REST VOL for HSDS and HDF Sharded Data Storage

John Readey

The HDF Group

Overview

- Sharded Data Storage
- REST VOL
- Direct Access

Sharded data concept

Instead of managing HDF5 objects (datasets, groups, chunks) within a POSIX file store

them as separate files (or as objects within an object storage system such as S3)

For meta data (datasets and groups), a self-descriptive format such as JSON can be used

For chunks, store as binary objects for efficiency

Why a sharded data format?

- Limit maximum size of any object
  - -> Object storage systems typically don't support partial writes, so large objects are inefficient to update
- Supporting parallelism is easier
  - -> no file locking needed
- No need to manage free space, key-value mappings, etc
  - -> storage systems have gotten pretty good at doing this for you
- No need to worry about system crash leaving you with a corrupted HDF5 files
  - -> worse case you lose one object, with object storage not even that

Case against sharded storage

- Convenience of having one file vs lots of small files
  - Maybe not as important given tooling to abstract this from the user
    - E.g. hstouch, hscopy, hsrm tools
- Filesystems have trouble dealing with large number of files (particularly within one directory)
  - Not sure this is a problem with modern Linux filesystems
  - Certainly not an issue with object storage systems

HSDS shard schema example

```
root_obj_id/
   group.json
   obj1_id/
      group.json
   obj2_id/
      dataset.json
      0_0
      0_1
   obj3_id/
      dataset.json
      0_0_2
      0_0_3
```

Observations:

- Metadata is stored as JSON
- Chunk data stored as binary blobs
- Self-explanatory
- One HDF5 file can translate to lots of objects
- Flat hierarchy – supports HDF5 multilinking
- Can limit maximum size of an object
- Can be used with Posix or object storage

Schema is documented here:
https://github.com/HDFGroup/hsds/blob/master/docs/design/obj_store_schema/obj_store_schema_v2.md

Storage Partitioning

- It can be useful to divvy up the objects within an HDF5 domain into roughly equal size collections – for instance we have n workers and we'd like to perform some action on the domain
- CRUSH algorithm approach: hash key and take modulo of number of workers
  - Decentralized, no book keeping required

Conversion from HDF5 files to sharded files

- The tool “hsload” will convert an HDF5 file to the sharded format (using HSDS)
- Conversely, the “hsget” tool will take the shaded format and reconstruct the HDF5 file
- Data is preserved after a round trip

HDF5 file linking

- Converting large HDF5 files (or a large collection of files) to the sharded format is time consuming and effectively doubles the storage requirements
- Rather than converting the entire file to the HDF Schema, just the metadata can be imported (typically <1% of the file)
- The sharded format will store a map to the chunks in the original file
- Dataset reads are converted to Range Gets on the stored file
- It is also possible to construct a server file that aggregates multiple stored files (similar to how the HDF5 library VDS feature works)

REST VOL Plugin

- The HDF5 VOL architecture is a plugin layer for HDF5
- Public API stays the same, but different back ends can be implemented
- REST VOL substitutes REST API requests for file i/o actions
- C/Fortran applications should be able to run with minor tweaks
- Downloadable from: https://github.com/HDFGroup/vol-rest

  For HDF5 1.12, use the hdf5_1_12_update branch

Features not yet supported

Feature	HSDS	h5pyd	RESTVOL
Object reference	☺	☺	☹
Region Reference	☹	☹	☹
Fill Value	☹	☺	☹
Virtual Datasets	☹	☹	☹
Variable Length Datatypes	☺	☺	☹
Dataset SQL Query*	☺	☺	☹

- *Need new VOL api?

REST VOL Wishlist

Interesting things that would be nice to have:

- Support multi-threaded clients
- Support asynchronous API
- REST VOL activation based on file prefix – e.g. “hdf5://”
- Paginated read/write for large dataset operations
- Retry logic for HTTP timeouts, Service Unavailable
- Support for other languages: Java, .Net, R, etc.

