SDFS Overview

By Sam Silverberg

Why did I do this ?

▸ I had an Idea that I needed to see if it worked….

Design Goals

▸ Create a dedup file system capable of effective inline deduplication for Virtual Machines

  ▸ Support Small Block Sizes (4k)
  ▸ File Based cloning
  ▸ Support NFS and CIFS

▸ Global Deduplication

  ▸ Between volumes
  ▸ Between Hosts

▸ High Performance

  ▸ Multi threaded
  ▸ High IO concurrency
  ▸ Fast reads and writes

▸ Scalable

  ▸ Able to store a petabyte of data
  ▸ Distributed architecture

▸

Why User Space and Object Based

- Quicker for my Iterative Development Cycle
- Platform Independence
- Easier to scale and cluster
- Interesting opportunities for data manipulation
- Provides flexibility for integrating with other user space services such as S3
- Leverage exists file system technologies such as replication and Snapshoting without re-inventing the wheel
- Deduplication lends itself well to an object based file system.
- Little or no performance impact
  - @128k chunks
  - 150 MB/s + Write
  - 150 MB/s + Read
  - 340 MB/s Re-Write
  - 3 TB of Data
  - 2 GB of RAM

Why Java

▸ Cross Platform Portability

▸ Threading and Garbage Collection works well for large number of objects

▸ Good integration with web/cloud based storage

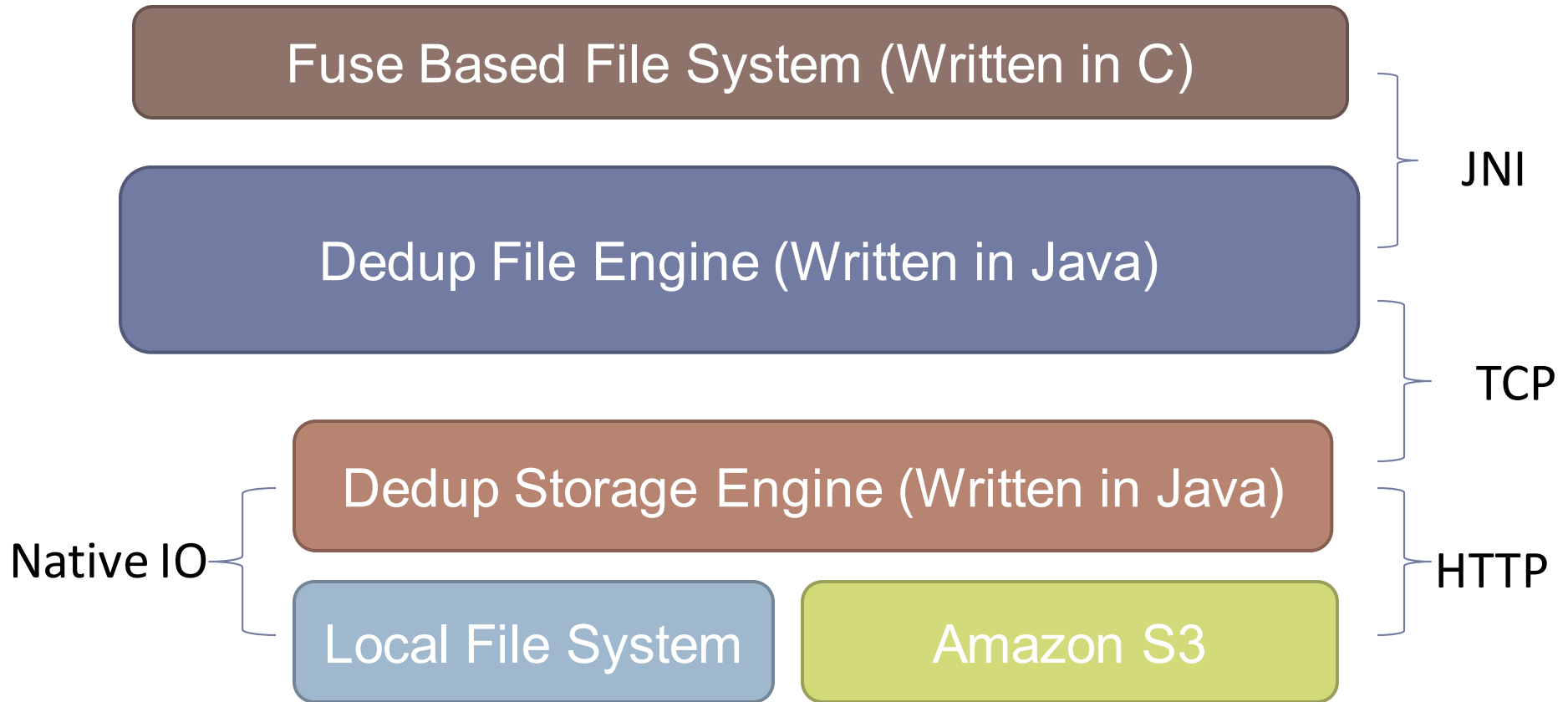▸ No performance impact - Using Direct ByteBuffers makes it as fast as native code

The Architecture

Terminology

▸ SDFS Volume : The mounted deduplicated volume. A volume contains one Dedup File Engine and one Fuse Based File System

▸ Dedup File Engine : The client side service, within a mounted SDFS Volume, that manages file level activities such as file reads, writes, and deletes.

