The Moab Design for Digital Object Versioning

Richard N Anderson
Digital Library Systems and Services
Stanford University Libraries
15 July 2013

Table of Contents

1 Introduction ........................................................................................................................................................................... 3
1.1 Abstract .................................................................................................................................................................................. 3
1.2 The Stanford Digital Repository ............................................................................................................................................... 3
1.3 Digital Object Diversity .......................................................................................................................................................... 3
1.4 The Need for Versioning ........................................................................................................................................................ 3

2 Versioning Requirements ....................................................................................................................................................... 4
2.1 Accessioning Requirements ....................................................................................................................................................... 4
2.2 Retrieval Requirements ............................................................................................................................................................ 4
2.3 Fidelity Requirements ............................................................................................................................................................. 4
2.4 Efficiency Requirements .......................................................................................................................................................... 4
2.5 Replication Requirements ....................................................................................................................................................... 4
2.6 Technology Requirements ....................................................................................................................................................... 5

3 Versioning Approaches .......................................................................................................................................................... 6
3.1 Whole Object Versioning ........................................................................................................................................................ 6
3.2 Delta Versioning .................................................................................................................................................................... 6
3.3 Forward-Delta Versioning ....................................................................................................................................................... 7
3.4 Reverse-Delta Versioning ....................................................................................................................................................... 8
3.4.1 CDL ReDD design ............................................................................................................................................................ 9
3.5 Content-Addressable Storage (CAS) ........................................................................................................................................ 10
3.5.1 Git ......................................................................................................................................................................................... 11
3.5.2 Boar ..................................................................................................................................................................................... 13
3.6 Summary of Versioning Approaches ................................................................................................................................... 14
3.7 Tape Archive Efficiency vs. Versioning Design ................................................................................................................ 14

4 The Moab Design .................................................................................................................................................................. 15
4.1 Online Storage Folder Structure ............................................................................................................................................... 16
4.1.1 Object Folder .................................................................................................................................................................... 16
4.1.2 Version Folder ................................................................................................................................................................ 16
4.1.3 Data Folder ...................................................................................................................................................................... 16
4.1.4 Manifests folder ............................................................................................................................................................ 17
4.2 File Manifestations ................................................................................................................................................................. 18
4.3 A Versioning Example ........................................................................................................................................................... 19

5 The Moab-Versioning Gem ................................................................................................................................................... 21
5.1 Moab Manifests ...................................................................................................................................................................... 22
5.1.1 The Version Inventory ........................................................................................................................................................ 22
5.1.2 The Signature Catalog ..................................................................................................................................................... 23
5.1.3 Version Additions Report ................................................................................................................................................ 23
5.1.4 File Inventory Differences ............................................................................................................................................. 25
5.1.5 Algorithm Used to Compare File Inventories ............................................................................................................... 25
5.1.6 Version Metadata ........................................................................................................................................................... 27
5.2 Moab Storage Objects .......................................................................................................................................................... 27

6 The SDR Preservation Workflow .............................................................................................................................................. 28

7 Conclusion ............................................................................................................................................................................. 29
List of Figures

Figure 1 - Whole Object Versioning .................................................................................................................. 6
Figure 2 - Delta Versioning .............................................................................................................................. 6
Figure 3 - Forward-Delta Versioning .............................................................................................................. 7
Figure 4 - Forward-Delta Insert File .............................................................................................................. 7
Figure 5 - Reverse-Delta Versioning .............................................................................................................. 8
Figure 6 - Reverse-Delta Insert File .............................................................................................................. 8
Figure 7 - Content-Addressable Storage ..................................................................................................... 10
Figure 8 - Git Object Model .......................................................................................................................... 11
Figure 9 - Git Revisions ................................................................................................................................. 11
Figure 10 - Git Loose File .............................................................................................................................. 12
Figure 11 - Git Packfile ................................................................................................................................. 12
Figure 12 - File Manifestation ...................................................................................................................... 18
Figure 13 - File with 2 Instances .................................................................................................................. 18
Figure 14 - Version 1 ..................................................................................................................................... 19
Figure 15 - Version 2 ..................................................................................................................................... 20
Figure 16 - Version 3 ..................................................................................................................................... 20
Figure 17 - Compare Manifests .................................................................................................................... 26
1 Introduction

1.1 Abstract
The Stanford Digital Repository has adopted the "Moab" design for versioned archiving of digital objects—a locally developed approach that optimizes data transfer, storage, and replication while providing efficient single file retrieval or full object reconstruction for any version of an object. This paper includes a review of various versioning strategies including forward-delta, reverse-delta and content-addressable mechanisms, the pro’s and cons of each, and highlights the relative advantages of the Moab design. In our approach, the fixity information of a file manifestation is used as its primary identifier and the filename is treated as metadata. Storage and retrieval of an object’s files is facilitated by mapping between a virtual version inventory and the physical location via a file signature catalog.

1.2 The Stanford Digital Repository
The Stanford Digital Repository (SDR) utilizes a set of interrelated accessioning workflows to assemble digital objects in a temporary workspace, augment them with metadata, make them publicly accessible, and preserve them in long-term storage. SDR uses Fedora in the course of this process to hold active copies of metadata in the form of XML datastreams and to specify the relationships between objects. We do not, however, use Fedora for the preservation storage component of our repository. Instead, we serialize the metadata as XML files and store them along with the original content files in a normal filesystem directory structure, where the object’s home directory is named using the object ID, and the metadata and content files are located in subdirectories of that parent.

1.3 Digital Object Diversity
Our repository aims to support the preservation of a wide variety of digital information being created and used by Stanford communities engaged in learning, scholarship, and research. The types of digital objects being preserved include electronic theses and dissertations, images, scanned books and manuscripts, audio files, and video files. Some of these objects are quite large, both in number and size of the component files. For example, a manuscript collection already ingested contains objects ranging in size from 1 to 600 Gigabytes. Objects containing video files may be in the Terabyte range.

1.4 The Need for Versioning
Some of our collections have proved to have volatile content. For example, significant number of manuscript pages have needed to be rescanned (in high-resolution mode) to correct problems that were found after initial ingest had occurred. Updating of metadata also occurs over time. In either case, the modifications of a digital object that result in a new version can be viewed as a mixture of add, delete, modify, or rename file operations. The repository storage system can most efficiently accommodate digital object versioning by archiving only the changed files and metadata in a way that preserves the history of changes to an object, retains the ability to retrieve earlier versions, and minimizes resource consumption.
2 Versioning Requirements
The versioning approach selected for the SDR had to meet the following set of basic requirements.

