Using Boar to Manage the Storage and Versioning of Digital Objects

Richard Anderson
Digital Library Systems & Services, Stanford University
17 September 2015

Introduction

Boar's catch phrase is "Simple version control and backup for photos, videos and other binary files". It is explicitly targeted toward the preservation and versioning of digital objects that contain large binary files. It claims support for point in time restores

Basic Boar References

- Boar project home
- Rationale Why boar?
- Frequently Asked Questions
- Discussion on Reddit.com

Strengths

- Uses a VCS-style workflow with check out, update, commit.
- Boar uses a centralized repository model, eliminating need to have a local copy of the entire repository in your workspace
- You can check out only part of a directory tree and work with that separately.
- Specifically designed for handling binary files
- There is no file compression or delta compression (at present)
- Designed to combine backup and versioning functions
- Checksums used for verification and corruption detection
- Simple storage format
- Supports replication by cloning
- Fast access to storage
- Protects all stored pathnames from being mangled by the local file system

Weakness

- This is a new (beta) software application this year
- Single developer, small community of users and contributors
- Initial target audience appears to be individual users needing a reliable backup application that integrates file versioning.
- There is a lack of documentation about the underlying storage design. The definitions of session and snapshot are not clearly articulated. I had to download and run the code to get a vision of how the file content and session data were organized.
- The vocabulary and design of *repository*, *session*, and *snapshot* makes it harder to figure out how best to store a collection of digital objects. Should each digital object be stored in its own repository? Or should a session within a larger repository correspond to an object?

In the latter case the versions of the various objects would be scattered across the sequential array of snapshot IDs in a large flat directory space.

- Has only a command-line API
- The API does not include commands to show version history of a session or a single file
- Does not yet work over network protocols
- Boar 1.0 will introduce block-level data deduplication (hopefully this will be optional)

Object Model

Major Commands

mkrepo

Syntax: boar mkrepo <repository path>
Create a new repository.

mksession

Syntax: boar mksession <session name>
Create a new session in a repository.

import

Syntax: boar import [--ignore-errors] [-w] [-n] [-v] [-m "log message"] <directory> <session name[/path/]>
Import the given directory into the given session [storing your data in the repository], optionally to a specific sub path in the session. “-w” turns the imported directory into a workdir (allowing you to easily update and check in changes by using “co”, “ci” and “update” commands).

ci

Syntax: boar ci [-m "log message"] [--add-only]
Commits any changes that have occurred in the workdir, thereby creating a new snapshot.

co

Syntax: boar co [-r <snapshot id>] <session name[/path/]> [workdir]
Checks out a session (or subdir of a session) as a new workdir. If no workdir path is specified, the name of the session will be used.
Normally the latest snapshot is checked out, but you can use the -r option to specify an older snapshot.

update

Syntax: boar update [-r <revision>] [--ignore]
Updates the workdir with any changes from the repository. If an revision is specified with the -r argument, the workdir will be updated to that revision. Otherwise, it will be updated to the latest revision. Note that "update" can be used to update to an earlier revision as well.
Modified files in the workdir will never be changed by an update. If you want to revert some changes to a file, just delete the modified file and execute "update" again.

Data Store Structure

```
repository
     blobs
          <first 2 letters of a blob’s checksum>
```

```
                <blob file named using checksum>
    sessions
         <snapshot number>
              bloblist.json
              session.json
```

It was only by installing and using Boar that I was able to obtain a clear picture of the data store.

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. The blob directory is organized by using the first 2 characters of the checksum to create sub-folders.
For instance, if a file "testimage.jpg" has the checksum bc7b0fb8c2e096693acacbd6cb070f16, it will be stored in blobs/bc/bc7b0fb8c2e096693acacbd6cb070f16.

A working directory containing an object's files can be stored using a concept called a "session". All session snapshots are captured in a flat structure under a top-level "sessions" directory. Each snapshot contains a bloblist.json file and a session.json file. The former contains a map between filename paths and checksums, and the latter contains metadata about the session. The application uses a forward-delta design, where each snapshot contains only the most recent changes plus a pointer to the previous snapshot.

