Network Coding Video File System

Documentation v0.2

October 29, 2012

1 Introduction

Cloud Storage has been playing an increasingly significant role in data storage for enterprises and individuals lately. For instance, *Amazon S3* and *Windows Azure* are two successful clould storage systems in real-life business. To achieve high data availability, a *distributed storage system* is the underlying system architecture adopted by clould storage systems, as it provides a reliable platform for storing a massive amount of data over a set of distributed storage nodes over a network. Current distributed storage systems mainly apply *data replication* to maintain data reliability, such as *HDFS* and *GFS*. Each piece of data in the system is replicated several times and each replica is distributed to different storage nodes. The *replication factor* determines the number of times each piece of data being replicated and therefore, data availability is positively related with the *replication factor* of the entire system.

However, one major drawback of data replication is that it requires relatively large amount of storage resources in order to achieve high availability, compared with employing *erasure coding* on data. Moreover, given the similar network bandwith and storage resources, systems employing erasure coding has better availability and durability than systems employing data replication [5]. Un-

der such circumstance, cloud storage systems like *Windows Azure* and *Hadoop Distributed File System* are switching their storage scheme from replication to erasure coding [1, 3].

Data repliacation may not fully utilize the parallelism benefit of distributed storage system if the size of data to be replicated is large. Hence, previous researches [2, 4] have proposed *object-based* storage systems to handle this issue. In general, an object-based storage system splits each file to be stored into equal size pieces called *objects*, and assigns different storage nodes to perform replication or coding on the objects. It can be found that object-based storage systems benefit more than file-based storage systems in a distributed environment due to the parallel processing.

The above-mentioned issues motivate us to design a brand new distributed storage system which could combine the advantages of both erasure coding and object-based stroage techniques to achieve high availability and performance. Hence, in this paper, we present our design and implementation of a network-coding-based distributed video file system, called NCVFS. The main features of NCVFS are summarized as follows:

- NCVFS is an object-based distributed file system targeting large video files. It supports general read and write operations in a distributed storage setting and optimizes the read operation based on the write-once-read-many feature of video files.

- NCVFS is not limited by erasure coding storage schemes, but can be extended to other coding schemes like network coding. Moreover, since NCVFS adopts object-based storage, it enables multiple coding schemes performed on a single file without changing the entire system logic.

- NCVFS excludes metadata management from the critical path for the read/write process to eliminate the bottleneck of metadata operations

and utilizes a centralized monitor to balance the workload of each storage node.

- NCVFS implements a distributed light-weight recovery scheme to handle storage node failures. In brief, once a recovery process is triggered for a particular storage node, NCVFS delays the process for a few minutes to confirm the storage node is dead permanently. Such delay saves the recovery operations for temporary failures like network partition while data on the failed node can still be accessed via degraded read during the delay.

2 Revision History

Version 0.2 (11 September 2012)

3 System Overview

3.1 Architecture

NCVFS is an object-based distributed file system and is composed of four components: metadata storage, object storage, clients and monitors. A single metadata server (`MDS`) is designed for NCVFS for storing all file metadata as well as managing operations on file metadata. On top of that, NCVFS contains multiple object stroage devices (`OSDs`) and each `OSD` stores file data objects with network coding. One key point of this architecture is that the `MDS` is not on the critical path when clients want to read or write files,as the `MDS` is contacted by clients first before clients operate with `OSDs` . Such design eliminates the bottleneck on the metadata side and enables parallel jobs on the object storage side. Finally, current architecture of NCVFS consists of a centralized monitor

(MONITOR) implementing OSD failure recovery functionality and load balancing among OSDs . The detail of each component is presented in Section 5.

