CloudKone Clone - Design Document

In this assignment, I have implemented a distributed task execution framework on Amazon EC2 using SQS. The framework is divided into two components, client which will read the tasks from a file and add them in SQS queue and worker who will fetch task from the queue and execute it. SQS provides requests handling across distributed workers.

Program Flow:

LOCAL (Command:* java Client -s <Queue Name> -w<Workload File> -t<no. of Threads>*)

The local experiment is implemented in t2.micro instance of Amazon by varying number of threads and workload files.

For local, I am creating two Java queues each for request and response and a thread pool using Java Executor API which will create fixed number of threads. Then I am reading all tasks from file in a queue and iterating the threads for workload size. Each thread will execute the tasks and add the status code in response queue.

Client:

1. Get command line arguments from user and parse them using apache command line API into queue name, no. of threads and workload file [1].
2. Create a Queue with the name specified
3. Read each line from the file and add into the queue along with a task id which in my case is a serial no.
4. Create a fixed size thread pool with the no. of threads passed in the arguments [2].
5. Iterate the Worker threads along number of tasks in the file.
6. After worker has finished off with the tasks, write the response queue to the file

Worker:

1. Fetch the task from the queue
2. Parse the string to get the required value (Time in milliseconds)
3. Execute the task for that particular time (Sleep Task)
4. Add Success or Failure code in the response queue

Remote (Command:* java Client -s <Queue Name> -w<Workload File> -t<no. of Threads>*)

The remote version is the implementation which will be executed on Amazon t2.micro instance. There are two separate classes Client and Worker. The Client will be running on one instance and separate Workers will be running on separate instances.

In this case, I am using Amazon SQS Queue for requests and responses. SQS queue is a distributed queue which can be used across multiple instances. Once the queue is created, with the help of queue URL, all send and receive requests can be made from any instances. Queue URL is fetch using queue name passed in command line arguments. SQS only guarantees at least one delivery of message because deleting all the copies of the message in the distributed queue will cost latency. Hence, I have used DynamoDB to avoid message duplicity. Before executing the task, each worker will check the table to see if there is an entry in it. If there is no entry in the table, it will put an entry in table, execute the task and delete the message from the request queue. Meanwhile, if another worker fetches the same message, it will not execute since an entry is present in the table. Once the task is executed, it will enter status code in a response queue which will be then written into a file by the client.

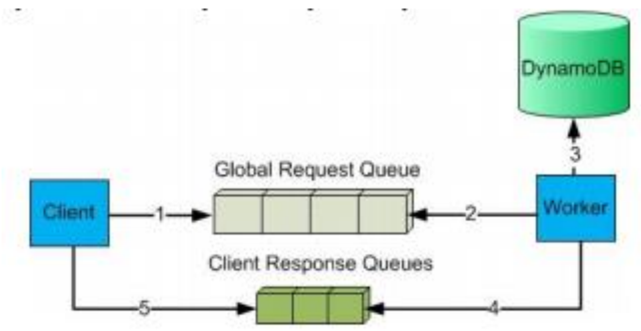
Client:

1. Get command line arguments from user and parse them using apache command line API into queue name, no. of threads and workload file.
2. Get AWS credentials from the credentials file.
3. Create an Amazon SQS Object passing credentials in the arguments and set its region
4. Create a queue with the name specified and get its URL
5. Read each line from the workload file and send as message in the queue along with a task id which in my case is a serial no.
6. Create a response queue and get its URL
7. Get the messages from the response queue and write it to the file
8. Iterate Step 7 till file size is not equal to number of tasks in workload file.
9. Purge both request and response queue

Worker:

1. Get command line arguments from user and parse them using apache command line API into queue name, no. of threads.
2. Get AWS credentials from the credentials file.
3. Create a table in DynamoDB with two columns of Key and Value
4. Create an Amazon SQS Object passing credentials in the arguments and set its region
5. Get the request and response queue URLs from the name specified in the argument
6. Fetch the task from the request queue
7. Parse the string to get the Time in milliseconds and task id
8. Check if the task id is present in the DynamoDB, if yes, delete the message from the request queue and if not, put the message in DynamoDB, execute the task and then delete it from the queue
9. Add Success or Failure code in the response queue
10. Iterate from Step 6 until the request queue is empty and response queue size is equal to original workload text file.

