CS 553 Programming Assignment #3

Google App Engine

Instructions:

- ***Due date: 11:59PM on Thursday, 11/07/14***
- ***Maximum Points including extra credit: 110%***
- *You should work in teams of 3 for this assignment.*
- *Please post your questions to the Piazza forum.*
- *Only a softcopy submission is required; it must be submitted to "Digital Drop Box" on Blackboard.*
- *For all programming assignments, please submit just the softcopy; please zip all files (report, source code, compilation scripts, and documentation) and submit it to BB.*
- *Name your file as this rule: "PROG#_LASTNAME1_LASTNAME2_LASTNAME3.{zip|tar|pdf}". E.g. "Prog1_Raicu_Li_Wang.tar".*
- *Late submission will be penalized at 10% per day (beyond the 7-day late pass).*

1. Introduction

This project aims to teach you how to use the Google App Engine to implement a non-trivial distributed system. Your first step is to get familiar with the Google App Engine (http://code.google.com/appengine/). You can sign up for an account at https://appengine.google.com/. You can download the Google App Engine SDK at http://code.google.com/appengine/downloads.html. The getting started guide can be found at http://code.google.com/appengine/docs/. You will find a wealth of information online at the Google App Engine website, however the following two tutorials seem to be quite complete and easy to follow in writing your initial (and trivial) Google App Engine application. They are http://www.vogella.de/articles/GoogleAppEngineJava/article.html and http://www.ibm.com/developerworks/java/library/j-gaej1/.

Once you have familiarized yourself with the Google App Engine, you are to familiarize yourself with two more sub-projects of the Google App Engine, specifically memcache and Google Cloud Storage or Google Cloud Storage Client Library:

- Memcache
  - B. Fitzpatrick. "Distributed caching with Memcached." Linux Journal, 2004(124):5, 2004; http://dl.acm.org/citation.cfm?id=1012894
  - Java: http://code.google.com/appengine/docs/java/memcache/
  - Python: http://code.google.com/appengine/docs/python/memcache/
- Google Cloud Storage Client Library
  - Java: https://cloud.google.com/appengine/docs/java/googlecloudstorageclient/
  - Python: https://cloud.google.com/appengine/docs/python/googlecloudstorageclient/

2. What you will implement?

You will build a distributed storage system on top of Google App Engine, using Google Cloud Storage Client Library. Your storage system should support storing arbitrary files using a unique key, think of a key/value store, where the key is a unique filename (could have some complex path associated with it, but it's not required), and the value would be a file of arbitrary size. The Google Cloud Storage will be a persistent storage solution, but could have particularly high latency overheads for small files. You are to improve the small file size performance by using memcache as a cache for small files (<=100KB). Since memcache is a distributed in-memory data structure, accessing small files from memory should be significantly faster than accessing such small files from persistent storage. Medium and large files should not be cached in memory, as in practice, large storage systems will generally have much more persistent storage capacity than volatile memory available.

2.1. Interface

Your system should be accessible through a web browser (it must work with Firefox or Chrome). We recommend you use JSP (Java Server Pages), but feel free to use any other technologies you see fit, to support the following operations on your distributed storage system:

- *Boolean = Insert(key,value)* //add a file named key with the contents stored in value to the distributed storage system; you should add the entry to the cache at this point as well
- *Boolean = Check(key)* //verify if a file named key exists in the cache or distributed storage system
- *Value = Find(key)* //retrieves the contents of a file named key from the cache or the distributed storage system
- *Boolean = Remove(key)* //remove the file named key from the cache and the distributed storage system
- *Key[] = Listing()* //retrieve a list of all file names as an array

Recall that the key should be a string that uniquely identifies a file, and the value should be the contents of the file. An insert on a file that already exists will simply overwrite the entire contents of the file. A find on a file that does not exist should return a specific value so that it is easy to determine if a file does not exist. A remove should return a Boolean value if it was able to remove a file; if a file does not exist, the remove should fail.

EXTRA CREDIT (10%)

Several other operations you might want to support, to make your implementation more complete. These operations are not required for full credit, but you can get up to 10% extra credit by implementing them.

- *Boolean = Check(key)* //verify if a file named key exists in the distributed storage system
- *Boolean = checkCache(key)* //verify if a file named key exists in the cache of the distributed storage system

- *Boolean = RemoveAllCache()* //remove all files from the cache
- *Boolean = RemoveAll()* //remove all files from the cache and the distributed storage system
- *Double = CacheSizeMB()* //retrieve the total space (in MB) allocated to files in the cache of distributed storage system
- *Double = CacheSizeElem()* //retrieve the total number of files in the cache of distributed storage system
- *Double = StorageSizeMB()* //retrieve the total space (in MB) allocated to files in the distributed storage system
- *Double = StorageSizeElem()* //retrieve the total number of files in the distributed storage system
- *Boolean = FindInFile(key, string)* //searches for a regular expression in file key
- *Key[] = Listing(string)* //retrieve a list of all file names as an array, whose names match the regular expression string

Once you implement some of these operations, some of your benchmarks in section 2.4 might be more efficient than naively running them without such more advanced operations. For example, the RemoveAll() call should be more efficient than first doing a key[] = List(), then for(all key[]) {remove(key[i])}.

2.2. Quotas

You are allowed to have 5GB in data storage free. Various other quotas apply, for how much data can be transferred, how many queries per second/hour/day, etc; see http://code.google.com/appengine/docs/quotas.html for more details.