To generate a real-world example, I perfomed the following actions:

- created a repository (mkrepo)
- initiated a session named "marc" (mksession)
- imported an existing directory into that session (import)
- performed a number of checkin operations after
    - adding a new file
    - modifying that file
    - deleting the file

At the conclusion of these steps, the blob folder contained a number of data files organized using checksums:

```
+---blobs
|  +---0d
|  |    0db681d54a4a82af6645a84cd14deb83
|  |
|  +---41
|  |    41fb12e05392a5f86d90f73e058ecbff
|  |
|  +---52
|  |    5247b1f801a118ddda9ca6e2a5267d15
|  |
|  \--- . . .
```

And the sessions folder contains a series of subfolders, named using integer ID numbers:

```
+---sessions
|  +---1
|  |      bloblist.json
|  |      d41d8cd98f00b204e9800998ecf8427e.fingerprint
|  |      session.json
|  |      session.md5
|  |
|  +---2
|  |      657cd087f9d670a666584e0b9b47b265.fingerprint
|  |      bloblist.json
|  |      session.json
|  |      session.md5
|  |
|  +---3
|  |      b9656f2aeb88a48387727839b794e08f.fingerprint
|  |      bloblist.json
|  |      session.json
|  |      session.md5
|  |
|  +---4
|  |      fa1372cff33ec668efcb02bb7ec2795c.fingerprint
|  |      bloblist.json
|  |      session.json
|  |      session.md5
|  |
|  \---5
|         657cd087f9d670a666584e0b9b47b265.fingerprint
|         bloblist.json
|         session.json
|         session.md5
|
\---tmp
```

Each of these numbered folders contains a point-in-time "snapshot" of the directory that was associated with the session. The integers are used as snapshot IDs. The session name is only viewable as a part of the metadata inside each snapshot's session.json file, which also contains a pointer to the previous snapshot ID for that session. The bloblist.json file contains fixity information about the files that were added or modified since the previous snapshot of the given session. Note that this implements a file-level forward delta mechanism. To reconstruct the session's original directory at a given point in time, the application must start with the first snapshot and work forward.

Snapshot 1 – Create a new empty session

bloblist.json

```
[]
```

session.json


```
{
  "base_session": null,
  "client_data": {
    "date": "Sun Oct 02 03:43:18 2011",
    "timestamp": 1317548598,
    "name": "marc"
  },
  "fingerprint": "d41d8cd98f00b204e9800998ecf8427e"
}
```


Snapshot 2 – Import a working directory's contents into the session and check it it

bloblist.json


```
[
  {
    "size": 20738,
    "md5sum": "67e3572b12802de074f87bd76f973c24",
    "mtime": 1215443688,
    "ctime": 1215443688,
    "filename": "NML MARC Samples.txt"
  },
  {
    "size": 71103,
    "md5sum": "a2cd505361ad7be384d900c2ba2754a9",
    "mtime": 1215442871,
    "ctime": 1215442812,
    "filename": "NML-MARC21.xml"
  },
  {
    "size": 112490,
    "md5sum": "6bd3ef5e2d25d72b028dce1437a0e89a",
    "mtime": 1215443139,
    "ctime": 1215443138,
    "filename": "MARC21slim2MODS3-2.xsl"
  }
]
```


session.json


```
{
  "base_session": 1,
  "client_data": {
    "date": "Sun Oct 02 03:45:36 2011",
    "timestamp": 1317548736,
    "name": "marc"
  },
  "fingerprint": "657cd087f9d670a666584e0b9b47b265"
}
```

Snapshot 3 – Add a new file to the working directory and check it in

bloblist.json


```
[
  {
    "size": 16,
    "md5sum": "c8fdfe3e715b32c56bf97a8c9ba05143",
    "mtime": 1317549043,
    "ctime": 1317549026,
    "filename": "mynewfile.txt"
  }
]
```


session.json


```
{
  "base_session": 2,
  "client_data": {
    "date": "Sun Oct 02 03:52:31 2011",
    "timestamp": 1317549151,
    "name": "marc"
  },
  "fingerprint": "b9656f2aeb88a48387727839b794e08f"
}
```


Snapshot 4 – Modify that same file and check it in the changed file

bloblist.json


```
[
  {
    "size": 26,
    "md5sum": "98844205481d4f3994902df393f22590",
    "mtime": 1317550813,
    "ctime": 1317549026,
    "filename": "mynewfile.txt"
  }
]
```


session.json


```
{
  "base_session": 3,
  "client_data": {
    "date": "Sun Oct 02 04:20:37 2011",
    "timestamp": 1317550837,
    "name": "marc"
  },
  "fingerprint": "fa1372cff33ec668efcb02bb7ec2795c"
}
```

Snapshot 5 – Delete the file from the working directory and check in the change

bloblist.json


```
[
  {
    "action": "remove",
    "filename": "mynewfile.txt"
  }
]
```


session.json


```
{
  "base_session": 4,
  "client_data": {
    "date": "Sun Oct 02 04:47:00 2011",
    "timestamp": 1317552420,
    "name": "marc"
  },
  "fingerprint": "657cd087f9d670a666584e0b9b47b265"
}
```