Basic Structure of Implementation [6]:



Environment Setup:

Software:

- Java version 8.0
- AWS SDK JAVA 1.10.74

Operating System: Ubuntu

System: Amazon AWS t2.micro instance, Simple Queue Service (SQS) and DynamoDB

Steps:

1. Install the “intall.sh” script in t2.micro instance
2. Configure the AWS Credentials (~/.aws/config) [3]:

   ```
   >> aws configure
   ```

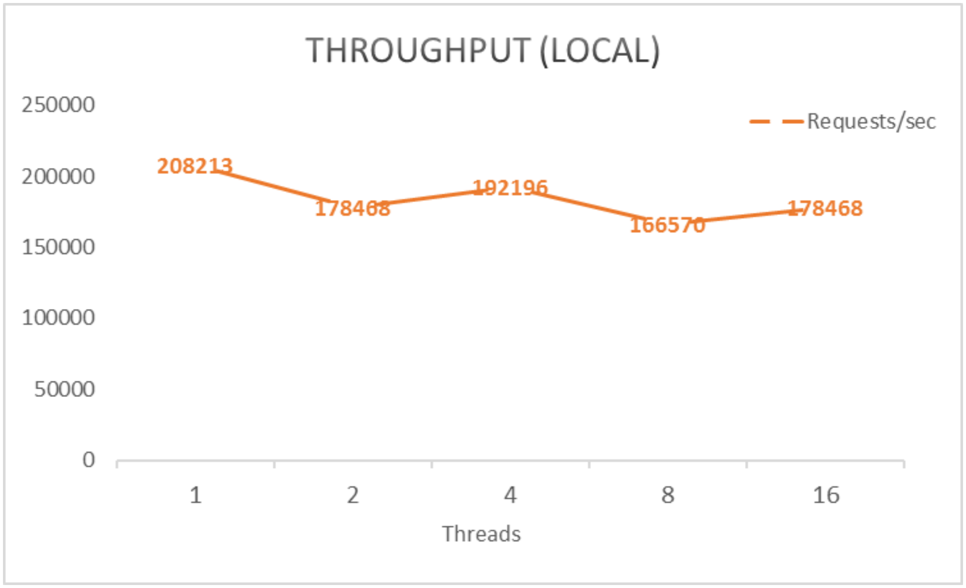
   Enter the KEY, SECRET ID, REGION and <BLANK>

3. Copy the source code in the instance
4. Take an AMI image of the current configuration.
5. Use this image to launch Amazon instances for Workers.