2.1 Accessioning Requirements
The pipeline between a depositing agent and the accessioning system should be flexible enough to accommodate various modes of new version submission. A depositor should be able to provide a full set of files comprising a new version (which may include files carried over from the previous version), or they should be able to just send only the files that have been added or modified and/or directives about which files should be deleted or renamed.

2.2 Retrieval Requirements
The mechanism for retrieval of a digital object should have a similar flexibility. One should be able to retrieve all or only a subset of the files from any version. It must be possible to retrieve, reconstruct, and deliver an exact copy of a digital object as it was originally submitted. The original filesystem properties for a file, such as relative pathname and last modification date should be faithfully preserved as metadata along with the file.

2.3 Fidelity Requirements
The preservation storage system should avoid introducing file content changes that cannot be reversed or recovered from. Any form of lossy compression is of course to be avoided, as is line-ending normalization of text files. And even lossless compression has been shown to exacerbate the preservation risk associated with bit rot. According to (Wright, Miller, & Addis, 2009) an uncompressed .WAV file with .4% errors will exhibit barely noticeable differences from the undamaged original whereas a similarly damaged MP3 file will not even open.

In order to confirm the fidelity of storage, checksums (message digests) are used to verify that file corruption has not occurred. The checksum for a file at the time of submission should be a constant that carries through to the ultimate storage location. For extra insurance the system should allow more than one checksum to be tracked for each file.

2.4 Efficiency Requirements
The storage system must accommodate all sizes of digital objects in a way that minimizes disk space consumption as well as the CPU and network resources required for such operations as file transfer, fixity checking, replication, and tape archiving. Greater efficiency of processing correlates with increased throughput in the ingest workflow.

Unintentional redundant storage of content files should be avoided. This is especially true for large binary files, such as video files or high-resolution TIFFs. If a new version of an object has only minor changes, then it should not be necessary to re-archive all the files in the object. If only a filename has changed, but not the internal content, then it should be possible to record that fact as metadata without creating a duplicate copy of the file.

2.5 Replication Requirements
A standard feature of any preservation system is the creation of extra copies of each digital object, either at other disk locations or on geographically dispersed tape media. The replication operation should consume a minimum of processing, I/O, and storage resource needed for making file copies or performing media migration.
SDR currently uses the Rsync[^2] file transfer utility to copy files from one location to another. It has the benefit that it can synchronize data in two locations by detecting and copying only the file differences from one location to another. It uses stat to figure out which files have changed and compare-by-hash to determine which data blocks are different between two locations so that it can send only the blocks with different hashes.

But even rsync's speed can suffer when one is replicating a digital object in its entirety. At a minimum, rsync needs to stat all the files in the specified directory structure and examine the file properties (size and date), which can take a appreciable time for an object containing a large number of files. Replication of a new version's files can therefore be made more efficient by using a directory structure that isolates a new version's changes to a known sub-location, reducing the number of files whose properties need to be compared.

Archiving of versioned objects onto tape is also impacted by the way in which digital object storage is structured. Archiving of the initial version of an object is straightforward, but archiving of a subsequent version requires a design that facilitates the copying of only the new or changed files.

### 2.6 Technology Requirements

We are currently minimizing costs through use of open source software and commodity hardware components as much as possible for our system. We have in the past experimented with a variety of vendor-supplied products, which have been quite promising. But we had difficulty addressing the long-term finances for scaling those systems to the size needed for our anticipated volume of content. We also need to minimize our system's dependency on vendor-supplied solutions. We should not need to recover from the business failure of a vendor, or from the obsolescence of a technology.
3 Versioning Approaches

Let us now turn our attention to a review of versioning approaches and then examine some implementations that illustrate those options. This list is by no means exhaustive, but summarizes many of the technologies that have been discussed by digital curators.

3.1 Whole Object Versioning

A foolproof mechanism for storing multiple versions of an object would be to ingest each new version as if it were a completely new set of files, and keep each version's files in its own separate bucket.

![Figure 1 - Whole Object Versioning](image)

This is a simple design, which should be easy to implement and would support fast reconstruction of any version. However, it entails a high level of file duplication and resource consumption.

3.2 Delta Versioning

One way to reduce duplication of file storage is for each new version to store only the changes that have occurred between versions. This mechanism has been used in virtually all software revision control systems.

![Figure 2 - Delta Versioning](image)

This design has a much lower level of file duplication than the whole version option, but does require more complex algorithms for storage and retrieval. A file rename operation can be treated as a combination of a file delete and a file add.
There are 2 varieties of delta versioning: forward-delta and reverse-delta.

### 3.3 Forward-Delta Versioning

In forward-delta versioning, you start by creating a storage bucket containing all of the initially ingested files. When later versions are added, each new version's bucket contains only the files that have been added or modified since the previous version (plus a list of which files have been deleted).

![Figure 3 - Forward-Delta Versioning](image)

This approach requires less work to add a new version and has less impact on replication and tape archive, but requires effort to reconstruct later versions, especially if the number of versions is large.

To illustrate an insert file composite change we start with a version 1 container holding all the files comprising the original submission. We need to insert a new file as page-3.tif in place of the existing file of that name, and move the remaining pages into new positions in the sequence. Broken down to smaller steps, this operation is a combination of a file addition and a couple of file renames. The simplest way to implement this would be to delete page-3.tif and the files that follow it in sequence, and then add in the new file and the renamed files.

![Figure 4 - Forward-Delta Insert File](image)

To retrieve a copy of the complete version 2 object, one would copy the files from version 1, then delete the version 2 deletes (pages 3 and 4), and finally add pages 3 to 5. If more versions exist, then the process must be repeated for each of the new versions.
3.4 Reverse-Delta Versioning

Reverse-delta versioning is the inverse of forward-delta versioning. In this approach, the storage bucket for the most recent version of an object will contain all the files comprising that version, and the storage buckets for previous versions will contain only the files that were "left behind" when a new version was created.

This approach requires less work to reconstruct the latest version, but more work to add a new version. And there is a greater impact on replication and tape archiving.

To demonstrate this, let’s again begin with the same version 1 object, and then see what the result is when we add a new version using the insert file scenario.

Version 2’s container now holds copies of all version 2 files, while version 1’s storage area has been revised to contain only the information required to restore version 1 using version 2 as a starting point.
### 3.4.1 CDL ReDD design

The California Digital Library has implemented a very rigorous and well-documented storage mechanism (Storage Micro-Service\(^3\)) based in part on a framework called Dflat\(^4\) and a reverse-delta implementation called ReDD\(^5\). Below is a skeleton example of a Dflat/ReDD/dNatural directory structure that is holding 2 versions of a digital object. V002 is the latest directory and contains a full complete version. V001 is a reverse-delta version.

