Indian Institute of Technology Delhi

Dept. of Computer Science

B.Tech Thesis

HStore

Integrated Storage on Hadoop

Authors:
Rayala Bharadwaj
VAVS Nikhil

Supervisor:
Dr. Suresh C Gupta

June 14, 2016

Acknowledgements

We would like to thank Prof. Suresh C Gupta for his guidance and motivation throughout the project. There were many times when we were drifting away and he motivated us to work harder. His constant guidance and tremendous experience were just what were needed for the successful completion of the project. Our sincere gratitude to him.

We also thank Prof. Vinay Ribeiro for his involvement in the committee and his encouragement.

Bharadwaj
Nikhil

Contents

Chapter 1

Introduction

1.1 Software Defined Storage

Software-defined storage (SDS) is an approach to data storage in which the programming that controls storage-related tasks is decoupled from the physical storage hardware. In this way, the pooled storage infrastructure resources in a software-defined storage (SDS) environment can be automatically and efficiently allocated to match the application needs of an enterprise. In SDS, the main emphasis is given to the storage related services rather than the storage hardware.
This decoupling of the data plane from the control plane adds a lot of flexiblity and enables centralized decision making about how resoures operate. For example, because the hardware is separate from the software layer, we can easily upgrade the hardware resources without affecting the storage services.

1.1.1 Key Characteristics

Software Defined Storage differs from traditional storage in several architectural elements.

1. Commodity Hardware - All the intelligence in SDS is placed in the software layer so that we can use simple commodity hardware for the physical storage.

2. Scale-out Architecture - SDS uses a building-block approach to storage that allows users to dynamically add and remove resources.

3. Resource Pooling - All the available storage resources are pooled into a single logical entity that is managed centrally.

4. Automation - Extensive automation is provided through which users can request storage resources in terms of capacity and performance rather than physical location of drives.

5. Programmability - In addition to the in-built automation, rich APIs are provided using which the administrators and third-party applications can integrate the control plane across storage layer and other layers to deliver work flow automation as per their requirements.

1.2 Motivation

Enterprise storage and cluster storage systems are currently dominated by vendor specific storage systems with proprietary protocols running on custom made hardware. Several examples of these would be SDN, NAS and various forms of cloud storage like Azure, Amazon S3 etc. These system are often costly and 'vendor-locked'. Since all the requirements of enterprise storage like reliability, scalability and fault-tolerance are implemented in hardware they are rarely customizable.

Motivation of this project is to develop a solution which while retaining the benefits of these solutions like reliability, scalability and fault-tolerance can do away with the problems like high cost, rigidity to customization, being vendor specific and proprietary. The concept of SDS defined above allows us to do this.

1.3 Target of Project

The aim of this project is to build a software defined storage solution which provides all three software systems - File Storage, Block Storage and Object Storage in software. A higher goal of the group of the projects this is a part of is to provide our own academic cloud with a software defined storage solution for its storage needs.

Hadoop is a well established, robust SDS which provides File Storage. We intend to build Block Storage and Object Storage on top of Hadoop, thus providing a complete software system. This is the target of our project.

Chapter 2

Literature Review

We use existing technologies in Software Defined Storage like HDFS and Hbase as building blocks and build on top it. We shall briefly review them before we discuss our object and block storage.

2.1 Hadoop

"Apache Hadoop is a framework that allows distributed processing of large data sets across clusters of computers using simple programming models".

It is designed to run on a cluster of commodity computers and rely on software for scalability, reliability and performance. It software layer converts a set of fault prone hardware into a single fault-tolerant system with graceful fault detection and handling. It is composed of two sub-components, Hadoop Distributed File System(HDFS) and Hadoop MapReduce. HDFS is a distributed fault-tolerant file system which is designed to run on commodity or cheap hardware.It is suitable for large data sets and streaming access to files with high throughput.

HDFS is a master/slave architecture. HDFS system consists of two type of nodes - a namenode and several datanodes. Namenode contains the information regarding which parts of a file is present in which datanodes. This information is maintained for every file in the namenode in its memory. Datanodes contain the data corresponding to files. Namenode allocates datanodes as to which blocks of a given file each datanode has to store. Clients then directly talk with the datanode to read data from the datanodes. Simply put, namenode handles all the namespace operations like opening and closing files, renaming files and storing mappings from blocks to datanodes. Datanodes handle data requests like read and write requests for a file from the file system clients, block creation, deletion requests etc. Data is replicated on different datanodes for fault tolerance and higher bandwidth depending upon the replication factor.

Mapreduce is a framework for scheduling jobs and managing cluster resources in hadoop. Hadoop bases itself one important assumption that if often true that

"Moving Computation is Cheaper than Moving Data". This becomes perticularly relevant when the dataset is huge. A MapReduce job usually splits the input data-set into independent chunks which are processed by the map tasks in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks. When the compute nodes and the storage nodes are the same, which is the case typically, it is computation that has actually moved to the data. This breakup of tasks into map and reduce gives us a great flexibility in running jobs parallely as all map tasks are independent of each other.

2.2 HBase

HBase is a scalable, distributed, big datastore on Hadoop. It supports very large tables with billions of rows and millions of columns. It is non-relational and modeled after Google's Bigtable. Some of it's important features are

- Random real-time read/write access to bigdata.
- Strictly consistent reads and writes
- Automatic failover support
- Linear and modular scalability.

Hbase easily integrates with hadoop, leveraging its Hadoop Distributed File System. We use Hbase in our system when we need to store massive amounts of data and want to read and write to it randomly and quickly.

