Assignment 1

BUILDING A DISTRIBUTED, REPLICATED, AND FAULT TOLERANT FILE SYSTEM: CONTRASTING REPLICATION AND ERASURE CODING

VERSION 1

DUE DATE: Wednesday September 25th, 2019 @ 5:00 pm

OBJECTIVE

The objective of this assignment is to build a distributed, failure-resilient file system. The fault tolerance for files is achieved using two techniques: replication and erasure coding. As part of this assignment, you should identify the trade-off space involving these techniques. For example, your analysis could contrast storage efficiency, CPU overheads, and memory utilization. This assignment has several sub-items associated with it.

There are 3 programs that you need to develop.

1. Chunk Server responsible for managing file chunks. There will be one instance of the chunk server running on each machine.
2. A controller node for managing information about chunk servers and chunks within the system. There will be only 1 instance of the controller node.
3. Client which is responsible for storing, retrieving, and updating files in the system. The client is responsible for splitting a file into chunks and assembling the file back using chunks during retrieval.

All communications in this assignment are based on **TCP**. The assignment must be implemented in **Java** and the external jar files that you can use are listed towards the end of the assignment. You must develop all functionality yourself. This assignment may be modified to clarify any questions (and the version number incremented), but the crux of the assignment and the distribution of points will not change.

Grading: This assignment will account for **20 points** towards your cumulative course grade. There are several components to this assignment, and the points-breakdown is listed in the remainder of the text. This assignment is to be done individually. The scoring process will involve a one-to-one interview session of approximately 30 minutes where you will demonstrate all the required functionality based on the inputs that will be provided to you. The slots for these interview sessions will be posted a few days prior to the submission deadline.

1 Fault Tolerant File System Design

In our discussions, we first start with fault tolerance using replication and then describe how the fault tolerance functionality achieved using replication can be achieved using erasure coding.

1.1 Fault Tolerance Using Replication

In this file system, portions (or ***chunks***) of a file are dispersed over the set of available machines. There are multiple chunk servers in the system: on each machine there can be at most one chunk server that is responsible for managing chunks belonging to different files. A chunk server stores these chunks on its local disk (in most cases, this will be /tmp).

Every file that will be stored in this file system will be split into 64KB chunks. These chunks need to be distributed on a set of available chunk servers. Each 64KB chunk keeps track of its own integrity, by maintaining checksums for 8KB slices of the chunk. The message digest algorithm to be used for computing this checksum is SHA-1: this returns a 160-bit digest for a set of bytes. Individual chunks will be stored as regular files on the host file system.

File writes/reads will be done via the chunk servers that hold portions of the file. The chunk server adds integrity information to individual chunks before writing them to disk. Reads done by the chunk server will check for integrity of the chunk slices and will send only the content to the client (the integrity information is not sent).

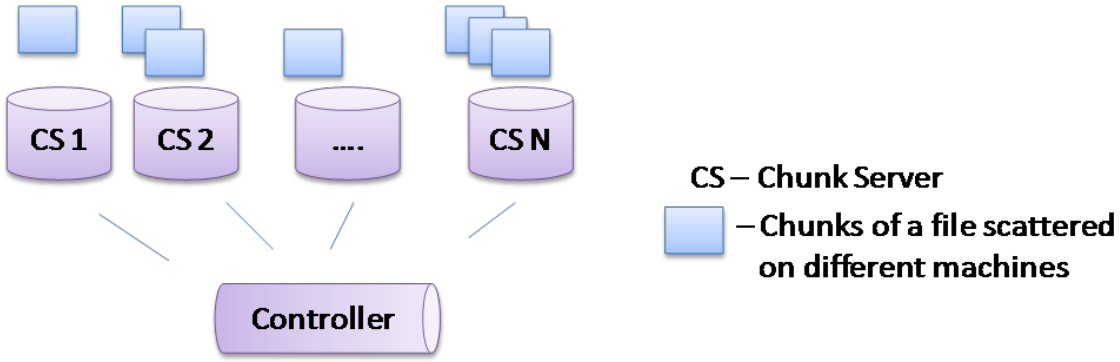

Figure 1: A file will be split into chunks and dispersed on multiple machines

Each chunk being stored to a file needs to have metadata associated with it. If the file name is `/user/bob/experiment/SimFile.data`, chunk 2 of this file will be stored by a chunk server as `/tmp/user/bob/experiment/SimFile.data_chunk2`. This is an example of the metadata being encoded in the name of the file. There will be other metadata associated with the chunk: this additional information should not be encoded in the filename; this includes –

- Versioning Information: Multiple writes to the chunk will increment the version number associated with the chunk.
- Sequencing Information: There will be a sequence number associated with each chunk.
- Timestamp: The time that it was last updated.