<table>
<thead>
<tr>
<th>&lt;object-identifier&gt;/</th>
<th># Dflat home directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ current.txt ]</td>
<td># pointer to current version (e.g. 'v002')</td>
</tr>
<tr>
<td>[ dflat-info.txt ]</td>
<td># Dflat properties file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v001/</th>
<th># reverse-delta version</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ manifest.txt ]</td>
<td># version manifest</td>
</tr>
<tr>
<td>delta/</td>
<td># ReDD directory</td>
</tr>
<tr>
<td>[ add/ ]</td>
<td># files to be added relative to subsequent</td>
</tr>
<tr>
<td>[ delete.txt ]</td>
<td># files to be deleted relative to subsequent</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v002/</th>
<th># current version</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ manifest.txt ]</td>
<td># version manifest</td>
</tr>
<tr>
<td>full/</td>
<td># dNatural directory</td>
</tr>
<tr>
<td>[ consumer/ ]</td>
<td># consumer-supplied files directory</td>
</tr>
<tr>
<td>[ producer/ ]</td>
<td># producer-supplied files directory</td>
</tr>
<tr>
<td>[ system ]</td>
<td># system-generated files directory</td>
</tr>
</tbody>
</table>

The CDL ReDD implementation works by first adding a new complete full version. At this point the latest version and the previous version are both full versions. Analysis is then performed to determine how one would generate the previous version from the new latest version, and a new reverse-delta version is created to replace the previous full version.

The SDR has imitated aspects this directory hierarchy in our own storage layer. We did, however, have concerns about the ReDD mechanism:

- Each time a new version is added there must be a subsequent backup of the new (full) version’s files as well as the now scaled-back previous version. This requires a complete archive of both directories to tape, and therefore a much higher amount of network traffic and tape I/O than would be required by a forward-delta, or content-addressable mechanism.

- There does not seem to be a provision for renaming files across versions other than to delete the original file and then add in the file with the new name, resulting in duplicate storage of the content under two different names.
3.5 Content-Addressable Storage (CAS)

The phrase "content-addressable storage" covers a broad range of meaning, but in the case of a version control system it refers to a practice of using a file's checksum value as an identifier and locator for the file in the storage system. This allows any given file instance to be stored only once using the checksum value as the key to its location, and enables multiple version manifests to reference that file location regardless of the filename used in a given version manifest. This usage is also known by the phrase "compare-by-hash" (Black, 2006), since the usual method for creating a suitable checksum (aka "message digest") is to use a cryptographic hash function.

![Content-Addressable Storage Diagram](image)

The CAS approach has a simple conceptual design that eliminates file storage duplication and facilitates easy reconstruction of any object version. It introduces, however, a dependency on the version manifests, since that file inventory becomes the only authoritative source for keeping track of which files are part of each version and what are their "real" filenames.

Another factor to consider has to do with statistical probabilities. What is the potential for 2 files having differing contents to hash to the same checksum value? If using SHA1 digests, the odds are 1 in $2^{160} \approx 10^{48}$. For SHA256 the odds are 1 in $2^{256} \approx 10^{77}$, which is comparable to the number of atoms in the entire universe. SHA2 is also much more immune to brute-force attempts to 'crack' the hash algorithm. The concern about algorithm cracking is mainly focused on the risk of someone figuring out how to create a bogus password or digital signature, but could also be raised in the context of a digital archive if there is a worry about fraudulent alteration of preserved digital assets.

The probability that a collection of unique files may contain a pair of files with the same checksum is actually larger than the above number, but would only be worrisome if you had an extremely large set of files. If using SHA256 and your repository's size were to approach $2^{128} \approx 10^{38}$ files, then duplicate checksums would begin to be observed. This is often called the "birthday paradox" because in a group of 57 people, there is a 99% probability that 2 of those people will have the same birthday.

I will next examine 2 existing open source implementations of Content-Addressable Storage that have been suggested as viable candidates for digital object versioning (Git and Boar), before finally describing the Moab design.
3.5.1 Git

Git\cite{12} is a popular distributed version control system that was initially created by Linus Torvalds for tracking changes to the Linux operating system. Due in part to its operating system heritage, it has an object model that looks very similar to a file directory hierarchy. In this design blobs are used to store the content of files, trees are used to represent directories, and commits are used to tag versions.\cite{13}

![Git Object Model](image1)

Figure 8 - Git Object Model

Git does not store files as they are received, but tacks on a header containing the string *blob* followed by a space, the length of the data in bytes, and a null byte. A *tree* functions similarly to a directory. It contains a list of all filenames (and subdirectories) contained in a directory at a given point in time along with the file permissions for those items. A *commit* is a special artifact that contains metadata about a revision (version) of the project.

SHA1 checksums are calculated for each for each object stored in Git. These checksum values are used to express relationships as well as to locate the listed files and sub-directories in the storage area.

Over time as revisions are made to a software project (and committed to the repository) a network structure evolves such that any given version of the project consists of a combination of old and new blob and tree references.\cite{14}

![Git Revisions](image2)

Figure 9 - Git Revisions
Note that content data stored in Git is usually also compressed, initially into separate loose files.

And eventually multiple loose items are combined into larger packfiles.

In spite of its sophisticated design, Git seems to be an undesirable choice for preservation storage of digital objects. Not only does it alter the contents of files in several ways, it also has repeatedly been observed to suffer from poor performance when dealing with large binary files. And because Git is a decentralized version control system, one needs to have a local copy of a full object and all its versions before commits of new versions can be made. We need to be able to transmit only a version’s changed files to the storage layer.
3.5.2 Boar

Boar\textsuperscript{151} is being developed by an author who felt that none of the existing open source backup and version control systems were a good fit for the management of collections of binary files. This software has a simple, but effective, storage design.

In the Boar design, each repository (storage node) has a top-level *blobs* directory that contains the entire repository's content files, each of which has been renamed using the value of the file's MD5 checksum. A simple hierarchy is used to subdivide the blobs folders in a manner similar to a Git repository.