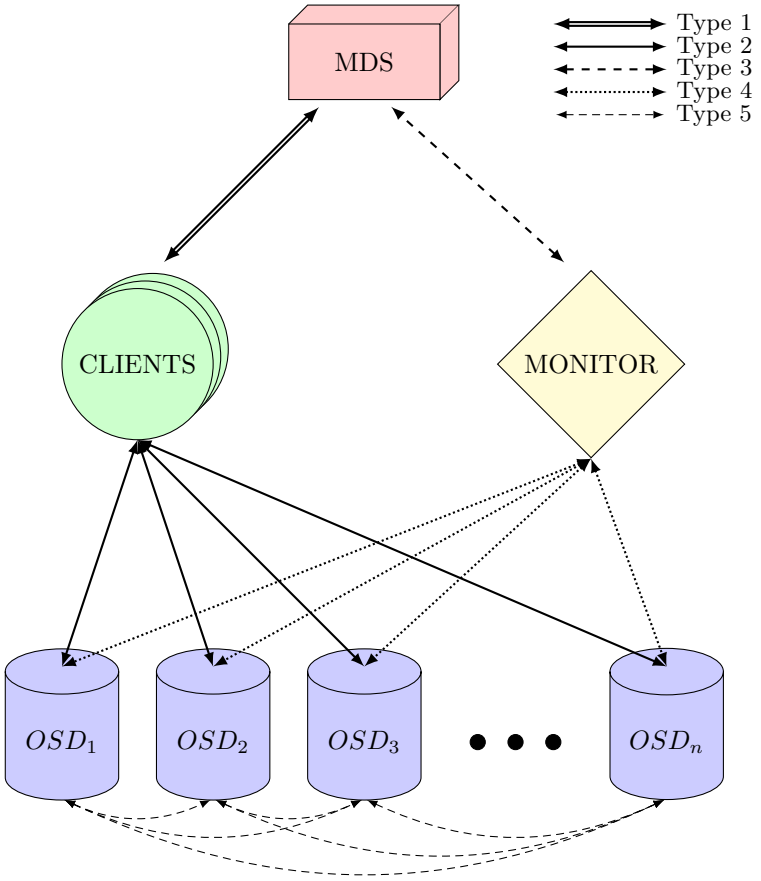

Figure 1: Architecture of NCVFS

For a particular file to be stored on NCVFS, its metadata, including file name, directory structure, access permission and file data locations, are stored in the MDS . Each file is split into several objects and the object size is flexible to configure. Each object is then assigned to one particular OSD to perform network coding and distribute the coded segments to other OSDs for storage. The detail workflow is described in Section 4. The entire picture of the archi-

tecture is illustrated in Figure 1. There are five types of communication within the entire system:

Type 1: Communication between `CLIENTs` and `MDS` initiates file read/write request and manages file metadata

Type 2: Communication between `CLIENTs` and `OSDs` transfers real file data during read/write processes

Type 3: Communication between `MDS` and `MONITOR` assists `MDS` to balance the workload of `OSDs`

Type 4: Communication between `OSDs` and `MONITOR` acts as heart-beat contact for detecting failures

Type 5: Communication between `OSDs` transfers coded data during read/write processes

3.2 System Requirement

3.2.1 Environment

NCVFS is tested on Ubuntu 12.04 (32-bit and 64-bit) and Arch Linux (32-bit). To compile NCVFS, GCC 4.6 or above (which supports the latest C++11 standard) is recommended.

3.2.2 Third-party Libraries

The third-party libraries that are used in NCVFS are listed below.

- **Google Protocol Buffers**

  Serializes control messages among different components

- **MongoDB**

  Supports metadata storage in `MDS`

- **Apache Runtime**

  Provides a memory pool to optimize memory allocation in different components

- **FUSE**

  Allows implementation of a file system in user space

- **OpenSSL**

  Supports crytopgraphic hash computations

- **Threadpool**

  Provides an extension to the Boost library for thread scheduling and management

- **Jerasure**

  Provides functions for computing various kinds of erasure coding scheme

- **Boost C++ Libraries**

  Prerequisite of Threadpool and MongoDB

4 Client Operation

We introduce the overall operation of NCVFS's components and their interations by describing client operations. An NCVFS client runs on each host and applications can access files through a mounted file system via `FUSE` .

4.1 File Upload

When a process initiates an upload process, the client sends a request to the `MDS` containing the filename, the number of objects to be uploaded and the desired coding scheme. In our setting, each object in NCVFS is 10MB. However, depending on the nature of file, the client could freely adjust the object size.

The MDS records the metadata by creating an entry in the database and sends a request to the MONITOR asking for a list of *primary* OSDs to handle this upload. The *selection module* in MONITOR tries to balance the workload among OSDs by selecting the OSDs that are relatively idle according to their remaining storage capacity and CPU loading to be the *primary* OSDs for this upload. After selection, the MDS replies the client with the list of *primary* OSDs and the respective locations. The client then establishes a TCP connection to each of the *primary* OSDs and transfers the object in chunks.

When a *primary* OSD receives all the chunks of an object from the client, it executes the coding module to encode the object into a list of segments with the coding scheme specified by the client. When this process is finished, it requests the MONITOR for a list of *secondary* OSDs . The number of *secondary* OSDs depends on the coding scheme of the object. For example, a replication coding scheme of n = 3 requires two *secondary* OSDs to hold the encoded segments. After the reply from MONITOR is received, the *primary* OSDs transfer the encoded segments to the assigned *secondary* OSDs . An acknowledgement will be sent to both the MDS and the client when the whole process is finished. At this point, the MDS records the ID of the OSDs that have stored a part of this object.

