(The case for) A high-level storage interface in Fedora

Aaron Birkland
Cornell University

January 26, 2010 [draft]

Abstract

Traditionally, the pluggable storage interface in Fedora has followed a "low-level" paradigm where objects and datastreams are presented to the storage layer as independent, anonymous blobs of data. This arrangement has proven simple, reliable, and generally flexible. In the past few years however, there has been an increasing need for Fedora to mediate storage in more complex scenarios. Managing large numbers of big datastreams, multiplexing storage between different devices, and archiving content in a transparent manner are tasks that are difficult to achieve through Fedora currently. One reason for difficulty is that all storage logic is essentially hard-coded in Fedora, and storage plugins are limited in what they can deduce from the anonymous blobs they are given. This proposal presents an alternate "high-level" interface where the storage layer is aware of the structural semantics of a Fedora object, and operations are performed on a whole-object basis. Such an architecture pushes the storage logic into external plugins, allowing greater flexibility in adapting to the needs of preservation, performance, or analysis-oriented use cases. This proposal will examine these use cases, compare architectures, and demonstrate how the high level storage interface can enhance existing components by inserting intelligence between the repository and the Akubra blob store.

1 Introduction

There are many ways to manage digital content with the Fedora repository. Fedora objects provide a mostly uniform encapsulation and service binding over content that can be stored and managed in many different ways. Out of the box, Fedora offers the capability to accept datastream content and store/manage it itself in Managed datastreams, as well as referencing content managed elsewhere in External datastreams.

To date, Managed datastreams in Fedora have been implemented in a very simple manner. Object XML and datastreams are serialized as blobs with opaque identifiers, and sent to a low-level storage module in a series of independent operations. This low-level storage module has always been a pluggable component - many implementations have been written for various storage devices.

Unfortunately, this architecture essentially hard-codes several storage assumptions, and makes it difficult to store content in in a dynamic or intelligent manner. For example, if one wanted to store content in different devices based upon some criteria (multiplexing use case), it is not clear where this multiplexing logic should be. The low level storage module is given very little information about the nature of the blobs it is asked to store, would potentially need to reflect into blob content in order to understand its nature. The DOManager layer would seem like a better choice as it is directly above the storage layer, but in reality that would be difficult to implement. DOManager is currently responsible for many important tasks, such as object validation and maintaining indexes, that are unrelated to storage. There is no clean place to insert storage logic without having to deal with other object management issues, so implementing a new or extended DOManager is perhaps more difficult than it could be.

As it stands now, the only way to manage data in a complex way with Fedora is to store Externally. This is a very powerful mechanism that places the logic and mechanics of storing data outside of Fedora entirely. An outside process stores datastream content wherever and however it wishes, then as a last step registers the location of this content with Fedora.

While managing content Externally is always a viable option with its own set of advantages and drawbacks, this proposal focuses on how Fedora could manage content internally in a more intelligent fashion. In other words, instead of an external service that coordinates storage resources, an internal approach would have applications merely introduce content to Fedora

through its own management APIs, while internal plugins to Fedora perform the complex storage tasks. This proposal does not seek to advocate one method over another, but rather to explore an architecture for which internally managed datastreams would be a viable option in complex or novel situations.

2 The `HighlevelStorage` layer

This paper proposes the addition of a high-level storage layer between the object management code and storage. Perhaps it is best to describe this layer in terms of Fedora architecture and separation of concerns.

Below is a description of various proposed or existing components within Fedora, and the tasks that could or should be performed within each layer. We start with the highest-level management layer, and work our way down to the lowest storage layer. Each layer is characterized with a brief descriptive label, and the most relevant internal java interfaces/components are listed.