<table>
<thead>
<tr>
<th>repository</th>
<th>blobs</th>
<th>bc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>bc7b0fb8c2e096693acacbd6cb070f16</td>
</tr>
<tr>
<td>sessions</td>
<td></td>
<td>sessions</td>
</tr>
<tr>
<td></td>
<td>bloblist.json (map between filename paths and checksums)</td>
<td>session.json (metadata about the snapshot)</td>
</tr>
<tr>
<td>&lt;snapshot name&gt;</td>
<td></td>
<td>&lt;snapshot name&gt;</td>
</tr>
<tr>
<td></td>
<td>bloblist.json</td>
<td>session.json</td>
</tr>
</tbody>
</table>

The application uses a forward-delta design with a linked-list aspect, wherein a series of *session* snapshots are stored in a flat structure under a top-level *sessions* directory. Each snapshot contains only the most recent changes to a filesystem object being backed up, plus a pointer to the previous snapshot of the given session.

It was somewhat challenging for me to figure out how this application could be applied to the preservation of digital objects. Boar's vocabulary of session, blob, and snapshot can be correlated to our terminology of digital object, data file, and version, but the need to create mental mappings between these concepts makes discussion of operations and storage somewhat confusing.

The main point to observe is that all of an object's files are stored in a pooled location that is content-addressable via file checksum values, and that the history of changes to the object are stored in a sequence of version deltas (the chain of bloblist.json files).

Although closer in functionality to SDR's needs than Git, and a source of design insights, Boar in its present incarnation does not seem to be a good fit for our repository. The software is designed for incremental backup of a working directory to a repository location on the same machine, which is a different concept from explicit archiving of a digital object version to a secure location. The pooling of all files in a single area is appealing on one level, but is not compatible with our requirement to segregate a new version's files in a way that simplifies replication.
3.6 Summary of Versioning Approaches
Here are the pros and cons of each of the versioning approaches we have examined so far.

<table>
<thead>
<tr>
<th>Versioning Mechanism</th>
<th>File Fixity</th>
<th>No File Duplicates</th>
<th>Replication Efficiency</th>
<th>Version Reconstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward-Delta</td>
<td>Good</td>
<td>Fair (file renames)</td>
<td>Good</td>
<td>Good for first Poor for latest</td>
</tr>
<tr>
<td>Reverse-Delta (CDL ReDD)</td>
<td>Good</td>
<td>Fair (file renames)</td>
<td>Poor (latest version is always full)</td>
<td>Good for latest Poor for earlier</td>
</tr>
<tr>
<td>CAS (Git)</td>
<td>Poor (headers &amp; pack files)</td>
<td>Good</td>
<td>Poor (pooled data)</td>
<td>Good</td>
</tr>
<tr>
<td>CAS (Boar)</td>
<td>Fair (MD5)</td>
<td>Good</td>
<td>Poor (pooled data)</td>
<td>Good for first Poor for latest</td>
</tr>
</tbody>
</table>

As you can see, there is no clear winner, but each of these system shines in at least one aspect. Note that a row in the table is included for Forward-Delta, but I have not examined any implementations that utilize that strategy.

3.7 Tape Archive Efficiency vs. Versioning Design
In many of the versioning approaches we have discussed, addition of a new object version could trigger duplicate tape storage of many or most of an object’s files, requiring a proportionately higher consumption of system and network I/O resources. This inference is based on the assumption that tape archiving will use replication at the directory level.

In the CDL reverse-delta (ReDD) design, the directory for an object’s latest version always contains copies of all files comprising that new version. Since this is in essence a new full object, the tape archive mechanism will copy all those files, even if they were previously archived from another location. Also, since the previous version has been converted from a full version to a reverse-delta version, that previous version must be re-archived as well.

In the Git and Boar designs, all of an object’s files are pooled in a common data store. Changing the content of this pool by addition of new files would likely require that a new tape archive be made of the common pool.

The forward-delta approach minimizes the volume of files that would need to be archived to tape. The new folders that are created only contain new files, and the previously existing folders of the object are not altered by versioning changes.

So it looks like the ideal would be to come up with a design that combines the replication efficiency of the forward-delta approach with the reconstruction efficiency of the content-addressable approach.
4 The Moab\textsuperscript{[16]} Design

Stanford has developed a storage architecture, dubbed the Moab Design, which aims to meet the SDR’s functional requirements by incorporating what we feel are the best aspects of the versioning mechanisms previously discussed. In this design:

- **A version inventory manifest specifies a version’s contents**
  Each object version has a full version inventory manifest that is used to specify the complete set of files that comprise a digital object version. Additional manifests are derived from the object’s complete set of version inventories: A signature catalog provides a union list of all file manifestations that have been preserved from all versions of the digital object. A version additions report lists which files were added to the object’s catalog in a given version. And a file inventory differences report shows the details of the changes that occurred between any two given versions.

- **Fixity data is used for file deduplication**
  At least two of MD5, SHA1, or SHA256 hash values are generated for each file ingested. These checksums are used along with the file size to determine whether a newly submitted file is a duplicate of another file in the same version or is an exact copy of a file that was previously ingested. This deduplication is facilitated by the use of a master file signature catalog that collects this data across versions. This file also contains the SDR storage location of all the object’s files making it possible to quickly reconstruct any version of an object using that version’s full inventory together with the catalog.

- **Files are treated as immutable**
  The application layer does not modify the internal bytestreams of files in the preservation system. Whatever benefit might be obtained performing any sort of file compression or other alteration of curated content files would be outweighed by the cost of processing cycles and the risk of data loss.

- **Files are stored using original filenames**
  A preliminary design mimicked Git and Boar by storing data files in a common pool after renaming them using the checksum value. This raised concerns about the resulting dependency on manifest files to preserve the filename information. The latest iteration of the design retains an object version’s original filenames during ingest.

- **Each version’s new files are stored in a separate data bucket**
  The CDL Dflat structure provides a good model for storage of a digital object and its versions, such as the use of v001, v002, etc. subdirectories within an object’s directory. The Moab design imitates this framework to keep each version’s new data files separate from those of previous versions. Note that a "new file" is defined in this context as a file manifestation whose checksum did not occur in any previous version of the digital object. A file instance having a filename that was present in a previous version, but which resulted from a modification of that old file will hash to a new digest value and thus be considered a new file.

- **File replication is simplified**
  The segregated storage of version data facilitates disk and tape replication with a minimum of processing and content duplication.

