of the object.
b. **dsLabel:** a human readable label describing the `datastream`.
c. **MIMEType:** the mime type of the data (for example, "image/gif")
d. **dsContent:** a byte stream of content intended to be stored in the repository as a `datastream`.

Returns: xsd:string – the ID of the `datastream`

Examples:

AddDatastreamManagedContent("uva-edu:123", "image123", "image/gif", imageBytes);
AddDatastreamManagedContent("uva-edu:456", "image456", "image/gif", imageBytes);

4.1.2.3 **AddDatastreamXMLMetadata(xsd:string PID, xsd:string dsLabel, xsd:string MDType, xsd:base64Binary dsInlineMetadata) –** creates a Implementer-Defined XML Metadata

`datastream` of the form Repository-Managed Content in the digital object from an XML byte array.

Parameters:

a. **PID:** the Persistent Identifier of the parent digital object in string format.
b. **dsLabel:** a human readable label describing the `datastream`.
c. **MDType:** a string that indicates the general type of Implementer-Defined XML Metadata the `datastream` represents. In Fedora, valid metadata types conform to the METS taxonomy of metadata. Thus, valid values for MDType are: ***digiprov***, ***rights***, ***technical***, ***source***, and ***descriptive.*** Refer to the METS documentation for more description of these types of metadata.
d. **dsInlineMetadata:** a bytestream of XML-encoded metadata intended to serve as a Implementer-Defined XML Metadata `datastream` in the digital object. The xml stream must contain well-formed XML.

Returns: xsd:string – the ID of the `datastream`

Examples:

AddDatastreamXMLMetadata("uva-edu:123", “descmeta123”, "descriptive", xmlbytestream);
AddDatastreamXMLMetadata("uva-edu:123", “rightsmeta123”, "rights" , xmlbytestream);

4.1.2.4 **AddDisseminator(xsd:string PID, xsd:string bMechPID, xsd:string dissLabel, fedora:DatastreamBindingMap dsBindMap)** – adds a disseminator to the object.

Input Parameters:

a. **PID:** the Persistent Identifier of the parent digital object is string format.
b. **bMechPID:** the PID of an existing `behavior mechanism` object that represents the implementation of the behaviors named in the `behavior definition` to which the `disseminator` subscribes. Among other things, the `behavior mechanism` object contains *service binding metadata* that describes how a set of methods can be invoked and executed. By associating the PID of a `behavior mechanism` object with a `disseminator`, a client essentially says that the `disseminator` represents the implementation of a set of abstract methods, and that, ultimately, the `disseminator` provides disseminations of content via these methods. The Mechanism Object also contains data binding metadata that defines a set of *data binding keys* that can be used to associate `datastreams` with the Mechanism. The data binding metadata also specifies some constraints on the kind of `datastreams` that can be used by the mechanism, such as appropriate MIME types. (See Appendix A for details.)
c. **dissLabel:** a brief description for the `disseminator` in string format. The descriptor should be meaningful to humans and should connote the purpose of the `disseminator`.
d. **dsBindMap:** an encoded structure that represents the relationships between `datastreams` and the `behavior mechanism`'s data input requirements. Specifically the dsBindMap associates *data binding keys* (defined by Mechanism) with *datastream identifiers*, thus establishing that particular `datastreams` play particular roles for the mechanism at runtime. Each key-to-datastream association may also be assigned a sequence number, if the Mechanism so requires. (See Appendix A for details.) A view of the existing `datastreams` of the target digital object will be necessary in order to form the dsBindMap. Clients that implement wizard-like interfaces for building digital objects likely need access to metadata that resides in the `behavior definition` and `behavior mechanism` objects to help the client build a `datastream` binding map.

Returns: nothing

Example:

AddDisseminator("uva-edu:123", "uva-edu-bmech:123", "UVA Image Disseminator", bindingMap);

4.1.2.5 **ModifyDatastreamExternal(xsd:string PID, xsd:string datastreamID, xsd:string dsLabel, xsd:string url)** – modifies a `datastream` of the form Referenced External Content by replacing currently stored pointer to content with a new pointer to remotely referenced content supplied by the user.

Parameters:

a. **PID:** the Persistent Identifier of the parent digital object is string format.
b. **datastreamID:** the internal identifier for the `datastream` to be modified.
c. **dsLabel:** a local identifier for the `datastream`.
d. **url:** a url pointing to content intended to *replace* the content currently stored as a `datastream` in the digital object.

Returns: nothing

Example:

ModifyDatastreamExternal("uva-edu:123", "DS1" , "imagexyz2", "http://foo.edu/img/xyz2.gif");

4.1.2.6 **ModifyDatastreamManagedContent(xsd:string PID, xsd:string datastreamID, xsd:string dsLabel, xsd:string MIMEType, xsd:base64Binary dsContent)** – modifies a `datastream` of the form Repository-Managed Content by replacing the content with the content from the bytestream supplied by the user.– modifies a `datastream` of the form Repository-Managed Content by replacing the content with the content from the bytestream supplied by the user.

Parameters:

a. **PID:** the Persistent Identifier of the parent digital object is string format.
b. **datastreamID:** the internal identifier for the `datastream` to be modified.
c. **dsLabel:** a local identifier for the `datastream`.
d. **MIMEType:** the mime type of the content (for example, “image/gif”)
e. **dsContent:** a byte stream (Base64) of content intended to *replace* the content currently stored as a `datastream` in the digital object.

Returns: nothing

Example:
ModifyDatastreamManagedContent("uva-lib:123", "DS1" , “imagexyz2”, “image/gif”, somebytestream);

4.1.2.7 **ModifyDatastreamXMLMetadata(xsd:string PID, xsd:string datastreamID, xsd:string dsLabel, xsd:string MDType, xsd:base64Binary xmlstream) –** modifies the contents of a `datastream` of the form Implementer-Defined XML Metadata by replacing the inline content in the digital object with the content from the bytestream supplied by the user.

Parameters:

a. PID: the Persistent Identifier of the parent digital object is string format.
b. datastreamID: the internal identifier for the Inline Metdata `datastream` to be modified.
c. dsLabel: a local identifier for the `datastream`.
e. MDType: a string that indicates the general type of Implementer-Defined XML Metadata the `datastream` represents. In Fedora, valid metadata types conform to the METS taxonomy of metadata. Thus, valid values for MDType are: ***digiprov***, ***rights***, ***technical***, ***source***, and ***descriptive***. Refer to the METS documentation for more description of these types of metadata.
d. xmlstream: a byte stream of XML-encoded metadata intended to replace an existing Implementer-Defined XML Metadata `datastream` in the digital object. The xml stream must contain well-formed XML.

Returns: nothing

Example:
ModifyDatastreamXMLMetadata("uva-lib:123", "DS1" , “metaxyz”, "descriptive", xmlbytestream");

4.1.2.8 **ModifyDisseminator(xsd:string PID, xsd:string disseminatorID, xsd:string bMechPID, xsd:string dissLabel, fedora:DatastreamBindingMap dsBindMap) –**

Parameters:

a. **PID:** the PID of the object.
b. **DisseminatorID:** the internal identifier for the `disseminator` to be modified. In METS XML, this is the ID attribute on a <behaviorSec> element.
c. **bMechPID:** if the intent is to modify the `behavior mechanism` for the `disseminator`, then bMechPID should be the PID of a new `behavior mechanism` object, or the PID of the object that represents a new *edition* of the currently associated `behavior mechanism` object. If no change is intended to bMechPID, then the value of this parameter should be the PID of the `behavior mechanism` object that is already associated with the `disseminator`.
d. **dissLabel:** a brief description for the `disseminator` in string format. The descriptor should be meaningful to humans and should connote the purpose of the `disseminator`.
e. **dsBindMap:** a `datastream` binding map must be in sync with whatever `behavior mechanism` is associated with the `disseminator`. Thus, if any change is made to bMechPID, the dsBindMap must also be updated. For example, if a different `behavior mechanism` object is being associated with the `disseminator`, then the dsBindMap must be updated to reflect the binding specification requirements of the new mechanism. A view of the existing `datastreams` of the target digital object will be necessary in order to form the dsBindMap. Clients that implement wizard-like interfaces for building digital objects likely need access to metadata that resides in the `behavior definition` and `behavior mechanism` objects to help the client build a `datastream` binding map.

Returns: nothing

Examples:

ModifyDisseminator("uva-edu:123", "DISS1", "uva-edu-bmech:123", "UVA Image Disseminator", xmlfragment);
ModifyDisseminator("uva-edu:123", "DISS1", "uva-edu-NEWmech:123", "UVA Image Disseminator", NEWxmlfragment);

4.1.2.9 **WithdrawDatastream(xsd:string PID, xsd:string datastreamID)** – withdraws a `datastream`, meaning it is made inactive, and inaccessible to everyone except the repository administrator.

Parameters:

a. **PID:** the PID of the object.

b. datastreamID: the internal identifier for the `datastream` to be withdrawn.

Returns: nothing

Example:
WithdrawDatastream("uva-edu:123", "DS1");

4.1.2.10 **DeleteDatastream(xsd:string PID, xsd:string datastreamID) –** used to remove a `datastream` from a digital object. This request marks a `datastream` as deleted without physically removing it from the digital object. Deleted `datastreams` can be physically removed using the PurgeDatastream method.

Parameters:
a. PID: the PID of the object.
b. datastreamID: the internal identifier for the `datastream` (Referenced External Content, Repository-Managed Content, or Implementer-Defined XML Metadata) to be flagged for deletion.

Returns: nothing

Example:
DeleteDatastream("uva-edu:123", "DS1");

4.1.2.11 **PurgeDatastream(xsd:string PID, xsd:string datastreamID) –** used to physically remove a `datastream` from a digital object.

Parameters:
c. PID: the PID of the object.
d. datastreamID: the internal identifier for the `datastream` (Referenced External Content, Repository-Managed Content, or Implementer-Defined XML Metadata) to be flagged for deletion.

Returns: nothing

Example:
PurgeDatastream("uva-edu:123", "DS1");

4.1.2.12 **WithdrawDisseminator(xsd:string PID, xsd:string disseminatorID) –** withdraws a `disseminator`, meaning it is made inactive, and inaccessible to everyone except the repository administrator.

Parameters:
a. PID: the PID of the object.

b. disseminatorID: the internal identifier for the `disseminator` to be withdrawn. In METS XML, this is the ID attribute on a <behaviorSec> element.

Returns: nothing

Example:
WithdrawDisseminator("uva-edu:123", "DISS1");

4.1.2.13 **DeleteDisseminator(xsd:string PID, xsd:string disseminatorID) –** used to remove a `disseminator` from a digital object. This request marks a `disseminator` as deleted without physically removing it from the digital object. Deleted `disseminators` can be physically removed using the PurgeDisseminator method.

Parameters:
a. PID: the PID of the object.
b. disseminatorID: the internal identifier for the `disseminator` to be modified. In METS XML, this is the ID attribute on a <behaviorSec> element.

