**CS553\_HW3\_Report**

Fan, Zexiang (A20313247)

Liu, Xin (A20314819)

Wang, Dingwen (A20304858)

(To use our application, please visit **http://qualified-cacao-745.appspot.com/**)

**Design**

**Overall program design:**

The program we developed is a cached distributed storage system based on Google App Engine.

Programming tools:

* The language we use is Java, JavaScript, and HTML.
* The libraries we include are Google App Engine SDK (including BlobStore API, DataStore API, and Memcache API) and Google Cloud Storage Client Library.
* The IDEs we use are Maven 3.1.1 and Eclipse Luna (to edit the imported maven project).

Program architecture: the program is a MVC architecture, where:

* Model is GCS objects (files in GCS), Memcache objects (small files in Memcache), and DataStore objects (metadata of the files in GCS stored in DataStore)
* Controller is Java Servlet classes.
* View is HTML pages integrated in JavaScript files.

Functionalities and implementations:

1. **Boolean = Insert (key, value)**: A Client uploads a list of files(HTTP Post request) to the server.

The servlet decodes the request’s header(key) and contents(value), generates a GCS object with the name = key and the contents = value, then store it into GCS.

If the size of the contents is less than 100KB, the servlets generate another Memcache object (same key and value) and stores into Memcache.

Then, the servlet stores the file information (file name and file size) into a DataStore object and stores it into DataStore.

At last, the servlet send a HTTP response to client, indicating whether the store operation succeed.

1. **Boolean = Check (key)**: A client provides a filename(HTTP Post request) to the server.

The servlet get the filename, make a query in DataStore with object name = filename, if there is a object, respond to client a “true” message; otherwise respond a “false” message.

If “true” in the above step, make another query in the memcache, and respond to client accordingly.

(The algorithm above is based on the fact that there CANNOT be an object that exists in memcache but don’t have a counterpart in GCS)

(This function also implement the Boolean = checkCache(key), which is in EXTRA CREDIT)

1. **Value = Find (key)** A client provides a filename(HTTP Get request) to the server.

The servlet get the filename, make a query in DataStore with object name = filename, if there isn’t such an object, send an “error” message to client. Otherwise, make another query in Memcache,

If there is such an object, read this object and generate it as a HTTP response to serve to client.

If there isn’t such an object in Memcache, read from GCS use the object key stored in DataStore, and use BlobStore API to serve this GCS object.

1. **Boolean = Remove(key)** A client provides a filename(HTTP Post request) to the server.

The servlet get the filename, make a query in DataStore with object name = filename, if there isn’t such an object, send an “error” message to client.

Otherwise, the servlet deletes the GCS object with this filename, and deletes corresponding DataStore object, which contains the metadata of the GCS object.

Then, the servlet make another query in Memcache, if there is such a file, delete it.

Then, the servlet send an “succeed” message to the client.

1. **Key[] = Listing()** A client asks the server to list all files(HTTP Post request).

The servlet lists all the filenames stored in the DataStore objects(they are the metadata of the files in GCS), and shows the lists to the client.

**EXTRA CREDITS**

Note: The algorithms of the extra functionalities are similar to the basic ones, some of them are based on the basic ones.

1. **Boolean = Check(key)** and **Boolean = CheckCache(key)** are already implemented in (2).
2. **Boolean = RemoveAllCache()** is implemented by simply calling a “MemcacheService.clearAll()” method defined in Memcache API.
3. **Boolean = RemoveAll()** is implemented by listing all the filenames, and remove each one in GCS, also remove the metadata in DataStore, and finally clear the Memcache.
4. **Double = CacheSizeMB()** and **Double = CacheSizeElem()** are implement by simply calling a “MemcacheService.getStatistics()” and using the Stat object to get the #elements and #bytes information.
5. **Double = StorageSizeMB()** and **Double = StorageSizeElem()** are implemented by listing all the DataStore object, counting the number of objects, and sum up all the “size” properties of the objects.
6. **Boolean = FindInFile(key, string)** and **Key[] = Listing(string)** are implemented using the Java standard package “regex” to match the pattern to the filenames.

**Design tradeoffs and Difficulties:**

* Uploading big files

To implement function (1), i.e. upload files, there is an “Incoming Bandwidth” limitation of GAE, that each HTTP request can be no larger than 32 MB.

To solve this problem, the client side’s JavaScript needs to partition the large file(the 100MB one) into smaller HTTP Post request, and let one final GET request command the GcsService.compose() method. Given that GAE only supports Servlet version 2.5, which don’t provide direct method to carve out multipart/form-data content from an http request, hand writing this function is error prone. But we managed to store exactly the same content uploaded, and we made sure that the result of a GCS compose operation results in the exact file.

* Downloading big files

To implement function (3), there is also a 32MB HTTP respond size limitation. To solve this, a BlobstoreService.serve() method is used to help client downloading large files. This mechanism requires to create a BlobKey with a GCS object name, using the “BlobstoreService.createGSkey()” method.