2.3 Existing Solutions

2.3.1 Ceph

Ceph is a distributed storage system supported by Inktank designed to provide Object, Block and file storage. It aims to be completely distributed without a single point of failure, fault tolerant by data replication and highly scalable.
Ceph provides seamless access to objects using native language bindings or RADOS (Reliable Autonomic Distributed Object Store) gateway, a RESTful interface that's compatible with applications written for Amazon S3 and OpenStack Swift.
Ceph's RADOS Block Device (RBD) provides access to block device images that are striped and replicated across the entire storage cluster. Ceph's RBD also integrates with Kernel Virtual Machines (KVMs). Several IaaS cloud platforms (i.e.: OpenStack, CloudStack) officially support Ceph to provide a block storage solution.
The Ceph File System (Ceph FS) is a POSIX-compliant filesystem that uses a

Ceph Storage Cluster to store its data. Ceph FS uses the same Ceph Storage Cluster system that provides object storage and block storage interfaces.
Ceph has 4 different kinds of daemons.

- Cluster Daemons - Keeps track of active and failed cluster nodes
- MetaData Server - Stores metadata of inodes and directories.
- Object Storage Devices - Stores the content of the files
- REST gateways - Exposes the object state layer as an interface compatible with Amazon S3 and OpenStack Swift APIs.

Chapter 3

Object Storage

3.1 Definition

Object Storage is a storage architecture that manages data as objects. A simple example for an application that requires object storage could be any software system that has large user-generated content, like Facebook, twitter and a lot of social networking sites. They would need to store unstructured data like images, music, videos, and documents. Microsoft's Azure, Amazon S3 are some examples of object stores in public clouds.
Objects in object storage system can be of any size ranging from small objects like text files to very large objects like HD videos. Each object also has some metadata associated with it. The metadata and data are often physically separated to achieve better management and indexing purposes.
Object storage system could be compared to a valet parking system at a restaurant. In valet parking, the customer gives the car to a valet and gets a receipt in return. The customer does not know where the car is parked or if the car was moved from one parking spot to another when he was in the restaurant. When he gives the receipt back, the car is returned by the valet. Similarly in the object storage system, the user gives the object to the object storage system and saves it. He does not how or where it is stored, and when he supplies the object key, the object's data is returned to the user.

3.2 Advantages of Object Storage

- Unlike files in a file storage system which have fixed metadata like file size and date of creation, objects can have rich user-defined metadata. Metadata in object storage system can have any number of custom defined metadata attributes.

- Protocol Support - Traditional file system protocols (CIFS and NFS) communicate on TCP ports that are available on internal networks, but are

not usually exposed to the Internet. But, object storage system is usually accessed through a REST API over HTTP.

- Scalability - Object storage works well as a scalable data store for unstructured data which are not frequently updated. In cloud storage systems, it is well suited for file content like images and videos.

3.3 Basic Concepts

3.3.1 Users

TODO ?? Fill something
Consider the URL https://objStore.iitd.ac.in/cs907263/btp/src.zip. In this URL, cs907263 is the owner user of the the object

3.3.2 Buckets

A bucket is a container for objects and contain many objects in it. Buckets serve several purposes: they organize the the system namespace at the highest level, they identify the account responsible for storage and data transfer charges, they play a role in access control.
In the URL https://objStore.iitd.ac.in/cs907263/btp/src.zip btp is the name of the bucket.

3.3.3 Objects

Objects are the fundamental entities in an object storage system. Each object has the object data and some metadata along with it. The metadata is a set of name-value pairs that describe the object. These might include some default metadata stating when the object was last modified or standard HTTP metadata like Content-Type, Character Encoding etc. The user can also add some additional metadata to the object while storing the object. In the URL https://objStore.iitd.ac.in/cs907263/btp/src.zip, src.zip is the name of the object.

3.3.4 Keys

Both the buckets and the object have a single identifier called keys. An object key is unique within a bucket. So, a bucket key and an object key uniquely define an object. In the URL https://objStore.iitd.ac.in/cs907263/btp/src.zip, cs907263 is the owner of the object with the key src.zip in the bucket with the key btp.

3.3.5 REST API

REST is short for REpresentational State Transfer. The REST API is an HTTP interface to our implementation. Each request from any client contains all the

information necessary to service the request i.e. no session or state is maintained on the server.The URI uniquely identifies the resource and the body contains the state (or state change) of that resource. Using REST, we use standard HTTP requests to create, fetch, and delete buckets and objects. We can use any toolkit (e.g. Postman) that supports HTTP to use the REST API.

3.4 Requirements

We expect the following functionalities to be provided by the Object Storage System.

1. Support multiple users.
2. Create/Delete Buckets - A bucket has a unique name. A bucket once created cannot be renamed and can be deleted only if it is empty.
3. List the objects in a bucket
4. Create/Read/Delete Objects in a bucket - Each object is in a bucket and the object key is unique within that bucket.
5. The system should be scalable and Crash Tolerant.

3.5 Problems with existing work

Each directory, file, block in HDFS take 150 bytes in the namenode's memory. So, 20 million small files would take up 3GB of ram. With this problem, scaling to huge number of files is not possible. But, an object storage system would need to handle a lot of files and this can include many small files. So, the namenode's memory could be easily filled up without properly utilising the cluster's storage capabilities. This would lead to wastage of cluster space.
In normal operation, the namenode must constantly track and check where every block of data is stored in the cluster. This is done by listening for data nodes to report on all of their blocks of data. As the number of blocks for which the datanode has to send the status to the namenode increases, the network bandwidth consumed by the datanode will increase. Even if there are high-speed interconnects between the nodes, at a large scale simple block reporting could become troublesome because of the number of blocks.
We need a design where large number of small files are handled efficiently without taking up too much memory in the namenode and reporting their status should not take too much bandwidth. The design of the proposed system is explained below.

3.6 Design

3.6.1 Overview

We use Hadoop and HBase to design our object store. Hadoop Distributed File System is a reliable, fault tolerant and scalable data. It is designed to run on commodity hardware. It is a master-slave architecture where the master (namenode) stores the metadata and location of the data, for each file. Files are stored in blocks in HDFS. These blocks are typically large with size around 64MB. This is because of the following reasons.