Some distributed storage systems such as the Windows Azure Storage perform erasure coding offline, objects are initially replicated when they are uploaded. Erasure coding are only performed on objects that are *sealed*. Instead of performing offline erasure coding, NCVFS performs coding on the write path. Compared with offline erasure coindg, the amount of network traffic among OSDs is smaller since we eliminate the need to transfer replicated copies at the beginning.

4.2 File Download

When a process initiates a download process, the client first contacts the `MDS` to retrieve the list of the *primary* `OSDs` . For each object, the client sends a download request containing the coding scheme to the *primary* `OSD` . When a download request is received, the *primary* `OSD` contacts the `MDS` to retrieve the list of *secondary* `OSDs` that are storing a part of the object. Once all segments for an object are received, the *primary* `OSD` reconstructs the object from the segments by applying the decode function of the same coding scheme that is used in the upload process. After decoding, the *primary* `OSD` sends the decoded object to the client.

Every `OSD` maintains a fixed-size LRU cache in the disk to store decoded objects. When an object is requested, the specified *primary* `OSD` first looks up the object in its cache. If there exists a cached copy on disk, the *primary* `OSD` simply returns the cached copy to the client. In this case, there will be a drastic improvement in throughput as there is not need to transfer segments from *secondary* `OSDs` and perform decoding.

5 Design and Implementation

NCVFS has four main components: the `MDS` , which stores metadata for objects and segments in a MongoDB; a cluster of `OSDs` , which collectively store all encoded segments and cache objects; the `MONITOR` . which keeps statistics of components and selects `OSDs` during upload; the `CLIENT` , which is implmented in `FUSE` and exposes a near-POSIX interface to a process.

5.1 Metadata Server (MDS)

NCVFS maintains a centralized storage of metadata in the MDS . The MDS in NCVFS is similar to the Namenode in the Google File System except the fact that we utilize the MongoDB as the backend data storage. The MDS plays an active role in most NCVFS operations.

5.1.1 Metadata Module

The *Metadata Module* is responsible for handling metadata operations. Metadata of a file contains the name, size, owner information and the list of its composing objects. Metadata of an object includes the coding scheme and the location of encoded segments. During an NCVFS operation, other components may request the MDS for a metadata update. The *Metadata Module* is invoked to interact the MongoDB. Since operations are atomic, the MDS guarantees that the metadata is written to the database before acknowledging the change to the other end.

Although the current single MDS design may serve as a single-point-of-failure in NCVFS, we can tackle this problem by making use of MongoDB's replication feature. MongoDB supports asynchronous replication across multiple servers for failover. In this setting, there is a primary server to serve write requests to ensure strong consistency. At the same time, data is replicated to secondary servers. In case of a failure in the primary server, a secondary server can be brought online to handle requests.

5.2 Object Storage Device (OSD)

NCVFS maintains a cluster of OSDs to store objects while exposing just a single file system interface to the processes. As NCVFS needs to handle simultaneous requests from a large number of clients, a central storage server will likely be

a bottleneck. By maintaining a cluster of OSDs , NCVFS tackles this problem in two ways. First, the OSDs can lessen the burden of the MDS and MONITOR as OSDs themselves are able to handle data replication, coding, data verification and failure recovery. Second, we ensure that the workload is balanced across the cluster by selecting different *primary* OSDs for each object according to their current CPU loading and remaining disk capacity.

5.2.1 Coding Module

The *Coding Module* provides two important functions: to encode an object into segments, and to reconstruct objects by decoding segments. Currently, the *Coding Module* supports RAID-0, RAID-1 and Reed-solomon coding schemes.

When a CLIENT finishes uploading an object to the Primary OSD , the *Coding Module* is executed to perform encoding according to the coding scheme specified. The client can adjust the parameters of a coding scheme by supplying an extra argument to the encode function. For each object, both the coding type and the chosen parameters are saved in the MDS since this information is necessary for an OSD to decode in the future.

A key feature of NCVFS is that we allow different objects in a file to employ different coding schemes. This allows the clients to select the coding schemes strategically according to the target workload. For example, if a client predicts that the first part of a file is more frequently read, a more decode-friendly coding scheme such as replication should be chosen for those objects to optimize read performance. In contrast, for some parts that are rarely read, more computation intensive coding schemes such as Reed-solomon could be used to optimize the storage space used.

