Digital Object Storage and Versioning in the Stanford Digital Repository

Richard N Anderson
Digital Library Systems and Services
Stanford University Libraries and Academic Information Resources
18 January 2012

Table of Contents

1 Introduction 3
1.1 Scope of this Document 3
1.2 The Stanford Repository Architecture 3
1.3 Role of Fedora 3
1.4 Digital Object Diversity 3
1.5 The Need for Versioning 4

2 Functional Requirements 4
2.1 Identity Requirements 4
2.1.1 Object Identifiers 4
2.1.2 Version Identifiers: 4
2.2 Modification Requirements 5
2.2.1 Atomic Operations 5
2.2.2 Composite Transactions 5
2.3 Accessioning Requirements 5
2.3.1 Submit full version 5
2.3.2 Submit delta changes 5
2.4 Retrieval Requirements 5
2.4.1 Retrieve any version 5
2.4.2 Retrieve any portion of an object 5
2.5 Fidelity Requirements 6
2.5.1 Preserve original filesystem properties 6
2.5.2 File content should not be modified by compression or headers 6
2.5.3 Support message digests for fixity checking 6
2.6 Efficiency Requirements 7
2.6.1 Must support efficient storage of large binary files 7
2.6.2 Must support storage of all media types including image and video 7
2.6.3 Minimize duplicate storage of identical file content 7
2.7 Replication Requirements 7
2.7.1 Facilitate copying of object store to another disk location 7
2.7.2 Facilitate creation of tape copies using Tivoli Storage Manager (TSM) 7
2.8 Cost Requirements 8

3 Versioning Approaches 8
3.1 Whole Object Versioning 8
3.2 Delta Versioning 9
3.3 Forward-Delta Versioning 9
3.4 Forward Delta Examples 9
3.5 Reverse-Delta Versioning 11
3.6 Content-Addressable Storage (CAS) 12

3.6.1 Using Checksums for CAS ... 12
3.6.2 Cryptographic hashes as Checksums ... 13
3.6.3 What About Collisions? ... 13

4 Versioning Implementations ... 14
4.1 CDL ReDD design ... 14
4.1.1 ReDD Process for a new version ... 15
4.1.2 CDL MicroService Concerns ... 15
4.2 Git ... 16
4.2.1 Object Model ... 16
4.2.2 Git Concerns ... 17
4.3 Boar ... 17
4.3.1 Boar Storage structure ... 18
4.3.2 Boar's version manifest ... 18
4.3.3 Boar Concerns ... 19
4.4 Summary of Desired Features ... 19

5 Moab Design ... 20
5.1 Moab Folder Structure ... 21
5.1.1 Data folder ... 21
5.1.2 Manifest file ... 21
5.1.3 Impact on replication ... 21
5.1.4 Hash collision risk ... 21
5.1.5 File renaming risk ... 21
5.2 Versioning Examples ... 22
5.3 Version Manifest ... 23
5.4 File Map ... 23
5.5 Reconstructing an Object ... 23

6 Replication and Tape Archives ... 24
6.1 Disk to Disk Replication ... 24
6.2 Disk to Tape Replication ... 24
6.3 TSM Archive Examples ... 24
6.4 Tape Archive Efficiency vs. Versioning Design ... 25

7 Version Manifest Structure ... 25
7.1 CDL Checkm specification ... 25
7.2 SDR's versionMetadata XML ... 25

8 Content & Metadata ... 26
8.1 Depositor Submitted Files ... 26
8.2 Stanford's Repository Datastreams ... 27
8.3 Digital Object Structure ... 27
8.4 contentMetadata Datastream ... 27
8.5 contentMetadata vs versionMetadata ... 28
8.6 Building a Version Manifest ... 28
8.7 Do we still need BagIt? ... 28

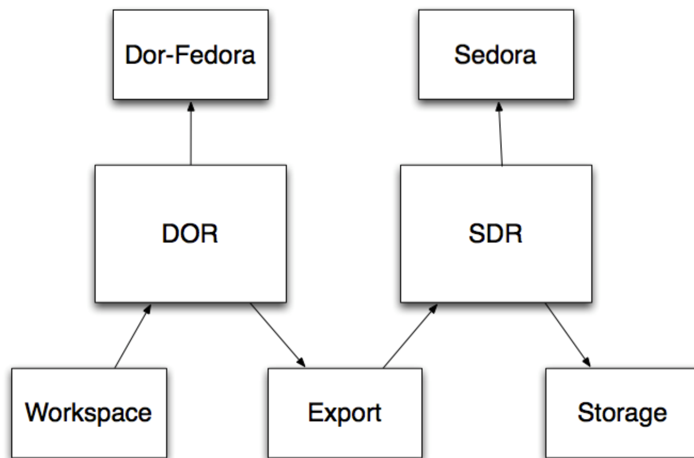
9 Ongoing Design Work ... 29
9.1 APIs and Tools for Accessioning ... 29
9.2 contentMetadata Revisions ... 29
9.3 Digital Stacks/Shelver Considerations ... 30
9.4 Versioning Workflow ... 30
9.5 Submission Package ... 30
9.6 DOR Work Steps ... 30
9.7 SDR Work Steps ... 31

1 Introduction

1.1 Scope of this Document

This presentation will work forward to arrive at a design for deep storage of digital objects (including object versions) and then look backwards at the implications of that design for the workflow used for submission and processing of objects. The architecting of processes throughout a preservation workflow will benefit from a more precise understanding of the data structures required by the later stages

1.2 The Stanford Repository Architecture



Stanford has evolved a repository architecture that actually consists of 2 interrelated repositories: DOR (Digital Object Repository) and SDR (Stanford Digital Repository). The DOR subsystem is used to accession digital objects into a temporary workspace, process the objects to add administrative metadata, and export the objects for ingest into the SDR subsystem. SDR provides a more secure deep storage system that is responsible for long-term preservation.

1.3 Role of Fedora

Both DOR and SDR utilize separate Fedora instances for administrative management of digital object metadata which takes the form of XML datastreams. Content files, however, are not stored as Fedora datastreams. Instead, content files are stored using a filesystem approach, which is currently based on a combination of a directory tree structure (to locate an object using its identifier) and Bagit (to store fixity information along with the content). SDR originally based object and file identifiers on UUIDs, but now uses a locally designed identifier called a "druid".

1.4 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 type 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.

1.5 The Need for Versioning

Some of our collections have proved to have volatile content. For example, a large number of manuscript pages have needed to be rescanned (in high-resolution mode) to correct problems that were found after initial ingest has occurred. The changeability of metadata is also likely over time. The storage system for SDR must therefore accommodate digital object versioning, and must do so in a way that efficiently processes **only** the changed files and metadata, so as to minimize resource consumption. We cannot use Fedora alone for this purpose not only because we do not store our content files as datastreams, but because Fedora's versioning support is only at the datastream level (and not at the object level). There is no simple way to ask Fedora for version #2 of a digital object.

2 Functional Requirements

The functional requirements for preservation of digital object versions are an extension of the normal requirements for preservation of an object. What is added is the need to capture, and preserve the history of changes to an object, while retaining the ability to retrieve earlier versions.

2.1 Identity Requirements

2.1.1 Object Identifiers

- primary key identifier
- alternate source ID

An object's identity is represented in the repository system by a system-generated primary key, such as "druid:ab123cd4567". In addition, the repository may store a 'source' identifier provided by the submitting agent (such as a manuscript number), which may be the key used in an external system, or some other text string that is unique in the submitter's context.

2.1.2 Version Identifiers:

- version number
- version label
- descriptive information

Similarly, each version of an object must have a unique sub-identifier, and we prefer a "natural" numbering system using sequential integers. An optional version label and/or more verbose descriptive text may also be associated with the version. One possible use of the label attribute could be to assigning a major.minor release number.

2.2 Modification Requirements

2.2.1 Atomic Operations

- add file
- delete file
- modify file contents
- rename file

Changes may occur to either content files or metadata files, or both. In either case, The modifications of a digital object that result in a new version can be viewed as a mixture of atomic add, delete, modify, or rename operations.

2.2.2 Composite Transactions

- insert file into a sequence

Some patterns of more complex changes can be viewed as a combination of add and rename operations, such as can occur when a set of files is named using a sequence of ordinal names, such as page-1, page-2, page-3, ... We already have 2 large collections whose content files are named using sequences of this sort. The insertion of a new page into the middle of such a sequence would require a cascade of file renames and a single new file addition. This is illustrated in examples which will be presented later in this document.

2.3 Accessioning Requirements

2.3.1 Submit full version

- An initial version
- A new full object version

2.3.2 Submit delta changes

- Only the changes that have occurred since the previous version

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 complete new 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. An API will need to be devised for this communication and tools implemented to enable either mode of submission.

2.4 Retrieval Requirements

The API for retrieval of a digital object should have a similar flexibility. One should be able to retrieve all or only a portion of the files from any version.

2.4.1 Retrieve any version

- latest version
- version by version number/label
- version at a point in time

2.4.2 Retrieve any portion of an object

- full version
- subset of files

2.5 Fidelity Requirements

2.5.1 Preserve original filesystem properties

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.

2.5.2 File content should not be modified by compression or headers

There should be no alteration of file content by the storage system. Any change to the internal structure of the content increases the risk of introducing changes that cannot be reversed or recovered from. File alteration also comes with a time penalty, which is especially noticeable for large files.