Returns: nothing

Example:
DeleteDisseminator("uva-edu:123", "DISS1");

4.1.2.14 **PurgeDisseminator(xsd:string PID, xsd:string disseminatorID) –** used to physically remove a `disseminator` from a digital object.

Parameters:
e. PID: the PID of the object.
f. disseminatorID: the internal identifier for the `disseminator`. In METS XML, this is the ID attribute on a <behaviorSec> element.

Returns: nothing

Example:
PurgeDisseminator("uva-edu:123", "DISS1");

4.1.2.15 **GetDatastream(xsd:string PID, xsd:string datastreamID, xsd:dateTime asOfDateTime) –** used to obtain a specific `datastream` that exists within a given digital object (i.e., Implementer-Defined XML Metadata, Referenced External Content, and Repository-Managed Content `datastreams`).

Parameters:

a. **PID:** the PID of the object.
b. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:Datastream – an XML-encoded data structure that represents a Datastream (where a `datastream` includes datastreamID, mimeType, other attributes, and either a URL or a base64-encoded stream).

Example:

GetDatastream("uva-edu:123", “DS1”, null);

4.1.2.16 **GetDatastreams(xsd:string PID, xsd:dateTime asOfDateTime)** – used to obtain all the `datastreams` that exist within a given digital object, Implementer-Defined XML Metadata, Referenced External Content, and Repository-Managed Content `datastreams`.

Parameters:

a. **PID:** the PID of the object.
b. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:ArrayOfDatastream – an XML-encoded data structure that represents an array of Datastreams (where a `datastream` includes datastreamID, mimeType, other attributes, and either a URL or a base64-encoded stream).

Example:

GetDatastreams("uva-edu:123", null);

4.1.2.17 **GetDisseminator(xsd:string PID, xsd:string disseminatorID, xsd:dateTime asOfDateTime)** – used to obtain a specific `disseminator` that exists within a given digital object.

Parameters:

a. **PID:** the PID of the object.
b. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:Disseminator – an XML-encoded data structure that represents a `disseminator` (where a `disseminator` includes disseminatorID, bdefPID, bMechPID, descriptor, and dsBindMap).

Example:
GetDisseminator("uva-edu:123", "DISS1", null);

4.1.2.18 **GetDisseminators(xsd:string PID, xsd:dateTime asOfDateTime) –** used to obtain all the `disseminators` that exist within a given digital object.

Parameters:
- **a. PID:** the PID of the object.
- **b. asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:ArrayOfDisseminator – an XML-encoded data structure that represents an array of `disseminators` (where a `disseminator` includes disseminatorID, bdefPID, bMechPID, descriptor, and dsBindMap).

Example:
GetDisseminators("uva-edu:123", null);

4.1.2.19 **ListDatastreamIDs(xsd:string PID, xsd:string state xsd:dateTime asOfDateTime) –** provides a list of ids of datastreams in the object.

Parameters:
- a. **PID:** the PID of the object
- **b. state:** the state of the IDs to list. This can be 'A', 'W', 'D', 'B', or null. If null, all datastream ids are returned regardless of state.
- **c. asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:ArrayOfString – the datastream IDs.

Example:
ListDatastreamIDs("uva-edu:123", null, null)

4.1.2.20 **ListDisseminatorIDs(xsd:string PID, xsd:string state, xsd:dateTime asOfDateTime) –** provides a list of ids of disseminators in the object.

Parameters:
- a. **PID:** the PID of the object
- **b. state:** the state of the IDs to list. This can be 'A', 'W', 'D', 'B', or null. If null, all disseminator ids are returned regardless of state.

c. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:ArrayOfString – the disseminator IDs.

Example:
ListDisseminatorIDs("uva-edu:123", null, null)

5.0 Access Service (API-A)

5.1 **API-A Definition –** the WSDL description for API-A can be found at http://www.fedora.info/documents/Fedora-API-A.wsdl.

5.1.1 **Access Methods**

5.1.1.1 **GetBehaviorDefinitions(xsd:string PID, xsd:dateTime asOfDateTime) –** used to obtain a list of the `behavior definitions` that are supported by the digital object. The PID of a `behavior definition` object identifies a `behavior definition` type. The GetBehaviorDefinitions method lists the set of PIDs that represent all `behavior definition` types found on all the `disseminators` of the digital object.

Parameters:
a. **PID:** the PID of the object.
b. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:ArrayOfString – the PIDs of the behavior definition objects.

Example:
GetBehaviorDefinitions("uva-edu:123", null);

5.1.1.2 **GetBehaviorMethods(xsd:string PID, xsd:string bDefPID, xsd:dateTime asOfDateTime) –** used to obtain a list of method descriptions for a given `behavior definition` type associated with the digital object. The PID of a `behavior definition` object identifies a particular `behavior definition` type.

Parameters:

a. **PID:** the PID of the object.
b. **BDefPID:** the `behavior definition` type identifier which is the PID of a given `behavior definition` object.
c. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: wsdl:definitions – pre-bound wsdl that the caller can use to make a call back to the repository for each method (GetDissemination requests).

Example:

GetBehaviorMethods("uva-edu:123", "uva-edu-bdef:123", null);

5.1.1.3 **GetBehaviorMethodsAsWSDL(xsd:string PID, xsd:string bDefPID, xsd:dateTime asOfDateTime) –** used to obtain the WSDL description of methods for a particular `behavior definition` object.

Parameters:

a. **PID:** the PID of the object.
b. **BDefPID:** the `behavior definition` type identifier which is the PID of a given `behavior definition` object.
c. **asOfDateTime:** the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: wsdl:definitions – pre-bound wsdl that the caller can use to make a call back to the repository for each method (GetDissemination requests).

Example:

GetBehaviorMethods("uva-edu:123", "uva-edu-bdef:123", null);

5.1.1.4 **GetDissemination(xsd:string PID, xsd:string bDefPID, xsd:string methodName, fedora:ArrayOfProperty parameters, xsd:dateTime asOfDateTime) –** used to obtain a particular view of content from the digital object (i.e., a dissemination of content). Essentially, the GetDissemination request encapsulates a behavior method that is defined by a particular `behavior definition` type. The GetDissemination request hides from the client all implementation details of how the behavior method is fulfilled (i.e., what `behavior mechanism` performs the work, and how the request is invoked upon the `behavior mechanism`). A client talks in terms of `behavior`

`definition` types when making a GetDissemination request, thus the client does not need to know anything about `behavior mechanisms`. The repository figures out at run time what `behavior mechanism` is associated with the digital object, and how to bind to that mechanism to fulfill the GetDissemination request.

Parameters:

a. PID: the PID of the object.

b. **bDefPID:** the `behavior definition` type identifier which is the PID of a given `behavior definition` object.

c. methodName: the name of a method defined in the `behavior definition` type represented by bdefPID. Access to the method definitions of a `behavior definition` will be necessary so the client can incorporate a valid method name in the GetDissemination request. The client can obtain this information by invoking the GetBehaviorDefType and GetBehaviorDefMethod requests.

d. parameters: an XML-encoded fragment that represents an array of properties that should be passed as name-value pairs to the named method. If the method does not take any parameters, `parameters` is null.

e. asOfDateTime: the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:MIMETypedStream – the result of the dissemination request. The actual returned data type is determined by the behavior mechanism in question.

Example:

GetDissemination("uva-edu:123", "uva-edu-bdef:123", "GetFoo", null, null);

5.1.1.5 **GetObjectMethods(xsd:string PID, xsd:dateTime asOfDateTime) –** used to obtain all method descriptions for all behavior definitions associated with a a particular digital object. Essentially, GetObjectMethods provides a way of reflecting on a digital object and its associated behavior definitions to discover all methods available to the object.

Parameters:

a. PID: the PID of the object.

b. asOfDateTime: the desired dated view of the object. If null, this method will run with the most current information available in the object.

Returns: fedora:ObjectMethodsDef – a data structure containing all methods and method parameters associated with the specified digital object.

Example:

GetObjectMethods("uva-edu:123", null);

System Implementation

6.0 Summary

The Fedora Repository System is implemented as a multi-tiered application. At the top is the web service layer. At this layer, the Access and Management service definitions (API-M and API-A) represent the public view of the Fedora Repository System. These two services definitions are externally expressed as WSDL. The core repository system is implemented as a set of collaborating subsystems written in Java using Sun's JDK 1.4. The Access Subsystem is a Java implementation of API-A. The Management Subsystem is a Java implementation of API-M. Within the core repository system, the Security Subsystem provides for access control, specifically policy management and enforcement. At the lowest layer of the repository is the Storage Subsystem, which is responsible for all read and write operations upon digital objects and `datastreams`.

In Phase I of the project, the Fedora Access web service is exposed over HTTP via SOAP bindings that connect to a SOAP-enabled service running on Apache Axis with Tomcat. Additionally, the GetDissemination operation of API-A is redundantly exposed via an HTTP GET service binding that connects to a Java servlet running on Apache Tomcat. The web service service layer interfaces with the core classes of the Access subsystem. The Management Web service is exposed via SOAP bindings that connect to a SOAP-enabled service running on Apache Axis with Tomcat. The Web service layer interfaces with the core classes of the Management subsystem. The development team will assess the strengths and weaknesses of each connectivity scenario during the testing period, and make appropriate modifications in Phase II.

Figure 4 depicts the full Fedora system implementation. The core subsystems are below the web services layer (API-M and API-A). The Phase I implementation includes the Access, Management, Security, and Storage subsystems. Connectivity to supporting web services is concentrated in the Storage Subsystem, as depicted the lower right-hand side of the diagram. Connectivity to remote content (i.e., for External Referenced Content `datastreams)` is depicted at the bottom of the Storage subsystem. In Phase I, the Security subsystem provides basic authentication and enforce a repository-wide access control policy based on user identity. Fine-grained object-level policy enforcement will be included in Phase II (e.g., unique policies for disseminations on digital objects). Other Phase II and III deliverables are marked in the diagram with diagonal fill lines. These features include: a cache for External Referenced Content `datastreams`; a cache for previously executed disseminations; alternative access clients ; alternative connectivity scenarios to external `behavior mechanism` services; alternative connectivity scenarios for External Referenced Content `datastreams;` and improved security for back-end connectivity to `behavior mechanism` services and remote content.