* Downloading benchmark

Problem is that JavaScript can receive the data as response from the server, but it cannot save them as files to local file system because of its security permission. In our benchmark, we use JavaScript to get the response data, and show size of these data in bytes, and discard it.

* Multithreading

Since the benchmark needs to measure the improvement of multithreading, the JavaScript on the client side has a functionality that partition a big number of HTTP request into four groups, and create four threads to send each group of requests. The Java Servlet Container will automatically handle this multithreading itself.

We first implement the 1 thread version. To make request sequentially actually as not a natural way to express in JavaScript. But by embedding recursive function call in within req.onreadystatechange, we manage to control that a request can only be made after a previous successful request.

Following the 1 thread version, implementing the 4 thread version was actually much simpler. Our JavaScript program first divide all works into 4 lists, and pass each of the lists to on 1 thread to execute.

* Memcache switch

Since the benchmark needs to measure the improvement of caching technique, the client HTTP page has a option to choose whether to enable memcache. If it’s checked, the uploading, downloading and removing processes are different from the ones without caching.

**Possible improvements and extensions**

The largest limitation of the program is that it can only handle text file uploading right now, due to the limitation of our Servlet decoding HTTP request technique. To fix this, we need to implement other techniques to handle Post request that contains images or videos.

**Manual**

**How to compile the program**

Having our source code (Servlet class file, index.html, web.xml and pom.xml), the compilation is followed by the official google website of GAE tutorial.

1. Downloading Java(version 7) and Maven(version 3.1.0 or above)
2. Create a maven project with architype = google app engine skeleton
3. In directory /src/main/java create a directory /com/google/test/
4. Copy all the Servlet class files into src/main/java/com/google/test/
5. Copy the web.xml into src/main/webapp/WEB-INF/
6. Copy the index.html into src/main/webapp/
7. Copy the pom.xml into the project root directory
8. Execute the following command in the project root directory