Chunk Server and the Controller Node

Each chunk server will maintain a list of the files that it manages. For each file, the chunk server will maintain information about the chunks that it holds.

There will be one controller node in the system. This node is responsible for tracking information about the chunks held by various chunk servers in the system. It achieves this via heartbeats that are periodically exchanged between the controller and chunk servers. The controller is also responsible for tracking *live* chunk servers in the system. The controller does not store anything on disk, all information about the chunk servers and the chunks that they hold are maintained in memory.

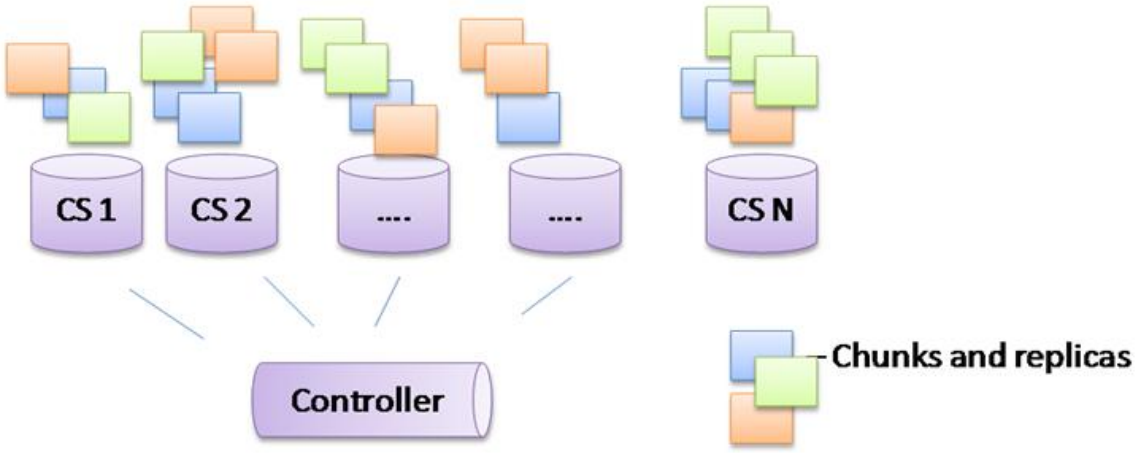

Figure 2: Distribution of chunks of a file and their corresponding replicas

Heartbeats

The Controller Node will run on a preset host/port. A chunk server will regularly send heartbeats to the controller node. These heartbeats will be split into two

1. A major heartbeat every 5 minutes
2. A Minor heartbeat every 30 seconds

At the 5 minute mark ONLY the major heartbeat should be sent out.

The major heartbeat will include metadata information about ALL the chunks maintained at the chunk server. The minor heartbeat will include information about any newly added chunks. Additionally, when a chunk server detects file corruption, it will report this to the Controller Node.

All heartbeats will include information about the total number of chunks and free-space available at the chunk server. Free space information should be one of the metrics used for distribution of chunks on the set of available commodity machines.

The Controller will also send heartbeats to the chunk servers to detect failures.

Replication of files

Each file should have a replication level of 3; this means that every chunk within the file should be replicated at least 3 times. When a client contacts the Controller node to write a file, the Controller will return a list of 3 chunk servers to which a chunk (64KB) can be written. The client then contacts these chunk servers to store the file. Rather than write to each chunk server directly, if there are 3 chunk

servers **A**, **B** and **C** that were returned by the controller, the client will only write to chunk server **A**, which is responsible for forwarding the chunk to **B**, which in turn is responsible for forwarding it **C**. Propagation chunks in this fashion has the advantage of utilizing the bandwidths more efficiently. After the first 64KB chunk of a file has been written, the client (this should be managed transparently by your API) contacts the Controller to write the next chunk and repeat the process. *A given chunk server cannot hold more than one replica of a given chunk.*

Chunk data will be sent to the chunk servers and not the controller. The controller is only responsible for pointing the client to the chunk servers: chunk data *should not* flow through the controller.