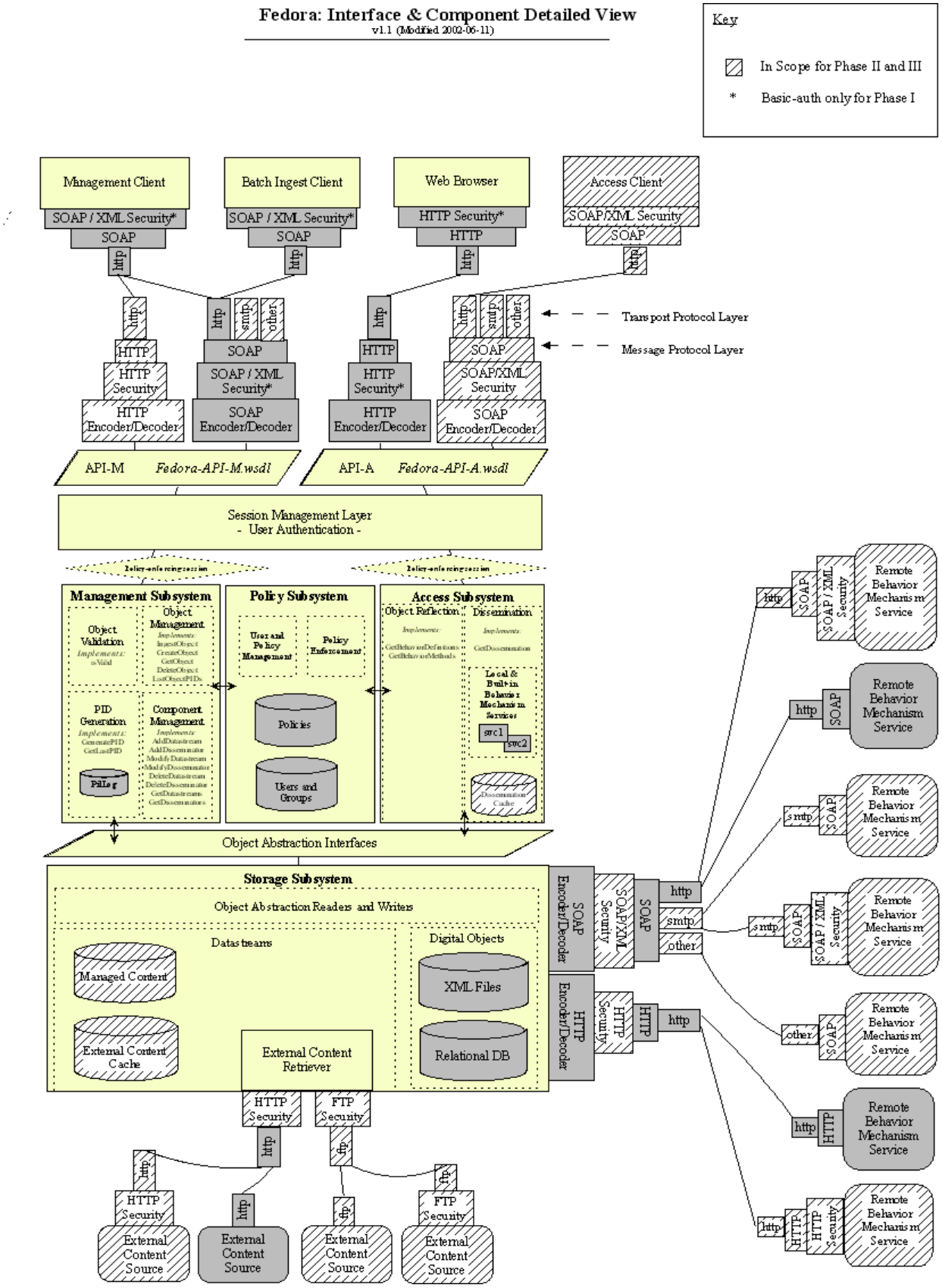

Figure 4. Detailed System Diagram

7.0 Digital Objects

7.1 XML Encoding using the METS Schema

Fedora digital objects are encoded in XML using the Metadata Transmission and Encoding Standard (METS). METS was developed under the auspices of the Digital Library Federation and is maintained by the Network Development and MARC Standards Office of the Library of Congress. It was originally designed for encoding metadata necessary to manage digital library objects within a repository and to facilitate exchange of such objects among repositories. METS is expressed using the XML Schema language and is freely available from the METS website at http://www.loc.gov/standards/mets/.

The use of METS is beneficial since Fedora digital objects are encoded using a community-accepted standard. The METS schema provides all of the functionality required to encode a digital object container with Fedora components: the PID, system metadata, `datastreams`, and `disseminators`. METS can also be used to record multiple versions of `datastreams` and `disseminators` within a digital object.

7.2 Mapping to METS XML Schema

Below is a macro-level mapping of the Fedora digital object components to appropriate METS elements and attributes. In the "METS Encoding" column of the table, the values in ***bold italics*** indicate equivalence to a data definition in the Fedora model described in Section 2.0. Values that are simply in **bold** are fixed strings required by Fedora. In both cases, these conventions indicate that the METS element or attribute is *required* from the Fedora perspective. For example, the OBJID attribute of the root METS element is required and is equivalent to the ***PID*** in a Fedora digital object. Also, the METS root element has a TYPE attribute that must be populated with the string "FedoraObject." Those METS attributes that do not have any string value depicted are deemed *optional* for a Fedora digital object. See the METS XML Schema (http://www.loc.gov/standards/mets/) for general validity rules for METS-encoded documents. See the Fedora extension of the METS XML schema (http://www.cs.cornell.edu/payette/mellon/fedora/mets-fedora.xsd) for elements and attributes added by the Fedora project. The Fedora team is working with the METS editorial board to have these extensions included in the official METS schema. Refer to Appendix A for a sample Fedora digital object encoded in METS.

Table 1. Mapping to METS XML Schema Example

Fedora	METS Encoding
Persistent Identifier (PID) + System Metadata	<!-- The root METS element contains the PID of the Digital Object, a human-readable --> <!-- LABEL, and a content model identifier. The METS TYPE attribute is --> <!-- fixed as 'FedoraObject'--> <METS:mets ID="" OBJID="***PID***" LABEL="***doLabel***" TYPE="FedoraObject" PROFILE="***contentModelID***">

<table>
<tr><td></td><td>

```
<!-- Fedora System Metadata is placed in the METS Header and also in a METS amdSec  that -->
<!-- wraps a special form of digiprov metadata that conforms to the Fedora Audit Trail schema. -->

<METS:metsHdr ID="" CREATEDATE="createdDT" LASTMODDATE="modDT" RECORDSTATUS="state">

    <METS:agent ID="" ROLE="createAgentRole" TYPE="createAgentType">
        <METS:name> createAgentName </METS:name>
        <METS:note> createAgentNote </METS:note>
    </METS:agent>
    <METS:agent ID="" ROLE="createAgentRole" TYPE="createAgentType">
        <METS:name> createAgentName </METS:name>
        <METS:note> createAgentNote </METS:note>
    </METS:agent>
</METS:metsHdr>

<!-- The Fedora Audit Trail is wrapped in a METS amdSec as a block of METS digiprov metadata. -->

<METS:amdSec ID="FEDORA-AUDITTRAIL">
 <METS:digiprovMD  ID="audRecID " CREATED="createDT "  STATUS="state ">
  <METS:mdWrap MIMETYPE="text/xml" MDTYPE="OTHER" LABEL="Fedora  Audit Trail Record">
      <METS:xmlData>
        <fedoraAudit:record>
             <fedoraAudit:process type=""/>
              <fedoraAudit:action>  </fedoraAudit:action>
              <fedoraAudit:responsibility> </fedoraAudit:responsibility>
              <fedoraAudit:date>  </fedoraAudit:date>
             <fedoraAudit:justification> </fedoraAudit:justification>
        </fedoraAudit:record>
    </METS:xmlData>
 </METS:mdWrap>
</METS:digiprovMD>

<METS:digiprovMD  ID="audRecID " CREATED="createDT "  STATUS="state ">
  <METS:mdWrap MIMETYPE="text/xml" MDTYPE="OTHER" LABEL="Fedora Audit Trail Record">
    <METS:xmlData>
      <fedoraAudit:record>
        <fedoraAudit:process type=""/>
        <fedoraAudit:action>  </fedoraAudit:action>
        <fedoraAudit:responsibility> </fedoraAudit:responsibility>
        <fedoraAudit:date>  </fedoraAudit:date>
        <fedoraAudit:justification> </fedoraAudit:justification>
     </fedoraAudit:record>
    </METS:xmlData>
  </METS:mdWrap>
 </METS:digiprovMD>
</METS:amdSec>
```

</td></tr>
<tr><td>Datastreams

Impl-Defined Metadata

+

Referenced External Content

+

Repository-Managed Content</td><td>

```
<!-- Implementor-Defined XML Metadata Datastreams: -->
<!-- A METS amdSec or dmdSec wraps implementor-defined inline metadata -->

<METS:dmdSec ID="dsVersionID " GROUPID="datastreamID" CREATED="dsCreateDT "
                STATUS="state ">
 <METS:mdWrap MIMETYPE="text/xml" MDTYPE="OTHER" LABEL="UVA desc metadata">

   <METS:xmlData>
         <uvadesc:desc>
             <uvadesc:date type="created" certainty="ca." era="bc">13th century</uvadesc:date>
              <uvadesc:identifier scheme="URN"> uva-lib:123</uvadesc:identifier>
              <uvadesc:identifier scheme="URL">http://uva.edu/cgi/imgdef.pl?file=arch/006- 007</uvadesc:identifier>
              <uvadesc:rights type="use">unrestricted></uvadesc:rights>
              <uvadesc:subject scheme="other" othertype="keyword">Bronze Age, Mycenean</uvadesc:subject>
              <uvadesc:subject scheme="other" othertype="keyword">Greek</uvadesc:subject>
              <uvadesc:subject scheme="other" othertype="keyword">Mycenae</uvadesc:subject>
```