During decode, the *primary* OSD first contacts the MONITOR to obtain a bitmap representing the availability of OSDs which store the segments correspond to the target object. A "0" in the bitmap indicates that the OSD contain-

ing that particular segment is currently unreachable. As this bitmap is passed to the *Coding Module*, it is able to determine whether there are enough available segments to perform decode or not. If there are enough segments, the *Coding Module* reconstructs the object by applying the decode function of the coding scheme, making use of the code segments when necessary. Otherwise, the *Coding Module* reports an error to the `OSD` . The `OSD` may then inform the `CLIENT` that the object is no longer available.

5.2.2 Storage Module

The *Storage Module* is the communication layer between the `OSD` and the underlying file system. In the current implementation, each `OSD` is attached to an *ext3* partition. In practice, any file system supporting the POSIX interface can be used in the underlying storage layer.

Both cached objects and segments are represented as physical files in the storage layer. Cached objects are named by the `objectId` which is a 64-bit integer while segments are named by the `objectId` appended by its 32-bit integer `segmentId`. Since the `objectId` acts as the unique identifier of an object in the system, the `OSD` can locate a deterministic file path of any cached objects or segments, eliminating the needs of saving the actual file path of objects and segments in either the `OSD` or the `MDS` .

5.3 Cluster Monitor (`MONITOR`)

NCVFS is equipped with a centralized `MONITOR` for multiple purposes as listed below.

- Coordinating with `MDS` to select optimal destinations for data placement
- Monitoring health status of each `OSD` for load balancing
- Trigger recovery process for `OSD` failure

`Monitor` behaves like a tracker in a peer-to-peer network. Therefore `MONITOR` is supposed to be an always-on process and `MDS` together with each `OSD` holds the connection information of `MONITOR` . The `MONITOR` consists of three major modules to serve the purpose above, i.e. *Statistics Module*, *Selection Module* and *Recovery Module*.

5.3.1 Statistics Module

The *Statistics Module* maintains the health status of each `OSD` and assists `OSDs` to establish connection with each other. Two typical workflows of this module are described below.

- When an `OSD` starts up, it registers to `MONITOR` with its connection information as well as its health status. `MONITOR` records the new `OSD` in its list and marks it `online`. `MONITOR` then broadcasts the new `OSD` to all other online `OSDs` . Also `MONITOR` tells the new `OSD` with all other `OSDs` information for them to connect with each other.

- `MONITOR` periodically requests `OSDs` in its list to update their health status. The health status includes current average loading and the disk capacity. `MONITOR` marks an `OSD` `offline` if it receives no reply from that `OSD` . The period can be configured manually for different situation and the current period time is set to be 10 seconds.

5.3.2 Selection Module

The *Selection Module* coordinates with the `MDS` to select optimal primary `OSDs` to perform coding and also assists primary `OSDs` to select optimal secondary `OSDs` to place coded data segments. The selection criterion is based on the headlth status of each online `OSD` which has been recorded in the *Statistics Module*. The design of this module is to balance the workload for each `OSD` .

Meanwhile, it is also convenient to design different selection criteria based on the statistics in order to achieve maximum load-balancing.

5.3.3 Recovery Module

The *Recovery Module* monitors the health status update of each `OSD` and triggers recovery process if an `OSD` goes offline. However, an `OSD` may go offline due to some usual reason like network partitioning and go online again a couple of seconds later. Therefore we delay the recovery process for a couple of minutes to avoid such cases in order to minimize the workload of the entire system. The delay time threshold for triggering the recovery process can be manually configured.

5.4 Client (`CLIENT`)

The `CLIENT` in NCVFS can perform the operations described in Section 4. The `CLIENT` design can be separated into two parts. First, a set of Client API functions are implemented to interact with NCVFS components. The API is used by the `FUSE` client and can be used by other applications to by-pass the POSIX interface and interact with NCVFS components directly. Second, the `FUSE` client utilizes the Client API and exposes a near-POSIX interface to the processes. With the `FUSE` client, any process can use NCVFS like an ordinary mounted directory.

5.4.1 Client API

The Client API provides the functionality for a client to access data in the NCVFS. For example, there are functions for clients to upload and download files. In addition, clients can also choose to transfer only a particular object. Besides our `FUSE` implementation of `CLIENT` , any client programs can be devel-

oped for NCVFS using the provided API.