mvn clean install

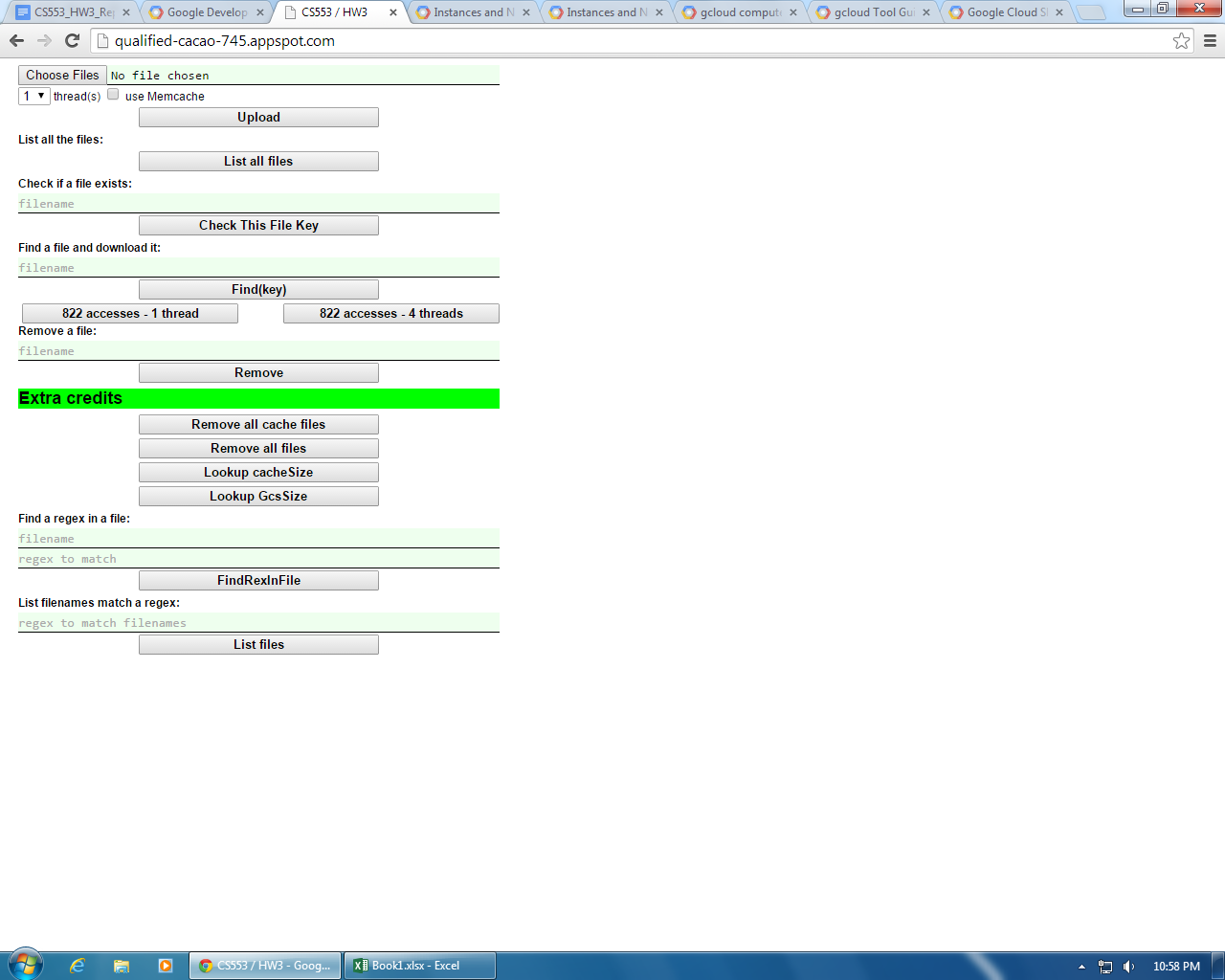
mvn appengine:update

Now, the program should be available on http://qualified-cacao-745.appspot.com/

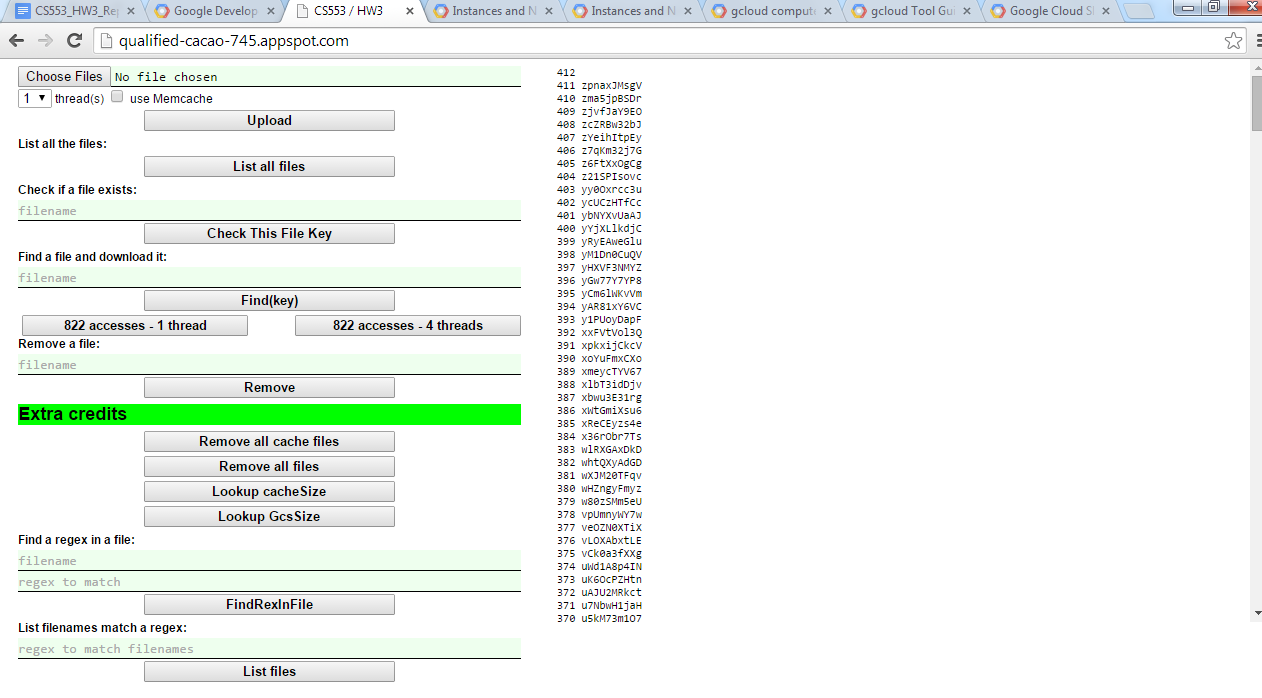
**How to run the program**

When the user access the program website(<http://qualified-cacao-745.appspot.com/>), the list of operations appears on the left hand side, the right hand side will show the result of each operation invoked by the user.

* HomePage (Control panel on the left, output log panel on the right)



* Example: list all files operation



* Upload files

Click the “Choose File” button, choose files to upload, and click “Upload” button.

(For now, we only support text file upload.)

* List files

Click the “List all files” button.

* Check a file

Type the filename in the textbox above, then click “Check This File Key” button.

* Download a file

Type the filename in the textbox above, then click “Find(key)”

* Remove a file

Type the filename in the textbox above, then click “Remove” button.

The **extra functionalities** are listed below the “Extra credits” line, user can use these functions by click the corresponding buttons, especially:

* To see if a filename contains a certain *regular expression*, a user can type the filename and an regular expression (e.g. ^\d, ^[A-Z], etc), then click “FindRexInFIle” button.
* To see the list of filenames that contain a certain regular expression, a user can type the regular expression in the textbox, then click “List files” button.