- **Hash Collision risk is minimized**
  Because each digital object has its own separate file storage structure (rather than a common file pool), the risk of checksum collisions is very low.
4.1 Online Storage Folder Structure
The file folder structure used for a digital object is similar in appearance to the CDL Dflat design.

```plaintext
| ab123cd5678 |
| v0001 |
| data |
| content |
| title.jpg |
| intro.jpg |
| page1.jpg |
| page2.jpg |
| page3.jpg |
| metadata |
| versionMetadata.xml |
| descMetadata.xml |
| identityMetadata.xml |
| manifests |
| versionInventory.xml |
| signatureCatalog.xml |
| versionAdditions.xml |
| fileInventoryDifference.xml |
| manifestInventory.xml |

| v0002 |
| data |
| content |
| page2.jpg |
| metadata |
| versionMetadata.xml |
| technicalMetadata.xml |
| manifests |
| versionInventory.xml |
| signatureCatalog.xml |
| versionAdditions.xml |
| fileInventoryDifference.xml |
| manifestInventory.xml |
```

4.1.1 Object Folder
Each digital object is stored in a separate filesystem directory named using the object ID. Within the repository as a whole, the path to an object is a tree structure derived by segmenting the object ID, such as `/ab/123/cd/4567/ab123cd4567`.

4.1.2 Version Folder
Each object-level folder is subdivided into version folders, named `v0001`, `v0002`, etc. The `v0001` folder holds the initial version of the digital object. Additional sub-directories of the form "v<nnn>" hold the files of subsequent versions of the object. The directory names are left padded with zeros so that the directory pathname is 5 characters long and will sort into the correct chronological sequence. Each version folder contains a `data` folder and a `manifests` folder.

4.1.3 Data Folder
The first version’s `data` folder will usually contain the majority of the object’s files. The `data` folder of subsequent versions will contain only those new files that were added in that version or which resulted from modifications of an older version’s files. In the SDR this data folder is subdivided into content and metadata subdirectories.
4.1.4 Manifests folder

Each version has a manifests folder containing file inventories, a location index, and version differences reports.

- The `versionInventory.xml` manifest is used to record the complete set of files that comprise a digital object version. This information is required to reconstruct the full version of the object.
- The `signatureCatalog.xml` manifest provides a union list of files from the current and all previous object versions. It serves as a lookup table mapping file signature to storage location.
- The `versionAddtions.xml` manifest lists the subset of files that were added to the object’s data folder in the current version.
- The `fileInventoryDifferences.xml` report documents the details of all changes that occurred between the current and previous version.
- The `manifestInventory.xml` file holds fixity information about the other manifest files.

The Moab manifests all contain size and checksums values for each of the files that are listed in them. The manifests therefore serve a fixity verification role similar to the manifest files that are a part of the BagIt specification. The `versionAddtions` manifest is most similar to BagIt's `manifest-xxx.txt" tag files used for payload verification, and the `manifestInventory` is similar in function to the `tagmanifest-xxx.txt` files that list checksums for the other tag files. But this is where the similarity ends. The Moab manifests have a structure in which the file fixity or signature is the key to a file’s identity, instead of the filename.
4.2 File Manifestations

The Moab Design is based in large part on the concept of a file manifestation (in the FRBR\cite{18} sense of manifestation). In this view, a file's uniqueness is defined by its bytestream content, which will have a unique signature (= file fixity). Multiple file instances might share the same content (and thus signature) even if they do not share the same name (filesystem path) and timestamp. In the version inventory files used in the Moab Design, a file manifestation consists of a combination of a signature and at least one file instance.

![Figure 12 - File Manifestation](image)

Above is a visual depiction of a file manifestation having one instance. Below is an example of a manifestation having 2 file instances.

![Figure 13 - File with 2 Instances](image)

This latter file manifestation would be represented in a version inventory by the following xml stanza:

```xml
<file>
  <fileSignature size="32915"
    md5="c1c34634e2f18a354cd3e3e1574c3194"
    sha1="8a354cd3e3e17328c364b2ea0b4a79c507ce915ed"/>
  <fileInstance path="Resume.txt" datetime="2012-01-01"/>
  <fileInstance path="Resume-copy" datetime="2012-03-04"/>
</file>
```
4.3 A Versioning Example
The figures below illustrate the basic logic and relationships of core components of the Moab Versioning Design.

In the first diagram, version 1 of an object has been ingested into the repository storage area. The v0001/data/content folder contains the physical files. The v0001/manifests/versionInventory.xml file contains a list of all the files in this version together with the signature of each of those files. The v0001/manifests/signatureCatalog.xml file contains a mapping from each file signature to the physical location in which the file is stored.

Figure 14 - Version 1
In version 2 of the object, one file (Intro) is removed, and the content of Page1 is modified. There is only one new physical file (Page1) which is placed into /v0002/data/content. The v0002/manifests/versionInventory.xml file is written to record the new composition of the object, and the v0002/manifests/signatureCatalog.xml file is extended to document the storage location of the new Page1 file. The arrows show the pathway used to reconstruct the full version 2 object.

Finally, a version 3 is added in which a new Page3 is inserted in the file sequence, with a domino effect that the old Page3 must be renamed to Page4. The version inventory of the new version doesn't need to know any of the previous history. It simply documents which file name/signature combinations define the new version. As before, the signature catalog only needs to be updated with the signature/location data for the newly added Page3 file. Reconstruction of version 3 remains a simple task of looking up storage locations via the signature catalog.
5 The Moab-Versioning Gem

A source code library has been written which implements a set of tools which can be used to create, parse, and work with Moab manifest files; and facilitates the storage of digital object versions in a filesystem structure. This library has been written in the Ruby programming language, and is used internally by the Stanford Digital Repository. The code project has been published on Github at https://github.com/sul-dlss/moab-versioning

API documentation can be viewed at
http://rubydoc.info/github/sul-dlss/moab-versioning/master/frames

The remainder of this section reviews the classes provided by this code library, providing more detailed descriptions and examples of the various Moab manifest files. The manifest field names are mostly self-explanatory, but you can look up additional definitions via the above links.

For examples of code that interacts with manifest files and storage structures, see the rspec files at: https://github.com/sul-dlss/moab-versioning/tree/master/spec/unit_tests
If you download the project and run the rspec test suite, then the fixtures folder of the spec directory will be completely filled out with data files that can be examined for more insight.

