CS553 : Programming Assignment #3

Due on Friday, November 07, 2014

Ioan Raicu

Thomas Dubucq / Tony Forlini / Virgile Landeiro

Contents

1 Introduction

The goal of this programming assignment is to enable us to gain experience programming with:

- Google App Engine (https://appengine.google.com/)
- Google Cloud Storage (https://cloud.google.com/storage/)
- Google's Memcache (https://cloud.google.com/appengine/docs/adminconsole/memcache)

This assignment topic is to discover Google App Engine (GAE) programming. Specifically, this assignment deals with Google Cloud Storage (GCS) and Google Memcache. The goal of this assignment is to create a distributed storage system using the 3 aforementioned Google services. This storage system should have a different behaviour depending on the size of the file to upload, to improve the responsiveness of the whole system.
Google App Engine is a Platform as a Service (PaaS), released in September 2011. GAE allows developpers to implement their applications in several different languages : Python, Java, PHP and Go. GAE apps need to use a web page interface and can be scaled automatically. With GAE comes a lot of different features such as Google Memcache that we will use in this assignment, but also storing, images handling, MapReduce and other features.
Google Cloud Storage is an Infrastructure as a Service (IaaS), made available in March 2010. It allows users to store data on Google's Cloud. Security and redundancy are handled by the cloud's apis, and this service can be accessed by other Google products. In this assignment, we will access it via Google App Engine.
Google Memcache is a GAE feature that allows users to store data of relatively small size on cached memory, which allows faster writing/reading than the GCS. Memcache is separated in 2 classes : dedicated and shared. Dedicated memcache is exclusively allocated to one's application, but has a billing price. On the other end shared memcache is split among all GAE apps and use a best-effort policy to provide space to them. In this assignment we will use the later.

2 Implementation

We decided to implement this assignement in Python.
The developement of a GAE application in Python requires, among others, both cloudstorage and google-cloud-sdk apis, both provided by Google.
The difficulty of this project is that the code is a server-based one, so data retrieving from the client has to be done via HTML requests. Unfortunately HTML requests cannot exceed 32MB. The whole system is subdivided in a front end and back end part. These two parts' implementation is detailed in the following sections.

2.1 Front end

The front end code is the actual interface with the user. It consists in a HTML page where the user can specify its requests to the storage system.
The front end interface has basically two main purposes, retrieving users' requests and sending them to the GAE app. The are fulfilled with two functions get and post, that respectively retrieve instructions from the user, and send them to the GAE app.
The functions get and post rely on the cgi library that allows the handling and processing of HTML forms. The htlm forms are linked to a particular URL in which the POST values are retrieved, then treated in a corresponding class with the StorageApi methods and finally displayed using the GET method. We defined one URL per implemented function from the StorageApi. We will explain the implementation of the

StorageApi methods in a further section. We used different kind of inputs in our forms to retrieve the file names or content as well as buttons.
We implemented a first version which doesn't handle the upload of big files, then we used blobstore to solve this problem even if the instructions were to not use blobstore since we only use it as a third party.
To retrieve a file from the user filesystem to the web application, we used a blobstore file upload input as a third party since we are unable to retrieve files bigger than 32MB from a simple file input within the html form.
Another level of indirection is added with the use of blobstore to retrieve big files because blobstore uses a "blobkey" and a "userkey" to directly put the file content within google cloud storage. The first key points on the file name while the second points to the file content. Thus a dictionnary is created by blobstore while inserting a file in google cloud storage to list the different keys. Consequently we have to make the correspondance between the filename from the user filesystem and the "userkey" from the blobstore dictionnary.
Eventually the web page consist in a list of the API's methods to insert, check remove a file or obtain information about the state of the storage (both cache and google cloud storage). After submitting a form, the user is redirected on another page with the URL coresponding with the method used from the API. Sessions are used to keep the blobstore key dictionnary actualised so that we can retrieve the data already added to the storage. Note that one needs to refresh the main page after adding a file to the storage in order to be able to retrieve the files afterwards.
The web interface is accessible in the following link: http://cs553-cloudcomputing-prog3.appspot.com

2.2 Back end

The back end code is the one running on Google App Engine servers. Its role is to treat the files send by the users and store them. It is also responsible for retrieving data from the storage system, and manage files deletions.
We implemented the following functions to interact with our storage system:

- insert(key, blobinfo) :

  adds the file located at "value" using "key" as a name to the whole storage system. If the file is small enough (<100 kB), it is added to memcache. The file is always added to GCS whatever its size. We use the blobstore to handle the upload of files of more than 32MB. Then the files are added to both the memcache and gcs id the files is less or equal than 100KB and only gcs when the file is more than 100KB. Before actually adding the file, the content is retrieved from the blobstore and then added to gcs or memcache. We delete the content of the file on blobstore right after we add it to the regular storage.