Performance Evaluation

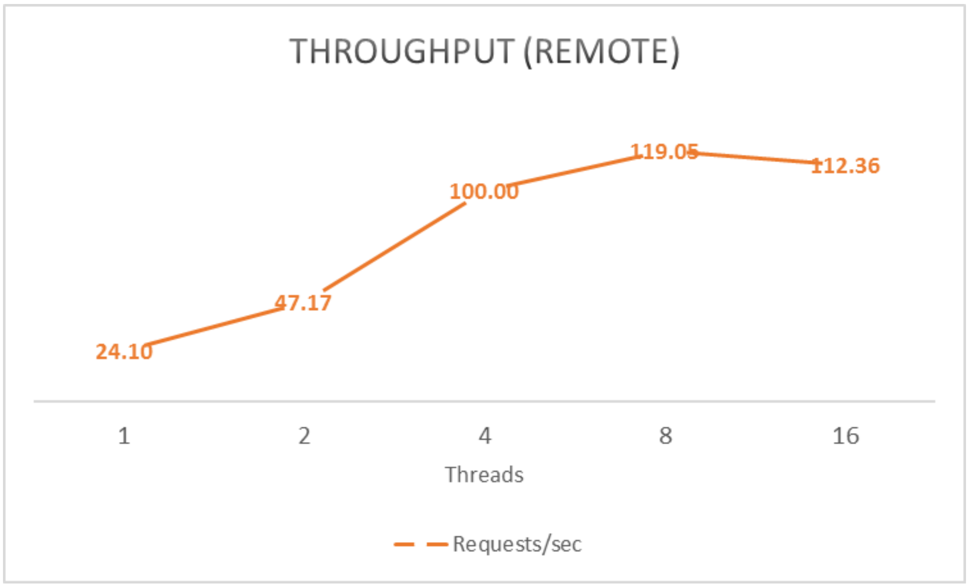
Performance of the implementation is done twice, one for throughput and one for efficiency. In all I have performed 20 experiments each for local and remote. For throughput, I have used “sleep 0” tasks and for efficiency I have used sleep task for 10 milliseconds, 1 sec and 10 seconds.

In case of local, to run the code for at least 10 seconds, I have used 15,000K “sleep 0” tasks and ran the code by varying number of threads. Below graph depicts the same. As we can see, the throughputs for single thread is more than rest of them since t2.micro instance is a single core processor and at a time 8 to 16 threads cannot run successfully. Also, as I am running the tasks in memory and all are “sleep 0” tasks throughput is in 200K requests/sec range.



Throughput (Local)	
Threads	Requests/sec
1	208213
2	178468
4	192196
8	166570
16	178468

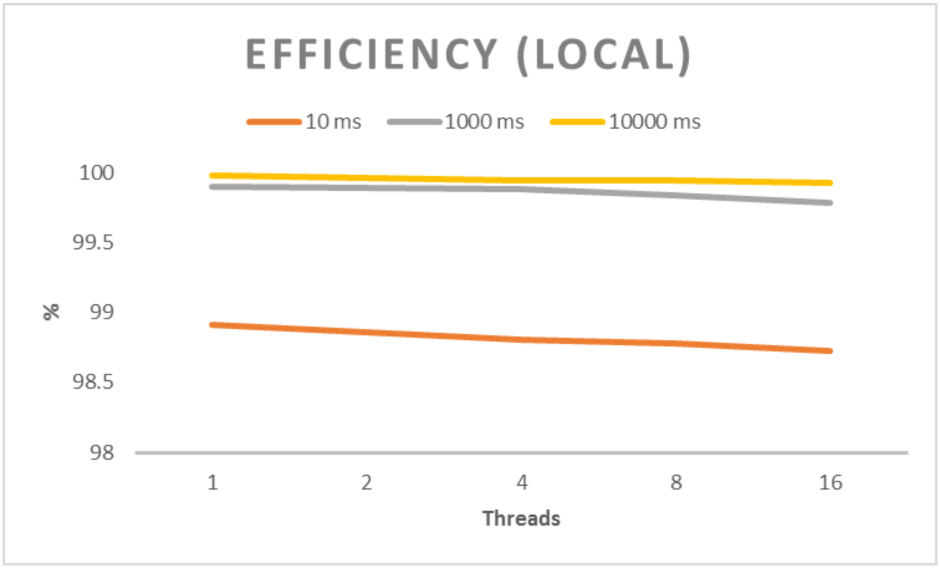
In case of remote, I have used only 10,000 “sleep 0” tasks and ran the code by varying the number of workers. As increasing number of workers are run on separate t2.micro instances, the throughput increases gradually and hence I found higher throughput for 8 and 16 threads.



Throughput (Remote)	
Threads	Requests/sec
1	24.10
2	47.17
4	100.00
8	119.05
16	112.36

For calculating the efficiency, I have performed 15 experiments for each local and remote. There are 1000 operations of “sleep 10”, 100 operations of “sleep 1000” and 10 operations of “sleep 10000”.