▸ Fuse Based File System : User level file system that is used to present files, contained within the Dedup Engine as a file system volume.

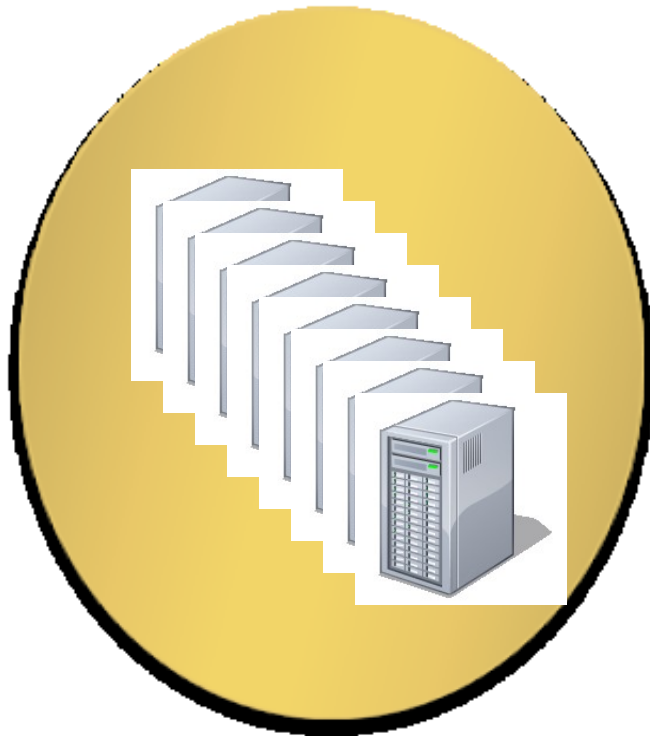▸ Dedup Storage Engine : The server side service that stores and retrieve chunks of deduped data
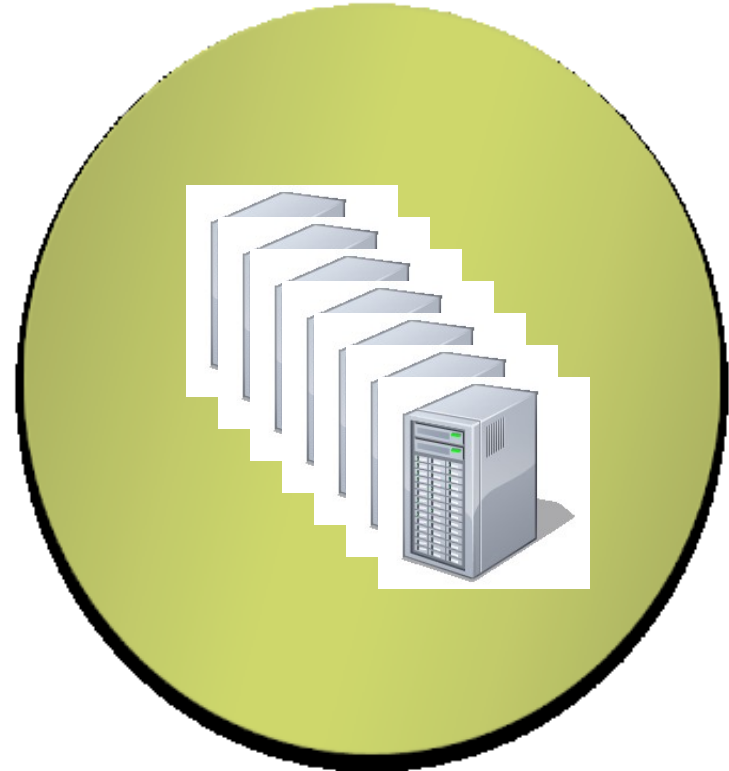
Overview

Fuse Based File System (Written in C)

Dedup File Engine (Written in Java)

JNI

TCP

Dedup Storage Engine (Written in Java)

Native IO

HTTP

Local File System

Amazon S3

Physical Architecture

Dedup File Engines

Dedup Storage Engines

TCP/UDP

Hashes are routed to Dedup Storage Engines based on the 1st byte of the Tiger hash.
Up to 256 Storage Engines allowed.

