

Task Execution Framework with Amazon EC2, S3, SQS and Dynamo DB

CS-553, Fall 2014

Summary:

The intent of the project is to develop a task execution framework using EC2, S3, SQS and Dynamo DB. This report estimates the throughput and efficiency of the framework when deployed in an EC2 instance.

1. Introduction:

The amazon services that were used in order to implement the task execution framework are

Amazon EC2:

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. Amazon EC2's simple web service interface allows you to obtain and configure capacity with minimal friction. It provides you with complete control of your computing resources and lets you run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 changes the economics of computing by allowing you to pay as you use.

Amazon S3:

Amazon Simple Storage Service is storage for the Internet. Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites.

Amazon SQS:

Amazon Simple Queue Service (SQS) is a fast, reliable, scalable, fully managed message queuing service. SQS makes it simple and cost-effective to decouple the components of a cloud application. You can use SQS to transmit any volume of data, at any level of throughput, without losing messages or requiring other services to be always available.

Amazon Dynamo DB:

Dynamo DB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. If you are a developer, you can use Dynamo DB to create a database table that can store and retrieve any amount of data, and serve any level of request traffic. Dynamo DB automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent and fast performance. All data items are stored on solid state disks (SSDs) and are automatically replicated across multiple Availability Zones in a Region to provide built-in high availability and data durability.

2. Design:

The application was built using python and a library called **BOTO** that is used to access the services of Amazon such as **SQS, Dynamo DB**.

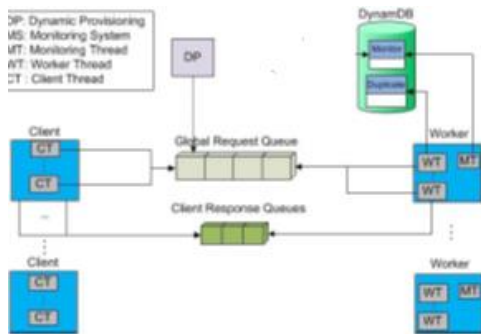
The application is a basic task execution framework which has a client, worker and scheduler. The tasks are given by the client are forwarded to the scheduler and the scheduler which is connected to Amazon SQS, the tasks are again sent to

Task Execution Framework with Amazon EC2, S3, SQS and Dynamo DB

CS-553, Fall 2014

the worker for execution. After the execution is complete the worker will send the **response queue** back to the scheduler. The tasks are sent through TCP connection and the tasks are sent as **JSON objects** through the network.

The other part of the framework is to check for duplication of tasks which is taken care by dynamo DB. In order to scale the program with more number of workers there is a dynamic provisioning system which will increase the number based on the number of tasks SQS. The architecture of the system is given below.



3. The Client:

The client program sends a text file containing sleep jobs or URL's to the scheduler for processing. **Batching** is implemented in the client side to send the files in parts. The client uses sockets in order to communicate with the scheduler which is based on TCP client/server model. It is done while passing 10k tasks to the worker.

4. The Front-End Scheduler/Local Worker:

A **queue** data structure is implemented where the incoming batch files from the client are processed and it is put into the **process queue**. The local back-end worker

fetches the task to be processed from the process queue and after completion, it is sent to the response queue. The scheduler monitors the response queue until the requested object from client arrives, upon arrival it prints the output to client. The scheduler is thread controlled and the threads are created based on the number of TCP connections established with the client.

5. Local Back-End Workers:

Connects with Amazon AWS and gets access to the process and response queue. Incoming files are pushed to SQS to execute the task in workers. The local back-end worker keeps **polling** from response queue for acknowledgement upon completion. The workers are also **thread controlled**, the number of workers created are based on the number of threads.

6. SQS:

We are using SQS for task queuing. There are two queues being used for storing tasks and sending response back to the client.

Sending Task to workers: **Task_to_process**
Sending Response to clients:

Processed_Task

We are storing all the tasks in the queue as **JSON** objects having a **Client_ID**, **Task_ID** and Job description as attributes. A worker can **uniquely identify** its task using **Task_ID**. Since the queue is passive in nature scheduler is supposed to push the task and worker is supposed to pull the task. This results in the increase in the scalability of the queue.

Task Execution Framework with Amazon EC2, S3, SQS and Dynamo DB

CS-553, Fall 2014

7. Remote Back-End Workers:

Remote back-end workers are thread controlled up to **16 threads** where **each thread fetches/polls messages from SQS** service and **checks** if the message is present in **Dynamo DB**, if not makes an entry in Dynamo DB. Based on Dynamo DB's result it proceeds. If the status is **True**, it **deletes** the entry. If the status is **False** it will process the task. Upon execution the result is returned to response SQS. If the **task fails, it is pushed back to process queue** and the status is updated to false so that some other worker can process the task.

8. Dynamo DB:

Dynamo DB makes sure that workers do not process duplicate tasks. The table performs four functions

- Add item
- Remove Item
- Check Status
- Update Status

The parameters of dynamo DB are **hash_key** and the **attributes** (Body-which contains the status of the task).

9. Dynamic Provisioning:

Application uses boto.sqs library to monitor the active queue "Task_to_process". A Threshold value for worker Dynamic provisioner can be set in DynamicProvisioner.py. On crossing the Threshold value a request to configure and launch an instance is made. To create a new instance application uses the AMI Id of existing image which is preconfigured with all the environment settings to run a worker.

10. Animoto Clone:

This module converts a set of images into a movie using "**ffmpeg**". All worker instances are configured with "**ffmpeg**" and the tasks from the client are received from SQS. Once the images are downloaded using "**wget**" and we use a **shell script** which will change the names to a common format as well as convert the images into a video. After the video file is done, it is **uploaded to S3** and the **URL is sent back as response** to the client.

11. Manual:

Environment Settings:

This application requires **Python2.7**, **BOTO**, **FFMPEG** before executing Client, Scheduler, Worker or Dynamic Provisioner.