</td></tr>
</table>

```
                <uvadesc:subject scheme="other" othertype="keyword">Greece, Argolis</uvadesc:subject>
                <uvadesc:title type="main">Mycenae</uvadesc:title>
              <uvadesc:type>image</uvadesc:type>
          </uvadesc:desc>
    </METS:xmlData>
  </METS:mdWrap>
</METS:dmdSec>

<METS:amdSec ID="datastreamID " >
  <METS:techMD ID="dsVersionID " CREATED="dsCreateDT " STATUS="state ">

    <METS:mdWrap MIMETYPE="text/xml" MDTYPE="OTHER" LABEL="UVA technical metadata">
      <METS:xmlData>
        <uvatech:tech>
          <uvatech:format>image/jpg</uvatech:format>
          <uvatech:compression>LZW</uvatech:compression>
          <uvatech:bitDepth BITS="8"/>
          <uvatech:colorSpace>RBG</uvatech:colorSpace>
          <uvatech:colorProfile CPLOCAT="FILE" CPFILE="UNKNOWN"/>
         <uvatech:resolution>100</uvatech:resolution>
       </uvatech:tech>
    </METS:xmlData>
   </METS:mdWrap>
 </METS:techMD>
</METS:amdSec>


<!—Referenced External Content and Repository-Managed Content Datastreams: -->

<METS:fileSec>
<METS:fileGrp ID="DATASTREAMS">

   <!-- A METS fileGrp wraps multiple versions of a particular datastream and -->
   <!-- contains a unique datastream identifier for the whole version group. -->

   <METS:fileGrp ID="datastreamID" VERSDATE="dsCreateDT" ADMID="inlinemetaIDs audrecIDs">

   <!-- A METS file element is used to describe a particular version of a datastream. -->
   <!-- All  datastreams refer to content via a URL, thus LOCTYPE is fixed as 'URL' -->
   <!-- and the xlink will always be an href to a URL.  -->

    <METS:file ID="dsVersionID" MIMETYPE="dsMIME" SEQ="dsSeq" SIZE="dsSize" GROUPID=""
               CREATED="dsCreateDT" CHECKSUM="" OWNERID="dsControlGrp"
               ADMID="inlinemetaIDs audrecIDs " DMDID="inlinemetaIDs" STATUS="state ">
       <METS:Flocat ID="" LOCTYPE="URL" xlink:href="dsLocation" xlink:title="dsLabel"/>
    </METS:file>
    <METS:file ID="dsVersionID" MIMETYPE="dsMIME" SEQ="dsSeq"   SIZE="dsSize" GROUPID=""
               CREATED="dsCreateDT" CHECKSUM="" OWNERID="dsControlGrp"
               ADMID="inlinemetaIDs audrecIDs " DMDID="inlinemetaIDs" STATUS="state ">
       <METS:Flocat ID="" LOCTYPE="URL" xlink:href="dsLocation" xlink:title="dsLabel "/>
    </METS:file>
  </METS:fileGrp>

   <!-- Another Datastream with just one version. -->

  <METS:fileGrp ID="datastreamID" VERSDATE="dsCreateDT" ADMID="inlinemetaIDs audrecIDs">
    <METS:file ID="dsVersionID" MIMETYPE="dsMIME" SEQ="dsSeq"  SIZE="dsSize" GROUPID=""
               CREATED="dsCreateDT" CHECKSUM=""  OWNERID="dsControlGrp"
               ADMID="inlinemetaIDs audrecIDs " DMDID="inlinemetaIDs" STATUS="state ">
       <METS:Flocat ID="" LOCTYPE="URL" xlink:href="dsLocation" xlink:title=" dsLabel "/>
    </METS:file>
  </METS:fileGrp>
</METS:fileGrp>
</METS:fileSec>
```

Disseminator	*<!-- Datastream Binding Maps for Disseminators (specifically for the Behavior Mechanisms). -->* <METS:structMap ID="*dsBinderMapID*" TYPE="fedora:dsBindingMap" LABEL="" STATUS="*state*" > <METS:div ID="" TYPE="*bmechPID*" LABEL=""> <METS:div ID="" TYPE="*dsBindKey*" ORDER="*dsBindSeq*" LABEL="" DMDID="" > <METS:fptr ID="" FILEID="*datastreamID*" /> </METS:div> </METS:div> </METS:structMap> <METS:structMap ID="*dsBinderMapID*" TYPE="fedora:dsBindingMap" LABEL="" STATUS="*state*" > <METS:div ID="" TYPE="*bmechPID*" LABEL=""> <METS:div ID="" TYPE="*dsBindKey*" ORDER="*dsBindSeq*" LABEL="" DMDID="" > <METS:fptr ID="" FILEID="*datastreamID*" /> </METS:div> </METS:div> </METS:structMap> *<!-- Disseminators -->* *<!-- A METS behaviorSec references a behavior definition (interface) and a behavior mechanism. -->* *<!-- It also points to METS structMap (datastream binding map) via the STRUCTID attribute. -->* <METS:behaviorSec ID="*dissVersionID*" GROUPID="*dissID*" STRUCTID="*dsBinderMapID*" BTYPE="*bdefPID*" CREATED=" *dissCreateDT* LABEL="*dissLabel*" ADMID="*inlinemetaIDs audrecIDs* " STATUS="*state*"> <METS:interfaceDef LABEL="" LOCTYPE="URN" xlink:href="*bdefPID*"/> <METS:mechanism LABEL="" LOCTYPE="URN" xlink:href="*bmechPID*" /> </METS:behaviorSec> <METS:behaviorSec ID="*dissVersionID*" GROUPID="*dissID*" STRUCTID="*dsBinderMapID*" BTYPE="*bdefPID*" CREATED=" *dissCreateDT* LABEL="*dissLabel*" ADMID="*inlinemetaIDs audrecIDs* " STATUS="*state*"> <METS:interfaceDef LABEL="" LOCTYPE="URN" xlink:href="*bdefPID*"/> <METS:mechanism LABEL="" LOCTYPE="URN" xlink:href="*bmechPID*" /> </METS:behaviorSec>

7.3 Digital Object Status Codes (Object State)

A digital object can be marked with different status codes that indicate the state of the object. Object state is a condition that applies to the entire digital object. In METS, the digital object state is recorded on the RECORDSTATUS attribute of the root METS element. The Management subsystem is responsible for setting object state in response to different operations on a digital object. Valid state values are:

Table 2. Digital Object Status Codes

Object State	Label	Definition
A	Active (default)	The digital object fully available for reading and writing, subject to specific access control policies.
N	Incomplete	Digital object is in the process of being created via the API-M component

		methods. The object is not available for reading.
W	Withdrawn	The digital object is accessible only to repository administrators and is effectively withdrawn from public readership.
D	Marked for Deletion	The digital object has been marked for physical deletion, to be reviewed by repository administrators before deletion.

7.4 Object Component State (Deletion, Withdrawal, and Inactivation)

Deletion of object components (i.e., `datastreams` and `disseminators`) is controlled by the Management subsystem. Actual physical removal of object components are allowed only to certain users, (e.g., the repository administrator). The Management subsystem can assign any of the following states to object components to provide the greatest flexibility for repository administrators to establish deletion policies for their institution.

Table 3. Digital Object Component States

Component State	Label	Definition
A	Active (default)	The object component is fully available to all users, subject to specific access control policies.
W	Withdrawn	The object component is accessible only to repository administrators and is effectively withdrawn from public readership.
D	Marked for Deletion	The object component has been marked for physical deletion, to be reviewed by repository administrators before deletion.
B	Broken Link	The object component references a link to external content that was not available via the network when the link was last checked.

7.5 Versioning of Digital Objects

The Fedora digital object versioning strategy allows for versioning of components within a digital object. Both `datastreams` and `disseminators` can be versioned. Each time a new version of a component is created an audit trail record is recorded in the system metadata section of the object. Below are examples of how a METS file should be encoded to reflect versioning in each area.

7.5.1 Recording an Audit Trail in Object's System Metadata

Every time a component within a digital object is versioned (i.e., either a `datastream` or a `disseminator`), the system metadata is updated to reflect an audit trail of changes. Specifically, each component version is associated with a block of administrative metadata that explains the nature of the change. The Fedora Audit Trail metadata schema (fedoraAudit.xsd) is used to capture the who/what/when/why of version updates. Thus, every time a version of a `datastream` or `disseminator` is added to the object, an audit record is also added. In METS, the audit records are encoded within a <METS:amdSec> element, specifically in a METS:digiprovMD element within the amdSec. Note that for `datastream` versions, the ADMID attribute of the METS:file points to the relevant <METS:amdSec> to associate the audit record with the `datastream` version. For `disseminators`, the ADMID attribute on the <METS:behaviorSec> does the equivalent association. See the METS example in 7.5.2.1.

7.5.2 Versioning of Datastreams

A `datastream` can have multiple versions. Each version must have a version identifier and a date of creation. `datastreams` cannot be modified, they must be versioned. Thus, any change to a `datastream` requires a new version of the `datastream` to be created. Every new version of a `datastream` must have an audit record in the Audit Trail section of the object.

7.5.2.1 **Using METS to Encode Datastream Versioning –** the METS <fileSec> is used to record `datastreams`. The METS <fileGrp> is used to group versions of the same `datastream`. Each `datastream` version is represented by a METS <file> element. The ID attribute of the inner <fileGrp> element is used to record the datastreamID and the ID attribute of the <file> elements is used as `datastream` version numbers.

```
<METS:fileGrp ID="DS1" VERSDATE="2001-08-31T06:32:00" STATUS="A">

    <!--This is the most current version of the medium sized image -->
    <METS:file ID="DS1.1" SEQ="2" CREATED="2002-05-22T06:32:00"
            MIMETYPE="image/jgp" OWNERID="E" ADMID="TECH2 AUDREC2"
            STATUS="A">

        <METS:FLocat LOCTYPE="URL"
            xlink:href="http://dl.lib.virginia.edu/data/image/saskia/006-007b2.jpg"
            xlink:title="Saskia medium jpg image"/>
    </METS:file>

    <!--This is an OLDER version of the medium sized image -->
    <METS:file ID="DS1.0" SEQ="1" CREATED="2001-08-31T06:32:00"
            MIMETYPE="image/jgp" OWNERID="E" ADMID="TECH2 AUDREC1"
            STATUS="A">

        <METS:FLocat LOCTYPE="URL"
            xlink:href="http://dl.lib.virginia.edu/data/image/saskia/006-007b1.jpg"
            xlink:title="Saskia medium jpg image"/>
    </METS:file>
</METS:fileGrp>
```

7.5.2.2 **Encoding Conventions for version control using METS –**

7.5.2.2.1 **Datastream version wrapper as <METS:fileGrp>:** one for each `datastream`. Used to group multiple versions of the same `datastream`. The required attributes for version control are:

- **a. ID:** This is the official DatastreamID, common to all versions of the `datastream` in the METS:fileGrp.
- **b. VERSDATE:** To support dateTime-stamped access requests, the VERSDATE attribute should reflect the dateTime that the <fileGrp> was originally created. This is the same as the CREATED attribute on the *original* <file> within the <fileGrp>, which is the first version of a `datastream`. Using VERSDATE as the created date for the <fileGrp> is consistent with METS intended use of this attribute. It may be tempting to update the VERSDATE when a `datastream` is versioned, thus having it reflect the CREATED date of the most recent <file> within it. This may present a problem in dateTime-stamped access requests because the <fileGrp> node would not reflect that there may be a <file> within that has a CREATED date earlier than that of the most recent <file>. For example, if we want a parsing program to quickly evaluate whether a <fileGrp> contains a <file> that is within a requested date, we *cannot* have VERSDATE be the latest modification date. This would require us to always evaluate all the <file> members within <fileGrp>. The VERSDATE on the <fileGrp> would be useless.

7.5.2.2.2 **Datastream as <METS:file>:** used to encode a particular version of a `datastream`. There must be at least one <file> within a <fileGrp>. The required attributes for version control are:

- **a. ID:** this is the `datastream` version identifier (dsVersionID), *not* the official datastreamID (which is found on the <fileGrp> that serves as the `datastream` version group wrapper).
- **b. CREATED:** the dateTime that a version of a `datastream` was created.