Disperse a file on a set of available chunks servers (2 points)
You will take a file and ensure the storage of chunks of this file on different chunk servers. Each chunk of the file should be replicated 3 times. This chunk should be available on the local disk (/tmp) of the chunk server.

Deductions

1. If you use the controller to forward chunk data to the chunk servers **(-2 points)**
2. If more than 1 replica of a chunk is stored at the same chunk server **(-2 points)**

Reading a previously stored file (2 points)
During the testing process, you will have to read the file that was previously scattered over a set of chunk servers. For reading each 64 KB chunk, the client will contact the Controller and retrieve information about the chunk server that holds the chunk. Assuming there were no failures, the file read should match the file that was dispersed.

Deductions

1. If you use the controller to forward chunk data from the chunk servers **(-2 points)**
2. If more than 1 replica of a chunk is accessed at the same time. A given read should result in only 1 copy of a chunk being accessed. **(-2 points)**

Tampering with chunks (4 points)
Next, we will go to an individual chunk file managed by your File System and tamper this by modifying the content of the file. This may be deleting/adding a line or a word to the file: this is done outside the purview of your chunk server. This should cause the file read to report a data corruption, and the specific chunk (and slice within it) that was corrupted.

Deductions

1. If you use the controller to detect corruptions of a chunk replica (2 points)

Error Correction (4 points)
The contents of one of your chunks will be tampered with. A subsequent read of the file should detect this corruption and initiate a fix of this chunk slice.

If it is detected that a slice of a chunk is corrupted, contact other valid replicas of this chunk and perform error correction for the chunk slice. Error detections will be performed outside the heartbeat control message scheme. The control flow is through the Controller, but the data flow is between the chunk servers.

Coping with failures of chunk servers (3 points)
We will terminate one/more of the chunk servers. In response to detection of failures of the chunk servers, the Controller should contact chunk servers that hold legitimate copies of the affected chunks and have them send these chunks to designated chunk servers. Note: The control flow is through the Controller, but the data flow is between the chunk servers.
The metadata maintained at the Controller is updated to reflect this. How are reads handled during this failure?

1.2 Fault Tolerance Using Erasure Coding (5 points)

In the previous subsection, fault tolerance was achieved by replicating chunks. The storage requirements in a replication-based setting increase proportional to the number of replicas. Erasure coding offers an alternative to achieve the same degree of redundancy without the corresponding increase in storage costs.

In your scheme with erasure coding, you will take individual chunks, break it into k fragments, expand and encode with redundant pieces of information, and store across different sets of locations. Specifically, your chunks will be broken up into **k** fragments, erasure coded and expanded into **n** fragments. These n fragments are then dispersed over the available servers. Note that **n** must be greater than **k**; furthermore, **m=n-k** is the *degree of redundancy* since any of the **k** fragments can be used to reconstitute the chunk. For the purposes of this assignment, we will work with *k=6* and *m=3*.

Similar to the GPS example that we looked at in class, one way to look at erasure coding is from the perspective of linear algebra. You have **k** variables and **k+m** equations. We will be using **Reed-Solomon** as the erasure coding algorithm. The terminology typically used in erasure coding settings include the following: (1) The first **k** fragments are often referred to as the *primary or data shards*, and (2) the next **m** fragments are referred to as the *parity shards*.