- Metadata for each block of each file is stored 'in memory' in the Hadoop namenode. This constraints the maximum number of blocks stored in HDFS. Thus higher the block size, higher the number of files that can be stored.
- Also each read file request to HDFS has a seek overhead and a transfer overhead. Seek overhead is the initial overhead to connect to the datanode and find the corresponding block. Transfer overhead is the overhead to transfer the block to the client. Seek overhead is governed by the number of blocks a file contains. Transfer overhead is governed by the network and system bandwidth. Having a very small block size would increase the 'seek overhead'.

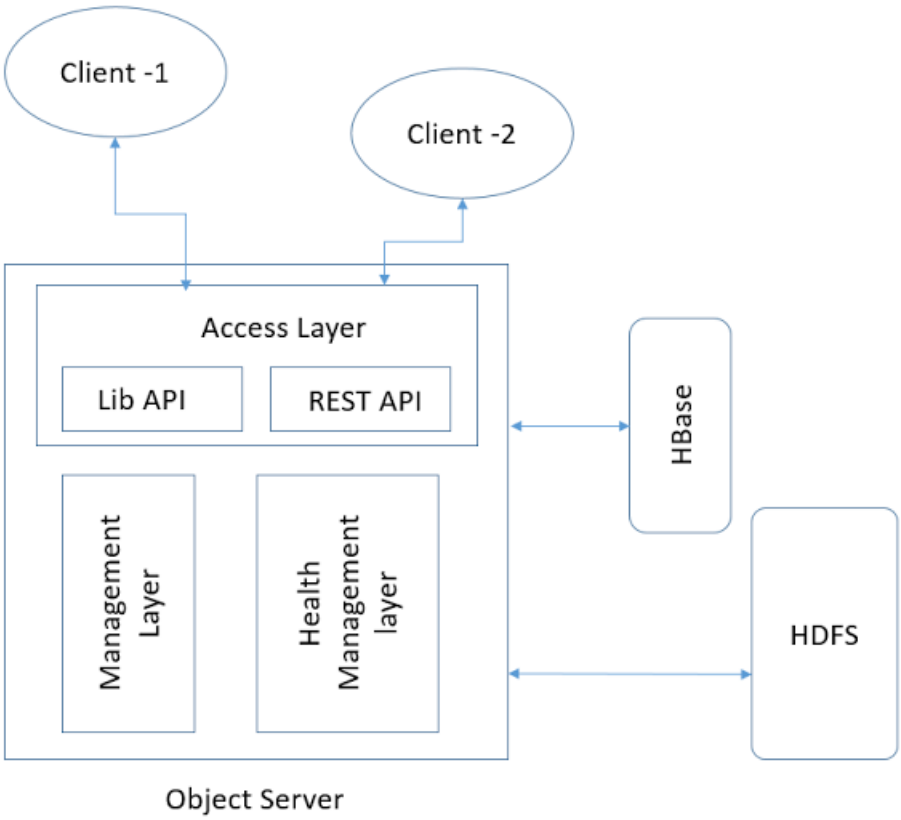
Thus HDFS is ideal for large files stored in large blocks.
Object store on the other hand should support a large number of objects with greatly varying sizes. Thus a large number of small files is a problem. We solve this problem by archiving several small files into one big file and storing this in HDFS. We use Hadoop's HarFileSytem and HadoopArchives classes to achieve this. Objects are divided into two classes for this purpose – Large object and Small Objects.
Small Files for this project is defined as any file that is less than half the default block size. This threshold is variable and can be set according to the cluster's storage and the namenode's memory. Any file larger than this is considered a large file.

For the remainder of this document, we define the following terminology.

Table 3.1: Object Store Terminology

Term	**Meaning**
Small File	File Size $<$ 0.5*Hadoop Block Size
Large File	File Size $\geq$ 0.5*Hadoop Block Size size
HAR file	Hadoop Archive File – generated by archiving multiple small files into a single archive

3.6.2 Architecture



3.6.3 HDFS

HDFS is used to store the object data. There are three kinds of files/directories in HDFS.

1. Directories, which are used to represent buckets. All files in a folder belong to the same bucket.

2. Files in the directories, represent objects. These objects are stored in the buckets directory only if they belong to the class of large objects.

3. Archive Files, store groups of small objects in single files. Before a small file is archived it is present in the 'small Files' directory of the user. After they are archived they are stored in an archive file in the 'har files' directory of the given user.

3.6.4 HBase

HBase is the database of the system. It is used to store metadata and several other tables like the key → URI mappings, the health of the archive files, buckets for a given user etc. There are four tables in the Hbase storing the following information.

Table 3.2: Hbase Tables

Table Name	**Information Stored**
Buckets Table	Stores the list of buckets for each user
Objects Table	Key to URI mappings, metadata and properties of the object(file size, time stamp)
Har Files Table	Maintains the health of har files.
Delete Files Table	Stores the list of files that should be deleted during clean up.

3.6.5 Access Layer

Access layer is interface through which clients connect to the object store. Currently, REST interface and JAVA library API have been implemented to access the object store. This layer communicates with the management layer for serving any queries or requests.

3.6.6 Management Layer

This layer is responsible for creating, deleting the buckets(folders in HDFS), storing the objects in proper folders in HDFS and maintaining proper key to URI mappings in the *Objects Table* in HBase.
This layer is called by the Access layer to perform the tasks and performs the following operations.

Put Bucket

This operation is used to create buckets for the given user. An entry is added to the *Buckets Table* for the user. A new directory is created in the HDFS with the same name in which the objects (only the large objects as explained in 3.1) in this bucket will be stored.

List Buckets

Lists all the buckets for a given user. This scans the *Buckets Table* based on the userid to retrieve all the bucket IDs and returns the list of buckets.

Delete Bucket

This operation succeeds only if there are no objects in the given bucket. Scans the *Objects Table* to find if any object is present in the given bucket. If no object is present, deletes the corresponding directory in HDFS and removes its entry in *Buckets Table*. If not, this operation fails.

List Objects

List all the objects in given bucket. Scans the *Objects Table* and returns the list of all objects in the given bucket.