Although compression technology is an inherent part of many file formats (such as JPEG), the introduction of any additional file compression by the preservation system should be avoided. Any form of lossy compression is of course to be avoided, as is line-ending normalization of text files. (See discussions of Git's core.safecrlf option). And even lossless compression has been shown to exacerbate the preservation risk associated with bit rot.

A CERN study on data corruption (Panzer-Steindel 2007)[1] found a data corruption rate of one error per 30 megabytes of data in a 8.7 terabyte set of files, with 1 out of 1,500 files affected. Furthermore, the CERN study found that a single bit error would make a compressed file unreadable with 99.8% probability. According to Wright, Miller & Addis (2009)[2], 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. Heydegger (2008)[3] found that a .01% byte error rate in a compressed JPEG2000 file resulted in at least a 50% loss of information; moreover, a single byte error in a compressed JPEG2000 file could produce obvious damage throughout the image.

2.5.3 Support message digests for fixity checking

In order to confirm the fidelity of storage, it should be as convenient as possible to use checksums to verify that file corruption has not occurred. For example, some popular version control systems, such as Git, tack on a header section at the beginning of a file, turning the file into a "blob" object. Such alterations are undesirable in a preservation environment because they change the value of a file's checksum (as does file compression). The checksum for an file at the time of submission should ideally 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.

---

[1] (Panzer-Steindel 2007) Data integrity. Bernd Panzer-Steindel. April 8, 2007.
http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797

[2] (Wright, Miller & Addis 2009) The Significance of Storage in the "Cost of Risk" of Digital Preservation. Richard Wright, Ant Miller, Matthew Addis. The International Journal of Digital Curation, 4(3), 104-122. (2009) -- (also presented at iPres 2008)
http://www.ijdc.net/index.php/ijdc/article/viewFile/138/160)

[3] (Heydegger 2008) Analysing the impact of file formats on data integrity. Volker Heydegger. *Proceedings of Archiving 2008*, Bern, Switzerland, June 24-27.
http://old.hki.uni-koeln.de/people/herrmann/forschung/heydegger_archiving2008_40.pdf

2.6 Efficiency Requirements

2.6.1 Must support efficient storage of large binary files

The storage system for SDR 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 object packaging, file transfer, fixity checking, replication, and tape archiving. Greater efficiency of processing correlates with increased throughput in the ingest workflow.

2.6.2 Must support storage of all media types including image and video

A digital object containing many large files will inevitably slow down processing. We have found that objects in the 100 – 500 GB range can require an hour to perform checksum verification or file transfer. Even deleting the files in a large object is a slow process. Any additional manipulation of such large files, such as compression or containerization, will have a decidedly negative impact on throughput. There is also the risk of out of memory errors if a process requires an in-memory copy of the file.

2.6.3 Minimize duplicate storage of identical file content

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.

2.7 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 tape media (which should ideally be geographically dispersed to mitigate disaster recovery). The replication operation should consume a minimum of processing, I/O, and storage resource needed for making file copies or performing media migration.

2.7.1 Facilitate copying of object store to another disk location

We would anticipate using a standard application such as Rsync for disk-to-disk replication of digital objects. Rsync was designed to facilitate the synchronization of files and directories from one location to another with a minimum of data transfer. 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 metadata (size and date), which can take a long 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 changes to a known sub-location, reducing the number of files whose metadata need to be compared.

2.7.2 Facilitate creation of tape copies using Tivoli Storage Manager (TSM)

Replication of versioned objects onto tape is even more 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. For Stanford the emphasis on this requirement derives from our use of the TSM 'archive' command for making copies. Unlike the TSM 'backup' command, the archive command does not save only the differences between the current state of a directory structure and the previously archived instance of that structure. If you change just one file in a folder, then the normal behavior of the archive command would be to make a new copy of the complete folder. One can override this default behavior by use of

the –filelist option, which allows you to explicitly specify the list of files to be archived, but this introduces a complexity of operation and risk of error that can be avoided by isolating a new version's files in a version-dedicated folder.

2.8 Cost Requirements

- Open Source Software
- Commodity Hardware

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 finding the long-term finances for scaling those systems to the size needed for our anticipated volume of content.

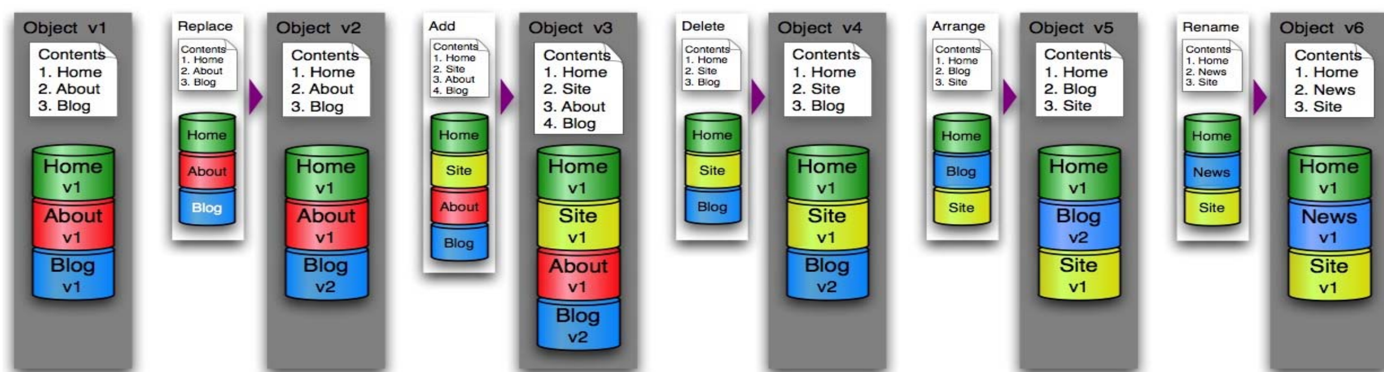
Related to this is the desire to minimize our system's dependency on vendor-supplied solutions. We should not need to recover from the bankruptcy of a vendor, or from the obsolescence of a technology.

3 Versioning Approaches

Let us now turn our attention to a review of versioning architectures and then examine some implementations that illustrate those approaches. This list is by no means exhaustive, but summarizes many of the technologies most recently 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.



Pro:

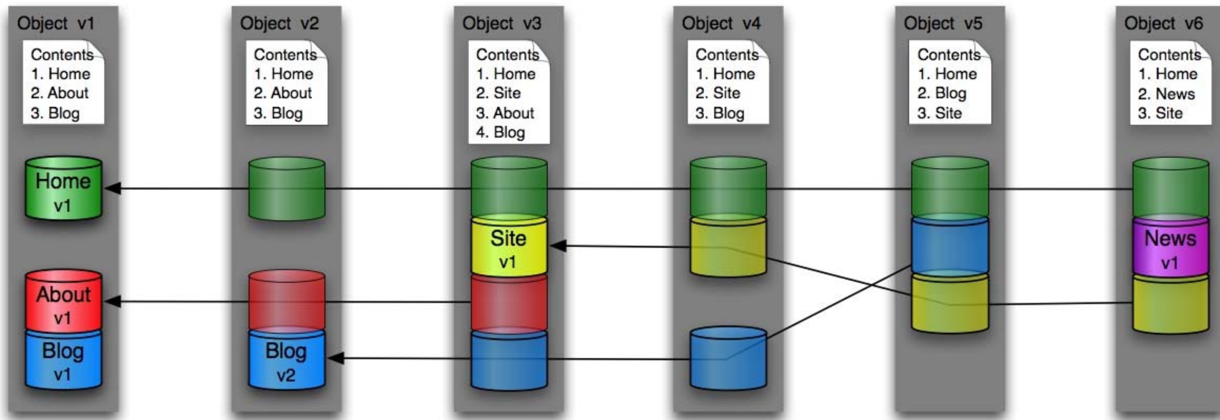
- Simple design
- Fast reconstruction of any version

Con:

- High level of file duplication
- Consumes extra resources

3.2 Delta Versioning

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



Pro:

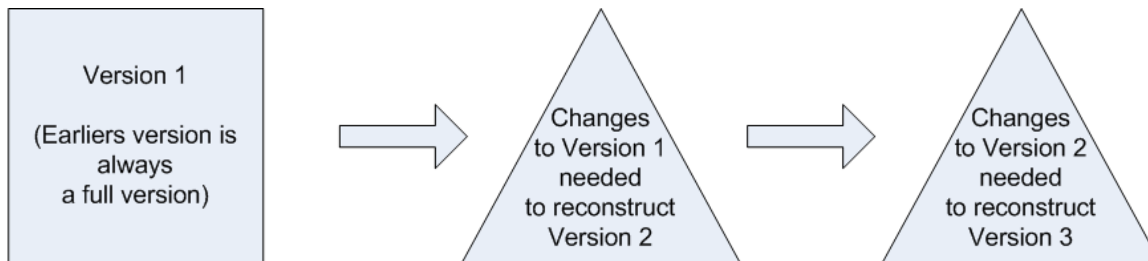
- Lower level of file duplication than the whole version option

Con:

- More complex algorithm for storage and retrieval
- Rename = delete and re-add

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 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).



Store complete file set in earliest version folder, and newer versions of files in later folders.