This gem is still being packaged and hosted internally at Stanford University Libraries, but if a community of users were to coalesce, then we could change that to use rubygems.org.
5.1 Moab Manifests

5.1.1 The Version Inventory

Each digital object version has a version inventory (serialized as versionInventory.xml) that is an instance of the FileInventory schema wherein the type attribute is set to "version" (this schema is shared by several document types). Every version inventory contains a complete list of all files comprising the specified version of a digital object. In the SDR examples below, the data storage area is divided into content and metadata subdirectories, which is mirrored by the content and a metadata fileGroups in the inventory. Within each fileGroup is an array of file manifestation (<file>) elements, each of which represents a point-in-time snapshot of a given file's properties. The fixity data for the file is stored in a fileSignature element, while the filename and last modification date are stored in one or more fileInstance elements (since copies of a given file may be present in multiple locations within an object).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fileInventory type="version" objectId="druid:jq937jp0017" versionId="3" inventoryDatetime="2012-04-19T12:12:48Z"
  fileCount="7" byteCount="187851" blockCount="191">
  <fileGroup groupId="content" dataSource="/Users/rnanders/Code/Ruby/moab-versioning/spec/fixtures/data/jq937jp0017/v3/content"
    fileCount="5" byteCount="171902" blockCount="170">
    <file>
      <fileSignature size="32915" md5="c1c34634e2f18a3543c3e1574c1914" sha1="0616a08d9279238c306b20a0b79c507ce915ed"/>
      <fileInstance path="page-1.jpg" datetime="2012-03-26T15:35:15Z"/>
    </file>
    <file>
      <fileSignature size="39539" md5="fe6e3ef34634e2f18a3543c3e1574c1914" sha1="43ced73d81687bc8e6f483618f0d6c7f7665e06a7"/>
      <fileInstance path="page-2.jpg" datetime="2012-03-26T15:35:15Z"/>
    </file>
    <file>
      <fileSignature size="39450" md5="82fc107c88446a3119a51a8663dfe955" sha1="d0857baf307a2e9eff42467b5ab4e1c0f0c05"/>
      <fileInstance path="page-3.jpg" datetime="2012-03-26T15:35:15Z"/>
    </file>
    <file>
      <fileSignature size="39450" md5="82fc107c88446a3119a51a8663dfe955" sha1="d0857baf307a2e9eff42467b5ab4e1c0f0c05"/>
      <fileInstance path="page-4.jpg" datetime="2012-03-26T15:35:15Z"/>
    </file>
    <file>
      <fileSignature size="39450" md5="82fc107c88446a3119a51a8663dfe955" sha1="d0857baf307a2e9eff42467b5ab4e1c0f0c05"/>
      <fileInstance path="page-5.jpg" datetime="2012-03-26T15:35:15Z"/>
    </file>
  </fileGroup>
  <fileGroup groupId="metadata" dataSource="/Users/rnanders/Code/Ruby/moab-versioning/spec/fixtures/data/jq937jp0017/v3/metadata"
    fileCount="2" byteCount="15949" blockCount="21">
    <file>
      <fileSignature size="30466" md5="a60bb487db0a1384e5b5b3ace2df2a" sha1="d0857baf307a2e9eff42467b5ab4e1c0f0c05"/>
      <fileInstance path="descMetadata.xml" datetime="2012-03-26T14:43:42Z"/>
    </file>
    <file>
      <fileSignature size="12903" md5="cc6b5f2e8e3c6d0b89692fa1e0145" sha1="3badb006aef40f14e4664d2594c606b9e9176"/>
      <fileInstance path="identityMetadata.xml" datetime="2012-03-26T14:44:43Z"/>
    </file>
  </fileGroup>
</fileInventory>
```

A fileGroup to be added to a version inventory can be generated by traversing a directory that contains the files being inventoried. This capability is provided by the FileGroup#group_from_directory method, which is also utilized by the FileInventory#inventory_from_directory method. A special case of the latter method will create both a content and a metadata fileGroup from a directory that contains two subdirectories having those names. A fileGroup can also be generated by transformation of a local file inventory in another format. At Stanford a contentMetadata datastream contains all the information needed to create the content fileGroup. A method to perform this transformation is provided in the Stanford::ContentInventory# group_from_cm method.
5.1.2 The Signature Catalog

A digital object’s signature catalog (serialized as signatureCatalog.xml) is derived from a filtered aggregation of all the version inventories of a digital object. It has a file signature entry for every file manifestation ever ingested for the object, along with a record of the SDR storage location that was used to preserve a copy of the manifestation. Once this catalog has been populated, it has multiple uses:

- An index of file signatures is used to determine which files of a newly submitted object version are duplicates of files previously ingested. (When a new version contains a mixture of new files and files carried over from the previous version we only need to store the files from the new version that have unique file signatures.)
- Reconstruction of an object version requires a combination of the version’s file inventory and signature catalog.
- The catalog can also be used for performing consistency checks of repository storage files.

A signature catalog is created by using the SignatureCatalog.new method and assigning the digital object ID. The SignatureCatalog#update method is run against the first version’s file inventory, which generates an entry in the catalog for each unique file. When a subsequent version of the digital object is deposited in the repository, the #update command is again run, but this time it only creates new entries for files whose signatures were not already present in the catalog.

Each new version’s manifests folder holds an updated copy of the signatureCatalog.xml file that was created when that new version was ingested.

5.1.3 Version Additions Report

A version additions report is an instance of the FileInventory schema (with type="additions") that lists which files manifestations were newly added to a given version. Because it is generated by applying a
new version’s file inventory against the previous version’s signature catalog, it is of secondary importance as a preservation artifact. It serves the following purposes:

- Provides a human-readable summary of new files
- Provides a file manifest for use in validating checksums of a given version’s storage files.

```xml
<fileInventory type="additions" objectId="druid:jq937jp0017" versionId="2" inventoryDatetime="2012-04-19T12:48Z"
    fileCount="2" byteCount="34050" blockCount="37">
  <fileGroup groupId="content" dataSource="" fileCount="1" byteCount="32915" blockCount="33">
    <file>
      <fileSignature size="32915" md5="c1c34634e2f18a354cd3e3e1574c319d" sha1="0616a0bd7927328c364b2ea04a79c507ce915ed"/>
      <fileInstance path="page-1.jpg" datetime="2012-03-26T15:35:15Z"/>
    </file>
  </fileGroup>
  <fileGroup groupId="metadata" dataSource="" fileCount="3" byteCount="2669" blockCount="4">
    <file>
      <fileSignature size="1135" md5="d74bfa778653b6c1b285b2d0c2f07c5b" sha1="0ee15e133c17ae3312b87247adb31b0b837ca3df"/>
      <fileInstance path="technicalMetadata.xml" datetime="2012-03-28T14:04:24Z"/>
    </file>
  </fileGroup>
</fileInventory>

A version additions report is created by using the SignatureCatalog#additions method of the previous version's catalog with the new version's file inventory as a input parameter.
5.1.4 File Inventory Differences