**Performance benchmarks**

**Experiment dataset, and how we generated it**

The dataset includes 411 files spanning 311MB of data, as described by the PA3 sheet.

We implemented a Java program to generate the files. The program has a sample string, containing the character eligible in the files. A random number generator is declared using the java.util.random package. Then, for each file to generate, the number of lines is calculated by size / 100 byte. For some sizes that are not multiple of 100, the module is calculated as the number of characters in the last line.

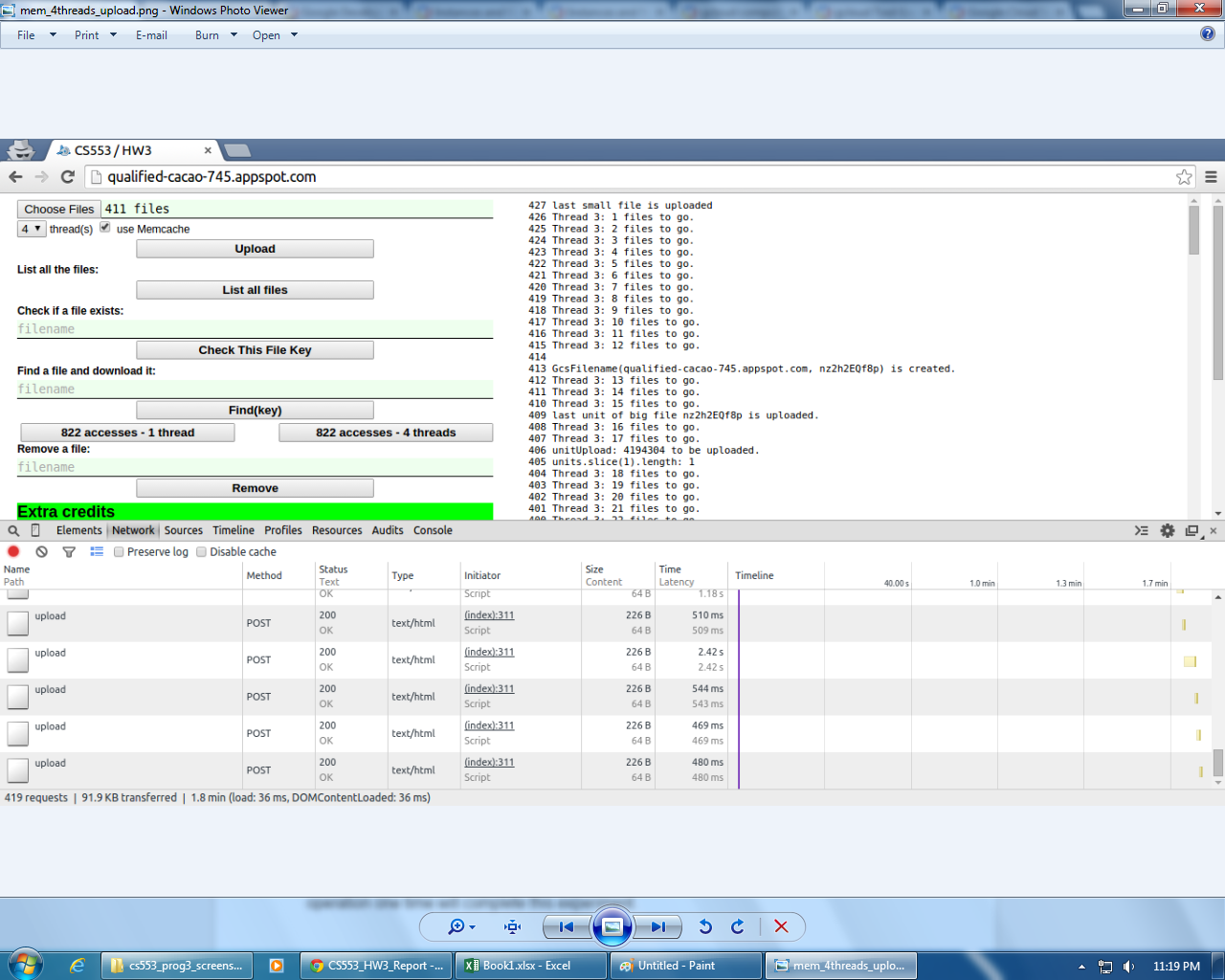
The filenames for these files are generated in a similar way.

**How we run the experiments**

1. Upload All 411 files

Since the HTML form has a multiple upload function, no further implement is needed. Simply invoke insert() operation, and choose all the files will complete this experiment.