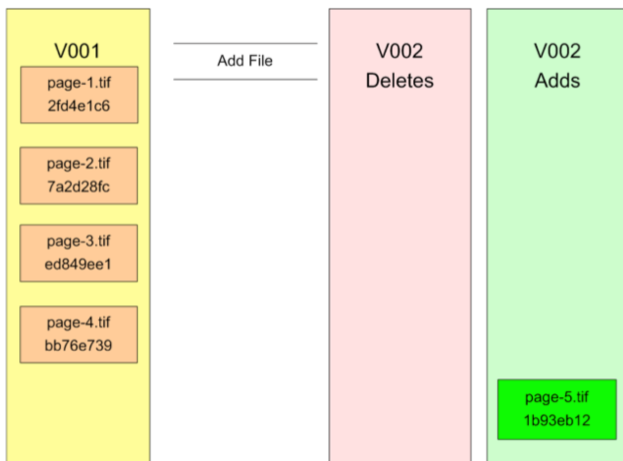
Pro:

- Less work to add a new version
- Less impact on replication and tape archive

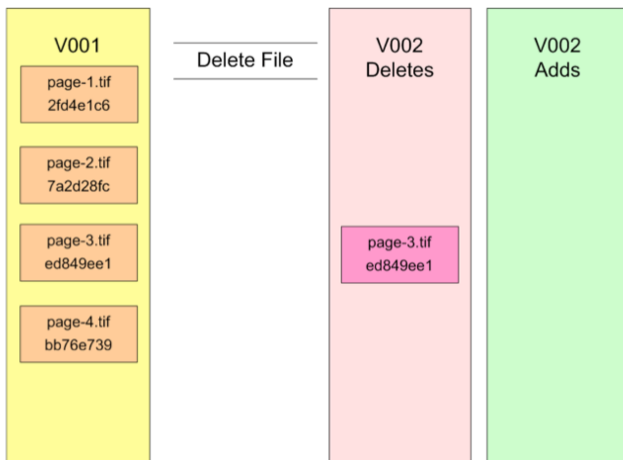
Con:

- More work to reconstruct latest version

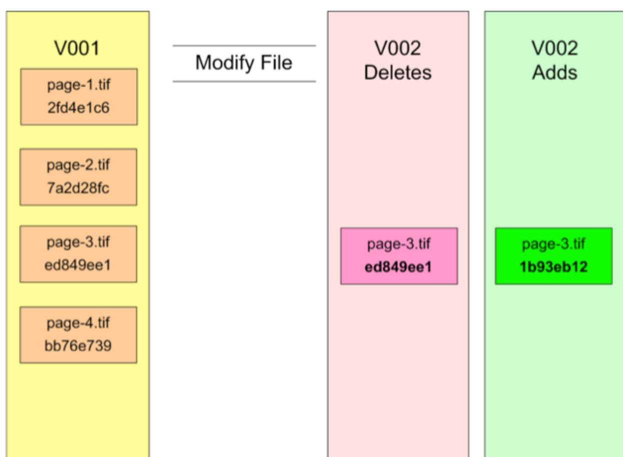
3.4 Forward Delta Examples



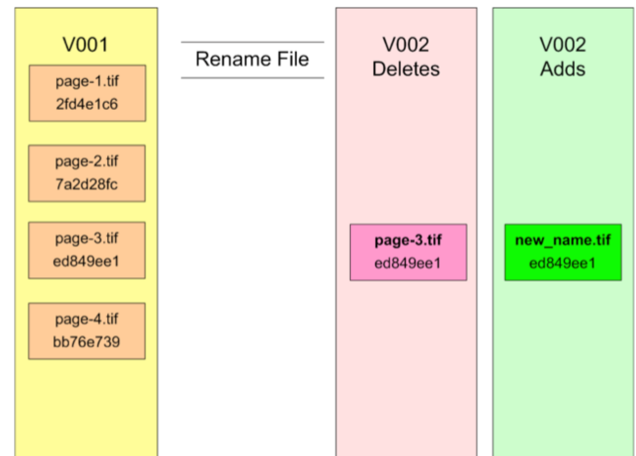
Here is an example that illustrates adding a new file (named page-5.tif). Version 1 contains all files comprising the original submission. The version 2 bucket contains only the new file.



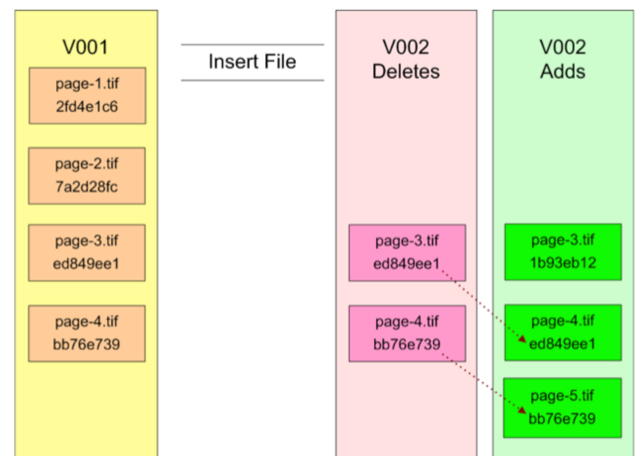
This example shows what would happen if page-3.tif were to be deleted to create version 2 of the object. In this case the only new storage data would be an entry in the deletes table indicating which file to delete.



This example show what would happen if you edited the contents of page-3.tif as part of creating a new version. The original file is marked for deletion and a new file with the same name (but different contents) is added.



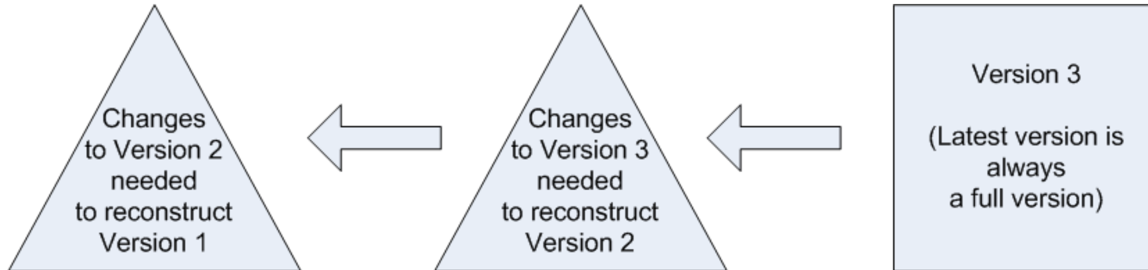
Renaming a file is also easy to demonstrate. In this example, the file page-3.tif is being renamed to new_name.tif. As in the Modify File example, the original file is marked for deletion and a new file with the new name (but same contents as the old file) would be added.



In this example we need to insert a different file as page-3.tif and move the remaining pages into new positions in a 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, then add in the new file and the renamed files.

3.5 Reverse-Delta Versioning

Reverse-delta versioning is the inverse of forward-delta versioning. In this approach, the storage bucket for most recent version of a 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.



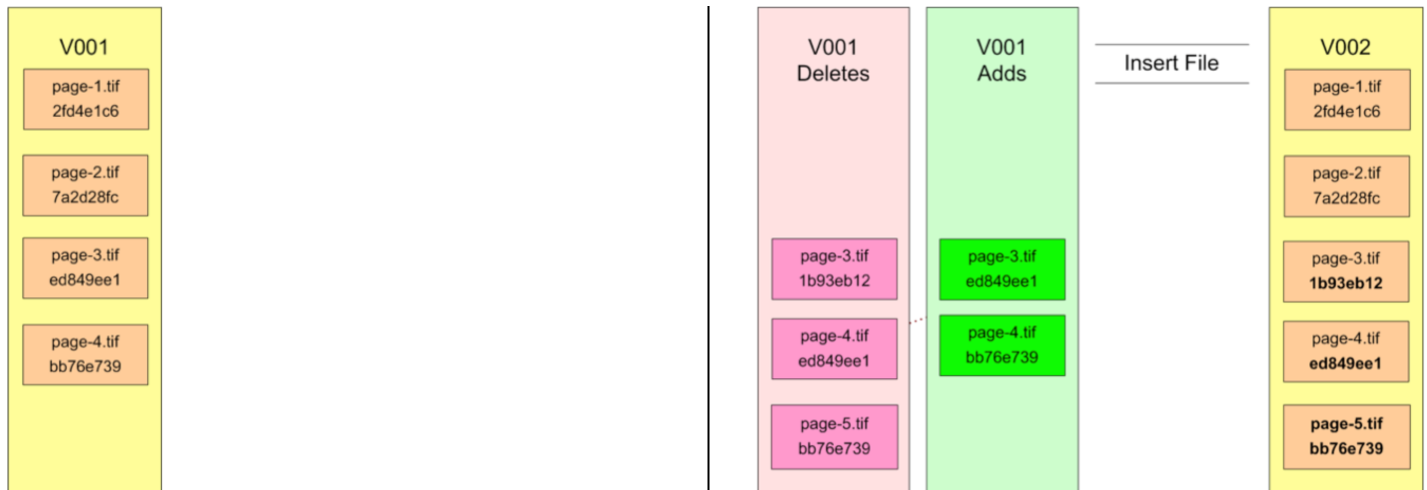
Store complete file set in latest version folder, but only older versions of files in previous folders

Pro:

- Less work to reconstruct latest version

Con:

- More work to add a new version
- More impact on replication and tape archive

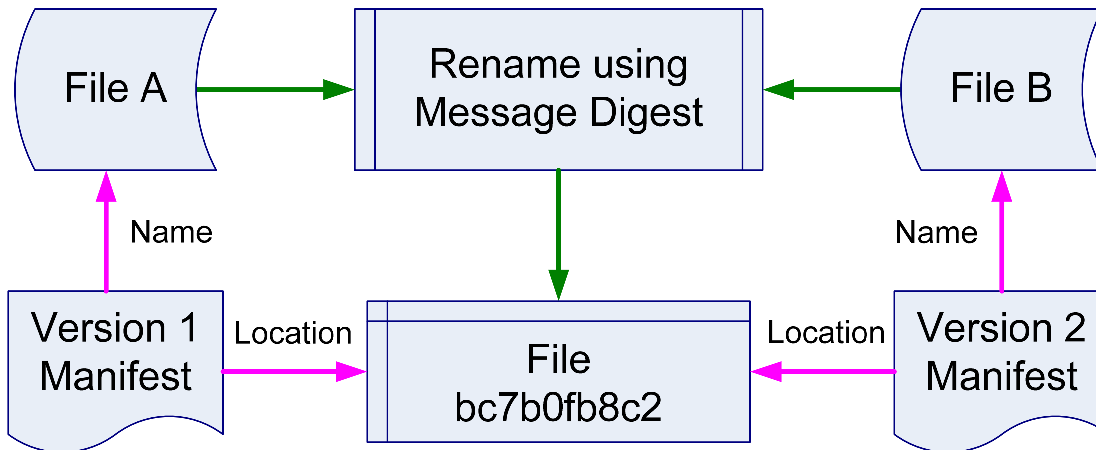


To illustrate this, let us envision the addition of that same version 1 of the object and see what the result is when we add a new version using our insert file scenario.