The below graph shows the efficiency for local performed on single t2.micro instance. As we can see, I am getting around 98-100% efficiency in all the cases since I am also increasing the aggregate workload size by increasing the number of threads. (e.g. 2 threads 2*1000, 4 threads 4*1000). In the throughput case, the workload was constant and huge hence, more amount of work was performed by multiple threads on a single core wherein in this case, the workload performing by a single thread on a single core is done by each thread. Also, in this case the workload is comparatively very small, therefore I am getting better efficiency.

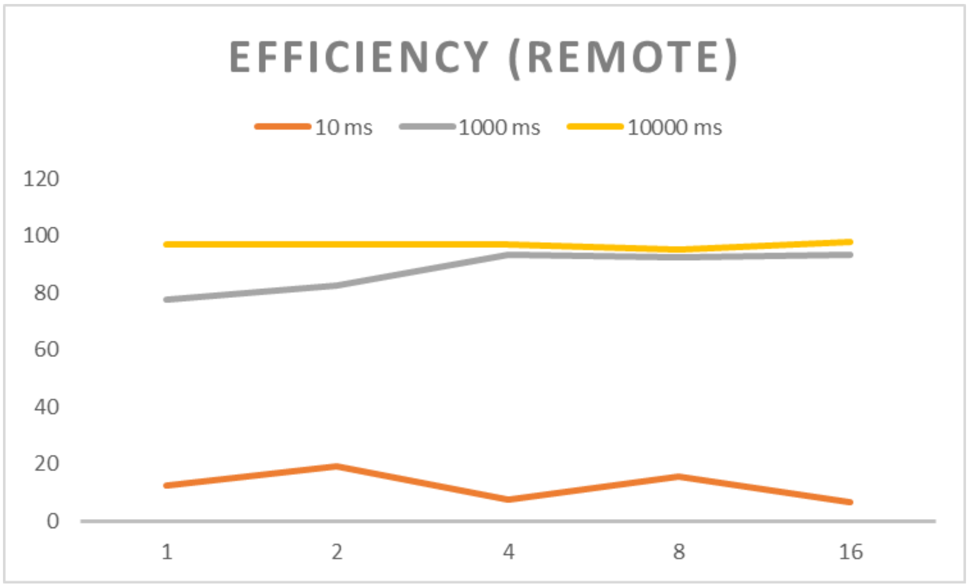


Efficiency in % (Local)			
Threads	10 ms	1000 ms	10000 ms
1	98.91	99.9	99.98
2	98.86	99.89	99.96
4	98.81	99.88	99.94
8	98.78	99.84	99.94
16	98.73	99.78	99.93

The below graph shows the efficiency for remote performed on multiple instances. As performed for local, I am performing the same 15 experiments with same workload files but on multiple instances.

From these experiments, I found very low efficiency for “sleep 10” tasks compared to other two “sleep 1000” and “sleep 10000” tasks. The reason is there are small jobs which take relatively less amount of time and workers have to wait for client to add messages to the queue. Wherein, in case of larger tasks, the number of jobs are less and till the time the workers execute old ones, new messages are waiting in the queue for them and hence much faster.

Also, to avoid Provisioned Throughput Exceeded Exception, I am gradually increasing the write capacity. Up to 2 threads, I am using 10L read and 20L write capacity, for 4 threads I increased the write capacity to 30L, for 8 threads I increased the write capacity to 40L and for 16 threads I used 50L read and 70L write capacity.



Efficiency in % (Remote)			
Threads	10 ms	1000 ms	10000 ms
1	12.5	77.52	97.09
2	19.23	82.64	97.09
4	7.46	93.46	97.09
8	15.38	92.59	95.24
16	6.67	93.46	98.04

Trade-offs:

One of the reasons, we have chosen Amazon SQS to implement this code is because its distributed. But the results for experiments we performed are not very efficient since it was on a small-scale. The same thing if we would have to implement on our system, we may have to use socket programming and the throughput and latency will not be efficient than SQS. I can say that since I have experimented a comparison between my distributed key/value storage systems and the ones which are boosting in the market.