Put Object

Puts an object into the given bucket key. If the object is a large object then the object is put in the folder corresponding to the bucket. If the object is a small object then it is put in the 'small Files' directory of the user.
The name of the file is the objectKey appended with the timestamp. This makes the path of an object unique even for newer versions of the object. Old versions of the object have to be deleted from the HDFS. Maintaining uniqueness of the path of an object even across versions is helpful in the case of archiving (as will be explained in Archiver in Section 3.5.6). This way both the old object and the new object for the same object key can co-exist in the HDFS. The entry in *Objects Table* always points to the newest version of the object.
The *Objects Table* entry for the corresponding object is retrieved to check if an object exists with the given object key. If an object exists and is a large file, then object is deleted. If an object exists and is a small file then the old object is added to the *Delete Files Table*. The entry in the *Objects Table* is now updated to point to the new entry.

Get Object

Fetches the object's URI in HDFS and metadata from the *Objects Table*. The object is then fetched from the HDFS using its URI and returned to the user.

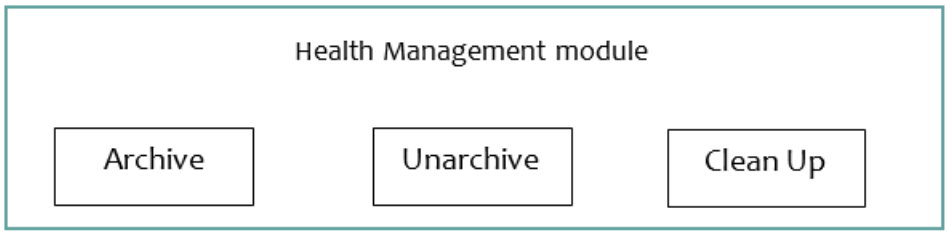
Delete Object

Corresponding entry for the given object Key in the *Objects Table* is deleted. If the object is a large object then it is deleted from the HDFS also. If it is a small object then its URI is added in the *Delete Files Table* which will be deleted by Clean Up module in Health management.

3.6.7 Health Management

This composes of various background tasks responsible to manage the health of the cluster. There are 3 jobs running in this layer.

1. Archiver job runs when there are too many small files in the cluster thus exhausting Hadoop namenode's memory.
2. Unarchiver runs when any archive has too much junk data that can be deleted.
3. Cleanup jobs runs to clean the HDFS of the junk data that present in the Filesystem.



Archiver

For every file in the hadoop cluster, some information about the file's location and metadata is stored in the namenode's memory. Thus a large number of small files (in this case, small objects) will fill up the namenode's memory prematurely without completely utilizing the space available in the cluster. This is solved by archiving many small files into one big HAR file (HAR – Hadoop Archives file). When the number of the small files crosses a certain threshold, or the size of the small files crosses some fixed size, this job selects a list of the small files, archives them into a single HAR file and updates the URI of the corresponding files (the small files which were selected to be archived) in the *objects Table*. While updating the entries, the timestamps of the file before and after archiving are compared to make sure that the entry in the table is the one corresponding to the file in the archive. If the time stamps are different, this means that the entry in the *objects Table* corresponds to a new object and not the one that is in the archive, and the file in the archive will be added to the *Delete Files table* to be deleted later.
Once the entries are updated in the table, the corresponding small files are no longer required as they are accessed through the HAR file. So, after the HAR file is created, all the small files are deleted which frees up the namenode's memory.

UnArchiver

HAR files are immutable i.e., after they are created they cannot be modified. So, a delete request to an object in the HAR file, will delete its entry in the *Objects Table* and this file will be deleted later by the Clean Up module. So, the HAR file will consist of some files which should be deleted.
We will define the health of a HAR file as the ratio of the amount of valid space in the archive (space occupied by valid files) to the actual space consumed by the HAR file. Initially the health of a HAR file will be 100%. But with each

delete request to an object in the HAR file, the number of valid files in the archive decreases, which means that the health of the archive file decreases.
This job scans the archive files and if the health of an archive file is less than a certain threshold, unarchives it (extracts the HAR file into individual small files), updates the entries of the files in the *Objects Table*, and deletes the HAR file.
Similar to the archive job, while updating the entries this job also compares the timestamps before and after unarchiving, and any small files which are no longer necessary are added to the 'Delete Files Table' to be deleted by the Clean Up module.

Clean Up

Delete request to a small file will result in only its entry being deleted from the *Objects Table*, and the path of the object is added to the *Delete Files Table* which will be deleted by this module. The file is not deleted directly. This is necessary because an archiver might be running in background and archiving this file, when the delete was called. The archiver expects the file to be present. So the file cannot be deleted as soon as the delete is called. Once the archiver finishes its job, the files can be deleted. Cleanup does these deletions.
Cleanup job is run every time after the Archiver or the Unarchiver job is run. It deletes all the files present in the *Delete Files Table* and clears the table entries.

Chapter 4

Block Storage

4.1 Definition

Block Storage is a storage abstraction where data is stored in volumes. Each volume is further divided into blocks. Each block is a sequence of bytes of fixed size. Each image will therefore contain a total of *imageSize/blockSize* blocks. Data is accessed whole blocks at a time rather than individual bytes.

Physical block storage devices are called hard disks and are connected to a computer with a SCSI or a SATA connector. In a cluster environment custom made block storage solutions are used which are connected to the servers with FC, FCoE, or iSCSI protocols. Virtualised block storage solutions provide this block and volumes abstraction in software using commodity network and existing storage resources. In block storage systems, raw storage volumes are created and these volumes can be treated as individual hard drives. This makes block storage usable for any kind of application including file storage, Virtual Machine File Systems etc. This abstraction of storage devices is then used by file systems or DBMS for use by applications and end users.

4.2 Purpose

In huge data centers and cluster environments often disk space(block storage) is provisioned using custom technologies like Network Attached Storage(NAS) or Storage Area Network(SAN). These technologies are costly and inflexible owing to their implementation in hardware and proprietary protocols. Software defined block storage can be a solution for these problems. A sub target of this project is thus to design and implement block storage in software on top of Hadoop.