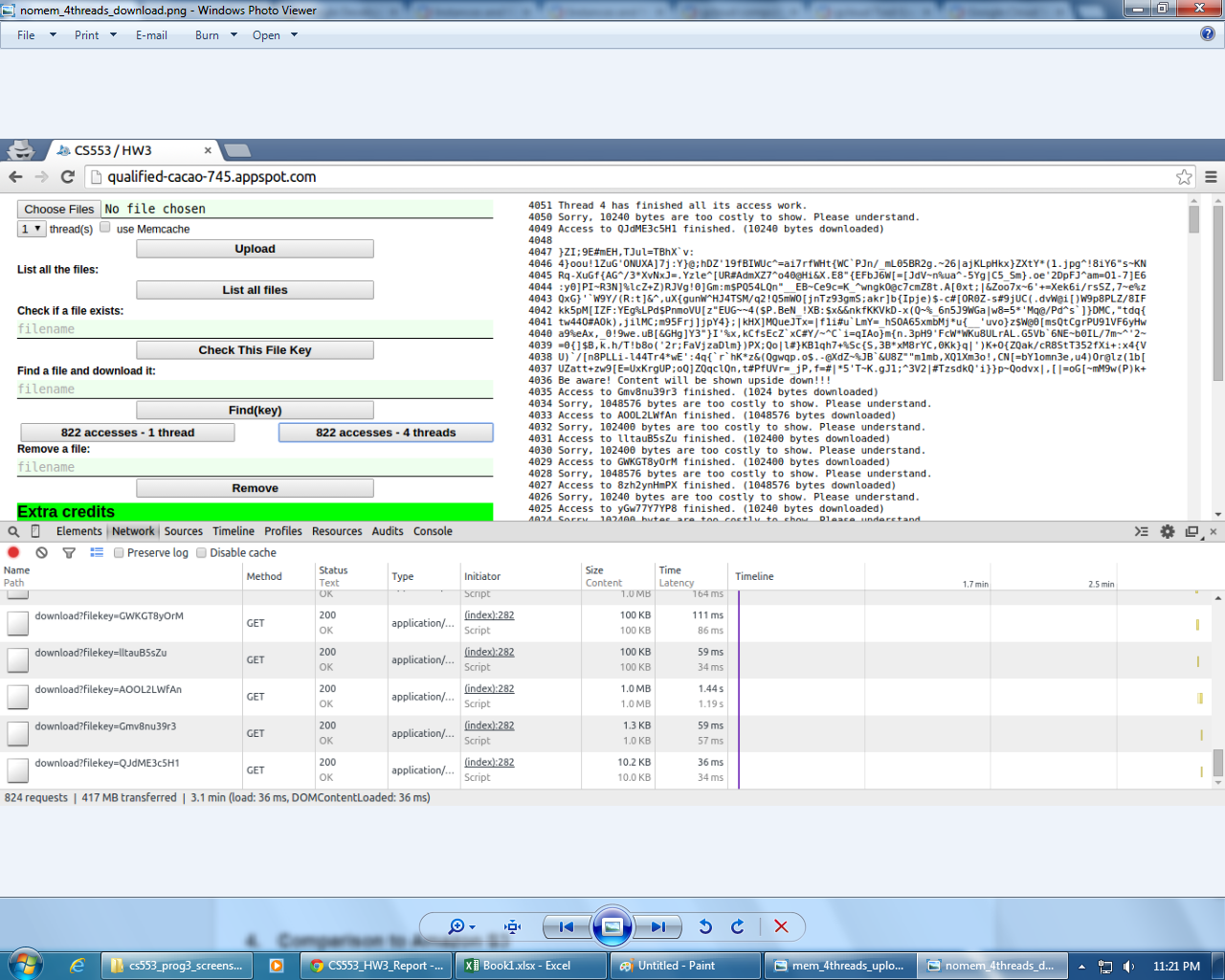
Experiment: 4 threads with MemCache, upload