7.5.2.2.3 **Algorithm to get Datastreams as of specified dateTime:**

- **a.** Program looks at ID attribute on <fileGrp> elements to locate the required `datastream` (search on datastreamID)

b. Optionally, the program can evaluate the VERSDATE attribute on the <fileGrp> to get a quick answer as to whether the `datastream` version group qualifies as within the dateTime boundary. VERSDATE reflects the oldest `datastream` date.

c. Program parses within <fileGrp> for <file> elements with requestedDateTime >= CREATED. This could yield more than one <file>.

d. If we have multiple qualifying <file> nodes, then we calculate which one has the dateTime closest to the requested dateTime.

7.5.3 Versioning of Disseminators

Since a `disseminator` points to a `behavior definition` object and a `behavior mechanism` object, the `disseminator` is subject to the version changes that have taken place in these objects. Refer to section 9.5.4 for a description of how `behavior definition` and `behavior mechanism` objects affect `disseminators` and disseminations.

There is one case where it may be appropriate to version a `disseminator` component itself. This is the case where the `disseminator` is updated to use an alternative `behavior mechanism`, but still subscribes to the same `behavior definition`. Such an event may occur when a better mechanism has been developed (e.g., faster, better), but the same overall functionality is maintained from the perspective of the object's `disseminator`. The METS example below demonstrates how a `disseminator` (METS behaviorSec) can be versioned for mechanism replacement. Notice that the METS:mechanism points to a entirely different `behavior mechanism` object in the new version.

```
<!--This is the most current version of the Std. Image Disseminator -->
<!--The mechanism has been replaced, but the behavior def is the same. -->
<METS:behaviorSec ID="DISS1.1" GROUPID="DISS1" STRUCTID="S1.1"
        BTYPE="uva-bdef:stdImage" CREATED=" 2002-05-20T08:32:00
        LABEL="UVA Std Image Disseminator" STATUS="A">
    <METS:interfaceDef  LABEL="UVA Standard Image Behavior Definition"
                        LOCTYPE="URN" xlink:href=" uva-bdef:stdImage "/>
   <METS:mechanism  LABEL="A NEW AND IMPROVED Image Mechanism"
                        LOCTYPE="URN" xlink:href="uva-bmech:BETTER-imageMech"/>
</METS:behaviorSec>

<!--This is the older version of the Std. Image Disseminator -->
<METS:behaviorSec ID="DISS1.1" GROUPID="DISS1" STRUCTID="S1.1"
        BTYPE="uva-bdef:stdImage" CREATED=" 2001-08-31T06:32:00
        LABEL="UVA Std Image Disseminator" STATUS="A">
    <METS:interfaceDef  LABEL="UVA Standard Image Behavior Definition"
                        LOCTYPE ="URN"  xlink:href ="uva-bdef:stdImage"/>
    <METS:mechanism   LABEL="Image Mechanism"
                        LOCTYPE ="URN"  xlink:href ="uva-bmech:imageMech"/>
</METS:behaviorSec>
```

7.5.4 Versioning of Behavior Definition and Mechanism Objects

A `behavior definition` object may have multiple versions of a `behavior definition`. In the `behavior definition` object, a distinct date/time stamped block of WSDL represents each version of an abstract service definition. Fedora best practice dictates that new versions are created when new methods are added to `behavior definitions`; however, for backwards compatibility reasons, old methods should never be deleted from any version of a behavior definition.

A `behavior mechanism` object may contain multiple versions of `behavior mechanism` implementations. Different versions are recorded as distinct, date/time-stamped blocks of WSDL service bindings for a given abstract service definition. A new version of a service implementation may be recorded when changes are made to underlying executables for the `behavior mechanism` service. Another case is when a `behavior mechanism` service is upgraded to conform to a new version of a `behavior definition`. Fedora best practice is that any change to an executable that affects the look and feel of content should result in a new version of that executable being maintained, and new service bindings should be recorded in the `behavior mechanism` object.

Table 4. Versioning of Behavior Definition and Mechanism Objects

	Best Practice
Versioning within Behavior Definition Objects	Record a new block of WSDL with a CREATED date/time in the `behavior definition` object whenever new methods are added to a `behavior definition`. Deletion and modification of methods not allowed.
Versioning within Behavior Mechanism Objects	Record a new block of WSDL with a CREATED date/time in the `behavior mechanism` object whenever significant changes have been made to the behavior service executables, and the former service executables are still available. Record a new block of WSDL with a CREATED date/time whenever a behavior service has been upgraded to implement a new version of a `behavior definition`.

7.5.4.1 **Implications for Disseminations** – The Fedora Access subsystem processes a dissemination request by first looking at the method implementation information in the `behavior mechanism` object that is associated with a digital object's `disseminator`. A dissemination request may have a date/time stamp on it; the default is the current date. When the Access subsystem processes the dissemination request it: (1) determines whether there are multiple service implementations

for the method specified in the dissemination request. It does this by looking for multiple WSDL blocks in the `behavior mechanism` object, (2) if there is just one version of the service, it uses that service binding information. Otherwise, it chooses the service implementation (WSDL block) whose CREATED date is closest to the date/time stamp specified in the dissemination request, and (3) runs a dissemination by invoking the method using behavior service binding information.

7.5.4.2 **Implications for Method Reflection on Digital Objects –** When the Access subsystem processes a GetBehaviorMethods request (see section 5.0) for a digital object, it must look at the `disseminator` and report back to the client what methods can be run via dissemination requests. A client cannot take advantage of a method unless the `behavior mechanism` object associated with the `disseminator` specifies an implementation for that method. In theory, a `behavior definition` may specify some abstract methods that a particular `behavior mechanism` object does not records service bindings for. The Access subsystem reports back to clients only those methods that are actually available via a `disseminator`'s `behavior mechanism` object. Thus, a `disseminator` subscribes to a `behavior definition`, but it can only run the methods actually implemented by the mechanism that it uses.

A client may invoke the GetBehaviorMethods request with a particular date/time stamp on it. In this case, the Access subsystem reports the set of methods that were implemented in a `behavior mechanism` object as of that date/time. Thus, the Access subsystem determines whether there are multiple versions of a service implementation in a `behavior mechanism` object (i.e., as multiple blocks of WSDL). It chooses the appropriate version based on the CREATED date of the WSDL block.

It should be noted that since the GetBehaviorMethods request contextualizes the view of a `behavior definition` based on what is implemented in the mechanism, a client who wants to view the official `behavior definition` must make a direct inquiry on a `behavior definition` object.

8.0 Management Subsystem

The Management subsystem provides a Java implementation of the API-M service definition. This subsystem is the workhorse for fulfilling client requests that originate at the web service layer. All requests for creating or manipulating a digital object are

processed here. The Management subsystem consists of four modules: Object Management, Component Management, PID Generation, and Object Validation.

In fulfilling API-M requests, the Management subsystem interacts with the Storage subsystem, which writes and reads objects and components to and from persistent storage. It is the job of the Management subsystem to instantiate a Digital Object Manager to interact with the Storage subsystem. The Digital Object Manager enables the Management subsystem to obtain an appropriate reader or writer for a digital object. An Object Reader or Object Writer enables the Management subsystem to work on a digital object in an abstract manner, free from details of how a digital object is actually stored on disk or on the network.

As the Management subsystem mediates between the web service layer and the Storage subsystem, it also ensures that all operations on a digital object maintain object integrity. It interacts with the Object Validation module to make sure that operations do not cause the object to violate the METS XML Schema or Fedora-specific rules. It is also responsible for Object State. Specifically, the Management subsystem ensures that the value of an object's state attribute (see section 7.3) is properly set for specific object-level operations.

To ensure proper access control, the Management subsystem interacts with the Security subsystem, which is in charge of enforcing policies that pertain to the use of API-M methods.

8.1 API-M Implementation

8.1.1 **Object Management Module** – The Object Management module focuses on the API-M operations that pertain to a digital object as a whole entity. These operations include adding, removing, and obtaining a copy of a complete digital object. Individual components of a digital object are not manipulated via the Object Management module. From the perspective of the Object Management module, the digital object is a stream of bytes with a PID. The Management subsystem translates object-level API-M requests into method calls upon an appropriate Object Reader or Writer that deals with the object at the storage layer.

8.1.1.1 **IngestObject** – fulfills a client request to import a complete digital object into the repository. In Phase 1, it only imports objects that are encoded in the METS format. Calls the PID generation module to obtain a PID for the object. Calls the Object Validation module to ensure that imported object meets all METS and Fedora integrity standards. Rejects objects that do not pass validation testing, and send report back to client. If object is valid, uses an Object Writer to set Object State to "A" (Active), and to put the object into persistent storage.

8.1.1.2 **CreateObject** – fulfills client request to create a new digital object. Calls the PID generation module to obtain a PID for the object.

Creates an empty digital object containing only a PID and initial system metadata. Uses an Object Writer to set Object State to "N" (Incomplete) since the object is not ready for circulation until it has been built up with content and behaviors via the component-level methods of API-M. Uses Object Writer to put the object to persistent storage. Note: clients must use the ObtainLock method before using the component management methods for adding `datastreams` and `disseminators`. Clients must ultimately issue the ReleaseLock method to commit all component additions, at which time the Object State gets set to "A" (Active).

8.1.1.3 **GetObjectXML** – fulfills client request to obtain a copy of a particular digital object in METS XML-encoded format. This method does not resolve `datastream` references within the object. Referenced External Content `datastreams` are returned as pointers to remote content. Repository-Managed Content `datastreams` are returned as local identifiers (i.e., DatastreamIDs). Implementor-Defined XML Metadata `datastreams` are returned as inline XML-encoded content. Uses Object Reader to obtain the object as a stream of bytes.

8.1.1.4 **ExportObject** – fulfills client request to obtain a complete copy of a particular digital object suitable for transporting to another repository in METS XML format. Referenced External Content `datastreams` are returned as pointers to remote content. Repository-Managed Content `datastreams` are returned as inline base64-encoded byte streams. Implementor-Defined XML Metadata `datastreams` are returned as inline XML-encoded content. Uses Object Reader to obtain the object as a stream of bytes.

8.1.1.5 **DeleteObject** – fulfills client request to delete an object from the repository without physically removing the object from the repository . Uses an Object Writer to set the Object State to "D" (Marked for Deletion). This makes the object unavailable, except to repository administrators. Objects can be physically removed from the repository using the PurgeObject operation.

8.1.1.6 **PurgeObject** – fulfills client request to purge an object from the repository which results in the object being physically removed. Objects must be in the "D" state (marked for deletion) before the can by physically removed.

8.1.1.7 **WithdrawObject** – fulfills client request to withdraw an object from circulation. Uses an Object Writer to set the Object State to "W" (Withdrawn). This makes the object unavailable, except to repository administrators. Objects that are withdrawn are never physically removed from the repository.