4.3 Requirements

The basic requirement of a block storage system is the ability and provision and de-provision volumes as per user needs. Users should be able to access the volumes using blocks. They should be able to read blocks to get data from the volume. They should be able to store data in form of blocks in the volume. Block size should be comparable to the block sizes used currently in stock hard drives. These range typically from 4KB to 128KB. Users should also be able to create snapshots of the volumes and restore from snapshots as necessary. Users should also be able to extend and shrink their volumes as necessary. Briefly following are the functional requirements that the system should provide. All the functions are self explanatory.

```
public interface BlockProtocol extends VersionedProtocol{
    public String createImage(long size);
    public File getImage(String imageKey );
    public boolean extendImage(String imageKey, long  newSize);
    public boolean trimImage(String imageKey, long  newSize);
    public boolean deleteImage(String imageKey);
    public String takeSnapShot(String imageKey);

    public boolean writeBlock(String imageKey, BlockData block);
    public List<BlockData> readBlock(String imageKey, long addr);
    public boolean commit(String imageKey);
}
```

Following are the non functional requirements that the system should satisfy. The system should be scalable and resistant to failures. Once a crash occurs the system should be able to recover from the crash automatically. Changes once committed to the system should not be lost. The system should reach a consistent state on recovery after failures such as power failures. The system should be running indefinitely for a long time and should recover quickly after intermittent failures. The system should be scalable to hundreds of Terrabytes or PetaBytes of storage. System should provide higher bandwidth than typical commodity hardware or atleast match them in performance.

4.4 Design Iterations

Several designs were formulated and tested as a part of this project. Each design had some limitations which were overcome by the subsequent design. Final design was a result of two major redesigns from the initial design. You can directly skip to the final design subsection to read about the current design. We advise you to read and understand all the designs as this would clarify the thought process as to how the final design was achieved and how the intricacies of HDFS architecture.

4.4.1 HBSv1

Block sizes currently used by the hard drives are in the range of 4KB to 128KB. Therefore typical blocks of sizes around 64KB have to be written to hadoop chunks(chunks is used to denote the blocks inside hadoop using which files are stored in hdfs.) whose size is around 64MB. Our simple idea was to store multiple blocks sequentially in files of size 'chunksize' which would be 64MB. So in each file there would be a total of *chunksize/blocksize* blocks. This segregation of blocks into chunks implies that given a block address, the chunk number in which the chunk is present and the file offset inside the chunk can be calculated.

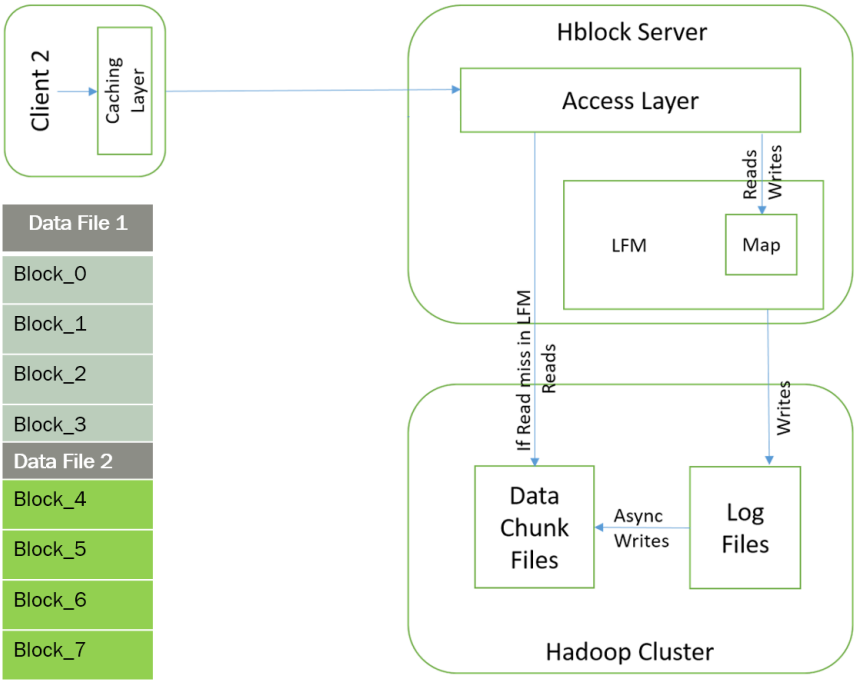
Overview

An image is represented by a folder in HDFS. A request for a new image results in creation of a new folder in the HDFS by the name of the image. Each file in the folder is of 1 chunk size. Each file is named chunk_0, chunk_1, chunk_2, etc. These files store data for the block device. Each chunk file contains data for cMB/bKB=(c/b)*1024 blocks and i^{th} chunk would contain blocks starting from (c/b)*i*1024.

Table 4.1: File contents

File Name	**Corresponding Address**
0	0 – c-1 MB
1	c – 2c-1 MB
2	2c – 3c-1 MB

Design



For a requested space of size x, we have x/c files in the corresponding HDFS folder. Writes to an address is stored in the corresponding files. Since these files are present in Hadoop file system which does not support modification of files, access (writes) to these have to be cached. Along with chunk files in each hdfs directory there is a logfile corresponding to that image. The protocol shall support following commands from the user. Logfile is an append-only file, one for each image in the HDFS, which stores the blocks that are received from the client.

Access Layer This layer provides the application calls that can be made by the client. All access to block storage have to be through this layer. All classes written in this layer should implement the BlockProtocol interface specified above. 'Hbs' is the default class which implements this protocol.

Caching Layer This layer is used for caching the calls to the network. This is implemented using the Apache's Java Caching System (JCS). By default we use a LRU memory cache of size 64MB, with MemoryShrinker enabled to shrink the unused memory space.