Version 2's bucket now contains copies of all the files, while version 1's bucket has been revised to contain only the information required to restore version 1 using version 2 as a starting point.

3.6 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 itself. This allows the file to be renamed using the checksum value as its new name, and allows multiple version manifests to reference the same file location regardless of the original filename used in a given version. This usage is also known by the phrase "compare-by-hash"



Rename files using a message digest (checksum), store files in a common pool, and use version manifests to specify a version's contents

Pro:

- Simple design
- Eliminates file duplication
- Easy reconstruction of any version

Con:

- Concern about file renames in preservation

3.6.1 Using Checksums for CAS

The following articles provide great explanations of how message digests (using cryptographic hash functions) can be used to generate digital fingerprints (analogous to coat check tickets) that can be used to store and retrieve the files of a digital object.

Content-addressable storage
http://en.wikipedia.org/wiki/Content-addressable_storage

The code monkey's guide to cryptographic hashes for content-based addressing
http://valerieaurora.org/monkey.html

Content addressable storage FAQ
http://searchsmbstorage.techtarget.com/feature/Content-addressable-storage-FAQ

The Survey on Content Addressable Storage. Bo Zhao. 2007
http://www.cse.psu.edu/~bhuvan/teaching/spring07/598d/_mail/B9B315EE7A42426A80529CB24EEEE23D/cas_survey_Bo_Zhao.pdf

Fast, Inexpensive Content-Addressed Storage in Foundation. Sean Rhea, Russ Cox, Alex Pesterev.
http://swtch.com/~rsc/papers/fndn/

3.6.2 Cryptographic hashes as Checksums

Cryptographic hash function
http://en.wikipedia.org/wiki/Cryptographic_hash_function

Nist Cryptographic Toolkit
http://csrc.nist.gov/groups/ST/toolkit/index.html

Which hash function should I choose?
http://stackoverflow.com/questions/800685/which-hash-function-should-i-choose

3.6.3 What About Collisions?

Algorithm	Checksum Length	Probability of Collision	Time to hash 500MB
MD5	16 bytes	1 in 2^128	1462 ms
SHA1	20 bytes	1 in 2^160	1644 ms
SHA256	32 bytes	1 in 2^256	5618 ms
SHA384	48 bytes	1 in 2^384	3839 ms
SHA512	64 bytes	1 in 2^512	3820 ms

What is the potential for 2 files having differing contents to hash to the same checksum value? The probability of such an occurrence is astronomically small. For SHA1, the odds of any given file having the same checksum as another is 1 in 2^{160}. ($2^{160} = \sim 10^{48}$). According to Black (2006) You are roughly 2^{90} times more likely to win a U.S. state lottery and be struck by lightning simultaneously.

SHA1 is definitely preferable to MD5 due to its much lower collision probability with only a modest increase in computing time. Any of the SHA2 algorithms would be preferable to SHA1, but require a lot more computing time to generate.

The odds 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 a extremely large set of files. If your repository's size were to approach 2^{80} ($=10^{24}$) files, then duplicate checksums would begin to be observed. This is often called the "birthday paradox": In a group of 57 people, there is a 99% probability that 2 of those people will have the same birthday.

There have been successful attempts to exploit vulnerabilities in MD5 and SHA1, but these cracks require enormous computing power and time. For example, a published brute-force attack on a SHA1 hash obtained a collision after about 2^{63} steps. This concern is really more directed at the risk of someone figuring out how to create a bogus web certificate or digital signature.

References:

Is it possible to get identical SHA1 hash? 2010
http://stackoverflow.com/questions/2479348/is-it-possible-to-get-identical-sha1-hash
http://en.wikipedia.org/wiki/Birthday_problem

What's the probability that 2 files of n bytes have the same hash using SHA1?
http://groups.google.com/group/sci.crypt/browse_thread/thread/9adc71ae65d50124

Compare-by-Hash: A Reasoned Analysis. J Black. USENIX Annual Tech. Conf (2006)
http://www.usenix.org/event/usenix06/tech/full_papers/black/black.pdf

Guidelines for Using Compare-by-hash. Val Henson & Richard Henderson. 2005
http://valerieaurora.org/review/hash2.pdf
http://www.nmt.edu/~val/review/hash2.pdf

4 Versioning Implementations

4.1 CDL ReDD design

The California Digital Library has implemented a storage mechanism (the Storage Micro-Service) based in part on a reverse-delta implementation called ReDD. Here is an skeleton example of a Dflat/ReDD/dNatural directory structure that is holding 3 versions of a digital object. V003 is the latest directory and contains a full complete version. V001 and V002 are reverse-delta versions.

```
<object-identifier>/    # Dflat home directory
    [ current.txt ]         # pointer to current version (e.g. 'v002')
    [ dflat-info.txt ]      # Dflat properties file
    v001/  # reverse-delta version
        [ manifest.txt ] # version manifest
        delta/              # ReDD directory
            [ add/ ]            # files to be added relative to subsequent
            [ delete.txt ]      # files to be deleted relative to subsequent
    V002/           # reverse-delta version
        [ manifest.txt ] # version manifest
        delta/              # ReDD directory
            [ add/ ]            # files to be added relative to subsequent
            [ delete.txt ]      # files to be deleted relative to subsequent
    v003/           # current version
        [ manifest.txt ] # version manifest
        full/               # dNatural directory
            [ consumer/ ]   # consumer-supplied files directory
            [ producer/ ]           # producer-supplied files directory
            [ system ]              # system-generated files directory
```

References:
http://www.cdlib.org/services/uc3/curation/index.html
http://www.cdlib.org/services/uc3/curation/storage.html
https://confluence.ucop.edu/display/Curation/D-flat
https://confluence.ucop.edu/display/Curation/ReDD
https://confluence.ucop.edu/display/Curation/Storage+Service+Download

4.1.1 ReDD Process for a new version

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.

- Compare new version manifest and previous version manifest
- Create temp delta version containing:
  - adds folder
  - deletes.txt file
- Replace previous full version with delta version

4.1.2 CDL MicroService Concerns

The SDR team greatly admires the intellectual rigor that went into the entire Storage Micro-Service specification, and we may very well adopt or imitate major portions of this design in our own storage layer. We do, however, have concerns about the ReDD mechanism:

- Reverse-delta design requires a subsequent backup of a full version's files to tape.

  As previously mentioned, a reverse delta design results in the creation of a new directory containing a full version of the object, and a modification of the previous version's directory by the removal of any files that have carried over to the new 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. If the frequency of digital object updates is low or size of objects is usually small, then this may be a minor concern. But it would be an impediment to high-frequency updates of large digital objects.

- 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.

  Of related concern, is the manner in which file renaming is handled. If the content of a file remains the same, but the filename changes, then this system will treat the transaction as a combination of delete and add, resulting in duplicate storage of the content under two different names.

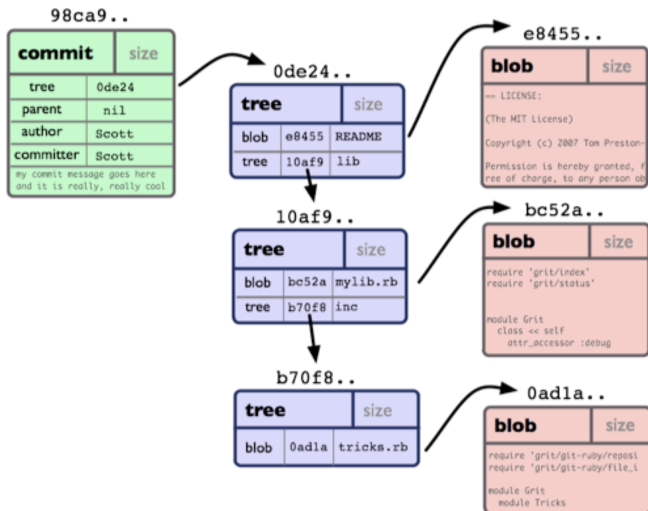
  Note that this analysis relies on examination a copy of the merritt.zip file from March 2010. In that code base, the addition of a new version required submission of a complete object (including all content files). I have been informed that an enhancement is in progress to address that behavior.