A file inventory differences report compares two FileInventory instances based primarily on file signatures and secondarily on file pathnames. Although the usual use will be to compare the content of 2 different temporal versions of the same object, it can also be used to verify a manifest document against an inventory harvested from a directory. The report is subdivided into sections for each of the file groups that compose the inventories being compared.

```xml
<fileInventoryDifference objectId="druid:jq937jp0017" differenceCount="5" basis="v1" other="v3" reportDatetime="2012-04-19T12:12:48Z">
  <fileGroupDifference groupId="content" differenceCount="4" identical="2" renamed="1" modified="1" deleted="1" added="1">
    <subset change="identical" count="2">
      <file change="identical" basisPath="title.jpg" otherPath="same">
        <fileSignature size="40873" md5="1a726cd7963bbdd6ceabf8335ce1e66" sha1="583220e05726a0b6d3dd9739d224ae863a6ad"/>
      </file>
      <file change="identical" basisPath="page-2.jpg" otherPath="same">
        <fileSignature size="19125" md5="a5099878d4e264432d6df44ca8827" sha1="c0ccac433cf82a6cee98c9f9ab6872a1444ca2"/>
      </file>
    </subset>
    <subset change="renamed" count="1">
      <file change="renamed" basisPath="page-3.jpg" otherPath="page-4.jpg">
        <fileSignature size="39450" md5="82fc107ca8446a31195a5ba663e1e955" sha1="d0857baa307a2e9ef4247b5ab41e1cf480fcd"/>
      </file>
    </subset>
    <subset change="modified" count="1">
      <file change="modified" basisPath="page-1.jpg" otherPath="same">
        <fileSignature size="25153" md5="3dee12fbb4f1c283517c48b7f6ff56a8" sha1="a6cc342f2f83ae55ce15743139d"/>
        <fileSignature size="34538" md5="c1c34634e62f0a354c3de3e15743139d" sha1="0616a0d0792728c36a4b2aa79b507c9e15ed"/>
      </file>
    </subset>
    <subset change="deleted" count="1">
      <file change="deleted" basisPath="intro.jpg" otherPath="">
        <fileSignature size="41981" md5="915c035b585513f1586295dc122" sha1="6044b056b60997f9ce6a65db4acef915178"/>
      </file>
    </subset>
    <subset change="added" count="1">
      <file change="added" basisPath="page-3.jpg" otherPath="">
        <fileSignature size="39539" md5="fe6e3faab02ced8b5d6a0a2c4a1e955" sha1="43c7636b167b8ce6f4f83618f0d6c7f766e80b7"/>
      </file>
    </subset>
  </fileGroupDifference>
  <fileGroupDifference groupId="metadata" differenceCount="1" identical="2" renamed="0" modified="1" deleted="0" added="0">
    <subset change="identical" count="2">
      <file change="identical" basisPath="descMetadata.xml" otherPath="same">
        <fileSignature size="3046" md5="a60b487da0aac0b60b5b3caedf2a" sha1="e9edc0e1d7c7f0b3d7c7b63b20d2c"/>
      </file>
      <file change="identical" basisPath="identityMetadata.xml" otherPath="same">
        <fileSignature size="12903" md5="cc85b72ae5c26aad06b09e2a6a01e45" sha1="3b9d0b6ad64e3e259ae6b09e7116"/>
      </file>
    </subset>
    <subset change="renamed" count="0"/>
    <subset change="modified" count="1">
      <file change="modified" basisPath="technicalMetadata.xml" otherPath="same">
        <fileSignature size="1619" md5="b98ecb8d4508884156b9160b9f840f0" sha1="b2328faaf25ca8df37cfc026386ad707c3a47a7"/>
      </file>
    </subset>
    <subset change="deleted" count="0"/>
    <subset change="added" count="0"/>
  </fileGroupDifference>
</fileInventoryDifference>
```

To generate a file inventory differences report, use the FileInventoryDifference#compare method, supplying it with two file inventories as parameters.

A convenience method, FileInventoryDifference#compare_with_directory, provides a simple interface for comparing an existing inventory against a directory containing physical files.

5.1.5 Algorithm Used to Compare File Inventories

In the diagram below, the word "Manifest" is used as a synonym for file inventory.
In the Moab design, the inventories for two different versions contain no indication of the history that occurred between those two versions. The differences must therefore be inferred by performing a comparison analysis between two version inventories.

This comparison is a 2-step process. We first compare the sets of signatures present in the two inventories, determining which signatures are present in both versions and which others are unique to either the old version or the new version.

Finally, the lists of signatures that are the output of the first step of the comparison are further analyzed to report the detailed nature of the differences between the two versions of an object:

A. If two files have the same signature in both manifests, and the same filename, then we consider the files to be identical.
B. If two files have the same signature in both manifests, but the filenames differ, then we conclude that a file rename has occurred.
C. If a signature only occurs in the newer manifest, and the filename is unique to that version, then we conclude that this is a newly added file.
D. If a signature only occurs in the newer manifest, but the filename was associated with a different signature in the older manifest, then we conclude that the file was edited or otherwise modified.
E. If a signature only occur in the older manifest, and the filename only exists in that manifest, then we conclude that the file was deleted.

Note that this logic gets more elaborate if there are multiple file instances within an object version that share the same checksum value.
5.1.6 Version Metadata

At the SDR a versionMetadata.xml file for each version is stored as part of the digital object metadata (and not in the manifests folder). It records simple descriptive information about a digital object’s collection of versions and is updated for each new version. The example presented here shows the simple schema adopted by Stanford. It is not an essential part of the Moab design, but is used at SDR as a place to record an optional tag value and a reason for the creation of the version.

```xml
<versionMetadata objectId="druid:ab123cd4567">
  <version versionId="1" tag="1.0">
    <description>Initial version</description>
  </version>
  <version versionId="2" tag="2.0">
    <description>Replacing main PDF</description>
    <note>Replaced at student request to fix citation errors</note>
  </version>
  <version versionId="3" tag="2.0.1">
    <description>Fixed title typo</description>
  </version>
</versionMetadata>
```

5.2 Moab Storage Objects

The Moab-Versioning gem also provides classes that can be used to represent the levels of an online folder hierarchy that can be used for storage and retrieval of digital object files and manifests.

The Class types that are provided include:

- StorageRepository, which represents a digital object repository storage node
- StorageObject, which represents a digital object’s repository storage location.
- StorageObjectVersion, which represents a version subdirectory within an object’s home directory.
- StorageServices, which supports application layer access to the repository’s objects, data, and metadata.
- A Bagger utility for creating Bagit packages for ingest or dissemination.