Log File Manager Log Manager Service logs all the writes to a log files in HDFS for recovery if the client fails after the cache write. Logfile is an append-only file, one for each image in the HDFS, which stores the blocks that are received from the client. This file is created with block size of 'clientBlockSize'.

Operations

Read Read request first hits the cache in the client. If it is a cache hit then the corresponding block is returned. If it is a cache miss, Hblock storage server is queried for the block. Storage server has to check the logfile for writes to the corresponding address. This is achieved by storing a map of the logfile in the storage server and querying the map. If map contains the address, the corresponding block is returned to the client. Else the block if fetched by reading the corresponding chunk from the HDFS. Read operation allows the storage server to send a list of blocks apart from the requested block to the client. This allows the storage server to guess the blocks that would be requested next or simply send the neighbouring blocks also.

Write When a write is sent to the Hblock storage server, it appends the write to an 'append only' log file and saves it in its write map. These writes are then asynchronously flushed to hdfs chunk files after the size of the logfile crosses a certain limit. Writes to the image when a logfile is being flushed are written to a secondary logfile which becomes the primary once the flush completes. Then the write map in the memory is also freed. The flush is done chunk by chunk so that all blocks belonging to the same chunk can be written at once.

Snapshot The directory for the current image is say A. On a takeSnapShot request, a new empty directory (say B) is created with a unique ID. This new directory (B) serves as the data point for the snapshot, 'A' still remaining as the current version. If any update (overwrite) is sent to A, in say chunk_001, B is checked to see if it contains this file. If it already contains the file, we simply update the A directory. If it doesn't, the file in A is moved to B, and then the file in A is overwritten with the new updated file. Read requests to current version is read directly from the corresponding chunk. In a read request to snapshot, B is checked to see if it contains that file. If it does the file is returned, or the same file in A is returned. On snapshot request, all the changes in the log will be applied before snapshot proceeds. This is similar to copy on write snapshot.

Problems

- In HDFS, an open operation involves opening a connection to the master for the information of the blocks, as to in which datanode they are present and so. Then the client opens a connection to the datanode and fetches data from the datanode. This process of opening multiple connections take time. In this design there are many files that are opened for a single block storage image. This too many number of connections to open hinders the

performance of Hadoop. Also since the connections are too many they cannot be all cached in the client. Even if the connections are cached they have to be closed and reopened for the changes to reflect in the input streams.

- There is a lot of work that is wasted when a chunk is written back. In worst case scenario of only one block being changed in a file, the entire chunk has to be read and rewritten back. This is a huge wastage of resources and would cause a performance hit. Although a cache is present to accumulate changes before they are written back, there would still be a significant quantity of the chunks which would contain very few block modifications.
- Larger the log file to accumulate changes, lesser would be the performance hit due to above concern. But a map which is a copy of the same file is stored in memory which constrains the number of clients as the log file maximum size is increased. This map cannot be deallocated similar to a cache as per need basis. This is because if for a read occurs for an address in the logfile and it is not present in the map then it would have to read from the logfile. This would require opening and closing the read stream of the logfile every time a read comes. This is prohibitively expensive.

4.4.2 HBSv2

Hbase is a database on top of hadoop. It used HDFS to store the table data. The features provided by Hbase had striking similarity with the requirements of block store. It supports storage upto multi petabyte data which is ideal for the scalability of block store. It can store large rows of size in KBs to MBs. It is row atomic, i.e, modification to a row either succeeds entirely or fails entirely, which can be used to provide block atomicity. The idea was to store entire data in hbase with the block address as key and data as the only column. Read and Write operations on blocks would be equivalent to read and writing the respective rows. Testing the feasibility of this solution showed us limitations of hbase that needed us to change the design.

Problems

- The usecase in which hbase works the best is mass upload of data once, multiple query scenario where huge amounts of data is uploaded at once, and then different queries were performed on this data. An exmaple of this kind of usage is the cluster that Yahoo uses for taking a snapshot of the internet. They take a snapshot of the internet using web crawlers which write huge data at once. Once they are written search engine computations are performed on the data. Block storage on the other hand has a diffrent usage pattern where data is continuously written to and read from. Hbase didnot give us good performance in this scenario.
- Hbase is memory and cpu intensive.

4.5 Final Design - HBSv3

4.5.1 Overview

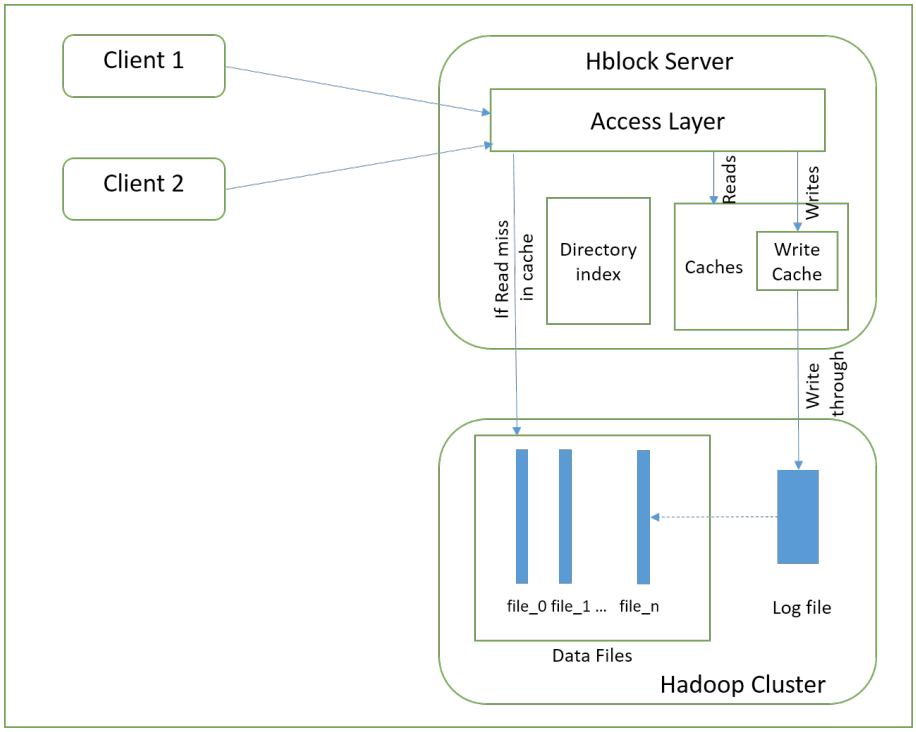
We use Hadoop's HDFS to store these images. Data of an image is stored in a folder with the same name in the user's directory in HDFS. All images are thus directories residing in their user directory. This directory contains multiple files which store the data of that image. We will refer to these files as data files. A write consists of the address – the address to which data is to be written and the data – the data to be written starting from that address. A new write to an image is simply appended to a data file. Any read from an address is served from the latest version of data in these files. Since, the location of each of these blocks is not fixed, we will have to store the location of each block in memory. These are stored as (block address -¿ (file, offset, sequence number)) mappings in memory for each block, which is a key value pair, key being the block address or number and value being the file number and the offset in the file at which the block's data is present and the sequence number represents the version of the data of the block.