1. Download 822 times with random filename generating

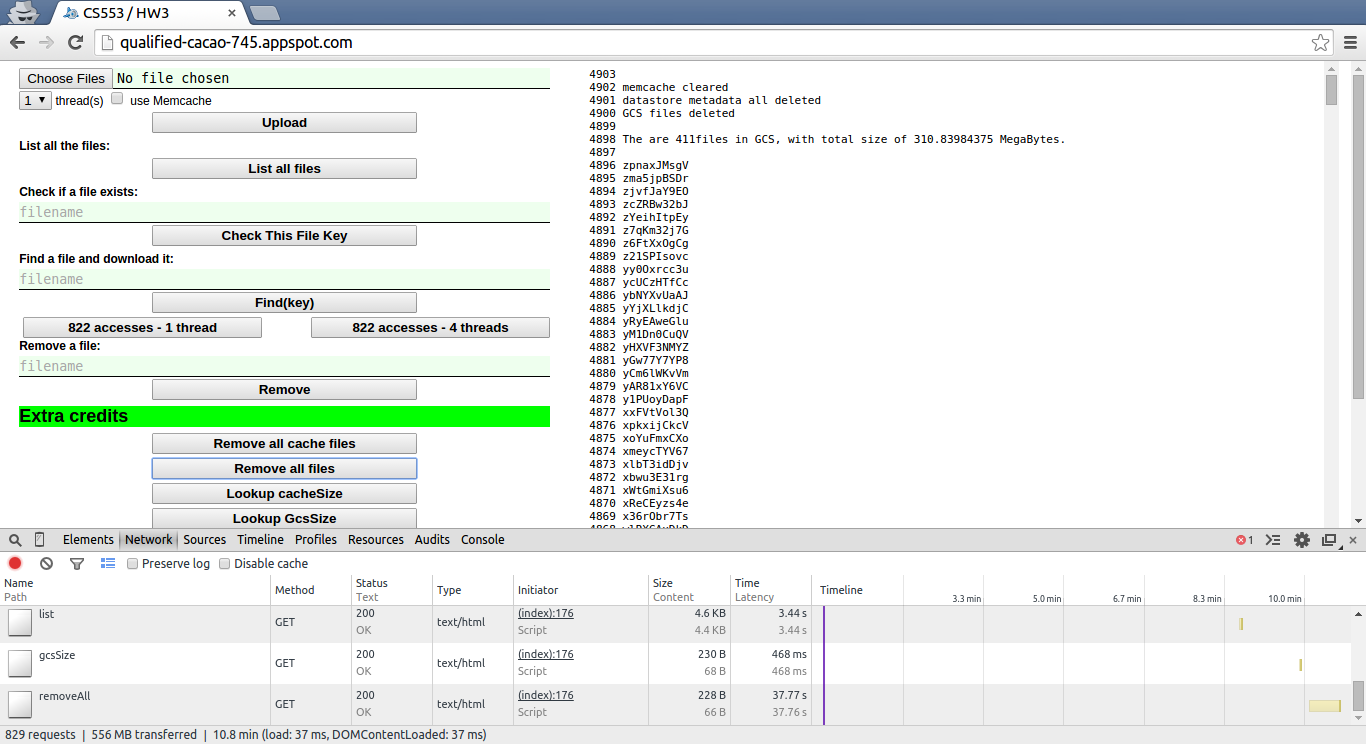
To generate random filename, first we list all the filenames stored in GCS, then make random choice on this sample. Generate 822 times will give us an array of 822 filenames(with some duplication). Then invoke the find() operation use this array.

Experiment: 4 threads with no MemCache, download



1. Remove All 411 files

Since we already implement the RemoveAll() operation, simply invoke this operation one time will complete this experiment.



**Result and analysis**

(*Note*: given that the file access benchmark is generated randomly, each run results in different size of of data being transferred. Our metrics are not affected however, since

*(File access)* ***Latency*** *= Duration of Transfer ÷ 822 (File accesses)*

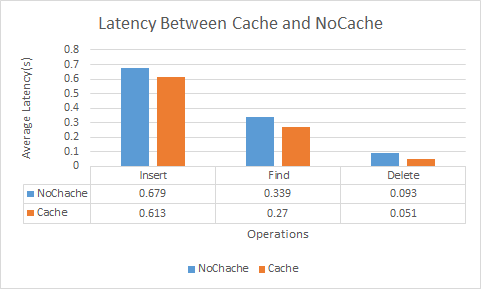
***Throughput*** *= Amount of Data Transferred ÷ Duration of Transfer*

.)

***Latency results:***

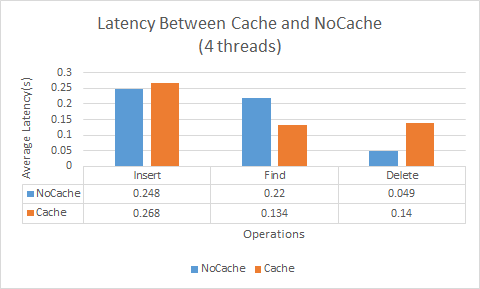
There are 12 experiments in total, as described by the PA3 sheet. The results are shown below:

1. **1 thread, with or without Memcache**



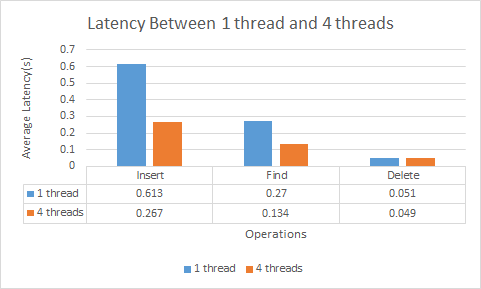
As is shown above, use of Memcache service provides quite some improvements on all of the three operations in terms of *latency*. Although in the case of running in single thread, inefficiency at the client side is high, Memcache still helps. However, the result in deletion is surprising considering that deletion operation with Memcache requires more work. (*Note*: throughout the experiments, we did found out that the deletion operation shew very inconsistent performance.)