- 1) Start the scheduler.py using **python scheduler.py**
- 2) Start the client using **Python client.py**
- 3) Enter the path of task file to be sent to the scheduler
- 4) In case of static provisioned worker. Execute **python worker.py** on the worker instance.
- 5) For dynamic provisioning Execute **python DynamicInstanceManager.py**
- 6) Dynamic provisioner will invoke new worker instance if a threshold value of task is reached.
- 7) In order to execute Animoto tasks there is another program called **workerAnimoto.py**, which should be executed in the instance.

12. Performance Evaluation:

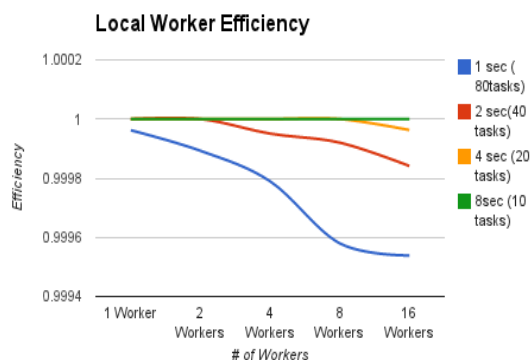
- 1) Dynamic Provisioning allows new instance to be created by checking the number of messages in SQS. A Threshold value for Queue size can be set considered in order to trigger a

Task Execution Framework with Amazon EC2, S3, SQS and Dynamo DB

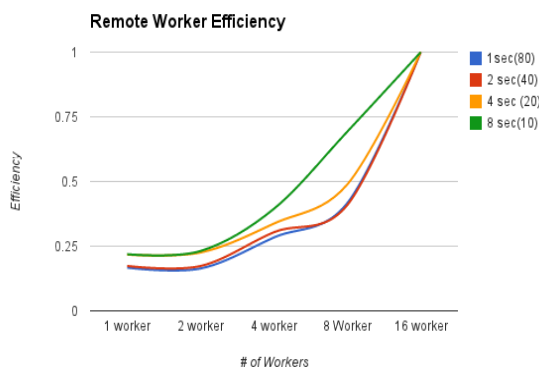
CS-553, Fall 2014

new instance. There are overheads such as creating and booting time of an instance. We compared the performance of Dynamic provisioned with static provisioned worker and found that the average boot time for a m2.medium instance is about 1min 20 sec. The dynamic provisioning values are given in the datasheet.

2) Efficiency



We can see that the efficiency of processing the task decreases after adding a large number of workers (threads), this is due to the overheads of worker thread management. The efficiency is calculated by dividing the Ideal time by the time taken by each process.



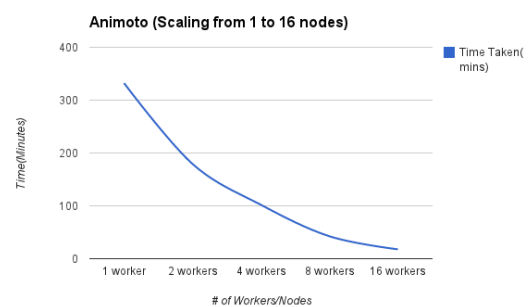
We can see from the following graph that as the number of workers increases the efficiency of the reaches 1. The maximum

efficiency is achieved when 16 nodes run in parallel.

3) Throughput

The values of throughput and efficiency are given in the datasheet below, and as we scale up the number of nodes, the throughput increase. Essentially the tasks executed per second increases.

4) Animoto



Initially to process 160 jobs of 60 pictures it approximately takes 330 seconds for one node to finish the job, but as we scale up the time taken to complete the job reduces significantly and jobs are done in parallel. Refer datasheet for values.

13. Conclusions:

- Amazon SQS provides high scalability and guarantees message delivery to the workers
- Dynamo DB prevents the execution of duplicate tasks
- Dynamic provisioning will be used in order to scale the system in order to accommodate heavy workloads

DataSets

1. Throughput Remote Worker.

Remote worker Throughput		
#Workers	Time	Task/sec
1	779	12.83697047
2	723	13.83125864
4	565	17.69911504
8	294	34.01360544
16	277	36.10108303

2.

Local Worker Efficiency				
	1 sec (80 tasks)	2 sec(40 tasks)	4 sec (20 tasks)	8sec (10 tasks)
1 Worker	80.102035	80.09159	80.0872	80.08548
2 Workers	80.1076	80.09184	80.08742	80.08495
4 Workers	80.11586	80.10295	80.08779	80.08542
8 Workers	80.13267	80.10539	80.09058	80.08573
16 Workers	80.13597	80.11171	80.10198	80.08583

3.

Remote Worker Efficiency				
	1sec(80)	2 sec(40)	4 sec (20)	8 sec(10)
1 worker	0.1670582677	0.1734110947	0.2174033423	0.2189375716
2 worker	0.1637942202	0.1732942494	0.2256460147	0.2313587328
4 worker	0.2838146427	0.3040618975	0.3374618665	0.3946474702
8 Worker	0.4171122451	0.4091769266	0.4889136691	0.6931465164
16 worker	1	1	1	1

DataSets

4.

Dynamic vs Static Provisioning	
# WorkerType	Time(s)
Dynamically Provisioned	72.79
Statica Provisioned	2.19

5.