See http://rubydoc.info/github/sul-dlss/moab-versioning/master/Moab/StorageRepository for more details
6 The SDR Preservation Workflow

We’ll end this tour of the Moab Versioning Design by describing the steps of the SDR accessioning and ingest workflow to illustrate an example of how a real-world implementation of this design is being utilized at Stanford for the long-term storage of digital objects.

### accessionWF Workflow Steps (DOR)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start-accession</td>
<td>Initiate the workflow</td>
</tr>
<tr>
<td>content-metadata</td>
<td>Verify presence of, or create, contentMetadata datastream</td>
</tr>
<tr>
<td></td>
<td><em>(This becomes the content fileGroup in the version inventory)</em></td>
</tr>
<tr>
<td>descriptive-metadata</td>
<td>Verify presence of, or create, descMetadata datastream</td>
</tr>
<tr>
<td>rights-metadata</td>
<td>Verify presence of, or create, rightsMetadata datastream</td>
</tr>
<tr>
<td>technical-metadata</td>
<td>Verify presence of, or create, technicalMetadata datastream</td>
</tr>
<tr>
<td></td>
<td><em>(Uses a version comparison to determine what files need characterization)</em></td>
</tr>
<tr>
<td>provenance-metadata</td>
<td>Verify presence of, or create, provenanceMetadata datastream</td>
</tr>
<tr>
<td>shelve</td>
<td>Transfer content to be held online to the Digital Stacks</td>
</tr>
<tr>
<td></td>
<td><em>(Uses a version comparison to determine what files are changed)</em></td>
</tr>
<tr>
<td>publish</td>
<td>Publish shared, public metadata to the Digital Library</td>
</tr>
<tr>
<td>sdr-ingest-transfer</td>
<td>Bag content and metadata file for export to preservation core ingest</td>
</tr>
<tr>
<td></td>
<td><em>(Heavy use of Moab versioning toolkit to record the version inventory, figure out which files have new signatures, then package only those files for export)</em></td>
</tr>
<tr>
<td>sdr-ingest-received</td>
<td>Callback from preservation core ingest indicating ingest completed</td>
</tr>
<tr>
<td>end-accession</td>
<td>Complete the accession workflow</td>
</tr>
</tbody>
</table>

### sdrIngestWF Workflow Steps (Preservation Core)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start-ingest</td>
<td>Initiate the workflow</td>
</tr>
<tr>
<td>register-sdr</td>
<td>Ensure that there is a Fedora entry for the object</td>
</tr>
<tr>
<td>transfer-object</td>
<td>Copy the object package from the export location</td>
</tr>
<tr>
<td>validate-bag</td>
<td>Verify package structure and checksums</td>
</tr>
<tr>
<td>populate-metadata</td>
<td>Update Fedora datastreams as needed</td>
</tr>
<tr>
<td>verify-agreement</td>
<td>Ensure that the object’s governing policy object has already been ingested</td>
</tr>
<tr>
<td>complete-deposit</td>
<td>Create the online storage folders for the object and/or version. Verify file checksums and the ability to reconstruct the full version.</td>
</tr>
</tbody>
</table>

The package that is exported from the DOR system is a normal BagIt bag containing content, metadata, and manifest files. The bag is transferred to the Preservation Core system using rsync, then validated a first time, before being used as the source for creating either a new digital object location in the storage system, or a new version location within a previously created digital object store. The steps are:

- create the new version’s storage folder
- copy the version’s data files into storage
- update the object’s signature catalog
- generate a version differences report
- generate an inventory of all the manifest files
7 Conclusion

The Moab Design for digital object versioning was born out of an urgent need to figure out an efficient, practical, and robust mechanism for making changes to digital objects that had already been ingested. The need for a versioning capability was holding us back from fixing some problems that had already occurred, or opening up the repository door to new sources of documents that we anticipated might get modified in a relatively short time.

Our archival philosophy restrains us from simply overwriting an old version of an object with a new one. We wish to retain all old copies of files in case a replacement proves worse than the original, or so that we can look back an object’s history for whatever reason might arise.

The nature of a preservation repository turns out to be different enough from that of a software change management system or file backup system that some technologies which initially appeared promising (such as Git and Boar) turned out to have design aspects or behaviors that ruled them out as solutions that we could adopt off of the shelf. The CDL Micro-Services approach had the best promise and well-written specifications, but we were disappointed when we realized that the ReDD reverse-delta mechanism would likely prove too resource consumptive for us. Inspiration was derived, however, from all of those designs and the result of assembling pieces from each approach into a new framework has yielded rewarding results.

Recapping the criteria used in section 3.6 to evaluate those other technologies, here is how the Moab design measures up:

- **File Fixity: Good**
  Moab does not alter any content of object files so that the fixity remains consistent throughout the system. Manifests record file sizes and checksums for use in verification.

- **No File Duplicates: Good**
  By using file signatures as the key to file identity, Moab guarantees that only one copy of any give file manifestation is stored at any given storage or replication location.

- **Replication Efficiency: Good**
  For the same reason, the transmission, processing, and storage of a new version’s files only needs to involve the files that were newly introduced in that version.

- **Version Reconstruction: Good**
  Any version of a digital object can be reconstructed and disseminated with equal ease to any other version.

SDR welcomes feedback and collaboration so that we can identify any deficiencies in our design or move the development forward with new useful enhancements.
8 Works Cited


End Notes

[3] https://confluence.ucop.edu/display/Curation/Storage
[5] https://confluence.ucop.edu/display/Curation/ReDD
   http://valerieaurora.org/review/hash2.pdf
   http://www.nmt.edu/~val/review/hash2.pdf
[10] I have not found a single authoritative reference that covers the issue of hash algorithm security and collision probabilities, but there are many well written commentaries to be found on the StackOverflow web site:
   http://stackoverflow.com/questions/800685/which-cryptographic-hash-function-should-i-choose
   http://stackoverflow.com/questions/2479348/is-it-possible-to-get-identical-sha1-hash
   http://security.stackexchange.com/questions/19705/is-sha1-better-than-md5-only-because-it-generates-a-hash-of-160-bits
Illustrations from the "Git Community Book" used by permission of Scott Chacon
Illustrations from "Power Your Workflow with Git" used by permission of Patrick Hogan
[16] The "Moab" name is not really an acronym, but is the name of the town in which this author is living. I have come up with some acronym expansions, however, for the fun of it: Multi-version Object Archive Bags; Make Objects Act Better; My Own And Best