8.1.1.8 **ObtainLock** – gives the calling client a *write* lock on a particular digital object. Keeps track of the user id and session associated with the locked object. Ensures that a *working copy* of the locked digital object is instantiated via an Object Writer. Ensures that the authoritative copy of the digital object is recorded as locked by the repository. When an object is locked the authoritative copy of the object can be read by others clients while the working copy is being manipulated.

8.1.1.9 **ReleaseLock** – releases a write lock that the calling client has on a particular digital object. The client will specify whether changes should be committed or abandoned. If changes are to be committed, ReleaseLock initiates a process to commit changes made to the object. It calls the Object Validation module to ensure that object meets all METS and Fedora integrity standards. If the object is valid, it initiates object replication routines to propagate changes through the repository. If the client requests that changes be abandoned, the lock is released without committing changes.

8.1.1.10 **GetLockingUser** – fulfills client request to obtain the user holding a lock for a particular digital object.

8.1.1.11 **ListObjectPIDs** – provides the calling client with a list of PIDs for objects stored in the repository. Returns only PIDs of objects whose Object State matches the value supplied by the State parameter. If invoked with no State parameter (i.e., State value is null), the method returns only PIDs for objects with an Object Status of “A” (Active).

8.1.2 **Component Management Module –** The Component Management module focuses on the API-M operations that pertain to a digital object components, specifically `datastreams` and `disseminators`. These operations include adding, removing, and modifying these components. The Management subsystem translates object-level API-M requests into method calls upon an appropriate Object Reader or Writer that deals with the digital object and its components at the storage layer.

8.1.2.1 **AddDatastreamExternal** – fulfills a client request to create a new `datastream` of the External Referenced Content variety. Uses an Object Writer to record the `datastream` information inside the digital object, particularly the URL of the referenced content. The Object Writer assigns a `datastream` identifier, a version identifier, and a created date/time. It also sets the `datastream` control group to "External" meaning the `datastream` content is not under the direct custodianship of the repository. The MIME type is obtained by doing an HTTP GET on the URL. Finally, the Object Writer sets the Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.2 **AddDatastreamManagedContent** – fulfills a client request to create a new `datastream` of the Repository-Managed Content variety. Uses an Object Writer to record the `datastream` information inside the digital object. The Object Writer assigns a `datastream` identifier, a version identifier, and a created date/time. The API-M request contains a stream of byes, and the Object Writer puts this content in a repository-specific storage location. It sets the `datastream` control group to "Internal" meaning the `datastream` content is under the direct custodianship of the repository. Finally, the Object Writer sets the Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.3 **AddDatastreamXMLMetadata** – fulfills a client request to create a new `datastream` of the Implementer-Defined XML Metadata variety. Uses an Object Writer to record the `datastream` information inside the digital object. The Object Writer assigns a `datastream` identifier, a version identifier, and a created date/time. The API-M request has a metadata type indicator (MDType) and a block of XML metadata as a stream of bytes. The Object Writer verifies that the incoming bytes are valid XML with a namespace, and if so, it stores the incoming XML inside the digital object (i.e., as inline XML within the METS XML). The Object Writer sets the `datastream` MIME type to "text/xml." It also sets the `datastream` control group to "Internal" meaning the `datastream` content is under the direct custodianship of the repository. Finally, the Object Writer sets the Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.4 **ModifyDatastreamExternal** – fulfills a client request to modify an existing `datastream` of the External Referenced Content variety. Uses an Object Writer to record updated `datastream` information inside the digital object, particularly the URL of the referenced content. *All modification requests result in the creation of a new version of the* `datastream`. The Object Writer uses the existing `datastream` identifier, but assign a new version identifier and an updated date/time for the version. It also sets the `datastream` control group of the version to "External" meaning the content is not under the direct custodianship of the repository. The MIME type is obtained by doing an HTTP GET on the URL. Finally, the Object Writer sets the new version's Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.5 **ModifyDatastreamManagedContent** – fulfills a client request to modify an existing `datastream` of the Repository-Managed Content variety. *All modification requests result in the creation of a new version of the* `datastream`. Uses an Object Writer and follows the procedure described above in *AddDatastreamManagedContent* for

interpreting the API-M request and storing the new version of `datastream` (attributes and content) in the appropriate manner. The Object Writer uses the existing `datastream` identifier, but assigns a version identifier and updated date/time for the version. Finally, the Object Writer sets the Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.6 **ModifyDatastreamXMLMetadata** – fulfills a client request to modify an existing `datastream` of the Implementer-Defined XML Metadata variety. *All modification requests result in the creation of a new version of the* `datastream`. Uses an Object Writer and follows the procedure described above in *AddDatastreamXMLMetadata* for interpreting the API-M request and storing the new version of the `datastream` (attributes and XML metadata) inside the digital object. The Object Writer uses the existing `datastream` identifier, but assigns a version identifier and updated date/time for the version. Finally, the Object Writer sets the Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.7 **AddDisseminator** – fulfills a client request to create a new `disseminator`. Calls upon the Object Validation module to pre-validate the referential integrity of the `disseminator's` parts. Specifically, ensures that the `behavior definition` and `behavior mechanism` objects specified on the API-M request are compatible. Also, ensures that the `datastream` binding map provided in the API-M request meets the requirements implied in the request (see section 4.1.2.4 for details). *AddDisseminator* uses an Object Writer to record the `disseminator's` information inside the digital object, particularly the PIDs of the `behavior definition` and `behavior mechanism` objects, plus the `datastream` binding map. The Object Writer assigns a `disseminator's` identifier, a version identifier, and a created date/time. Finally, the Object Writer sets the Component State to "A" (Active) and insert an audit trail record in the digital object describing the transaction.

8.1.2.8 **ModifyDisseminator** – fulfills a client request to modify a new `disseminator`. *All modification requests result in the creation of a new version of the disseminator.* Calls upon the Object Validation module to pre-validate the referential integrity of the updated `disseminator's` parts, as described above in the description of *AddDisseminator*. Uses an Object Writer to record the updated `disseminator's` information inside the digital object, particularly the PIDs of the `behavior definition` and `behavior mechanism` Objects, plus the `datastream` binding map. The Object Writer uses the existing `disseminator's` identifier, but assigns a new version identifier, and an updated date/time for the version.

Finally, the Object Writer sets the Component State to "A" (Active) and inserts an audit trail record in the digital object describing the transaction.

8.1.2.9 **DeleteDatastream** - fulfills a client request to remove a `datastream` from a digital object. Uses an Object Writer to set the Component State to "C" (Marked for Deletion). All versions of the `datastream` are marked for deletion (i.e., there is no method for deleting a particular version). Object Writer inserts an audit trail record in the digital object describing the transaction.

8.1.2.10 **WithdrawDatastream** – fulfills a client request to withdraw a `datastream` from service. Uses an Object Writer to set the Object Component State to "W" (Withdrawn). This makes the `datastream` unavailable, except to repository administrators. All versions of the `datastream` are marked as withdrawn (i.e., there is no method for deleting a particular version). Object Writer will insert an audit trail record in the digital object describing the transaction. `Datastreams` that are withdrawn are never physically removed from the repository.

8.1.2.11 **DeleteDisseminator** - fulfills a client request to remove a `disseminator` from a digital object. Uses an Object Writer to set the Component State to "C" (Marked for Deletion). All versions of the `disseminator` are marked for deletion (i.e., there is no method for deleting a particular version). Object Writer inserts an audit trail record in the digital object describing the transaction. Note that *DeleteDisseminator* does *not* affect the `behavior definition` and `behavior mechanism` objects to which the `disseminator` referred. These are independent objects in their own right.

8.1.2.12 **WithdrawDisseminator** – fulfills a client request to withdraw a `disseminator` from service. Uses an Object Writer to set the Object Component State to "W" (Withdrawn). All versions of the `disseminator` are marked as withdrawn (i.e., there is no method for deleting a particular version). This makes the disseminator unavailable, except to repository administrators. Object Writer will insert an audit trail record in the digital object describing the transaction. Note that *WithdrawDisseminator* does *not* affect the `behavior definition` and `behavior mechanism` objects to which the `disseminator` referred. These are independent objects in their own right. `Disseminators` that are withdrawn are never physically removed from the repository.

8.1.2.13 **GetDatastreams** - fulfills a client request to obtain all of the `datastreams` within a digital object. Uses an Object Reader to obtain attributes and content for each `datastream`. Ultimately, the client receives an array of `datastreams` encoded in XML (as defined in the API-M WSDL). Default is to return just current version of each

`datastream`. If request has date/time stamp, returns version of each `datastream` closest to requested timeframe (see algorithm is section 7.5.2.2.3). Does NOT return `datastreams` with Component State of "W" (Withdrawn) or "D" (Marked for Deletion).

8.1.2.14 **ListDatastreamIDs** – provides the calling client with a list of identifiers for `datastreams` in the digital object. Returns only identifiers of `datastreams` that are accessible by clients. The method returns only identifiers of `datastreams` whose Object Component State matches the value supplied by the State parameter. If invoked with no State parameter (i.e., State value is null), the method returns only identifiers for `datastreams` with an Object Component Status of "A" (Active).

8.1.2.15 **GetDatastream** - fulfills a client request to obtain a specific `datastream` within a digital object. The API-M request contains a `datastream` identifier. Follows procedure for *GetDatastreams*, but for a single `datastream`.

8.1.2.16 **GetDisseminators** - fulfills a client request to obtain all of the `disseminators` within a digital object. Uses an Object Reader to obtain attributes for each `disseminator`. Ultimately, the client receives an array of `disseminators` encoded in XML (as defined in the API-M WSDL). Default is to return just current version of each `disseminator`. If request has date/time stamp, returns version of each `disseminator` closest to requested timeframe. Does NOT return identifiers for `disseminators` with Component State of "W" (Withdrawn) or "D" (Marked for Deletion).

8.1.2.17 **ListDisseminatorIDs** – provides the calling client with a list of identifiers for `disseminators` in the digital object. Returns only identifiers of `disseminators` that are accessible by clients. The method returns only identifiers of `disseminators` whose Object Component State matches the value supplied by the State parameter. If invoked with no State parameter (i.e., State value is null), the method returns only identifiers for `disseminators` with an Object Component Status of "A" (Active).

8.1.2.18 **GetDisseminator** - fulfills a client request to obtain a specific `disseminator` within a digital object. The API-M request specifies a `disseminator` identifier. Follows procedure for *GetDisseminators*, but for a single `disseminator`.

8.2 Internal PID Generation Module

The Fedora PID Generation module runs as part of a Fedora repository within the Management subsystem. As mentioned earlier, a PID is a persistent identifier for a digital object. PIDs must be unique within a given repository. However, the

possibility of distributed repositories requires generating PIDs that are unique across all repositories. Therefore, each repository is configured with a unique identifier which serves as the namespace for PIDs generated within that repository.