I will next examine 2 existing open source implementations of Content-Addressable Storage that have been suggested as viable candidates for digital object versioning, as well as a new design we are fleshing out at Stanford that combines some of the best design ideas from several versioning approaches.

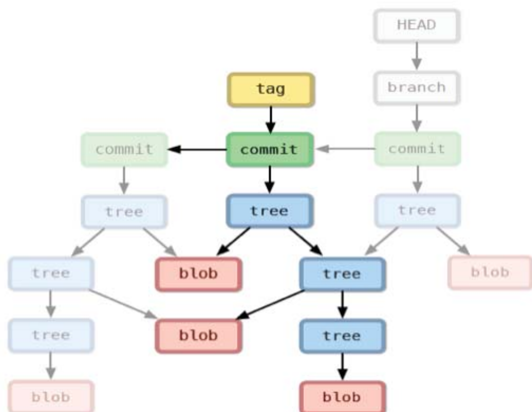
4.2 Git

Git is a popular distributed version control system that was initially created by Linus Torvalds for tracking changes to the Linux operating system.

4.2.1 Object Model



Due 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.



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.

blob

```
blob 109\0
#import <Cocoa/Cocoa.h>

int main(int argc, const char *argv[])
{
    return NSApplicationMain(argc, argv);
}
```

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.

tree

```
tree 84\0
100644 blob cd98f   README
100644 blob a3f6b   Info.plist
040000 tree bfef9   Source
```

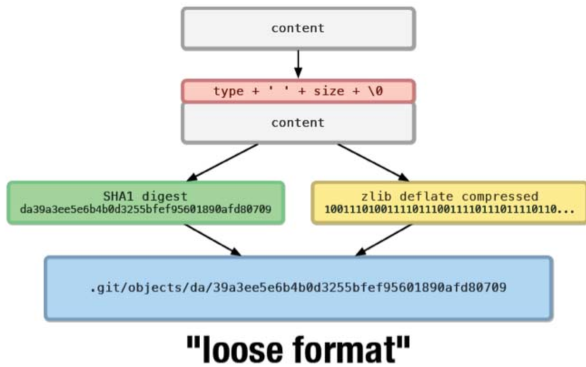
A tree functions similarly to a directory. It contains a list of all filenames (and subdirectories) contained in a directory at a given point in time along with the file permissions for those items, and SHA1 checksums for each. The checksum values are used to actually locate the listed files and sub-directories in the Git repository's storage area.

commit

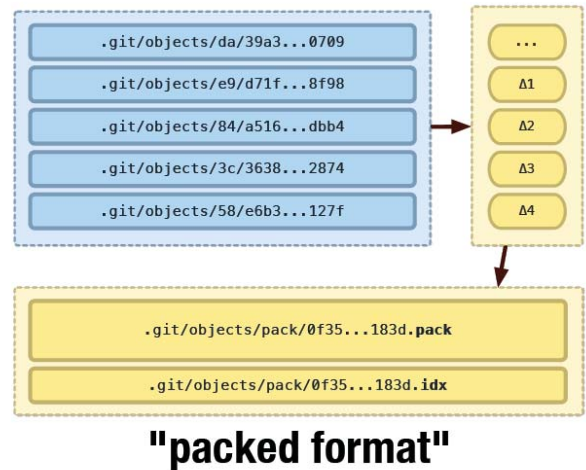
```
commit 155\0
tree 9a3ee
parent fb39e
author Patrick Hogan <pbhogan@gmail.com> 1311810904
committer Patrick Hogan <pbhogan@gmail.com> 1311810904

Fixed a typo in README.
```

A commit is a special Git artifact that contains metadata about a revision (version) of the project along with a pointer to the top level tree of the project as of that point in time.



Note that the content data stored in Git is usually also compressed, initially into separate "loose" files.



But eventually multiple loose items are combined into larger "packfiles"

References
http://book.git-scm.com/1_the_git_object_model.html
http://www.slideshare.net/pbhogan/power-your-workflow-with-git

4.2.2 Git Concerns

- Does not store unaltered original files unless you use a 3rd-party plugin (adding complexity)
- Adds header structure to each file, then "packs" files into a container
- Requires special configuration settings to avoid zlib compression and delta compression
- Requires a local copy of the entire repository in order to make revisions

In spite of its sophisticated design, for the reasons listed above 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.

4.3 Boar

Boar (http://code.google.com/p/boar/) 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's simple, but elegant, storage design.

In the Boar design, each repository (storage node) has a top-level "blob" 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.

4.3.1 Boar Storage structure

```
repository
	blobs
		bc
			bc7b0fb8c2e096693acacbd6cb070f16
	sessions
		<snapshot name>
			bloblist.json
			session.json
		<snapshot name>
			bloblist.json
			session.json
```

Each content file is checksummed, and the MD5 digest is used to rename and store the file inside the blobs hierarchy

A local working directory containing a set of files can be backed up using a concept called a "session". 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 session, plus a pointer to the previous snapshot of the given session

- Bloblist.json contains a map between filename paths and checksums.
- Session.json contains metadata about the snapshot.

4.3.2 Boar's version manifest

bloblist.json


```
[
  {
    "size": 16,
    "md5sum": "c8fdfe3e715b32c56bf97a8c9ba05143",
    "mtime": 1317549043,
    "ctime": 1317549026,
    "filename": "mynewfile.txt"
  }
 {
    "size": 112490,
    "md5sum": "6bd3ef5e2d25d72b028dce1437a0e89a",
    "mtime": 1215443139,
    "ctime": 1215443138,
    "filename": "MARC21slim2MODS3-2.xsl"
  }
]
```


The above is an example of the kind of data stored in a snapshot's bloblist.json file. It is similar in function to the tree component of the Git object model. The main point to observe is that files are stored in a location that is content-addressable via the file's checksum value, and that these values are stored in a type of version manifest. You would have to look at a chain of bloblist.json files to figure out the contents of any given version of an object.

It was somewhat challenging for me to figure out how this application could be applied to the preservation of digital objects. I would suggest giving each digital object its own repository node, in which case each snapshot would be considered a new version.

4.3.3 Boar Concerns

- Limited documentation of internal design
  - The definitions of session and snapshot are not clearly articulated.
- Pools content files of all versions in a single folder structure
- Has a limited command-line API
- Does not yet work over network protocols
- Single developer, small community of users
- Target audience is individual users needing a backup system that integrates file versioning.

Although much closer in functionality to our needs than Git, Boar in its present incarnation does not seem to be a good fit for our repository either. 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 mappings between these concepts makes discussion of operations and storage somewhat confusing. Also 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 in a secure location. Nevertheless Boar provides inspiration for an alternative design of our own creation.

It has been pointed out to me that some of concerns I have raised about this product's maturity (newness, small community, target audience) can also be directed against the new Moab design we are proposing. In answer to that I would respond that these concerns were not the main reasons we have decided not to directly adopt the Boar approach. But we also would rather have a dependency on our own developer resources than on a single person's "spare time project". (http://code.google.com/p/boar/wiki/FAQ)

4.4 Summary of Desired Features

This is a good time to review and summarize the pros and cons of each of the storage and versioning approaches we have examined so far.

Versioning Mechanism	File Fixity	No Disk Duplicates	No Tape Duplicates	No Local Repo Copy
Forward-Delta	**Good**	Fair	Fair	**Good**
Reverse-Delta (CDL ReDD)	**Good**	Fair	**Poor**	**Good**
Git	**Poor**	**Good**	**Poor**	**Poor**
Boar	**Good**	**Good**	**Poor**	**Good**

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

5 Moab[§] Design

Since we did not find an existing storage and versioning system that we wanted to adopt or adapt, we decided to "invent" our own. SDR's Moab Design aims to meet the functional requirements stated at the beginning of this presentation by incorporating the best aspects of the approaches previously discussed:

- **Uses a simple folder structure**
  The CDL Dflat structure provides a good framework for storage of a digital object and its versions, such as the use of V001, V002, etc subdirectories within an object's directory to keep each new version's data separate from that of previous versions.

- **Renames files using SHA1**
  Although it introduces some preservation risk, the advantages of renaming content files using the values of their checksums is compelling. As you will see it greatly simplifies the processes used for archival storage, fixity checking, and object reconstruction. Unlike Git & Boar, however, the Moab design uses a separate content file bucket for each version.

- **Files are immutable**
  Unlike the behavior of a software version control system, the internal bytestreams of files in a preservation system should not be modified by that system. There is little advantage in performing any sort of file compression or other alteration of curated content files. On the contrary, compression reduces the potential for data recovery in the event of bit rot, as well as introducing significant processing cost.

- **Uses version manifests**
  All the systems examined use some sort of manifest file or files to inventory the files comprising a digital object version. CDL's Dflat uses a manifest.txt file, Git uses a set of linked tree objects, and Boar uses a set of bloblist.json files. Similar to the CDL approach, the manifest used by the Moab design describes an entire object at the point in time at which the version was accessioned.

- **Stores new content files in new buckets**
  In order to facilitate disk and tape replication with a minimum of processing and content duplication, this design will store each version's new or modified content files in a separate bucket within the version's home directory. This adds a slight bit of additional location lookup when reconstructing a given version from its manifest, but since files are named using the SHA1 hash of the content, there will never be 2 files of the same name to disambiguate.