1. **4 threads, with or without Memcache**



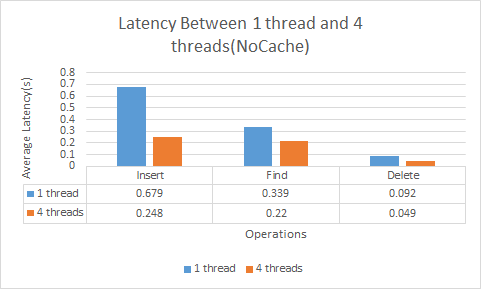
Another thing to notice here is that the deletion in the case of using Memcache uses less time is not reasonable, since by using Memcache, the deletion will require clearing of Memcache in addition to deletion of files in GCS and entities in Datastore, which should lead to worse performance when compared with the without-cache case. In our experiment, deletion without Memcache failed in its first try at 59.7 second. Then we pressed the list function which shows that all files are there (list according to Datastore records). When we try the second time, immediately after, the deletion without Memcache finished in 20 seconds.

1. **With Memcache, 1 or 4 thread(s)**



As is shown above, multi-thread at the client side provides significant improvement. The major reason is that HTTP requests are asynchronous in nature, and making multiple requests simultaneously better utilizes available bandwidth, thus hides latency.

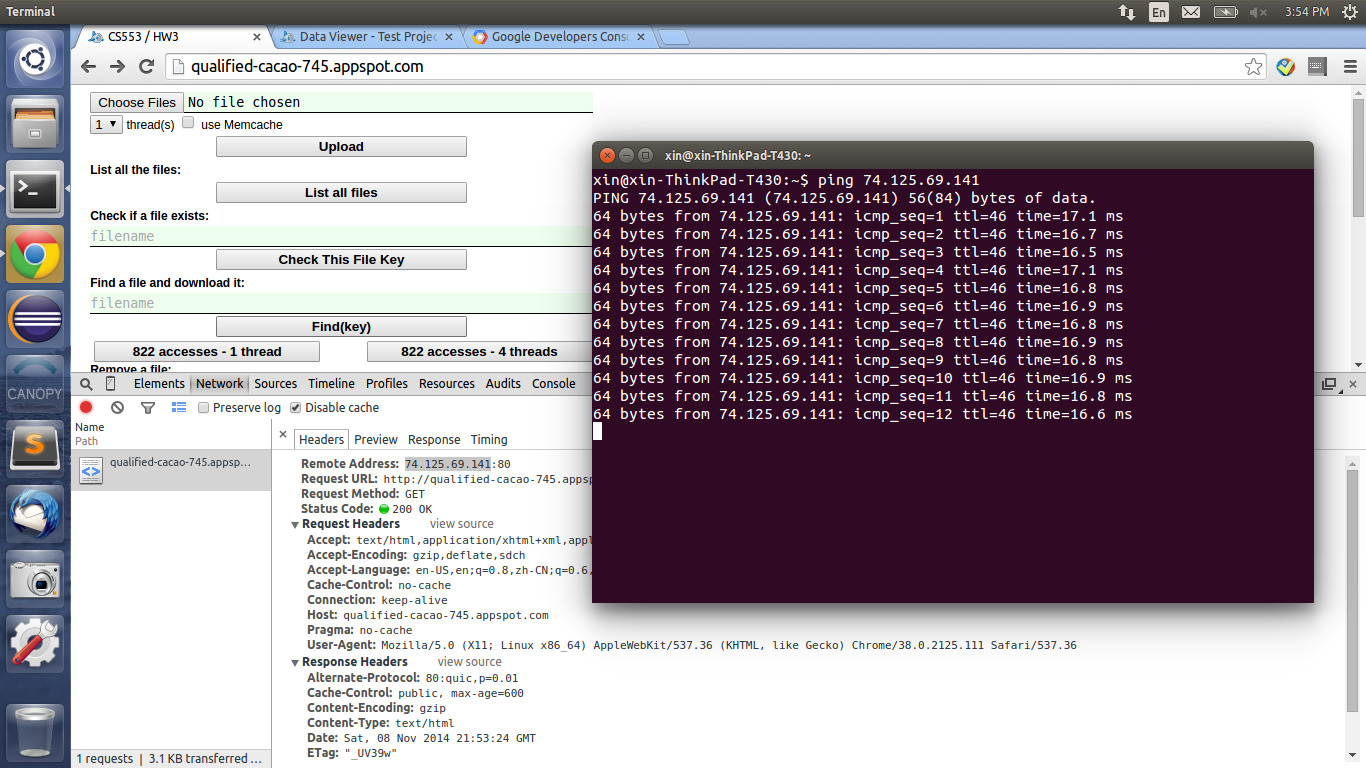
1. **Without Memcache, 1 or 4 thread(s)**



In this graph showing 1 thread v.s. 4 thread, we see the significant improvement as in with Memcache experiments. It again tells us that the client side, if running in 1 thread, is very inefficient, in terms of bandwidth utilization. Nonetheless, by not using Memcache, this experiment gives lower performance than the one with help from Memcache.