2.3. Dataset

You are to create a dataset comprising of 411 files spanning 311MB of data, according to the following rules:

- 100 files of 1KB (100KB total)
- 100 files of 10KB (1MB total)
- 100 files of 100KB (10MB total)
- 100 files of 1MB (100MB total)
- 10 files of 10MB (100MB total)
- 1 files of 100MB (100MB total)

These files should be given random alpha-numerical names of 10 characters long. The contents of the files should be random strings of 100 bytes long per line.

You will generate these files on a local client (e.g. your laptop). These files will be transferred to the Google Cloud Storage through the Google App Engine, and make accessible to other clients who want to access these files.

You must include the workload generator source code as part of the overall project source code.

2.4. Benchmark

You will first transfer (e.g. insert(key,value) for all files) the entire dataset from your client (e.g. laptop) to the Google Cloud Storage via the API defined in section 2.1. Careful timings should be done to measure each insert operation. Your client should be multi-threaded so that you can explore using threads to hide latency; run experiments with 1 and 4 threads to measure the performance of insert.

You are to generate a random workload that covers all 411 files with uniform distribution. Repeat file access is acceptable, as long as they are generated under the uniform distribution. Generate a list of 822 file accesses, which should give each file access about 2 accesses. Perform the retrieve workload (e.g. find(key) for all 822 file accesses), pulling all the data twice from the Google Cloud Storage locally. Careful timings should be done to measure each find operation. Your client should be multi-threaded so that you can explore using threads to hide latency; run experiments with 1 and 4 threads to measure the performance of find.

You are to remove all 411 files from the Google Cloud Storage (e.g. remove(key) for all files). Careful timings should be done to measure each remove operation. Your client should be multi-threaded so that you can explore using threads to hide latency; run experiments with 1 and 4 threads to measure the performance of remove.

Run the above 3 workloads with memcache, and without memcache.

2.5. Your results

You have 12 experiments to run:

- Operations: Insert, Find, Remove
- Number of client threads: 1, 4
- Memcache: Yes, No

You can plot all 12 experiments on one figure, but you can also split them up into several figures if you think it is more clear. The metric you should be measuring is overall throughput (MB/s) and average file access latency (ms). You should make sure you understand the minimum latency to the Google Cloud Storage from your client machine, please report on the results of the ping utility between the client machine and the Google Cloud Storage. Please explain your results, comparing the different operations, how the # of threads affected performance, and if memcache did anything to improve the performance.

The workloads are small enough that you are likely to not run into your quota limits assuming you only run these workloads several times a day. Please monitor your usage, and make sure you do not surpass your quotas, as I cannot waive any fees you might incur.

2.6. Comparison to Amazon S3

Assume that your dataset and workload increased 1 million times. That means that your dataset is now 411 million files, that stores 311 TB. The 1 million users need to transfer 311 TB of data to the cloud, and then read 622 TB of data from the cloud. Do a simple analysis of which cloud platform is the most cost effective to run such a distributed storage system on, Amazon

S3 or Google App Engine? You can assume that the workload is spread out over 1 month time. Compute the costs in dollars for each provider, how much you expect to pay to transfer the data in (311TB of data), store the data (311TB of data), and retrieve the data (622TB of data), broken up in these 3 categories. If there is charge per I/O throughputs, you can assume the workload is uniformly spread over the 1 month (30 days) period, with each user's inserts happening first, and then each users finds next. You can use the Amazon Web Services Simple Monthly Calculator at http://calculator.s3.amazonaws.com/calc5.html.

3. What you will submit?

When you have finished implementing the complete assignment as described in section 2, you should submit your solution to 'digital drop box' on blackboard. Each program must work correctly and be detailed in-line documented. You should hand in:

1. **System (50%):**
   - i. **Source code (45%):** All of the source code, scripts, etc; in order to get full credit for the source code, your code must have in-line documents, must compile, and must be able to pass sanity check workloads (based on the 5 APIs defined in Section 2.1).
   - ii. **Deployed System (5%):** Provide a link to your application web address.
2. **Report (50%):**
   - i. **Design (7.5%):** This section should be 1~2 page describing the overall program design, and design tradeoffs considered and made. Also describe possible improvements and extensions to your system (and sketch how they might be made).
   - ii. **Manual (7.5%):** This section should describe how the program works. The manual should be able to instruct users other than the developer how to compile and run the program step by step. Remember to include both client instructions (e.g. what would run on a user's laptop) and the server side instructions (e.g. the code that would be deployed in the cloud, in your case, the Google App Engine).
   - **iii.** **Performance (30%):** Answer the questions posed in Section 2.5. You need to explain each graph or table results in words. Hint: graphs with no axis labels, legends, well defined units, and lines that all look the same, are likely very hard to read and understand graphs. You will be penalized if your graphs are not clear to understand.
   - iv. **Comparison to Amazon S3 (5%):** Answer the questions posed in Section 2.6.

Please put all of the above into one .zip or .tar file, and upload it to 'digital drop box' on blackboard'. Name your file as this rule: "PROG#_LASTNAME1_LASTNAME2_LASTNAME3.{zip|tar|pdf}". E.g. "Prog1_Raicu_Li_Wang.tar". Please do NOT email your files to the professor or the TA!! You do not have to submit any hard copy for this assignment, just a soft copy via Black Board. •