PIDs are generated at the time that digital objects are added to the repository. The PID Generator is called via methods of API-M (i.e., IngestObject, CreateObject).

8.2.1 PID Generation Interface Definition

8.2.1.1 **GeneratePID(string namespaceID) –** creates a new PID with the specified namespace as a prefix.

Input Parameters:
a. namespaceID: The namespace is the repository identifier that is configured for the Fedora repository. Each Fedora repository has a unique identifier.

Return Value: a PID in string format

Client pre-requisites:
(1) the client is a calling program in the Fedora Management subsystem. This client program must have access to the repository identifier that is configured for the repository

Example:
GeneratePID("uva-lib");

8.2.1.2 **GetLastPID() –** returns the last PID generated by reading the last entry in the PID log file.

Input Parameters: none

Return Value: a PID in string format

Client pre-requisites:
(1) none

Example: GetLastPID();

8.3 PID Generator Implementation

8.3.1 PID Syntax

PIDs are based on the URN Syntax as described in RFC 2141 (see http://www.rfc.net/rfc2141.html), however, we do not prepend the PID with the "urn" prefix. A PID string should be opaque, meaning it should not carry any system-specific meaning within them. Keeping PIDs opaque keeps them from being bound to a specific machine, repository, or location. There are

several possible methods for generating unique strings. The PID Generator uses a simple counter mechanism that provides low risk of duplicate PIDs. A PID consists of two parts, a namespace and an object identifier string, separated by a delimiter character:

(1) **Namespace prefix**: a string that identifies the namespace of a specific repository or a group of repositories. The namespace is a logical construct. From a global uniqueness perspective, the participants in Fedora must avoid overlap of their namespace identifiers. Allowed characters include alpha characters a-z and the dash(-) character, but no other special characters (e.g., uva-lib, uva-cs, cornell-cs, uva, oxford).

(2) **Delimeter** – a single character delimiter consisting of a colon (":").

(3) **Object ID String**: a sequential number generated by an incremental counter. This string uniquely identifies an object within a namespace. Allowed characters include digits 0-9 with unbounded length (e.g., 1, 12, 12345).

Examples of valid PIDs:

uva-lib:1234567890
uva-lib:123
uva-lib:12345
uva-cs:123
cornell.cs:123

8.3.2 **Method Implementation –** The methods of the PID Generator Interface are implemented as follows:

8.3.2.1 **GeneratePID –** The PID is generated in accordance with the syntax specified above. Each time this request is issued, an entry to a PID log file is made indicating the PID that was generated, and a date/time it was generated. The log file also serves as the official counter file for the PID Generator. Thus, each time a new GeneratePID request is processed, the last PID is obtained from the log file (this can be done via the GetLastPID method). The Object ID string is parsed out of the last PID and incremented to create the next PID. The log file must be initialized to bootstrap the first GeneratePID request.

8.3.2.2 **GetLastPID –** This method is typically called by the GeneratePID method to obtain the last PID, (which is then incremented to create the next PID). The GetLastPID method can also be used for utility or debugging purposes.

8.4 Object Validation Module

8.4.1 **METS XML Schema Validation –** Both the Object Management and Component management modules call upon the Object Validation module to ensure that digital object validate against the METS XML schema. This validation module contains a pluggable XML schema validating parser (e.g., Xerces).

8.4.2 **Fedora Integrity Rules –** Beyond the METS schema validation, there are a set of Fedora-specific integrity rules for digital objects. Many elements and attributes in the METS schema are optional. Furthermore, there are some integrity rules that are not easily expressed using XML schema language. Thus, the Object Validation module implements a secondary integrity check. This integrity check is implemented using both java code and the Schematron. Schematron provides a means of declaring a set of rules in an XML format, and uses an XSLT-based approach for validating an XML document against the rules. The table below outlines some of the Fedora-specific integrity rules for digital objects:

Table 5. Selected Fedora Integrity Rules

Fedora Integrity Rule	**Additional METS restrictions**
1. Every digital object must have a PID, a created date/time, and an Object State indicator	OBJID attribute on <mets> required CREATEDATE on <metsHdr> required RECORDSTATUS attribute on <metsHdr> required.
2. Every digital object must be typed as a Fedora digital object	TYPE attribute on <mets>must have fixed value of "FedoraObject"
3. Every `datastream` must have a `datastream` identifier	ID attribute on <fileGrp> required GROUPID on metadata section (mdSecType) required
4. Every `datastream` version must have a version identifier	ID attribute on <file> required ID attribute on metadata section (mdSecType) required
5. Every `datastream` version must point to a Fedora audit trail record	ADMID attribute on <file> must reference a metadata section within an <amdSec> whose ID attribute id "FEDORA-AUDITTRAIL"
6. Every `disseminator` must have a `disseminator` identifier	GROUPID attribute on <behaviorSec> required
7. Every `disseminator` version must have a version identifier	ID attribute on <behaviorSec> required
8. Every `disseminator` version	One <interfaceDef> element and

must have one `behavior definition` and one `behavior mechanism`	one <mechanism> element must be within <behaviorSec>
9. Every `disseminator` version must have one `datastream` binding map	STRUCTID attribute on <behaviorSec> must reference a <structMap> with a TYPE attribute whose value is "fedoraBindMap"
10. Every `disseminator` version must point to a Fedora audit trail record	ADMID attribute on <behaviorSec> must reference a metadata section within an <amdSec> whose ID attribute id "FEDORA-AUDITTRAIL"
11. Every `datastream` and `disseminator` version must have a created date/time	CREATED attribute on <file> required CREATED attribute on <behaviorSec> required CREATED attribute on metadata section (mdSecType) required
12. Every `datastream` and `disseminator` version must have a Component State indicator	STATUS attribute on <file>, metadata section (mdSecType), and <behaviorSec> required
13. Object State may only make the following transitions (before→after): N → A A → W W→ A D → W A → D W → D D → A	RECORDSTATUS attribute on <metsHdr> cannot change to a state that violates the state transition paths.
14. Component State may only make the following transitions (before→after): A → W W→ A D → W A → D W → D D → A A → B B → A	STATUS attribute on <file>, metadata section, or <behaviorSec> cannot change to a state that violates the state transition paths.

9.0 Security Subsystem

The Security subsystem enables repository managers to define access control policies for the repository. It also provides the mechanism to enforce these policies at runtime. In Phase I, the basic repository management functions (API-M) will be secured through a username/password scheme using simple HTTP authentication. Once authenticated, users will either be allowed or denied access to API-M operations. In Phase I, API-A can be protected using IP restriction. This minimal implementation will provide a simple means of securing the repository while the Security Subsystem is under development for delivery

in Phase II. The Security Subsystem will be based as much as possible on existing and emerging standards that are appropriate for the web services environment. This includes a distributed authentication solution (e.g., Shibboleth), an XML policy expression language (e.g., XACML, SAML, XrML), and a policy enforcement mechanism capable of supporting fine-grained object-level policies.

10.0 Access Subsystem

10.1 **API-A Implementation**

The Access subsystem supports digital object reflection and disseminations of digital object content. A digital object aggregates content in the form of `datastreams`, and assigns behaviors (access methods) in the form of `disseminators`. A `disseminator` references an abstract definition of a set of methods and a mechanism (service) for running those methods. When clients issue dissemination requests for a behavior method, supporting services are called to release `datastreams` from the object, or provide transformations of the `datastreams`. The Fedora Access subsystem acts as a service mediator for clients accessing digital objects.

The primary function of the Fedora Access subsystem is to fulfill a client's request for dissemination by evaluating the behavior associations specified in a digital object, and figuring out how to dispatch a service request to an external service with which the digital object associates. The Access subsystem facilitates all external service bindings on behalf of the client, simply returning a dissemination result. A client can be a web browser, a web application with embedded dissemination requests, or a custom client built to interact with Fedora.

Clients can interact with the Access subsystem either via HTTP or SOAP. The WSDL for each repository service defines bindings for both modes of communication. In Phase I of the project, the HTTP GET/POST service bindings connect to a Java Servlet running on Apache Tomcat, and SOAP bindings connect to a SOAP-enabled web service running on Apache Axis with Tomcat.

10.1.1 **Object Reflection Module** – implements the two methods described in the Access Service definition. The Object Reflection module enables clients to discover the kinds of disseminations that are available on the object.

- **GetBehaviorDefinitions** – identifies the types of `behavior definitions` the object subscribes to.
- **GetBehaviorMethods** – returns a data structure containing definitions of the methods for a particular `behavior definition`
- **GetBehaviorMethodsAsWSDL** – returns a WSDL description of the methods for a particular `behavior definition`
- **GetObjectMethods** – returns a data structure containing all method definitions for a particular digital object.

The Object Reflection module is also linked with the Security subsystem and the Storage subsystem. In Phase II of the project, the Security subsystem will enable repository managers to define access control policies for methods in the Object Reflection module. The Object Abstraction interface provides access to the Storage subsystem that manages all data in the repository.

10.1.2 **Dissemination Module** – implements the dissemination method described in the Access Service definition. The Dissemination module provides the sole means of accessing content from digital objects.

The Dissemination module is also linked with the Security subsystem and the Object Abstraction interface. The Security subsystem enables the application of access control policies to each dissemination request at runtime. Note that the Security subsystem is scheduled for Phase II of the project.

The Object Abstraction interface is used to access the relational database in the Storage subsystem. In Phase I, the relational database replicates the information contained in the XML-encoded digital objects for the most current version of each object functioning as a cache to enhance performance and retrieval. The relational database will probably be deprecated in later phases of the project and dissemination requests will directly target the XML-encoded digital objects. See Section 11.0 for additional information regarding the relational database.

10.1.2.1 **GetDissemination** – runs a method on the digital object to produce a dissemination.

10.1.2.2 **Local and Built-in Behavior Mechanism Services Module** – some behavior services are local to the repository as opposed to being external services and are accessible directly through the Local Behavior Mechanism Services Module.

10.1.2.3 **Dissemination Cache** – the Dissemination cache provides enhanced performance for frequently requested dissemination requests. Since the implementation of API-A uses a Java Servlet, servlet caching is used to enhance performance of dissemination requests.

10.2 WSDL for Behavior Mechanisms

A digital object aggregates content in the form of `datastreams`, and assigns behaviors in the form of `disseminators`. A `disseminator` references a `behavior definition` object and a `behavior mechanism` object. A `behavior definition` object contains a special `datastream` whose content is a WSDL definition of an abstract set of methods. A `behavior mechanism` object contains a special `datastream` that is a WSDL definition describing the run-time bindings to an external service for these abstract methods. In essence, the `behavior`

`definition` and `behavior mechanism` objects function as surrogates for external services. The primary function of the Fedora Access subsystem is to satisfy a client's request for dissemination by evaluating the behaviors specified in a digital object and then dispatching a service request to an external service with which the digital object associates. WSDL provides a standards-based way of expressing both the abstract method definitions and the bindings to a external services in `behavior definition` and `behavior mechanism` objects.