Operation logic (`Management`)

- Translate API operations into specific change sets for an object
- Initiate policy enforcement
- Manage timestamps and propagate writes to `DOManager`
- Generate new audit entries

Object logic (`DOManager, DOReader, DOWriter`)

- Manage datastream versioning
- Add audit records to object
- Set properties or object state
- validation
- Indexing/registering
- RELS-EXT manipulation (add/purge relationship)

Storage logic (`HighlevelStorage`)

- Perform multiplexing decisions

- Grouping of components into file archives
- Managing transactions and atomicity between storage components
- Locking strategies
- Serializing
- indexing/registering
- Sending data to storage implementation

Storage implementation (Akubra blob store, other)

- Connect to storage implementations
- Store bytes, manage failures
- Present storage device as a transactional resource

As is evident, 'storage logic' is an apt characterization of the proposed HighlevelStorage module. In order to perform functions such as multiplexing decisions, serialization, locking, etc, this module necessarily has to be aware of the logical structure of fedora objects, and have an awareness of operations in at least a whole-object scope. Thus, the interface it presents to the object logic components must not be blob-based.

Table 1 presents a proposed `HighlevelStorage` interface, and compares it with the existing `ILowlevelStorage` interface currently in Fedora. As we can see, the proposed HighlevelStorage interface is oriented towards pid identifiers and whole-object representations (`DigitalObject`[1]), where the ILowlevel-Storage is oriented toward opaque identifiers and InputStream blobs.

There are a few things that are notable in the proposed HighlevelStorage interface. To begin, we see that the `update` method is unusual in that it takes two arguments. The intent is to convey (a) a complete representation of the updated object and (b) the structure of the old object *as known to the thread or process performing the update.*

[1]This is an existing internal Fedora interface representing the structure of a Fedora object, though a final proposal might end up using some simplification or variant of the same concept.

```
interface ILowlevelStorage {

 void addObject(
        String objectKey,
        InputStream content);

 void replaceObject(
        String objectKey,
        InputStream content);

 void removeObject(
        String objectKey);

 InputStream retrieveObject(
        String objectKey);

 void addDatastream(
        String dsKey,
        InputStream content);

 void replaceDatastream(
        String dsKey,
        InputStream content);

 void removeDatastream(
        String dsKey);

 InputStream retrieveDatastream(
        String objectKey)

}
```

```
interface HighlevelStorage {

 void add(
        DigitalObject object);

 void update(
        DigitalObject oldVers,
        DigitalObject newVers);

 DigitalObject read(
        String pid);

 void remove(
        String pid);

}
```

Table 1: Comparison of existing low-level (abridged) and proposed high-level interfaces

For (a), knowing the complete representation of the updated object would allow *all* mutations of an object to be performed as a single operation. For

example, suppose that we wish to add a datastream to an object representing an image, add another datastream representing a thumbnail representation, and update RELS-INT to signify a file name for the image and a relationship indicating that one datastream is a thumbnail of another. If future revisions of Fedora's external API allows multiple mutations of an object to occur as a single atomic operation, this HighlevelStorage api would be suitable for truly implementing the storage of this change set as a single atomic operation.

Additionally, since the semantics of the whole object is known to the HighlevelStorage layer, storage decisions may be made based upon various known attributes such as content model, datastream mime type, owner, etc.

For (b), knowing the old version of the object in an update operation allows the HighlevelStorage impl to derive the change set to an object, and determine if the version of the object in storage matches the version of the object that was assumed to be in storage while the object was being manipulated. This gives the HighlevelStorage module the tools it needs in order to implement a variety of locking strategies or other concurrency-control strategies.

3 Use cases and possible solutions

This section contains several use cases, and examples of how they may be satisfied within the HighlevelStorage paradigm

3.1 Multiplexing storage between different devices

Akubra provides the ability to multiplex multiple backing stores behind a single, unified interface. In order to achieve this, an underlying `AbstractMuxConnection` must be provided that is able to pick a BlobStore based on a *blob URI*, and some *hints*. In other words,

```
getBlob(URI blobid, Map<String, String> hints)
```

connects to a different store based solely upon the information present in the blob URI and the map of hints that is passed along to AbstractMuxConnection.

In order to send storage to the proper location, an appropriate selection of blobid URI and hints must be chosen. HighlevelStorage seems like it would be a good place to plug in such logic. For example, suppose that

we have a multiplexing Akubra instance which will send a blob to a given store based upon the presence of the hint `store = uri:/storeUri` where `uri:/storeUri` is the uri of a configured blob store. Logic can be inserted in the HighlevelStorage layer that picks a store URI based upon the desired criteria (e.g. object XML goes into store A, datastreams declared in model Y go into store B, datastreams with mimetype video/* go into store C, etc).