[§] 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 Address Behavior
* Make Objects Act Better
* My Own And Best

5.1 Moab Folder Structure

The Moab design's file folder structure has a similarity in appearance to the CDL Dflat design, and it should be possible to make it compatible with the CDL storage framework. Unlike CDL, however, the design uses aspects of the forward-delta and CAS approaches.

```
druid:ab123cd4567/
    v001/
      manifest.txt & manifest.sha1
      filemap.txt & filemap.sha1
      data/
         2fd4e1c6
         7a2d28fc
         ed849ee1
         bb76e739
    v002/
      manifest.txt & manifest.sha1
      filemap.txt & filemap.sha1
      data/
         1b93eb12
```

5.1.1 Data folder

In our file layout, the first version's data folder will usually contain the majority of files that are referenced by the manifests of all versions. Each new version will have its own data directory containing only those new files that were added in that version or which resulted from modifications of an older version's files. Each content file is renamed using the file's SHA1 checksum. Although these content files are stored in a series of separate data folders, the aggregation of those folders can be conceptually be considered to be a single storage space. A filemap file is updated with each new version to contain location information for all files ingested so far.

5.1.2 Manifest file

Each object version has a manifest file providing a complete inventory of the content files that comprise that version. This file contains at a minimum each file's original name, path, size, and date along with a SHA1 checksum. The checksum is used not only for fixity verification, but also as a key to locate the actual content file on disk. The file will be found in either a given version's data folder, or in a previous version's data folder. The manifest.sha1 file is a signature file that contains the checksum of only the manifest.txt file, since the manifest file cannot contain the checksum of itself.

5.1.3 Impact on replication

This structure is conducive to simple mechanisms for both disk-to-disk replication and tape archiving, since all the changes are isolated in the new version's folder. Reconstruction of any given version is slightly impacted due to the need to traverse multiple data folders while locating files, but doing so should be fairly trivial.

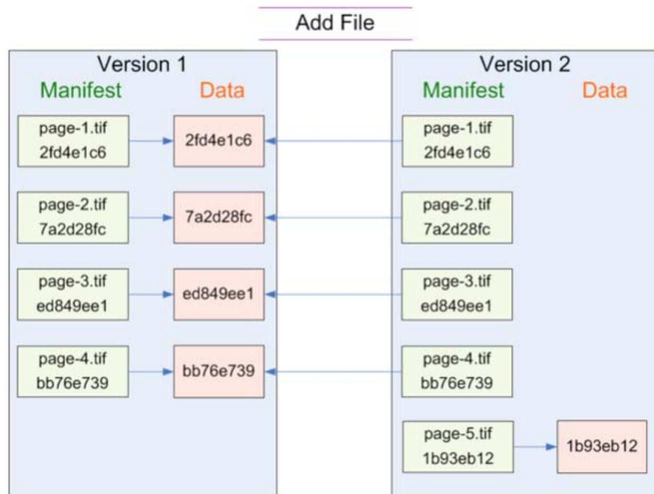
5.1.4 Hash collision risk

Note that in the Moab design the risk of two files having the same checksum is reduced even further due to the isolation of each digital object's files within that object's folder, instead of in a common storage area shared by the whole repository. We can nevertheless do a file diff in the extremely unlikely case that two files being added in a new version hash to the same value.

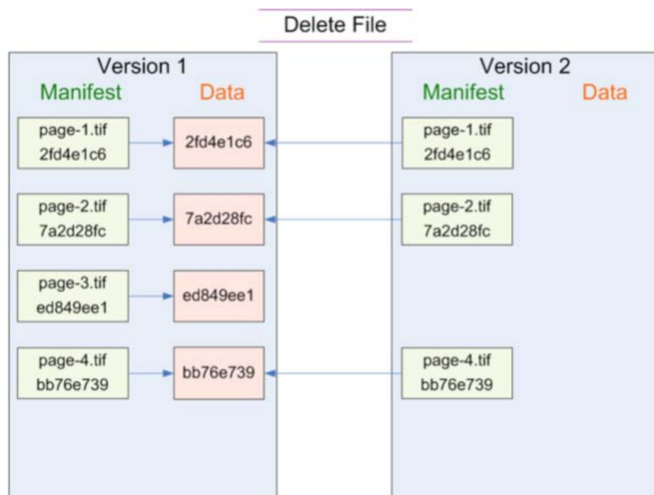
5.1.5 File renaming risk

The concern remains about renaming of content files, and the resulting dependency on the version manifest to preserve the filename information. This concern could also be voiced about any of the major DVCS systems in widespread use today, for which the recommended recovery strategy is to rely on your replication copies. It is essential that one store multiple copies of each manifest in each replication node for additional insurance.

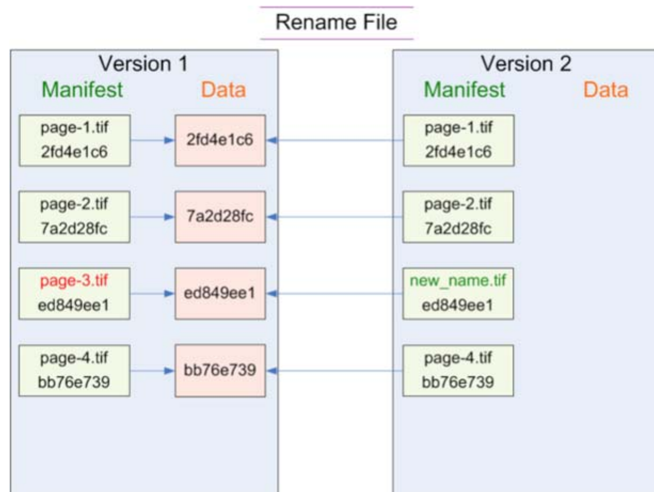
5.2 Versioning Examples



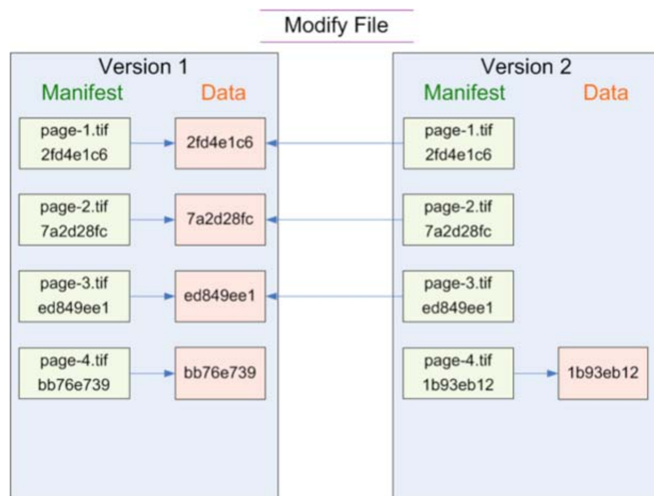
Here is an example of 2 sequential version manifests. Version 1 contains 4 files, renamed using their SHA1 checksums, and stored in the version 1 data folder. In version 2, a new file is added (page-5.tif). The file is renamed to 1b93eb12 and stored in the version 2 data folder. The version 2 manifest contains an inventory of all the original filenames and SHA1 hashes, allowing simple reconstruction of the object version as needed.



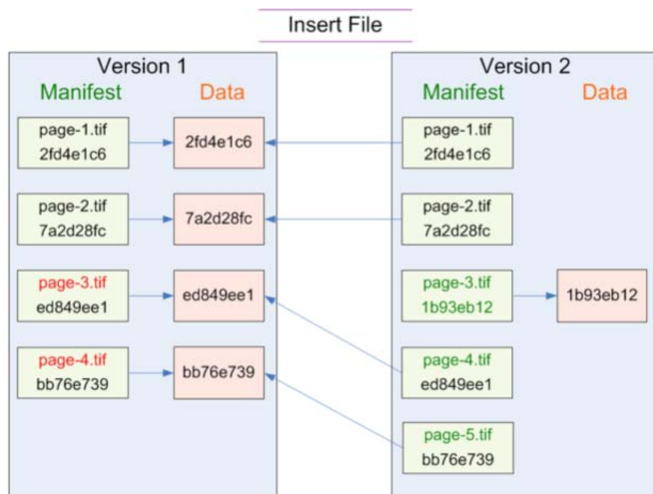
This example shows what would happen if a file were to be deleted to create version 2 of the object. In this case the only change would be the removal of the entry for page-3.tif from the manifest of version 2.



Renaming a file is also easy to demonstrate. In this example, the file page-3.tif is being renamed to new_name.tif, but the SHA1 hash is unchanged. In this case both version manifests point to the same file in the version 1 data folder.



If you modify a given file (page-4.tif) as part of creating version 2, then a new SHA1 checksum will be generated for what is essentially a new file. The new file is added to the version 2 data folder, and the new version's manifest will contain a reference to the new checksum instead of the previous version's checksum.



Inserting a new file into a sequence of files is a fairly difficult concept to illustrate clearly, but let us attempt to show it as a combination of a file addition and a couple of file renames. To make room for the new file inserted in place of the existing page-3.tif, that version 2 manifest is edited so that page-3.tif and page-4.tif are renamed to page-4.tif and page-5.tif. A new entry is then inserted for page-3.tif, which points to the new data file.

5.3 Version Manifest

As stated earlier, the manifest for any given version will contain at a minimum, an inventory of all files constituting that version and the checksum value for each of those files.

File Name	Checksum
relative/path/page-1.tif	2fd4e1c6
relative/path/page-2.tif	7a2d28fc
relative/path/page-3.tif	1b93eb12
relative/path/page-4.tif	ed849ee1
relative/path/page-5.tif	bb76e739

5.4 File Map

In order to locate any given file, based on its checksum, we need to generate a map from checksum to a file's physical storage location.