In the example depicted in Figure 3, a digital object has a Watermarker `disseminator` that can dynamically apply a watermark to an image. The `disseminator` has two notable attributes: a `behavior definition` identifier and a `behavior mechanism` identifier that reference their respective `behavior definition` and `behavior mechanism` objects. In this example, the service is one for applying a watermark to an image. A `behavior definition` object contains a special `datastream` whose content is a WSDL definition of abstract methods for watermarked images (e.g., getImage). A `behavior mechanism` object contains a special `datastream` that is a WSDL definition describing the run-time bindings to an external service for these watermarking-related methods (operations). Service bindings can be via HTTP GET/POST or SOAP. See Appendix C for an example of a `behavior mechanism` object containing the WSDL definitions for a simple image rendering service.

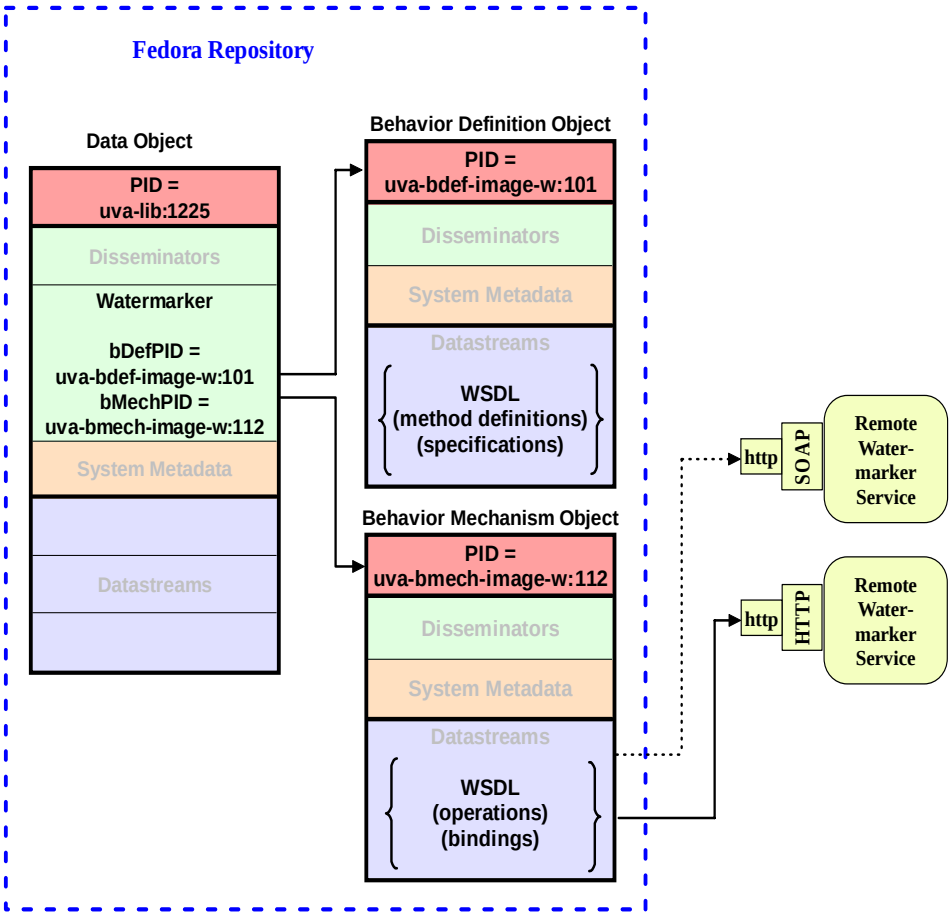

Figure 5. WSDL for Behavior Mechanism Objects

11.0 Storage Subsystem

11.1 **Internal Storage Interface Definition**

The Access and Management APIs reflect upon and manipulate digital objects by means of the reader/writer interfaces, Object Reader (DOReader) and Object Writer (DOWriter). Concrete implementations of these interfaces encapsulate the code and calls necessary to fulfill API-A and API-M requests.

When a digital object needs to be reflected upon or manipulated, the caller provides a PID. This PID is passed in a call to the DOReaderFactory or DOWriterFactory's getReader() or getWriter() method, and a DOReader or Writer is returned, respectively.

A DOReaderFactory and/or DOWriterFactory is provided to the Access or Management subsystem by the Object Manager (DOManager). It is expected that once a factory is obtained, a reference is kept by the caller if repeated calls to getReader() and/or getWriter() is made.

The following diagram illustrates the top-down request flow on digital object requests, assuming an appropriate DOReader or DOWriter has already been obtained by the appropriate factory.

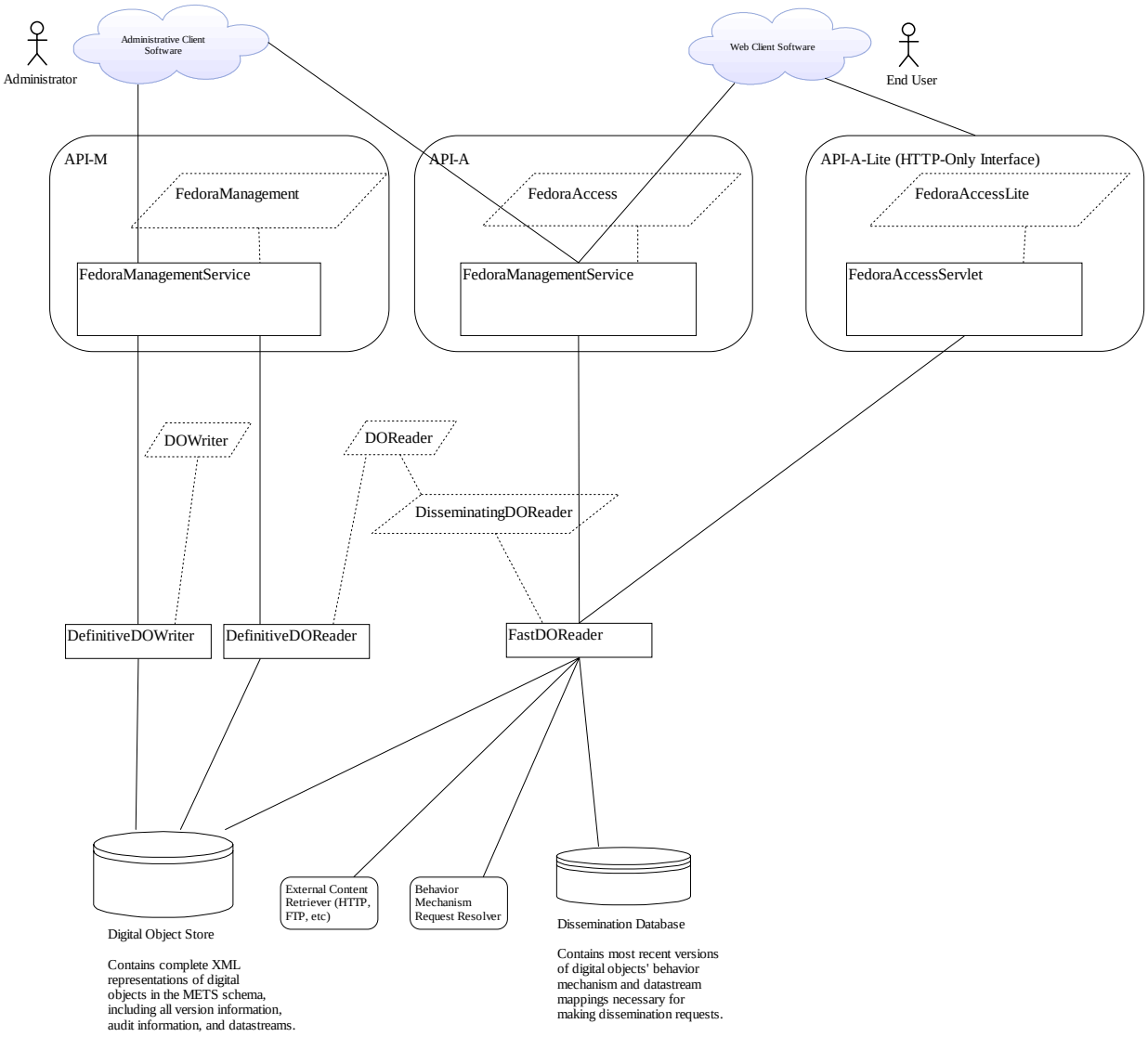

Figure 6. Top-down Request Flow Diagram

11.2 Persistent Storage Implementation

The reference implementation stores digital objects in a partially redundant manner. It is expected that in production, the majority of the calls to a fedora repository will be getDissemination requests on the most recent version of digital objects. Much of the information necessary for dissemination is stored in an SQL database to allow for quicker performance.

11.2.1 Digital Object XML Storage

Digital Objects are stored in XML form in the METS schema on the filesystem. DOReaders and DOWriters use this to read from and write to the freshest copy of the information.

11.2.2 **Digital Object Registry**

The registry is used to tell the DOReaders and DOWriters where on disk a digital object's information is stored and to fulfill the API-M request to enumerate all Digital Object PIDs. The registry is available via the getRegistry() method of the DOManager. getRegistry() is a factory method that provides an appropriate DORegistry object based on the configuration parameters provided to the constructor of DOManager.

DORegistry provides the following methods:

File find(PID objId);
File register(PID objId);
string[] getAll();

11.2.3 **Fedora Dissemination Database**

The dissemination database is used to support high-performance dissemination requests on digital objects. The database schema can be found in Appendix D.

The database is used by FastDOReader. When providing a DOReader to the Access subsystem, the DOReaderFactory takes care of ensuring that a connection to the database is available to the reader.

12.0 Future: PID Resolver Service Implementation

12.1 **General**

A PID Resolution Service may become necessary in the future when multiple Fedora repositories are collaborating. It will be a separate service, running independently of any particular repository. The PID Resolution Service assumes multiple Fedora repositories, and the need for some clients to determine *which repository* a particular digital object is stored in. As mentioned earlier, a PID is a persistent identifier for a digital object that does *not* imply the specific location of the digital object. The Fedora PID Resolution Service will support one or more repositories by (1) maintaining a database of PIDs and their current repository locations, (2) resolving PIDs to their current location. Thus, clients (including a Fedora Repository, itself, as a client) can make requests to the PID Resolution Service to find the current location of any digital object represented by an existing PID. The location may change over time if a digital object is moved to different repository, but the PID itself will remain stable.

Appendices

Appendix A: Example Digital Object

Please see attached file **obj-sizer-image.xml**

Appendix B: Example Behavior Definition Object

Please see attached file **bdef-simple-image.xml**

Appendix C: Example Behavior Mechanism Object

Please see attached file **bmech-sizer-image.xml**

Appendix D: Database Schema

Please see attached file **db-schema.doc**

Appendix E: Glossary

coming soon.