3.2 Storing entire object in self-contained file archives

Currently, Fedora stores objects and managed datastream versions as separate blobs in storage. When creating or updating an object, these various blobs are inserted separately into storage (indeed, ILowlevelStorage has parallel sets of operations for object XML blobs vs datastream blobs, see table 1).

Suppose that, for preservation purposes, we wanted Fedora to natively store objects into self-contained archives such as zipped BagIt packages[2] or Fedora AtomZip. This storage paradigm violates many of the assumptions present in Fedora currently, and would be difficult to implement cleanly in the current architecture without creating a new DOManager.

HighlevelStorage is an appropriate layer in which to serialize objects into an archive before sending to storage. The read and write operations in the HighlevelStorage interface do not make any assumptions about the serialized structure of the stored object. Additionally, since the 'add' and 'update' operations take whole-object representations (`DigitalObject`) as arguments, it would be trivial write an implementation that serializes the object into the desired archive.

3.3 Non-blob storage

Since the interface to HighlevelStorage is whole-object based and not blob-based, there is no implicit assumption about how the object is serialized and stored. While most of this proposal has focused on a blob store such as Akubra as a final storage implementation under HighlevelStorage, that does not have to be the case.

Some examples of non-blob storage devices into which one could potentially store object structure include:

[2]http://www.digitalpreservation.gov/library/resources/tools/docs/bagitspec.pdf

- serializing fedora object model into rdf and inserting/manipulating in a triple store
- storing FOXML in an XML database
- Storing object XML and datastreams in a column-oriented store such as HBase[3] or Vertica[4]
- Storing an object as multiple rows/column values in a relational database.

Again, HighlevelStorage interface makes no assumptions about the serialization and storage of an object – it only assumes that somehow an object's data can be reconstituted into an object (`DigitalObject`) that represents the object's logical structure and provides access to datastreams. The fact that an object may or may not be serialized into a stream of bytes is an implementation detail that the interface is not concerned with.

3.4 Lock-free concurrent updates

Since `update` requires as arguments both the updated object *and* the (assumed) old object version, explicit locking is not necessarily needed in the DOManager layer. HighlevelStorage is free to implement optimistic locking by comparing the actual old version of the object in storage with the declared old version from the updating thread. At this point, a HighlevelStorage implementation may take advantage of any locking or atomic update capabilities that may be present in the underlying storage device.

Additionally, this property can allow for HighlevelStorage implementations that are suitable for use in a distributed shared-nothing environment with multiple Fedora instances managing a single repository using shared storage.

3.5 Large-scale processing/analysis of data

Multiplexing can be leveraged in order to allow reasoning about the storage location of certain classes of data in the repository. For example, text documents introduced into the repository might be sent to their own bucket in Amazon S3. The data in this bucket (which could be constructed to contain

[3]http://hadoop.apache.org/hbase/
[4]http://www.vertica.com/enterprise-data-warehouse

all the relevant indexable data) may then simply be used as an input to an "Elastic MapReduce[5]" process that scans all data and builds a search index. For massive datasets that need to be traversed entirely, this may be a more performant method than reading data through Fedora's APIs.

Likewise, if a suitable non-blob datastore such as HBase or Vertica is employed for storage, one may use their built-in analysis facilities (MapReduce) to perform analysis of all data, or certain subsets thereof.

4 Default Fedora implementation

If the additioon of `HighlevelStorage` is accepted as a valid approach, it would allow a vastly wide array of possible storage configurations. Many long-standing assumptions (such as serialization to FOXML in files) will no longer remain valid universally. Despite this, it may make sense to ship Fedora "out of the box" with a configuration that results in a familiar pattern of file and datastream storage, with a few features (such as multiplexing and plugging in different storage devices) readily added and manipulated.

In other words, it may make sense to provide a default configuration as follows:

`Management` → `DOManager` → `HighlevelStorage` → Akubra

with a simple default multiplexing Akubra implementation, and clear instructions on where and how to insert add-on multiplexing logic and additional storage devices.

[5]http://aws.amazon.com/elasticmapreduce/