Fuse Interface

- Uses Fuse-J JNI interface
- File Statistics and Advanced file manipulation is done though getting and setting extended file attributes (getfattr setfattr)
- Each mounted volume uses it own Fuse-J interface and Dedup File Engine instance

Dedup File Engine

- Dedup File Engine is made of 4 basic components
  - MetaDataDedupFile. Contains all the meta-data about a file in the mounted volume. File Path,Length, Last Accessed, Permissions…
  - DedupFile. Contains a map of Hashes to File Locations for dedup data. Persisted using Customer Persistant Hashtable (Really Fast and Light Memory)
  - Dedup File Channel. Interfaces between Fuse-J and other DedupFile for IO commands
  - WritableCacheBuffer. Contains a chunk of data that is being read or written to inline. DedupFile(s) cache many of these during IO activity.
- MetaDataDedupFiles are stored in the MetaFileStore. The MetaFileStore persists using JDBM
- DedupFiles are stored in the DedupFileStore. They are persisted independently as an individial disk based hashtable.

Steps to Persisting A Chunk of Data

1. WritableCacheBuffer is aged (LRU) out of the DedupFile write cache and is tagged for writing
2. Byte Array within WritableCacheBuffer is hashed and passed to HCServiceProxy
3. HCServiceProxy looks up route, to the appropriate dedup storage engine, for the hashed chunk (WritableCacheBuffer) based on the 1st byte of the hash.
4. Based on route HCServiceProxy queries the appropriate dedup storage engine to see if the hash is already persisted.
5. It the hash is peristed then no other action is performed. If not the hash is sent to the dedup storage engine either compressed or not based on configuration settings.

Dedup Storage Engine

- **3 Basic Pieces to the Architecture**
  - HashStore. The hash store contains a list of all hashes and where the data represented by that hash is located. The hashstore is persisted using custom hashtable. The custom hashtable is fast and memory efficient
  - ChunkStore. ChunkStore is responsible for storing and retrieving the data associated with a specific hash.Currently three ChunkStores have been implemented
    - FileChunkStore – Saves all data in one large file
    - FileBasedChunkStore – Saves all chunks as individual files
    - S3ChunkStore – Saves all data to AWS (s3)
  - The chunk of Data. Data is passed as byte arrays between objects within the Dedup Storage Engine.

Steps to Persisting A Chunk of Data

1. ClientThread recieves a WRITE_HASH_CMD or a WRITE_COMPRESSED_CMD and reads the md5/sha hash and data

2. ClientThread passes hash to the HashChunkService

3. HashChunkService passes hash and data to the appropriate HashStore

4. HashStore stores the hash and sends the data to the assigned ChunkStore

5. ChunkStore stores data.

▸ Only One type of ChunkStore can be used per Dedup Storage Engine Instance.

Deduplication Concerns

▶ Data Corruption

  ▶ Files can be corrupted if the Dedup File Engine is killed unexpectantly. This is because written data is cached in memory until persisted to a backend Dedup Storage Engine.

  ▶ Solution : cache data to memory and write locally until write to Dedup Storage Engine is complete, then flush both. This option is enabled using safe-sync=true.

▶ Memory Footprint

  ▶ The Dedup Storage Engine uses about 8 GB of memory for every TB used at 4k blocks. BTW … This is linear so it would be 32 TB at 128k blocks

  ▶ Solution : Scale out with RAIN configuration for DSE.

▶ Files with a high rate of change

  ▶ Files with a high rate of change and lots of unique data, such as page files or transaction logs will not effectively use dedup.

  ▶ Implemented Solution : Selective block level deduplication.

▶ Performance suffers with copying large number of small files (< 5 MB)

  ▶ A meta data dedup file instance is created per File. This is fast, but not as fast as IO.

  ▶ Possible Solution : Create a pool of Hash Tables and allocate them on the fly as new files are needed

▶

Today's Solutions - Considerations

▸ Scalability : Most dedupe solutions are per volume or per host. SDFS can share dedupe data across hosts and volumes providing expanded storage.

▸ Store at 128k chunks and not tuned for small block size deuplication. This will effect memory usage considerable and possibly increase memory requirments 32x.

▸ Large block size does not work for VMDKs

▸ Today's dedup solutions only solves the problem of storing dedup data not reading and writing inline.

Native File System Considerations

▸ A volume is monolithic which would make inter host deduplication difficult.

▸ Deduplication requires a hash table. Scalability could not be maintained if a hash table is required per instance of a volume.

▸ Hashes will be difficult dedup between volumes

▸ File Level Compression will be difficult with file system deduplication