Since the workload contains “sleep 0” & “sleep 10” tasks the actual latency was in fetching and inserting in DynamoDB rather than executing time and hence the throughput was not efficient in case of first two experiments. Which can be proved in the next two experiments which contained 1 and 10s sleep.

These were few trade-offs I faced, which can be overcome by varying the jobs performed.

Extra Credit: “Animoto”

In this experiment, I have created a video out of images and uploaded it to Amazon S3. I have taken URLs of some images from google.com and placed it in the workload file. Later the client will read the URLs, add it to the queue and the workers will fetch URLs and download images from the web. Each worker will then rename all the files (“rename.sh”) with a standard convention for convenience. Then it will call the “ffmpeg” tool to create a movie file. Once the file is created, worker will create a bucket and upload the file in S3. It will also fetch the URL of the S3 location of the file and add it into response queue.

References:

[1] https://commons.apache.org/proper/commons-cli/usage.html

[2] http://www.javatpoint.com/java-thread-pool

[3] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ProvisionedThroughput.html

[4] http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html

[5] http://docs.aws.amazon.com/AWSSdkDocsJava/latest/DeveloperGuide/java-dg-samples.html

[6] http://datasys.cs.iit.edu/publications/2014_CCGrid14_CloudKon.pdf

[7] http://docs.aws.amazon.com/AmazonS3/latest/dev/ShareObjectPreSignedURLJavaSDK.html

[8] http://www.ffmpeg.org/faq.html#HowdoIencodesinglepicturesintomovies_003f

Note:

Manual is attached as last part of the document.

Jars are downloaded online from the “install.sh” hence I am not including with the deliverables.

Screenshots are in a separate document called “Screenshots.pdf”

“rename.sh” is used to rename image names in Animoto.java

Manual

Steps to perform before running the code:

1. Please run "install.sh" first, which is a script to setup Java and AWS and add the required Jars.

   Since Jars are big in size, I have used "wget" in the above script to fetch online.

2. Configure AWS Credentials by giving the following command and Enter the required information when prompted,

   ```
   aws configure
   ```

Steps to run the code:

1. Complete the environment setup as mentioned above.

2. Create the 17 workload files for the experiments (3 files for each thread (15), Local Throughput file (15,000K ops), Remote throughput file (10,000 ops)).

3. Run the Client code for local throughput by varying number of threads (1,2,4,8 and 16)

```
Javac Client.java (in local folder)

Java Client -s LOCAL -w <WORKLOAD FILE> -t <THREADS>
```

4. Now, run the client code for local efficiency (10ms) with varying number of threads (1,2,4,8 and 16)

```
Java Client -s LOCAL -w <WORKLOAD FILE for Efficiency> -t 1

Java Client -s LOCAL -w <WORKLOAD FILE for Efficiency> -t 2

Java Client -s LOCAL -w <WORKLOAD FILE for Efficiency> -t 4

Java Client -s LOCAL -w <WORKLOAD FILE for Efficiency> -t 8

Java Client -s LOCAL -w <WORKLOAD FILE for Efficiency> -t 16
```

5. Repeat step 4 for other two experiments (1000ms and 10000ms)

6. Now for remote, start instances for workers

7. Run Client on original instance and Worker on worker instances simultaneously. Perform throughput experiment by varying no. of threads

```
Javac Client.java (in Remote folder)

Java Client -s <QNAME> -w<WORKLOAD FILE>


Javac Worker.java

Java Worker -s <QNAME> -t <THREADS>
```

<THREADS> will be 1 for remote experiments.

8. Run the above experiments for efficiency by changing the workload file