5.4.2 FUSE

The `FUSE` client implements functions using the Client API to support POSIX file operations such as `open`, `read`, `write` and `close`. There are mainly two adaptations that we have to make when we design the `FUSE` client. First, POSIX read operations can access at any offset an arbitrary number of bytes. However, in our design, the smallest atomic unit is an object. Therefore, we have to design the `FUSE` client such that during a read operation, it will always request the correct object from the `OSD` even when the number of bytes it needs is smaller than the object size. Second, many POSIX metadata operations such as `chmod` and `chown` are currently not implemented in NCVFS. In order to provide a fully-functional POSIX interface, we use a *shadow directory* approach. Each `CLIENT` creates and mount a *shadow directory* in the `MDS`. When a file is uploaded, an empty file of the same name is created in the *shadow directory*. By channeling subsequent POSIX metadata operations to the shadow copy of the file, we are able to support most POSIX functions without actually implementing all of them in the `MDS`.

5.5 Shared Modules

5.5.1 Communication Module

The Communication Module facilitates interaction among all the components. The Google Protocol Buffers provides very effective serialization and deserialization methods and is used to support communication of control messages among components.

In our design, for each control message, a *prepare* function and a *handle* function are implemented. Each message consists of a message header. To send

a control message, the communication module of the corresponding component first prepares a message header which stores the massage type, the protocol message size and the payload size. The communication module obtains an object from the *MessageFactory* by specifying the type of message. By executing the *prepare* function of the object, the information that the component passed into the object is serialized into a binary byte stream. The message header, serialized protocol message and the payload are sent via TCP socket to the target component. The receiver of this message can then retrieve the information stored by parsing the binary byte stream. The handlers for the obtained information are then executed by calling the *handle* function defined in the corresponding message class.

5.5.2 Memory Pool

In NCVFS, the *Communication Module* requires frequent memory allocation and deallocation. This introduces two performance issues in our system. First, frequent calls to malloc and free can introduce significant performance overhead. In our design, we tackle this problem with the help of a memory pool. The memory pool obtains a large chunk of memory from the system at a time. When a component calls malloc, the memory is obtained from the pool. When a piece of memory is freed, the memory is returned to the pool and can be allocated to subsequent malloc calls. Second, although each control message is small, the size is variable. Even with the help of a memory pool, memory fragmentation may occur when these small pieces of memory are allocated and deallocated frequently. To tackle this problem, we add padding to each control message to ensure that the same amount of memory is allocated for each message created.

In NCVFS, we use Apache APR Memory Pools to handle the memory allocations. The APR Pools are a fundamental building block of Apache, and are the basis for all resource management. They serve to allocate memory, either

directly (in a malloc-like manner) or indirectly (e.g. in string manipulation), and, more crucially, ensure the memory is freed at the appropriate time. But they are extended much further, to ensure that other resources such as files or mutexes can be allocated and will always be properly cleaned up. They can even deal with resources managed opaquely by third-party libraries.

5.5.3 LRU Cache

Both the `CLIENT` and `OSD` need a cache for performance. A generic cache is implemened to support a fixed-size cache with least-recently-used deletion policy. Our implementation makes use of a list together with C++11 `std::unordered_map` to provide efficient key valued accesses.

In the `FUSE` implementation, a cache is used to store file handles since looking up the full path in every `FUSE` operation is very expensive. In addition, the Primary `OSDs` also use a cache to remember the location of the Secondary `OSDs` for recently accessed objects to exploit temporal locality of the access pattern. When an object is requested, the Primary `OSD` first checks the cache for the list of Secondary `OSDs` storing parts of the object. If there is a cache hit, the round trip time to the `MDS` can be saved.

6 Evaluation

6.1 Iozone Benchmark

6.1.1 Sequential Read

6.1.2 Sequential Write

6.1.3 Random Read

6.1.4 Random Write

Terminology

References

[1] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[2] Michael Mesnier, Gregory R. Ganger, Erik Riedel, Carnegie Mellon, and Carnegie Mellon. Object-based storage, 2003.

[3] Maheswaran Sathiamoorthy. Hdfs-raid. `http://wiki.apache.org/hadoop/HDFS-RAID`, 2011.

[4] Feng Wang, Scott A. Brandt, Ethan L. Miller, and Darrell D. E. Long. Obfs: A file system for object-based storage devices. In *IN PROCEEDINGS OF THE 21ST IEEE / 12TH NASA GODDARD CONFERENCE ON MASS STORAGE SYSTEMS AND TECHNOLOGIES, COLLEGE PARK, MD*, pages 283–300, 2004.

[5] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 328–338, London, UK, UK, 2002. Springer-Verlag.