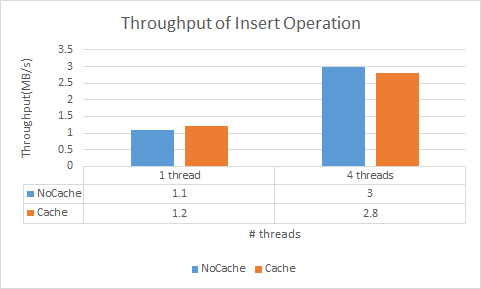
1. **Understand the minimum latency**

We measure the minimum latency from our local machine(which did the experiment) to the remote server of our program using ping utility. ping the server’s remote address

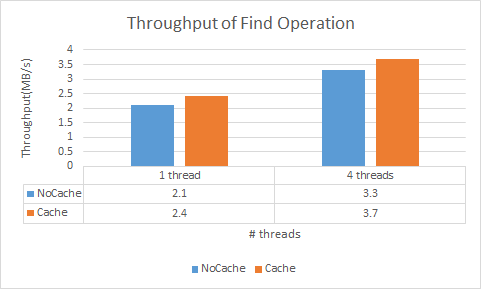


From the graph above, the minimum latency is around **16.8 ms**.

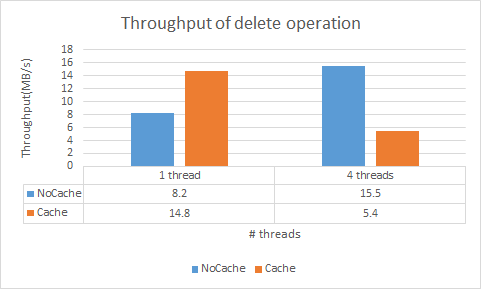
***Throughput results***



As is shown in previous graphs, improvements brought by multithreading is obvious. On the other hand, in 4-thread case, throughput of insertion is lower when using Memcache, which is reasonable, since the insertion with Memcache enabled requires more work.



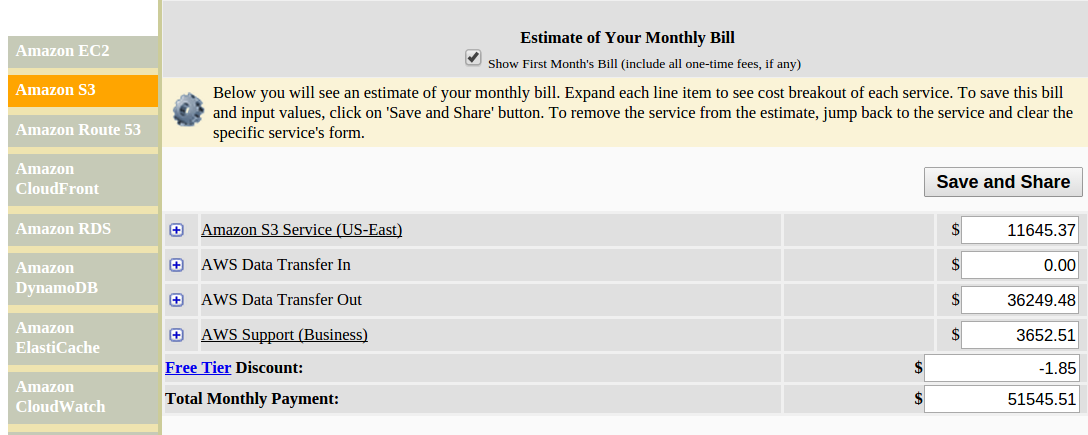
Here we see consistent improvement brought by both multithreading and the use of Memcache service. The result concerning the use of Memcache is not surprising since for every find access, our application checks the target’s availability in Memcache first, and if it is in Memcache, the target is served to user directly.



As we illustrated before, deletion performance is not consistent on GCS. But in the 1 thread case, the result is reasonable, since deletion with Memcache requires more work.

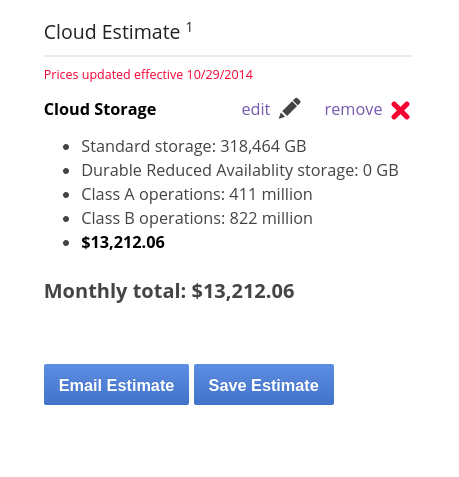
**Comparison to Amazon S3**

The Object is to compare the cost efficiency between Amazon S3 and Google Cloud Storage. Using AWS Simple Monthly Calculator for S3, the result shows below:



It show the total cost of Amazon S3 for one month is 51545.51 dollars.

Using Google Cloud Platform Pricing Calculator, the result shows below:



It shows the total cost of GAE storage for one month is 13212.06 dollars

To conclude, for the given assumption, the better choice is GAE storage.