Animoto Performance	
No of nodes	Time Taken(mins)
1 worker	332.2252
2 workers	179.3065
4 workers	101.9036
8 workers	42.4345
16 workers	17.7643

SCREENSHOTS

SQS:

The screenshot shows the AWS SQS Management Console in Google Chrome. The browser tabs include 'PA4 benchmark - Go', 'Cloud Assignment 4', 'SQS Management C', 'EC2 Management C', 'Python Lists', and '[kubuntu] Get SSH'. The address bar shows the URL: <https://console.aws.amazon.com/sqs/home?region=us-west-2#queue-browser:selected=https://sqs.us-west-2.amazonaws.com/509286232126>. The console shows a list of queues with the following data:

Name	URL	Messages Available	Messages In Flight	Created	Visibility Timeout
Processed_Task	https://sqs.us-west-2.amazonaws.com/509286232126/Processed_Task	4	0	2014-12-06 21:35:53 GMT-06:00	0 seconds
Task_to_Process	https://sqs.us-west-2.amazonaws.com/509286232126/Task_to_Process	1,054	0	2014-12-05 17:02:03 GMT-06:00	0 seconds

Below the table, the details for the selected queue 'Processed_Task' are shown:

- Name: Processed_Task
- URL: https://sqs.us-west-2.amazonaws.com/509286232126/Processed_Task
- ARN: arn:aws:sqs:us-west-2:509286232126:Processed_Task
- Created: 2014-12-06 21:35:53 GMT-06:00
- Last Updated: 2014-12-06 21:35:53 GMT-06:00
- Delivery Delay: 0 seconds
- Default Visibility Timeout: 0 seconds
- Message Retention Period: 4 days
- Maximum Message Size: 256 KB
- Receive Message Wait Time: 0 seconds
- Messages Available (Visible): 4
- Messages In Flight (Not Visible): 0

The footer shows the copyright notice: © 2008 - 2014, Amazon Web Services, Inc. or its affiliates. All rights reserved. and links to Privacy Policy and Terms of Use. A 'Feedback' button is also present.

Dynamo DB:

The screenshot shows the AWS DynamoDB console in Google Chrome. The browser tabs include 'PA4 benchmark - Go', 'Cloud Assigni', 'SQS Manag', 'EC2 Manag', 'DynamoDB', '[kubuntu] C', 'Quick Note', 'Review Subr', and 'Inbox (8,999)'. The address bar shows the URL: https://us-west-2.console.aws.amazon.com/dynamodb/home?region=us-west-2#table:name=Task_Monitor. The console shows the 'Task_Monitor' table with the following details:

Name	Status	Hash Key	Range Key	Sum of Read Throughput	Sum of Write Throughput
Task_Monitor	ACTIVE	Job_ID	-	10	10

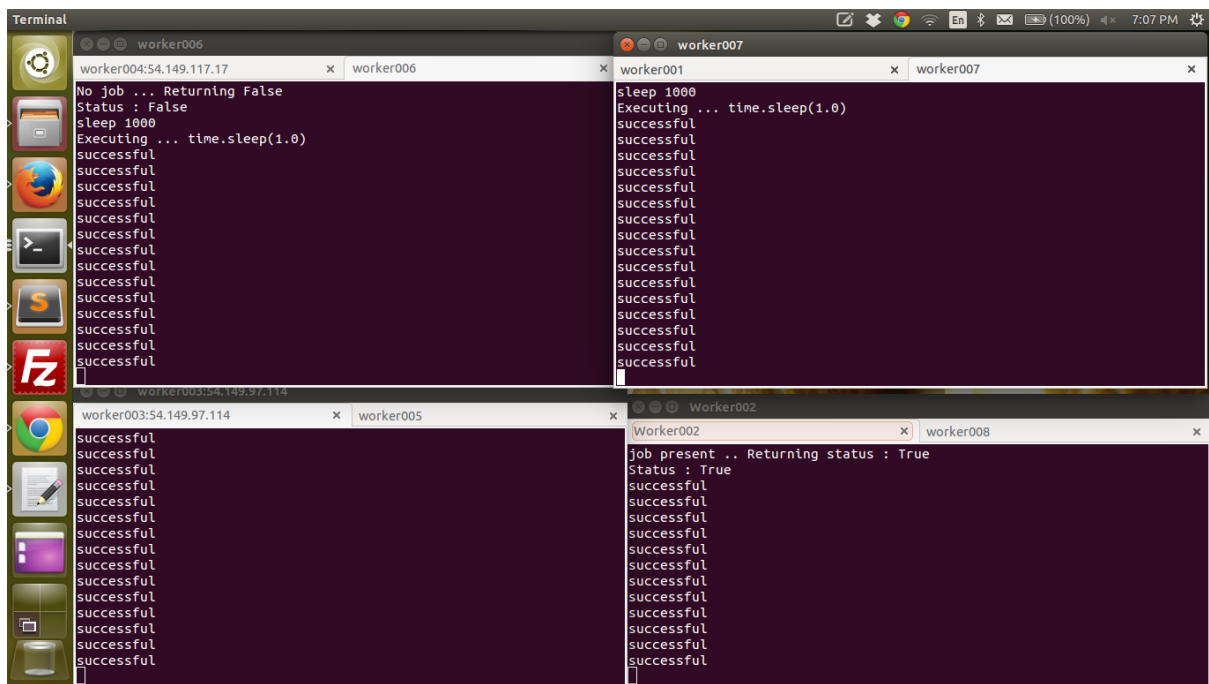
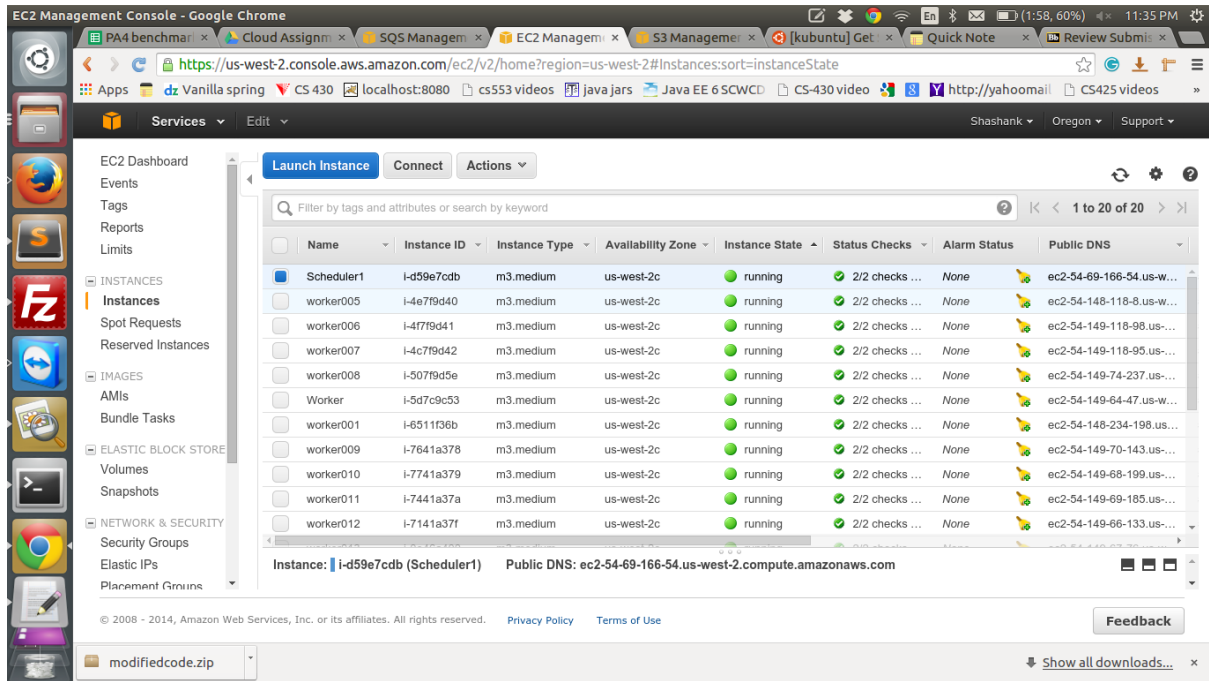
Below the table, several performance metrics are displayed as line graphs:

- Write Capacity (Units/Second - 5 Minute Average):** Shows Provisioned Write Capacity (red line) and Average Consumed Write (blue line).
- Throttled Write Requests (Count):** Shows the count of throttled write requests for Put, Update, Delete, and Batch Write operations.
- Get Latency (Milliseconds):** Shows the latency for Get operations.
- Put Latency (Milliseconds):** Shows the latency for Put operations.
- Query Latency (Milliseconds):** Shows the latency for Query operations.
- Scan Latency (Milliseconds):** Shows the latency for Scan operations.
- User Errors (Count):** Shows the count of user errors.

The footer shows the copyright notice: © 2008 - 2014, Amazon Web Services, Inc. or its affiliates. All rights reserved. and links to Privacy Policy and Terms of Use. A 'Feedback' button is also present.

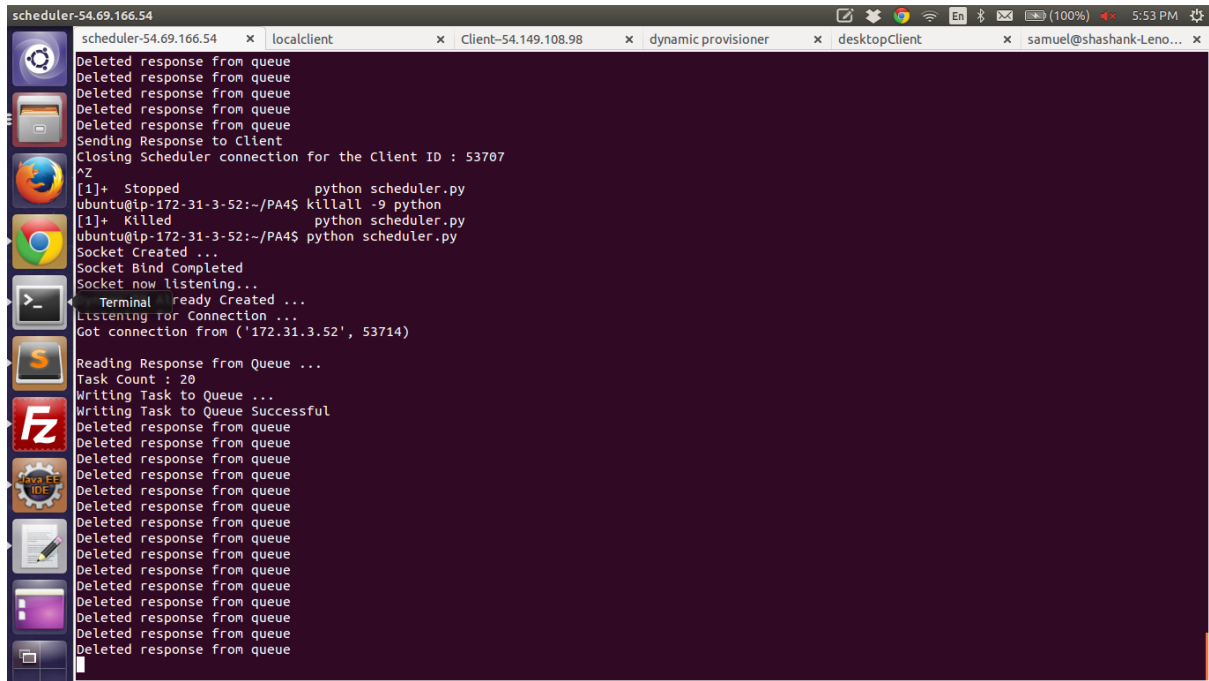
SCREENSHOTS

Workers (1-16):



