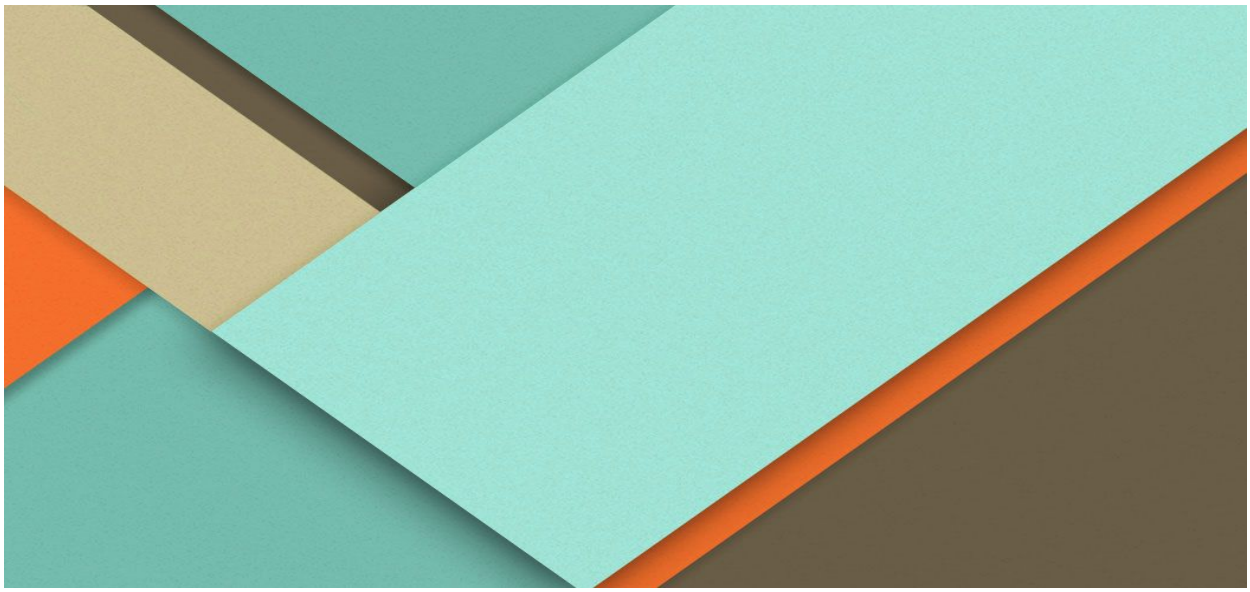

Cloud Computing

Disk Virtualization

16.08.2017

Submitted by: [GROUP 15]

Aditi (2014CS10205)
Aditi Singla (2014CS50277)
Aditi Gupta (2017BSY7508)

Design :

The diskPhysical.py file contains the following classes :

1. **class Block :**

   This class is the basic block that contains the data and the virtual block number of its replica.

2. **class Patch :**

   This class is defined so as to be able to efficiently store the patches occupied by disks, and the unused patches available in virtual space. It is stored by maintaining the starting block number and the number of blocks in the patch.

3. **class Disk :**

   This class signifies a virtual disk that the APIs are used to create. It stores all the metadata about where its patches are, and also stores the information required for snapshots of the disk.

This file also instantiates the physical memory, stored as an array of objects of class Block for each physical disk.

We also store the mapping from a block number in the continuous virtual space to the physical disk number and corresponding block number.

The information about unoccupied blocks is also stored as a list of patches (with blocks indexed in virtual space)

The following methods are implemented in this file, which are internally called by the APIs exposed to the users :

a. **writePhysicalBlock()**
b. **readPhysicalBlock()**
c. **getBlockReplica()**
d. **setBlockReplica()**

Implementing Disk Virtualization :

I. Consolidation & Partitioning

Here are the details of the APIs implemented in this section :

a. **createDisk -**

This involves first checking if there is enough space for this disk and a disk with such an id does not exist already. Then, it calls a recursive function called createPatch.

The createPatch function will look for the smallest unoccupied patch which can accommodate the complete disk. If no such patch exists, we need fragmentation. In that case, the largest patch is first engaged for the disk, and then createPatch recursively looks for other unoccupied patches for the remaining portion, updating the unoccupied patches' list.

b. **readDiskBlock -**

Here, the main task is to convert the given block id, which is indexed relative to the particular virtual disk, to the block number in the continuous virtual space. This is done by iterating over all the patches in the disk. For this, a helper function named getVirtualDiskNo() is called.

c. **writeDiskBlock -**

Here, similar to the reading case, we first need to call getVirtualDiskNo() function to convert the block number to an index in the virtual space. Then, it just calls the function provided by the diskPhysical library to write data to the physical block.

d. **deleteDisk -**

We iterate over the patches in the disk, and merge them with the list of unused patches maintained by diskPhysical. Also, the count of used blocks is reduced by the size of the disk. We do not delete the data actually stored in these blocks. They are considered as garbage values.

Helper functions :

- **getVirtualDiskNo()**

II. Replication

The functions in 3.1 were modified to keep a replica of all the blocks that contain data. We assume that when a disk of size x is allocated, it actually has only x/2 blocks , and the other half stores replicas of the first half.

The details of the implementation of the APIs are as follows :

a. **readDiskBlock -**

   We first find a random number between 1 and 100.

   If it is more than 10, we simply read that block and return.

   If it is less than 10, it means we need to simulate a read error. Hence, we find an unallocated block to store the new replica, and then, we update the list of patches of the disk by replacing the original block by the first replica, and replacing the first replica by the new replica we just created. Note that at this point we might need to split a few patches and create a new one, due to the adjustments made.

   The corrupt block is removed from the disk, and also NOT put back to the list of unused patches. This means that block can NEVER be accessed again.

b. **writeDiskBlock -**

   In the write command, we need to update both the original and its replica.

   We first find out if the current replica block number is correct with respect to the current disk. If it's not, we set it to the virtual block number of block number + disk_size/2. This needs to be done to avoid access of any garbage value from a block. Now, we just need to write the data to both the original block and the replica block.

III. Snapshotting & Rolling back

This was implemented as an extension over 3.1. The class Disk maintains a list of all the commands executed over this disk since creation. It also stores checkpoints as an array, where ith element of this array is the index in the list of commands which are included in this checkpoint.

Eg. 10 writes, 5 reads, checkpoint, 5 writes, checkpoint

In this case, the 1st checkpoint will store index 15, and 2nd will store index 20.

Only slight changes were done in the APIs, listed as follows :

a. **writeDiskBlock**
b. **readDiskBlock**

These APIs do one extra thing - add the corresponding operation to the list of commands. The rest of the code is the same.

The implementation of the new APIs in this section is described as follows :

- **checkPoint : This function simply appends the current length of commands list to the list of checkpoints. Also, it returns the index of this new check point.**

- **rollBack : This function first fetches the list of commands and checkpoints of the current disk, and then deletes it. Now, it makes a new disk, and then executes all the commands in the command list upto the checkpoint index. The new disk now stores only the list of commands it executed, and only the checkpoints that were done before the one we rolled back.**