The Reed-Solomon encoding/decoding library as well as the following code snippets are adopted from the open source code implementation available in https://github.com/Backblaze/JavaReedSolomon. Following code snippet demonstrates how to use the provided library for encoding a given payload using Reed-Solomon scheme. Code used for some of the data manipulation using Java is omitted for brevity and to focus more on how to use the encoding and decoding APIs. Please follow the comments closely and implement the necessary sections. Also this code snippet assumes the number of data shards (**k**) is 4 and the number of parity shards **(m)** is 2.

```
    public static final int DATA_SHARDS = 4;
    public static final int PARITY_SHARDS = 2;
    public static final int TOTAL_SHARDS = 6;

    public static final int BYTES_IN_INT = 4;


    // file size
    int fileSize = (int) inputFile.length();

    // total size of the stored data = length of the payload paylod size
    int storedSize = fileSize + BYTES_IN_INT;

    // size of a shard. Make sure all the shards are of the same size.
    // In order to do this, you can padd 0s at the end.
    // This particular code works for 4 data shards.
    // Based on the numer of shards, use a appropriate way to
    // decide on shard size.
    int shardSize = (storedSize + DATA_SHARDS - 1) / DATA_SHARDS;

    // Create a buffer holding the file size, followed by the contents of
the file
    // (and padding if required)
    int bufferSize = shardSize * DATA_SHARDS;
    byte [] allBytes = new byte[bufferSize];

    /* You should implement the code for copying the file size, payload and
padding into the byte array in here. */

    // Make the buffers to hold the shards.
    byte [] [] shards = new byte [TOTAL_SHARDS] [shardSize];

    // Fill in the data shards
    for (int i = 0; i < DATA_SHARDS; i++) {
        System.arraycopy(allBytes, i * shardSize, shards[i], 0, shardSize);
    }

    // Use Reed-Solomon to calculate the parity. Parity codes
    // will be stored in the last two positions in 'shards' 2-D array.
    ReedSolomon reedSolomon = new ReedSolomon(DATA_SHARDS, PARITY_SHARDS);
    reedSolomon.encodeParity(shards, 0, shardSize);

    // finally store the contents of the 'shards' 2-D array
```

The corresponding code snippet for decoding and recovering the original file is shown below.

```
public static final int DATA_SHARDS = 4;
public static final int PARITY_SHARDS = 2;
public static final int TOTAL_SHARDS = 6;

public static final int BYTES_IN_INT = 4;

// Read in any of the shards that are present.
// (There should be checking here to make sure the input
// shards are the same size, but there isn't.)
byte [] [] shards = new byte [TOTAL_SHARDS] [];
boolean [] shardPresent = new boolean [TOTAL_SHARDS];
int shardSize = 0;
int shardCount = 0;

// now read the shards from the persistance store
for (int i = 0; i < TOTAL_SHARDS; i++) {
   // Check if the shard is available.
   // If avaialbe, read its content into shards[i]
   // set shardPresent[i] = true and increase the shardCount by 1.
}

// We need at least DATA_SHARDS to be able to reconstruct the file.
if (shardCount < DATA_SHARDS) {
   return;
}

// Make empty buffers for the missing shards.
for (int i = 0; i < TOTAL_SHARDS; i++) {
   if (!shardPresent[i]) {
      shards[i] = new byte [shardSize];
    }
}

// Use Reed-Solomon to fill in the missing shards
ReedSolomon reedSolomon = new ReedSolomon(DATA_SHARDS, PARITY_SHARDS);
reedSolomon.decodeMissing(shards, shardPresent, 0, shardSize);
```

In your support for fault tolerance using erasure coding you are allowed to develop your own metadata schemes. The points distribution for the 5 points for this component are as follows:

1. Successful retrieval and assembly of erasure coded fragments into individual chunks and reconstruction of the entire file. **(4 points)**
2. Report contrasting erasure coding and replication based schemes. **(1 point)**

Third-party libraries and restrictions:

You are allowed to use a 3rd party library for the SHA1 hash function and the Reed-Solomon Codes. The jar file for the Reed-Solomon codes will be posted on the course website. You are not allowed to download *any* other code from *anywhere* on the Internet. You are also not allowed to use RPC or distributed object frameworks to develop this functionality (there is a **15 point deduction** for this). You should not build GUIs for this application; in the context of this assignment, GUI-building is an auxiliary path (there is a **15 point deduction** for building a GUI). You can discuss the project with your peers at the architectural level, but the project implementation is an individual effort.

Testing Scenario

You will be asked to launch between 10-20 processes possibly on different machines. The port number on which your chunk server runs should be configurable. There will be only 1 chunk server per machine.

Submission deadline:

Please submit the source codes for your project by the 5:00 pm on the due date. This should be mailed to cs555@cs.colostate.edu. There is a **2 point deduction** for mailing it directly to the Professor or GTA's personal e-mail account. We will rely on the honor system: please do not make any modifications to the codebase after the submission deadline has elapsed.

Change History

Version	Date	Change
1.0	8/28/2019	First public release of the assignment