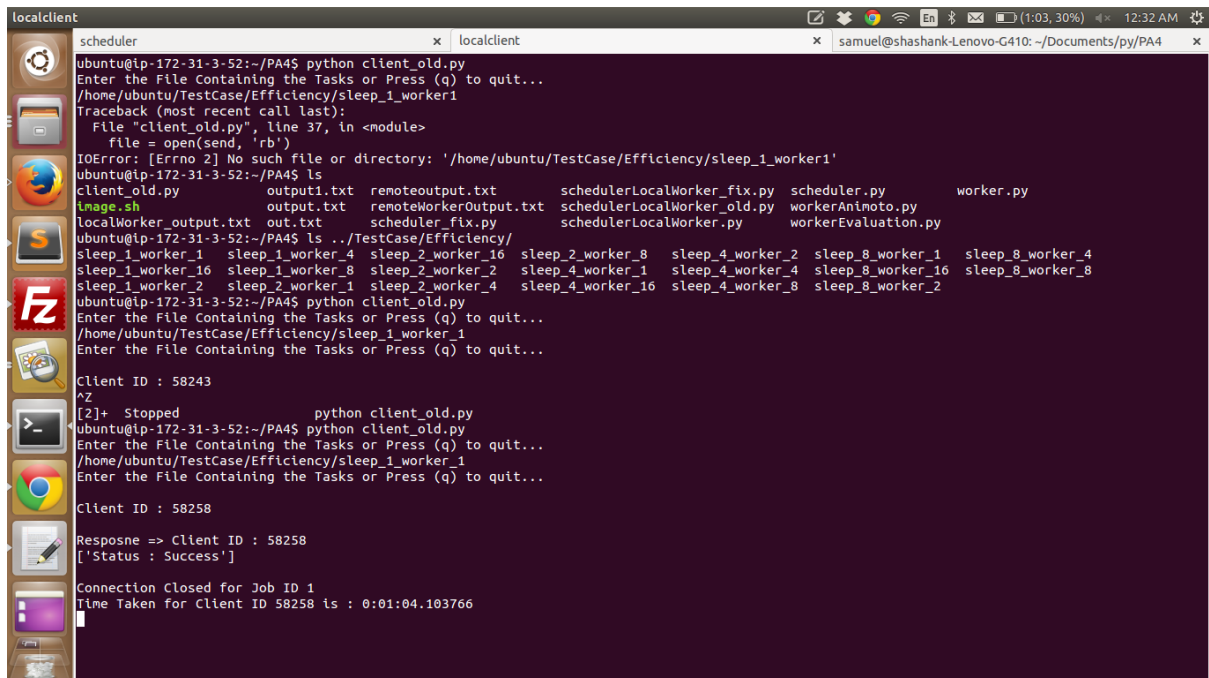
SCREENSHOTS

Scheduler:



```
scheduler-54.69.166.54
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Sending Response to Client
Closing Scheduler connection for the Client ID : 53707
^Z
[1]+  Stopped                  python scheduler.py
ubuntu@ip-172-31-3-52:~/PA4$ killall -9 python
[1]+  Killed                    python scheduler.py
ubuntu@ip-172-31-3-52:~/PA4$ python scheduler.py
Socket Created ...
Socket Bind Completed
Socket now listening...
Terminal ready Created ...
Listening for Connection ...
Got connection from ('172.31.3.52', 53714)
Reading Response from Queue ...
Task Count : 20
Writing Task to Queue ...
Writing Task to Queue Successful
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
Deleted response from queue
```

Client:



```
localclient
scheduler
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/Efficiency/sleep_1_worker1
Traceback (most recent call last):
  File "client_old.py", line 37, in <module>
    file = open(send, 'rb')
IOError: [Errno 2] No such file or directory: '/home/ubuntu/TestCase/Efficiency/sleep_1_worker1'
ubuntu@ip-172-31-3-52:~/PA4$ ls
client_old.py      output1.txt  remoteoutput.txt  schedulerLocalWorker_fix.py  scheduler.py  worker.py
image.sh          output.txt  remoteWorkerOutput.txt  schedulerLocalWorker_old.py  workerAnimoto.py  workerEvaluation.py
localWorker_output.txt  out.txt    scheduler_fix.py    schedulerLocalWorker.py
ubuntu@ip-172-31-3-52:~/PA4$ ls ../TestCase/Efficiency/
sleep_1_worker_1  sleep_1_worker_4  sleep_2_worker_16  sleep_2_worker_8  sleep_4_worker_2  sleep_8_worker_1  sleep_8_worker_4
sleep_1_worker_16  sleep_1_worker_8  sleep_2_worker_2  sleep_4_worker_1  sleep_4_worker_4  sleep_8_worker_16  sleep_8_worker_8
sleep_1_worker_2  sleep_2_worker_1  sleep_2_worker_4  sleep_4_worker_16  sleep_4_worker_8  sleep_8_worker_2
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/Efficiency/sleep_1_worker_1
Enter the File Containing the Tasks or Press (q) to quit...
Client ID : 58243
^Z
[2]+  Stopped                  python client_old.py
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/Efficiency/sleep_1_worker_1
Enter the File Containing the Tasks or Press (q) to quit...
Client ID : 58258
Response => Client ID : 58258
['Status : Success']
Connection Closed for Job ID 1
Time Taken for Client ID 58258 is : 0:01:04.103766
```

SCREENSHOTS

Scheduler and Local Worker with throughput of 100k sleep 0 tasks and output in client:

The screenshot shows a Windows desktop environment. On the left side, there is a vertical taskbar with icons for a clock, calendar, and several web browsers (Chrome, Firefox, Edge). The main area of the screen is occupied by a terminal window titled 'scheduler'. The terminal output shows a series of 'Executing ... time.sleep(0.0)' commands, indicating a loop that is running. At the bottom of the terminal, it says 'Sending Response to Client' and 'Closing Scheduler connection for the Client ID : 58237'. The top of the screen shows the Windows taskbar with icons for network, volume, and battery, along with the system clock showing 11:58 PM on 11/48/48.