This a trivial operation, requiring only that one combine the directory listings of all the version data folders (ls v*/data/*), and generate an index on the checksum string.

Checksum	File Location
2fd4e1c6	v001/data/2fd4e1c6
7a2d28fc	v001/data/7a2d28fc
1b93eb12	v002/data/1b93eb12
ed849ee1	v001/data/ed849ee1
bb76e739	v001/data/bb76e739

There are reasons one might wait to create this index until a reconstruction is needed.

- It is trivial to produce
- It will need to be updated each time a new version is added
- Retrieval of object from archival storage is assumed to be a rarity.

However, it would also serve as a convenient table for

- Determining if a file being submitted has already been placed into storage
- (this is especially desirable at the time when a submission package is being created)
- Doing consistency checks between the manifest's and the disk file locations
- Doing fixity checks on the data files

5.5 Reconstructing an Object

In order to reconstruct an object version, a combination of a version manifest and the file map is used. All that is needed is to create a temporary directory into which symbolic links (named for the original filename) can be made to actual files located in the data folders.

Version Manifest

File Name	Checksum
relative/path/page-1.tif	2fd4e1c6
relative/path/page-2.tif	7a2d28fc
relative/path/page-3.tif	1b93eb12
relative/path/page-4.tif	ed849ee1
relative/path/page-5.tif	bb76e739

File Map

Checksum	File Location
2fd4e1c6	V001/data/2fd4e1c6
7a2d28fc	V001/data/7a2d28fc
1b93eb12	V002/data/1b93eb12
ed849ee1	V001/data/ed849ee1
bb76e739	V001/data/bb76e739

6 Replication and Tape Archives

6.1 Disk to Disk Replication

Stanford currently uses the Rsync (http://rsync.samba.org/) file transfer utility to copy files from one location to another. It has the benefit that it can synchronize two locations by detecting and coping only the file differences from one location to another. This synchronization is simplified in the Moab design since file changes associated with a new version are isolated to that version's folder.

6.2 Disk to Tape Replication

For tape backup we use IBM's Tivoli Storage Manager (http://www-01.ibm.com/software/tivoli/products/storage-mgr/).

TSM has two flavors of command for storing files on tape: 'backup' and 'archive'.

Backup Command

- usually run daily by a scheduler
- picks up incremental changes
- wipes out deleted files after expiration period

Archive Command

- requires explicit scripting to run
- copies full directory structure to an archive
- persistent storage of archive until explicit delete

The Backup command makes it simple to backup incremental changes to a disk structure, but has two drawbacks that prevent us from using it for preservation purposes:

- Any files that are inadvertently modified would get backed up (again), and there is a limit on the number of previous versions of a given file path that can be restored.
- Any files that are inadvertently deleted would cause an expiration of that file's backup after a relatively short time period.

The Archive command is safer in that the archive copy of a file is considered permanent, provided that the TSM policies for a given TSM storage node are set up to prevent expiration of archives. Unlike the backup command which can easily be run automatically from the TSM scheduler, archive commands should be explicitly requested--manually, by script, or by cron.

6.3 TSM Archive Examples

The value of the -desc flag is recommended as a way to differentiate an archive version from any other copies of the same digital object that have been saved to tape.

Archive the directory containing object xyz, including its child folders.

```
dsmc archive druid:xyz -subdir=yes -desc="druid:xyz"
```

Archive only the version 002 folder from within the xyz object's directory.

```
dsmc archive druid:xyz/v002 -subdir=yes -desc="druid:xyz v002"
```

6.4 Tape Archive Efficiency vs. Versioning Design

- Whole Object
  - poor performance (must transfer full versions)
- Reverse-Delta
  - poor performance (latest version is always full)
- Forward-Delta
  - fair performance (new versions are deltas)
- Content-Addressable
  - poor if data for all versions is pooled
  - good if new data is segregated by version

In many of the versioning designs we have discussed, addition of a new object version would 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.

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 replication mechanism of the storage system must copy all those files, even if they were previously archived. Also, since the previous version has been converted from a full version to a reverse-delta version, those changes much be 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 require that a new tape archive be made of the common pool. TSM archive has a 'filelist' option that allows one to specify a list containing the paths of only those files that need to be archived, but that would be like adding a kludgy patch to a design that should be re-thought.

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

7 Version Manifest Structure

7.1 CDL Checkm specification

The initial conception of the version manifest was as a tab-delimited flat file, using CDL Checkm standard format. (https://confluence.ucop.edu/display/Curation/Checkm) This format records the object-relative pathname, the digest type and value, and the file length and date. It is important to record the digest type on a per-file basis in case a decision is later made to move on to a new cryptographic hash algorithm.

<pathname> <digest-type> <digest-value> <file-size> <modtime> ...

7.2 SDR's versionMetadata XML

Use of an XML format (or JSON) adds the ability to add a header section that can be used to store metadata about the version itself: the version number, the timestamp of the version, and a label or other descriptive text related to provenance. SDR has come up with the above structure to hold this information in a single datastream.

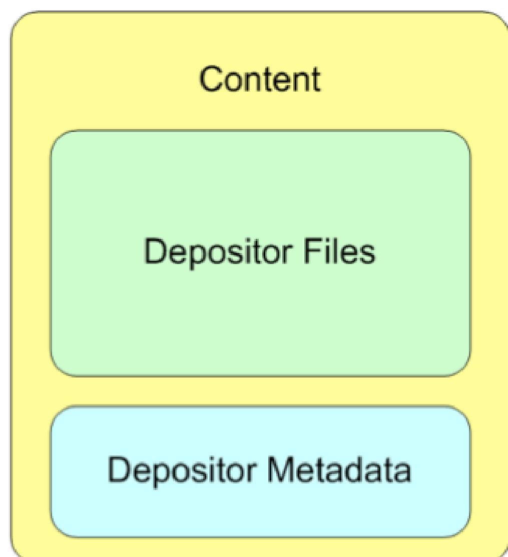
An XML structure also allows sub-dividing the inventory of files into sections, such as "content" (the depositor submitted files) and "metadata" (the administrative datastreams added by DOR). We currently use the same strategy when creating Bagit bags for transfer of digital objects.

```
versionMetadata
    |_objectId
    |
    |_versionIdentity
    |  |_versionId
    |  |_label
    |  |_timestamp
    |  |_description
    |
    |_versionData
    |_dataGroup
    |   |_group_name    -> id = "content|metadata"
    |   |_baseDirectory
    |   |
    |   |_file (repeating)
    |      |_id (relative_path)
    |      |_size
    |      |_signature (SHA1 hash)
    |      |_modtime
    |      |_
    |      |_checksum (repeating)
    |         |_type
    |         |_value
```

8 Content & Metadata

I now broaden the discussion and as promised begin working backward from our storage and versioning design to an assessment of the implications of that design for the workflow used for submission and processing of an object.

8.1 Depositor Submitted Files



The content producer will typically submit a package containing normal data files (such as the page images of a scanned book), plus any ancillary or auxiliary files that contain user metadata about the object. An example would include a METS file provided by the external system.

For the purpose of the repository workspace, both sets of files will be grouped together and considered "content" files.

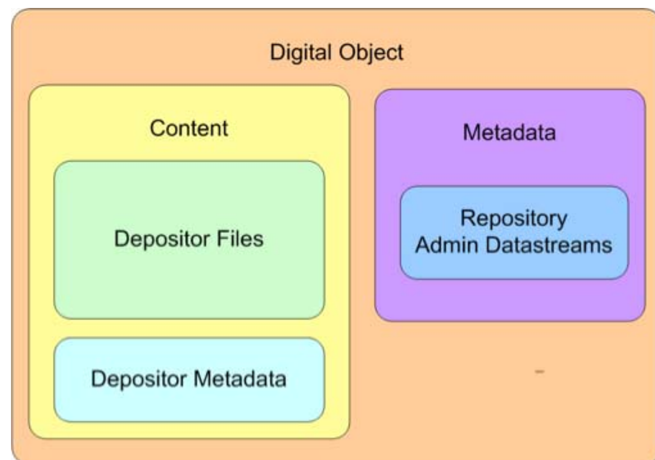
8.2 Stanford's Repository Datastreams

As an outcome of the processing of a digital object through its accessioning workflow, DOR creates a number of additional metadata files that also need to be preserved and versioned. DOR generated metadata datastreams are also preserved as data files.

- identityMetadata
- descMetadata
- relationshipMetadata
- provenanceMetadata
- technicalMetadata
- contentMetadata
- versionManifest

Note that it may be the case that the only change from one version to the next is the modification of one of theses DOR datastreams.

8.3 Digital Object Structure



A digital object will therefore consist of 2 major data groups, each of which will have a separate base directory. When being processed in the DOR workspace or packaged in a Bagit bag for transfer to SDR:

- Depositor submitted content and metadata files should be stored in the data/content directory.
- Repository generated metadata datastreams will be stored in the data/metadata directory.

8.4 contentMetadata Datastream

One of Stanford's core metadata files (named contentMetadata) is used to store the object's structural information and associated metadata.

- structural information about how the files are arranged within the object
- key technical information to aid delivery
- fixity information (checksums)
- tags to categorize resource types, including differentiation of data files from ancillary files

It is analogous to information contained in the fileSec and StructMap segments of a METS document or an OAI-ORE Resource map. For a book object it can function as a page sequence file. For selected file formats, it carries a subset of technical metadata information, such as image dimensions.