Caches are maintained in memory where every read and write request is cached. Write cache is supported by a small logfile where writes are written, to recover in case of a crash/failure. Once the write cache/logfile reaches a certain threshold the changes are flushed to the data files and the write cache is cleared. Read cache caches the read requests to an image and serves them directly from the cache instead of reading the block -¿ file mappings and then fetching the block from the data file.

Term	Meaning
Block	A sequence of bytes with a maximum length. Any data transfer is done in block sized units.
Data Files	The files in which data of the images is present.
Directory Index	Stores (block addr ->(file, offset, sequence number)) mappings for each image in the server's memory.

Table 4.2: Terms used

4.5.2 Architecture



Access Layer

This layer provides the application calls that can be made by the client. All accesses to block storage have to be through this layer. All classes written in this layer should implement the BlockProtocol interface specified above. 'HBS' is the default class which implements this protocol.

Data Files

These files are the storage components of the system. Data files for an image reside in a directory with the image name. These files store the writes to the image. New writes are appended to a data file instead of replacing an old write to the same address in the file. Each data file has a size threshold, and if the size of the current file reaches a limit, a new data file is created and any future writes are appended to the new data file.

Each data file is a sequence of ¡block address, sequence number, block data¿. The sequence number denotes the version of the data for the corresponding block address. New writes to the same address are appended to the data file with an incremented sequence number. Changes are made in the directory index in memory to point to the new file and new offset.

Directory Index

This stores the locations of the blocks' data for each block in each image in the storage system. This is an 'in memory' structure present in the memory of the storage server. When the server is starting up, it checks for any data files that are already present on the system. If there are any data files, it reads these files and creates this index. Because a data file is a sequence of ¡block address, sequence number, block data¿, a mapping ¡block address -¿ (file number, offset, sequence number)¿ is created. In this mapping, the key is the block address and value stored is the file number, the offset in the file for this block address, and its sequence number.

Caching Layer

This layer is used for caching the calls to the network. This consists of two caches- write cache and a read cache. By default we use a LRU memory read cache of size 64MB for each client. The write cache is a small 6.4MB cache used to cache write access and is backed by a logfile. Write cache is a write-through cache. Once the cache/logfile crosses a certain threshold the writes are flushed to the most recent data file and the write-cache and log files are flushed.

4.5.3 Operations

Read

Read request first checks the cache in the Hblock storage server. If it is a cache hit, then the corresponding block is returned directly. Cache hit can be either due to hit in read cache or the write cache. If it is a cache miss, the directory index is accessed to get the location of the block. Location of the block would include the file number and offset of the block in the data file. The block is then fetched by reading the corresponding chunk from the HDFS. Read operation allows the storage server to send a list of blocks apart from the requested block to the client. This allows the storage server to guess the blocks that would be requested next and fetch them or simply pre-fetch the neighbouring blocks to the cache.

Write

Write request is first recorded in the write cache of the Hblock server. It is then appended to a small 'append only' log file. If an entry is present in the read cache for this address, the entry is invalidated and the write operation is then returned as successful. If the write log reaches a certain threshold, the log file is flushed to the current data file. Directory mappings are made to point to the current data file and corresponding offsets. Write cache is flushed only after the directory mappings are made. If the current data file reaches its own threshold a new data file is opened and the write is appended to the new file. Each image

thus keeps growing in size beyond its maximum capacity till the compaction job runs.

Compaction

As each image accumulates writes to itself in data files, the total size of the data files keeps growing. These files contain many writes to a single location which can be discarded as only the last write is required. Compaction operation is run in the background for every image after it reaches a certain threshold like *current_size_on_disk = k * actual_size_of_image* . It merges all the data files except the current data file into one single data file. The size of data file which is the result of the compaction process cannot be more than the *actual_size_of_disk* , since it contains at most one write for a single address. This way the space that an image uses in HDFS is kept at check.

Snapshot

Since all data files of an image reside in a single directory, the snapshot of the image is equivalent to the snapshot of the directory in the HDFS. We use Hadoop snapshot API to snapshot the entire directory in HDFS. The write cache and the log files are cleared and the compaction job i run before a snapshot operation starts. Also, on a write operation original blocks are not copied to the snapshotted image as in a Copy-On-Write (COW) snapshotting but their blocks redirected to point to the new file as in Redirect-On-Write (ROW) snapshotting technology. Recovering from the snapshot would require creating the directory index structure out of the data files which is similar to the crash recovery operation.

4.5.4 Crash recovery

The directory index is stored in the memory of the storage server. If the storage server crashes, the directory index is lost and needs to be reconstructed. This can be done by reading the data files and reconstructing the directory index. Because each data file stores the data in ¡block address, sequence number, data¿ format, we can read the address and construct its entry in the directory index corresponding to the entry in the data file with the largest sequence number.