localclient scheduler x localclient

```
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/Throughput/sleep_0_100k.txt
Enter the File Containing the Tasks or Press (q) to quit...

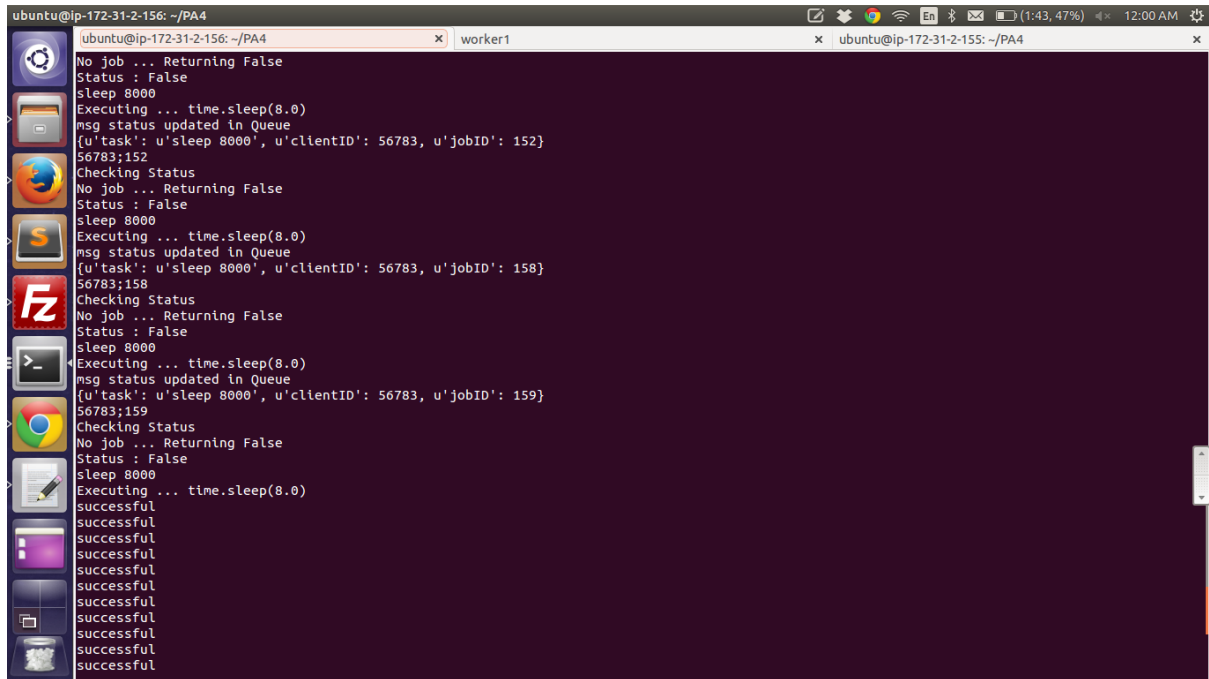
Client ID : 58237

Resposne => Client ID : 58237
Status : Success

Connection Closed for Job ID 1
Time Taken for Client ID 58237 is : 0:00:10.076575
```

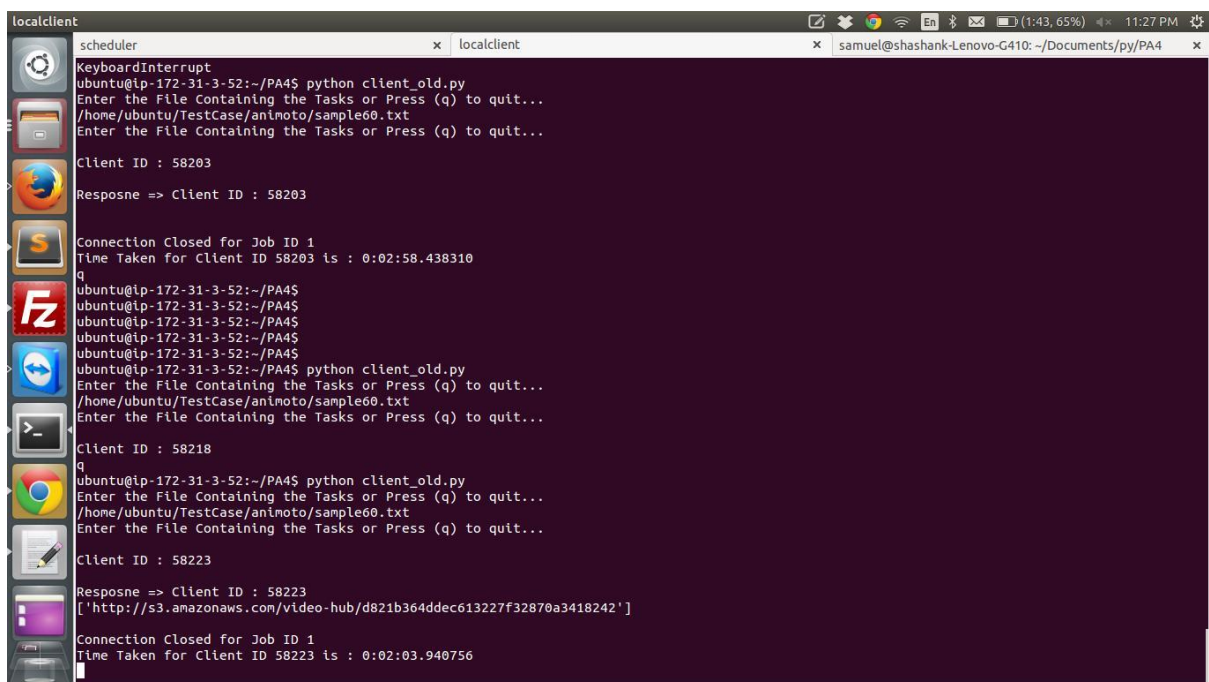
SCREENSHOTS

Remote Worker:



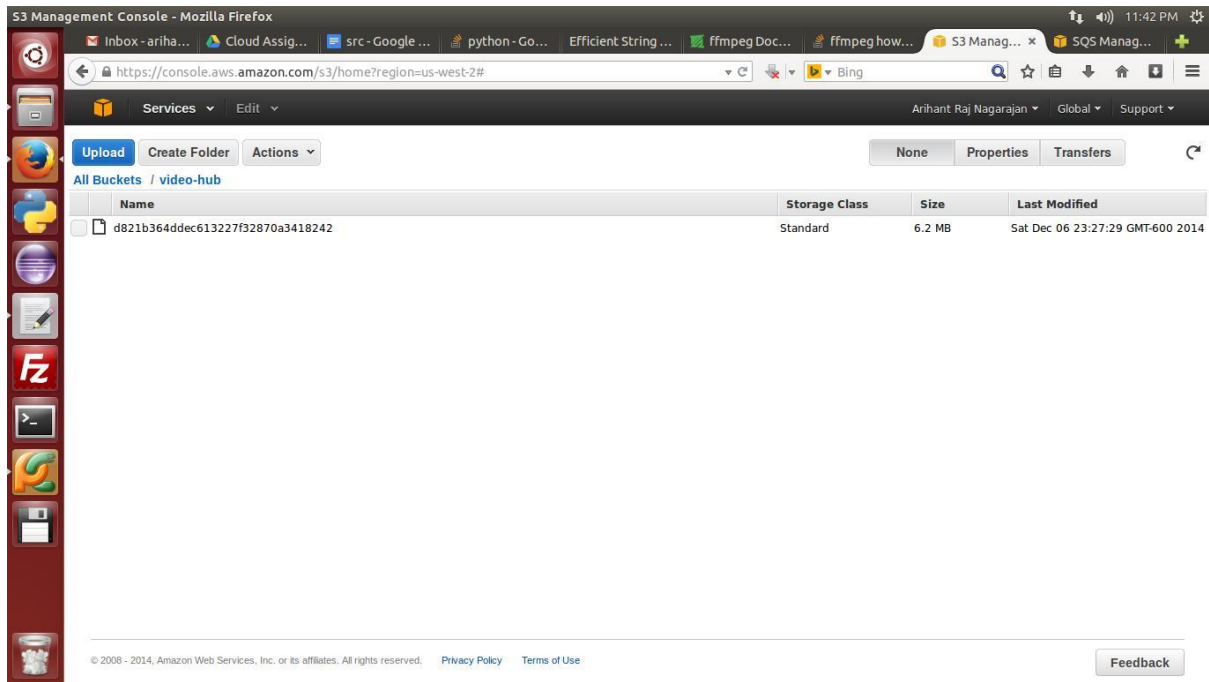
```
ubuntu@ip-172-31-2-156: ~/PA4
No job ... Returning False
Status : False
sleep 8000
Executing ... time.sleep(8.0)
msg status updated in Queue
{'u'task': 'u'sleep 8000', 'u'clientID': 56783, 'u'jobID': 152}
56783;152
Checking Status
No job ... Returning False
Status : False
sleep 8000
Executing ... time.sleep(8.0)
msg status updated in Queue
{'u'task': 'u'sleep 8000', 'u'clientID': 56783, 'u'jobID': 158}
56783;158
Checking Status
No job ... Returning False
Status : False
sleep 8000
Executing ... time.sleep(8.0)
msg status updated in Queue
{'u'task': 'u'sleep 8000', 'u'clientID': 56783, 'u'jobID': 159}
56783;159
Checking Status
No job ... Returning False
Status : False
sleep 8000
Executing ... time.sleep(8.0)
successful
successful
successful
successful
successful
successful
successful
successful
successful
successful
```

Animoto and the file stores in S3 bucket:

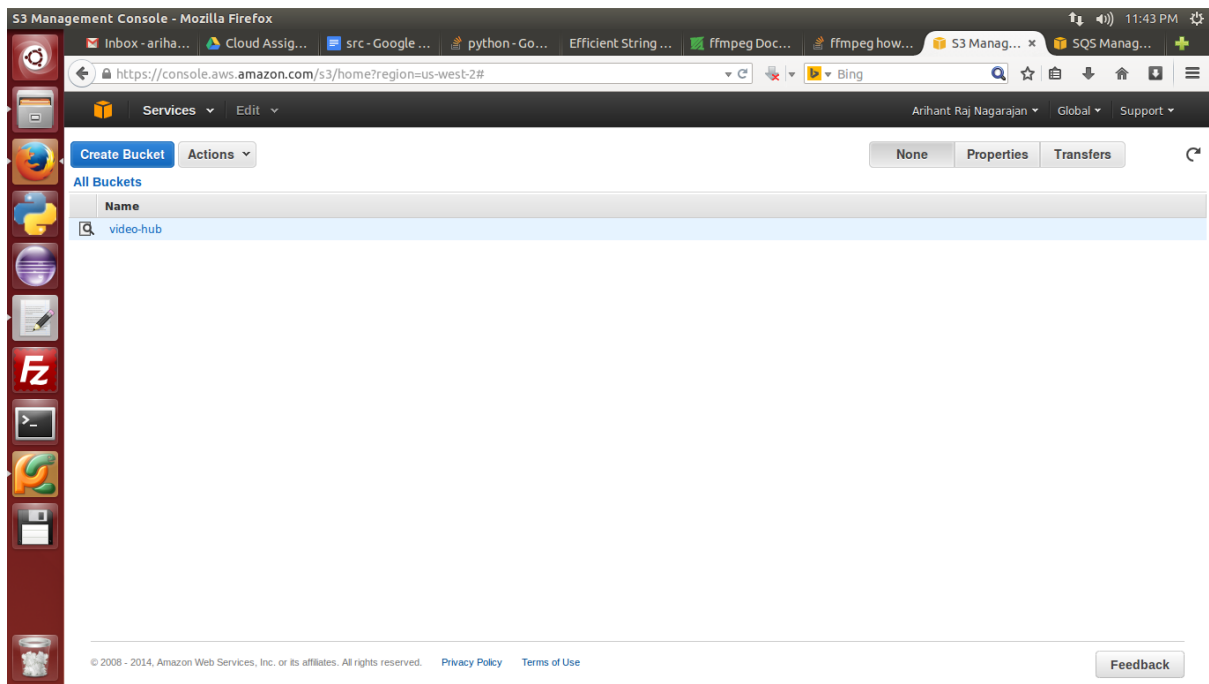


```
localclient
scheduler
KeyboardInterrupt
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/animoto/sample60.txt
Enter the File Containing the Tasks or Press (q) to quit...
Client ID : 58203
Resposne => Client ID : 58203
Connection Closed for Job ID 1
Time Taken for Client ID 58203 is : 0:02:58.438310
q
ubuntu@ip-172-31-3-52:~/PA4$
ubuntu@ip-172-31-3-52:~/PA4$
ubuntu@ip-172-31-3-52:~/PA4$
ubuntu@ip-172-31-3-52:~/PA4$
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/animoto/sample60.txt
Enter the File Containing the Tasks or Press (q) to quit...
Client ID : 58218
q
ubuntu@ip-172-31-3-52:~/PA4$ python client_old.py
Enter the File Containing the Tasks or Press (q) to quit...
/home/ubuntu/TestCase/animoto/sample60.txt
Enter the File Containing the Tasks or Press (q) to quit...
Client ID : 58223
Resposne => Client ID : 58223
['http://s3.amazonaws.com/video-hub/d821b364ddec613227f32870a3418242']
Connection Closed for Job ID 1
Time Taken for Client ID 58223 is : 0:02:03.940756
```