- check(key) :

  verifies if a file named key exists in the whole storage system using the check_cache() and the check_storage() functions.

- find(key) :

  Retrieves the contents of a file named key as a string from the cache or the distributed storage system by reading in the gcs file corresponding to the right blobkey found in the dictionnary.

- remove(key) :

  deletes the file named key from the cache and the distributed storage system using the delete() functions from memcache and gcs APIs bound with the corresponding blobkey.

- listing(string=None) :

  retrives a list of all file names stored as an array. It is possible to specify a regex to this function to only get the matching file subset using the gcs listbucket() function.

- check_storage(key) :

  checks if a file named key exists in the GCS bucket by checking that the key appears in the return value of the listing() function.

- check_cache(key) :

  checks if a file named key exists in memcache using the memcache get() method with the corresponding key.

- remove_all_cache() :

  removes all files from the cache using the memcache flush_all() method.

- remove_all() :

  removes all files from the whole storage system (GCS & memcache) using the remove_all_cache() method and by using the gcd delete() method on all the files of a bucket found in listbucket().

- cache_size_mb() :

  returns the space allocated to the files in memcache (in MB) using the get_stats() method with the "bytes" attribute.

- cache_size_elem() :

  returns the total number of files in memcache using the get_stats() method with the "items" attribute.

- storage_size_mb() :

  returns the total space allocated to files in GCS by summing on the size of each file in the bucket found in file.st_size

- storage_size_elem() :

  returns the total number of files stored in GCS by summing on the number of items in listbucket().

- find_in_file(key,string) :

  searches for a regex in the file named key using the search() method.

- listing_regex(key,string) :

  retrieves the file names matching the regex string as an array using a filter on every files content that matches the string and returning the corresponding key.

3 Performance

The performance evaluation of our implementation consists in testing 3 different types of request : upload, access and deletion of a dataset. These request will be answered using 1 or 4 threads, and with and without the memcache feature.

3.1 Dataset

The dataset used for this performance testing includes 411 files for a total size of 311MB. These files are distributed as following

- 100 files of 1KB
- 100 files of 10KB
- 100 files of 100KB
- 100 files of 1MB
- 10 files of 10MB
- 1 files of 100MB

These files are generated with a python script that is part of the code submission for this assignement. They have random alpha-numerical 10-characters-long names and consist in 100 bytes random string lines.

3.2 Benchmark

There is a total of 12 experiements to conduct. The testing will measure the average throughput in MB/s for the upload and access phases, and the latency for the 3 phases. We implemented two version of the application, the first one doesn't handle the upload of files of more than 32MB, and the second one handles it by using blobstore. Nevertheless, we were unable to measure the transmission time for an upload using the blobstorage. Thus we only computed the benchmark on a version which doesn't hanlde upload of big files.