4.6 Performance

4.6.1 Cluster Specifications

We tested with 4 VMs on baadal as our cluster. The master/namenode has 16GB RAM and the three datanodes have 8GB RAM each. Each of them has a hard disk capacity of 80GB.
All the VMs are backed by storage by a filer. So, the maximum bandwidth

we can get is the filer's bandwidth. We used Bonnie++ to benchmark the raw cluster bandwidth for different operations. The results are tabulated below.

Table 4.3: Raw Cluster Bandwidth

Test	**Speed**
Random Write	108.3 MB/sec
Random Read	75.75 MB/sec
Random Read Write	39.67 MB/sec

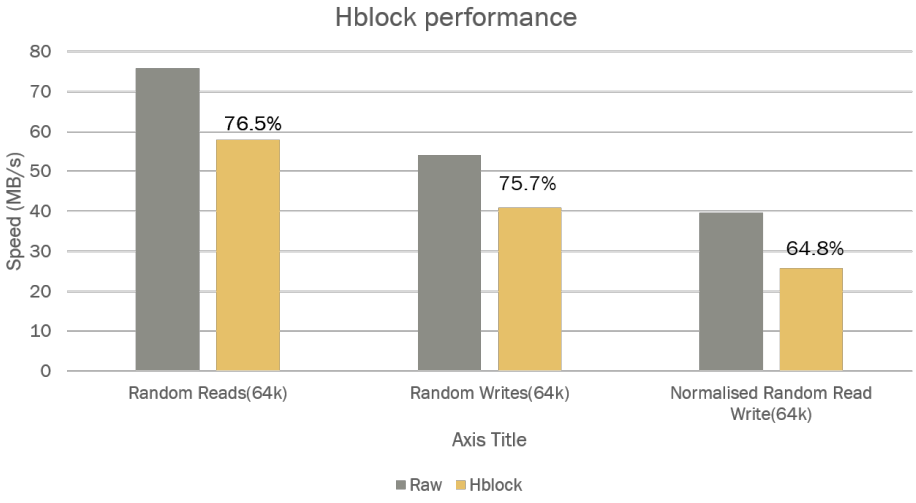
4.6.2 Performance of HBlock Server

The same tests when run on the block storage server gave the following results.

Table 4.4: HBlock Bandwidth

Test	**Speed**
Random Write	41.02 MB/sec
Random Read	57.95 MB/sec
Normalised Random Read Write	25.71 MB/sec

In random writes, the maximum bandwidth we can obtain from a HDFS cluster is $n*throughput\ of\ each\ disk/replication\ factor$, where n is the number of datanodes in the cluster. In the case of our cluster, since all the datanodes are connected to the same filer, the maximum bandwidth we can get is the filer's bandwidth and we used a replication factor of 2, this maximum obtainable random write bandwidth translates to $filer's\ bandwidth/2 = 54.15MB/sec$.



In the above figure, the raw (grey) column represents the raw bandwidth than can be supported by the cluster and the Hblock (yellow) column represents the average bandwidth we obtained by HBlock server. The percentage above the Hblock column shows the percentage of raw bandwidth provided by HBlock.

4.7 Limitations

- It is assumed that the storage space is sufficiently cheap and available. Performance was given importance at the cost of storage space. As a result for every image, actual space of 2x-3x(multiples of image size, depending on the configuration) times the image size is used. Factoring in the minimum replication of 2 of hadoop, 4x-6x size of the required size is being used. It might still be comparable to hardware solutions present in the market since this system uses commodity storage which is much cheaper than the vendor specific storage systems.
- High memory storage servers are required. These storage servers store caches and directory structure of each image in memory. This limits the number of active images that can be served by the storage server. However this limitation can be mitigated by running multiple storage servers on the same cluster. Each storage server acts independently and serves their own set of images. This would make it scalable.

Chapter 5

Future Work and Conclusion

5.1 Future Work

- **Kernel drivers and RPC**

  To mount our block storage system as a block device, kernel module has to be written to implement ioctl calls in terms of BlockProtocol interface. These calls would in turn be made using Remote procedure call to the storage server, which would return back results to the kernel module.

- **Test Support**

  Tests have to be written to validate that the system is working with every incremental change made to the software. These tests are absolutely necessary as kernel sensitive code would be using our APIs.

- **ACL**

  Present implementation of Block Storage and Object Storage does not have any security mechanism in place to control access of users to others data. These have to be implemented for data, file and object sharing.

- **Integration with baadal**

  Kernel drivers and RPC is a prerequisite for this task. After they are complete, this software can be integrated with baadal for this software needs.

- **Improve**

  Current implementation is a proof of concept implementation of the system. There are limitations to both block and file stores that can be improved upon. There are many features that are to be added like QOS, pricing etc. for this product to be a fully fldeged product. Scope of improvement is immense.

5.2 Conclusion

The ultimate goal of this project or 'the group of projects of which this is a part of' is to successfully provide our academic cloud(Baadal) with a software defined storage solution for all its storage needs. It should be able to provision disk images for VMs from block storage, provide object storage for required users, provide file storage for performing Map reduce and other distributed computing operations. In this project we have designed and implemented Object Storage improving on the previous work. We have designed and implemented Block Store thus achieving a significant part of higher goal. There is a lot more work left to complete it.

With our utmost thanks to Prof. Suresh C Gupta

THE END

Bibliography

[1] Konstantin Shvachko et al. *The hadoop distributed file system.* Incline Village, NV

[2] Sage A. Weil et al. *Ceph: a scalable, high-performance distributed file system* OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation

[3] Fay Chang et al. *Bigtable: A Distributed Storage System for Structured Data* adf

[4] Apache Hbase reference guide *http://hbase.apache.org/book.html*

[5] Apache Hbase Developer API *https://hbase.apache.org/devapidocs/index.html*

[6] Hadoop Developer API reference *http://hadoop.apache.org/docs/current/api/index.html*

[7] Ceph Architecture guide *http://docs.ceph.com/docs/hammer/architecture/*