SCREENSHOTS

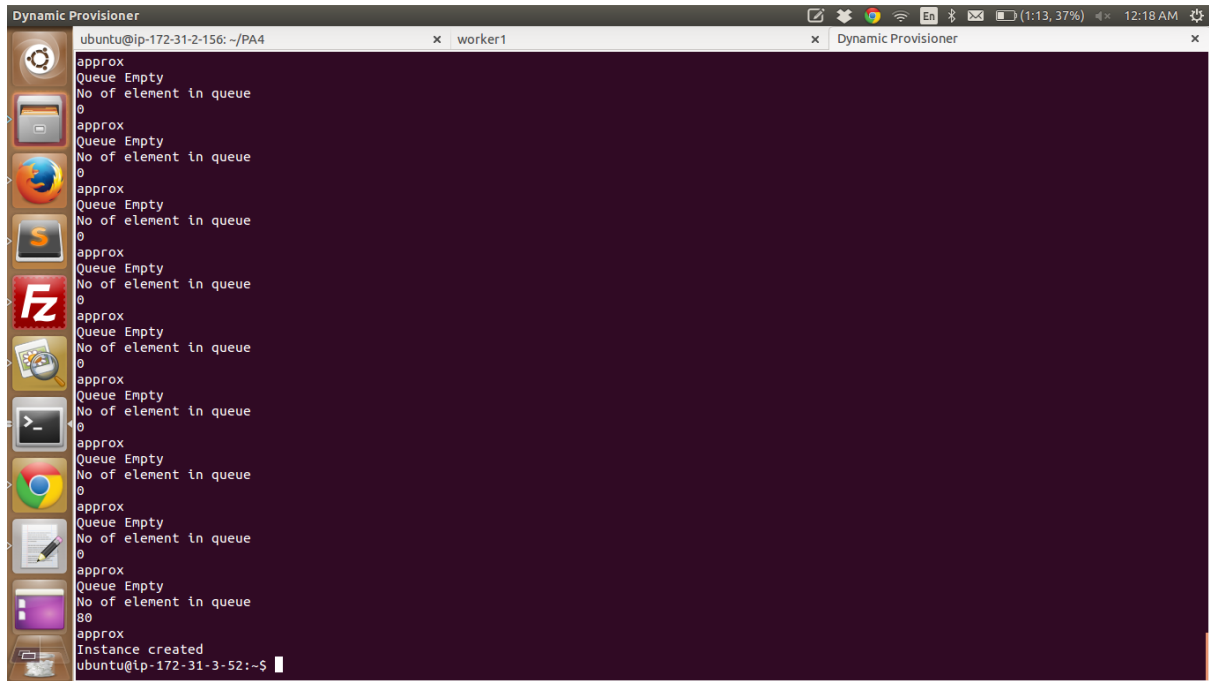


S3 video-hub bucket created for Animoto:



SCREENSHOTS

Dynamic Provisioning:



References:

CloudKon: http://www.cs.iit.edu/~iraicu/teaching/CS554-F13/lecture03_CloudKon-overview.pdf

FalKon: http://datasys.cs.iit.edu/publications/2007_SC07_FalKon.pdf

Amazon AWS Documentation: http://aws.amazon.com/documentation/?nc2=h_l2_su/

Tutorials Point python: http://www.tutorialspoint.com/python/python_networking.htm

Tutorials Point python: http://www.tutorialspoint.com/python/python_multithreading.htm

Boto Documentation for S3: <http://boto.readthedocs.org/en/latest/ref/s3.html>

Boto Documentation for dynamoDB:
<http://boto.readthedocs.org/en/latest/ref/dynamodb.html>

Boto Documentation for SQS: <http://boto.readthedocs.org/en/latest/ref/sqs.html>

Boto Documentation for EC2: <http://boto.readthedocs.org/en/latest/ref/ec2.html>

Ffmpeg : http://www.ffmpeg.org/faq.html#How-do-I-encode-single-pictures-into-movies_003f

Ffmpeg: <http://askubuntu.com/questions/432542/is-ffmpeg-missing-from-the-official-repositories-in-14-04>

S3 api: <http://ceph.com/docs/master/radosgw/s3/python/>

Python: <https://docs.python.org/2.7/tutorial/index.html>

Credits:

1. Arihant Raj Nagarajan(**A20334121**)
2. Rahul Krishnamurthy(**A20330185**)
3. Shashank Sharma(**A20330372**)