It also allows for additional attributes that can aid in the management and delivery of an object in our Digital Library. The contentMetadata design uses 'resource' elements that can have 'type' and 'contains' attributes. The type attribute is currently used to specify such values as 'page'. We will plan to use the contains="ancillary" attribute value to mark user supplied ancillary or auxiliary files so they can be filtered appropriately.

8.5 contentMetadata vs versionMetadata

The inventories contained in these two files overlap in that they both contain a list of all user deposited files, but only the versionMetadata will contain a list of the repository datastreams added by the repository workflow.

contentMetadata datastream

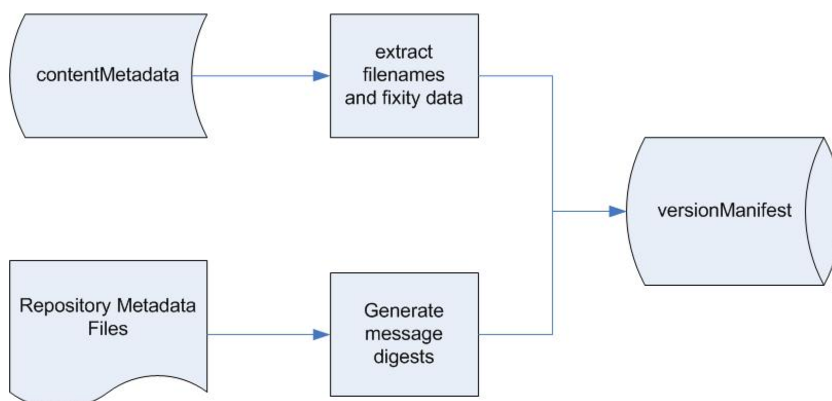
- inventories only user deposited files
- contains metadata beyond fixity info

versionMetadata datastream

- inventories all files, both user & repository MD
- records file paths and directory info
- records fixity information

8.6 Building a Version Manifest

The versionMetadata file will be created by a DOR robot. It will consist of an extract from the contentMetadata datastream listing the file paths and checksums, etc. of all content files plus an inventory of all DOR datastreams and their checksums. It should be generated in the robot sequence after all other datastreams have been created or modified, but before the sdr-ingest-transfer robot is run. It may also need to precede the running of the shelve robot, depending on how we design digital stacks updates.



The manifest file thus generated should also be considered a repository metadata datastream, with DOR's Fedora always containing a copy of the latest copy, and the deep preservation storage (SDR) containing all versions. Up until the point where a new object version is transferred to SDR (and the manifest saved to DOR Fedora), the new version manifest can be diffed against the previous version manifest in order to figure out what changes have occurred, and which files will need to be transferred to SDR via a BagIt bag. (Related to philosophical discussion of DOR's retention of information about previous versions).

8.7 Do we still need BagIt?

The functionality of BagIt also overlaps with that of the versionMetadata file

- BagIt is just a data directory plus manifest files that contain checksums.
- Can still use BagIt for transfer, but no need to re-calculate checksums -- just extract them from the version manifest.
- We may be transferring only a subset of files for a new version. Would need to filter out the manifest entries for files not present
- Would not use BagIt in deep storage.

We have used the phrase "sparse bag" to summarize the idea of a bag containing only the new or changed files that are part of a new version. This should not be confused with the phrase "holey bag" that is a part of the BagIt spec. A bag with 'holes" includes a file named fetch.txt that contains the URLs of the files that are missing from the bag being transmitted. A sparse bag will be a normal bag, but will contain a data directory holding only a subset of an object's content and metadata files. The versionMetadata datastream will be placed in the data/metadata directory along with any other DOR datastreams that have changed. The BagIt payload manifest files (e.g. manifest-sha1.txt) can be derived from the versionMetadata, but will only contain the subset of file data for the files actually being transferred in the bag's data directory.

9 Ongoing Design Work

I'd like to keep going at the previous level of detail, but have reached the point where local collaborative work is needed to flesh out some aspects of the end-to-end design of a versioning workflow

9.1 APIs and Tools for Accessioning

How will we implement the interface between the depositor and the accessioning system?

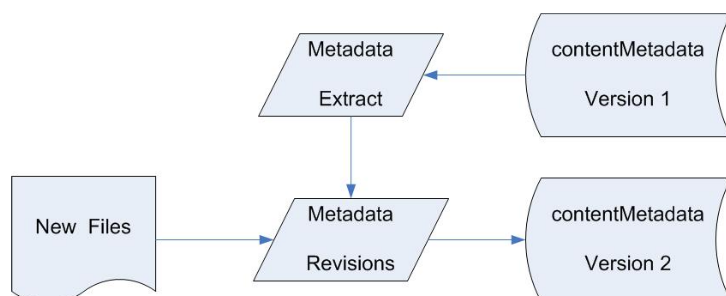
- Command line
- Procedural (e.g. java or ruby API)
- Web API (RESTful services)
- Graphical User Interface (GUI)

The communication between a submitting agent and the accessioning system should be flexible enough to accommodate various modes of new version ingestion. A depositor should be able to provide a complete new 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 plus directives about which files should be deleted or renamed. An API will need to be devised for this communication and tools implemented to enable either mode of submission.

9.2 contentMetadata Revisions

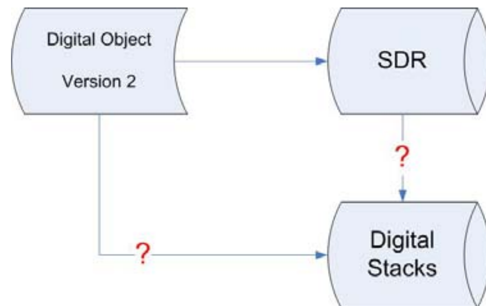
The objective here is to facilitate the revision of the contentMetadata datastream with the least amount of effort and risk. If we have a whole new set of files for the new version, then we may be able to just generate a new contentMetadata file. But if dealing with a subset, then we need to figure out a way to edit the existing datastream. It would be risky to allow direct editing of the contentMetadata datastream by the depositor.

A better design might be to present a skeleton extract of the contentMetadata and allow the user to revise that extract. The accessioning workflow would then revise the contentMetadata accordingly and fill in the new checksums, etc, based on analysis of the new files.



9.3 Digital Stacks/Shelver Considerations

Our Digital Stacks shelver needs a parallel mechanism for communicating changes and transferring new versions of files. We have not yet decided if this should this be done from DOR or should new version's files be retrieved from SDR after version storage is complete.



9.4 Versioning Workflow

A complete processing workflow that supports versioning, will have the following components

- Submission Package
- DOR work steps
- SDR work steps

9.5 Submission Package

The main objective of the DOR workflow is to enable the accessioning of a digital object or a new version of an existing object. The list below specifies the files or other information that must be supplied by the depositor in a submission package container.

- A digital object identifier that is either the druid or the submitting agent's sourceID.
- An explicit indication that this submission is a new version of an existing object.
- A folder (or other container) holding the set of versioned files.
- A submission manifest that lists relative path names and digests of *ALL* files comprising the new version

For new versions, inclusion of unchanged files is optional. If the only changes are file renames, file deletes, or DOR datastream changes, then this container may be empty. Any user-submitted metadata should be considered as being part of the content.

The version manifest must inventory all files, including unchanged files that may have been omitted from the submitted file container. This manifest will be used to infer/deduce any changes, additions, deletions, or renames.

9.6 DOR Work Steps

Below is a proposed workflow for the DOR component of digital object processing:

1. Bootstrap a versioning workflow
2. Determine which driud is involved (may require lookup using sourceID), and verify that a DOR object exists
3. Transfer the submitted content into a workspace object/content directory and the manifest into a object/metadata directory.
4. Perform a fixity validation procedure to make sure that all content files being submitted are correctly included in the submission manifest.

5. Compare the submission manifest against the current DOR content manifest for this object. This procedure will allow us to determine:
    a. unchanged files (same filename, same digest)
    b. renamed files (new filename for a given digest)
    c. modified files (same filename, different digest)
    d. added files (new filename, new digest)
    e. deleted files (old filename nor digest
6. Verify that any modified or added files were included in the submission package.
7. Run JHOVE against any new or modified files and merge the results into the technicalMetadata datastream. If there are deleted or renamed files, make the appropriate revisions to the datastream.
8. Update the DOR contentMetadata datastream to incorporate structural and other changes, as well as extracts from technical metadata
9. Update descMetadata and other datastreams, if appropriate. This may be derived from depositor-submitted metadata that was included as part of the submitted package.
10. Add a new stanza to provenanceMetadata
11. Generate a datastream manifest, that itemizes all DOR datastream names and fixity data.
12. Generate new versionMetadata that includes both content and DOR datastream file info.
13. Run sdr-ingest-transfer in a mode similar to what occurs during initial accessioning. The sdr-ingest-transfer robot would create a BagIt bag in the export area and bootstrap the SDR Ingest workflow. The difference would be that only modified or added files would be included in the bag payload. The bag's data directory will also contain the content and metadata manifest files that document the full inventory of the object version. This will be essential for SDR storage versioning.

9.7 SDR Work Steps

Below is a proposed workflow for the SDR component of digital object processing:

1. Bootstrap the workflow
2. Determine whether this is a new object or a new version of an existing object
3. Transfer the bag to a temp location
4. Validate the bag
5. Update any standard SDR Fedora datastreams
6. Create the new version in storage, using the storage mechanism described previously