Pros and cons of running a service

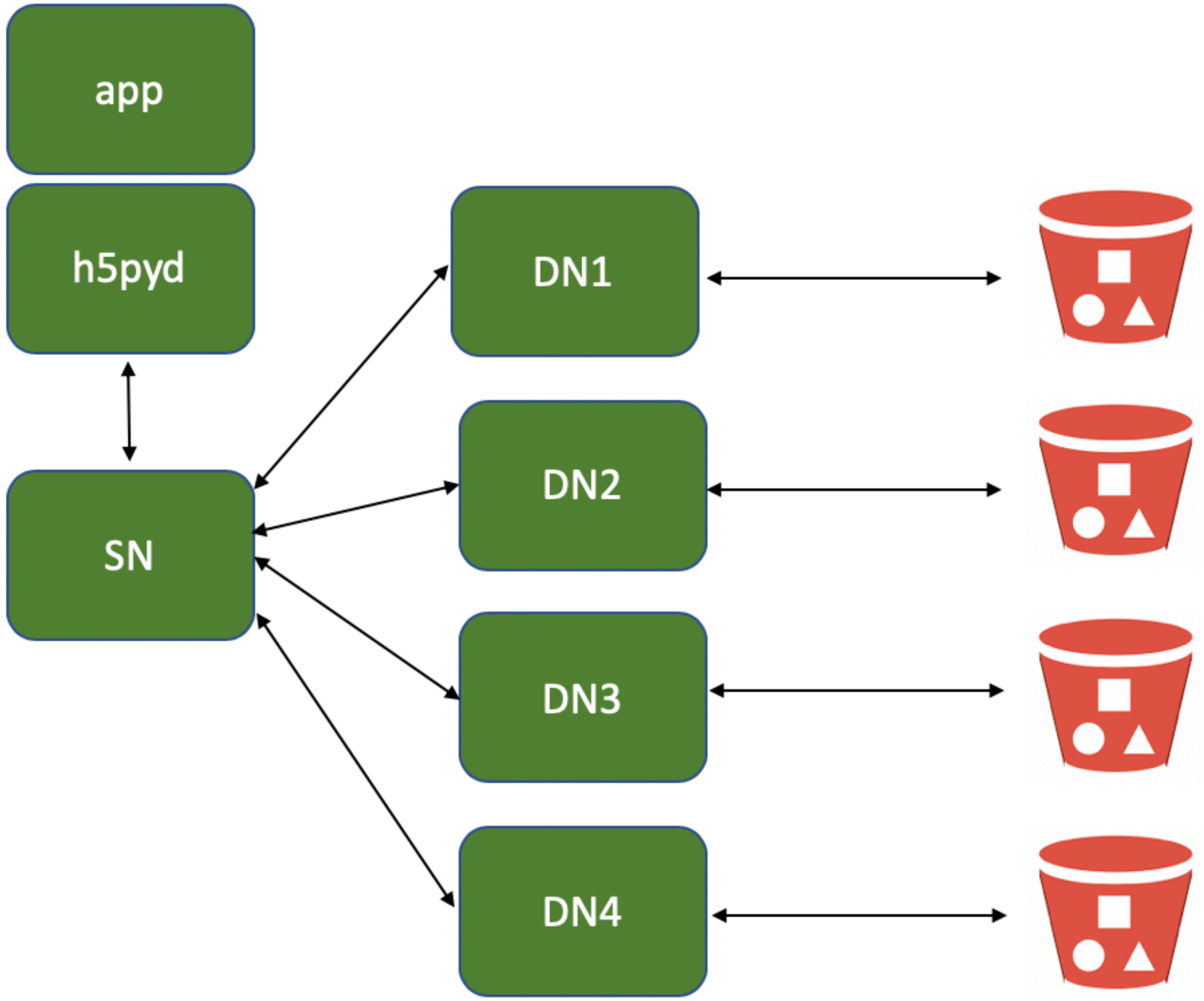
- Accessing a sharded data store via a service (HSDS) is nice:
  - Server mediates access to the storage system
  - Server can speed things up by caching recently accessed data
  - Only the data the client needs needs to be transmitted outside the data center
  - HSDS running on a large server or cluster can provide more processing capacity than a client might have
- Unless it’s not:
  - Don’t want to bother setting up, running service
  - Challenge to scale capacity of service to clients

Direct Access Project

Provide equivalent functionality of HSDS in a library

- SN code would run in a sub-process
- DN code would run in one or more sub-processes (e.g. based on number of cores)
- Sub-processes would directly access storage system
- Communication between parent processes and sub-processes would be http via localhost
- Sub-processes shutdown when last file is closed
- The same HSDS storage schema would be used
  - Can switch between direct access and server as needed

Diect Access System Diagram



Direct Access VOL plugin

- For C/C++ apps, the direct access model could be implemented as a VOL connector
- Other than launching the sub-processes the VOL would work in the same way as the REST VOL, so it probably makes sense to include this functionality in the REST VOL rather than create a new VOL
- With direct access HDF5 lib + REST VOL enables sharded data as an alternative to the HDF5 file format
  - Enables multi-threading
  - Cloud optimized storage
  - Crash-proof

Questions?

Try it out!

Get the software here:

- HSDS: https://github.com/HDFGroup/hsds
- H5pyd: https://github.com/HDFGroup/h5pyd
- REST VOL: https://github.com/HDFGroup/vol-rest
- REST API documentation: https://github.com/HDFGroup/hdf-rest-api
- Example programs: https://github.com/HDFGroup/hdflab_examples