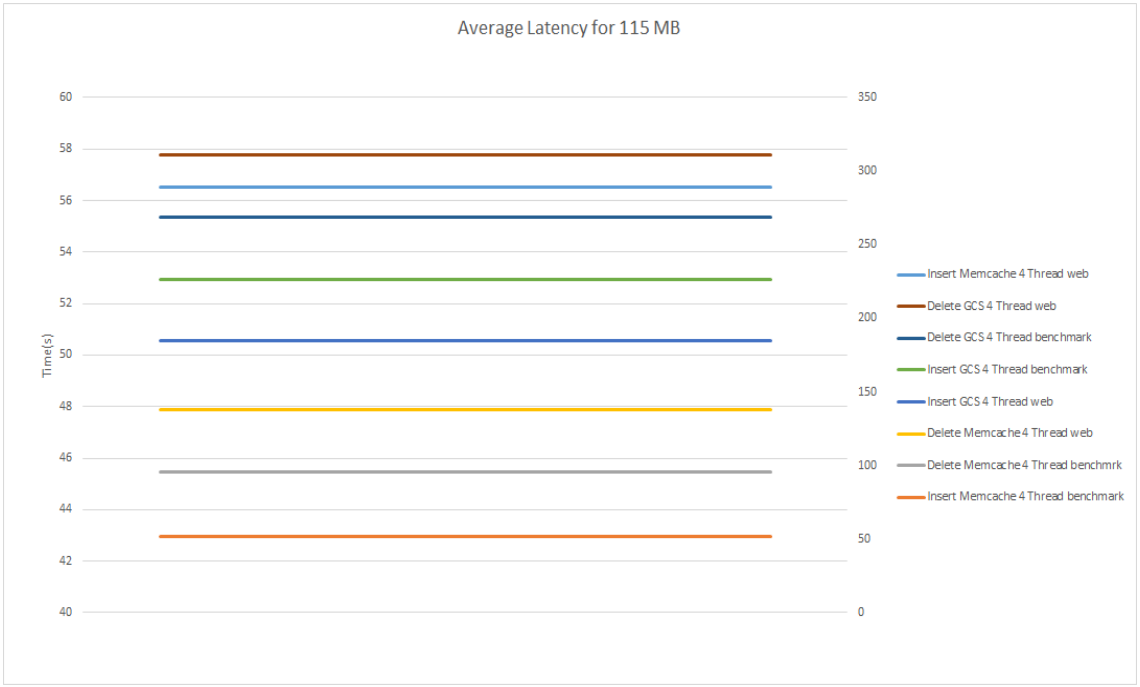

Figure 1: Average Latency for 115 MB

Figure 2: Throughput for 115 MB

3.3 Results

We were unable to mesure the benchmark on the whole dataset, because the http requests were longer than 60 seconds for 311MB which caused deadline exceeded errors. Thus we only uploaded 115 MB and we measured the Insert function and the delete function since the find function since the memory available on the GAE instance wasn't big enough to handle such intensive memory usage. We both measured the latency and throughput with the benchmark code and the web browser tools in order to get objective results and to make sure that our results were in a right range of values. We found that the operations are slower when using both memcache and gcs which makes sense because there is an overhead involved in removing extra files from memcache. Our results with the benchmark are slightly the different as the web browser tools because the latency to access the GAE insance is added with the web browser latency computation.

3.4 Comparison to Amazon S3

In this section we will study the pricing details of high workload applications assuming that our dataset is now of 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.We assume that the workload is uniformly spread out over one month. We first study the Amazon S3 case using the Amazon Web Services Simple Monthly Calculator: http://calculator.s3.amazonaws.com/calc5.html

Amazon S3	
Storage	311 TB
PUT/COPY/POST/LIST Requests	1000000
GET and Other Requests	1000000
Data Transfer Out	622 TB
Data Transfer In	311 TB
Total	110,607$

Now let's take a look at the GAE pricing fo our high workload application. We gathered the pricing informations from google app engine pricing page: https://cloud.google.com/appengine/pricing

GAE Pricing	
Instances (Instance hours)	$0.05
Outgoing Network Traffic (Gigabytes)	$0.12
Incoming Network Traffic (Gigabytes)	Free
Datastore Storage (Gigabytes per month)	$0.18
Blobstore, Logs, and Task Queue Stored Data (Gigabytes per month)	$0.026
Dedicated Memcache (Gigabytes per hour)	$0.06
Logs API (Gigabytes)	$0.12
SSL Virtual IPs(Virtual IP per month)	$39.00

We compute the estimated price on the basis of the use of data transfer and the use of the memcache and google cloud storage according to the size of the file. Here is the estimates total price for our application:

Google App Engine	
DataStore 300 TB	55,296 $
Memcache 11TB	675 $ $
Data Transfer Out 622 TB	76,431 $
Data Transfer In 311 TB	FREE
Total	132,438$

Finally we observe that the overall price for Google App Engine is more expensive than the price of deploying the application on Amazon Web Services.
First the fact that one needs to setup the whole architecture on AWS in order to deploy an application explains the lower cost. Second, high workload and high throughput application are usually better designed using AWS rather than GAE which is more startup application oriented. The higher workload, the lower the monthly price per unit on AWS, besides the pricing is fixed on GAE which prevent from making savings with heavy workloads.

Sources

Google App Engine (https://appengine.google.com/
Google Cloud Storage (https://cloud.google.com/storage/)
Google's Memcache (https://cloud.google.com/appengine/docs/adminconsole/memcache)
Google app engine pricing page: https://cloud.google.com/appengine/pricing
Amazon Web Services Simple Monthly Calculator: http://calculator.s3.amazonaws.com/calc5.html
http://www.vogella.de/articles/GoogleAppEngineJava/article.html
http://www.ibm.com/developerworks/java/library/j-gaej1/