CS 553 Programming Assignment #3

CloudKon clone with Amazon EC2, S3, SQS, and DynamoDB

Instructions:

- ***Due date: 11:59PM on Wednesday, 04/20/16***
- ***Maximum Points: 100%***
- ***Maximum Extra Credit Points: 15%***
- *This programming assignment must be done individually.*
- *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#_LASTNAME_FIRSTNAME.{zip|tar|pdf}". E.g. "Prog3_Raicu_Ioan.tar".*
- *Late submission will be penalized at 10% per day (beyond the 7-day late pass).*

1. Overview

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

- Amazon Web Services, specifically the EC2 cloud (http://aws.amazon.com/ec2/), the SQS queuing service (http://aws.amazon.com/sqs/), S3 (http://aws.amazon.com/s3/), and DynamoDB (http://aws.amazon.com/dynamodb/).
- You will learn about distributed load balancing and how to distribute work between clients and workers

This assignment will involve implementing a distributed task execution framework on Amazon EC2 using the SQS. The assignment will be to implement a task execution framework (similar to CloudKon, http://datasys.cs.iit.edu/publications/2014_CCGrid14_CloudKon.pdf; you are highly encouraged to read the CloudKon paper to better understand the context of this assignment. You are to use EC2 to run your framework; the framework should be separated into two components, client (e.g. a command line tool which submits tasks to SQS) and workers (e.g. which retrieve tasks from SQS and executes them). The SQS service is used to handle the queue of requests to load balance across multiple workers.

This assignment can be done in any programming language (as well as libraries) you prefer.

2. The Task Execution Framework

2.1. The Client (15 points)

The client should be a command line tool, which can submit tasks to the SQS. You are free to choose the implementation language, as well as the communication protocol between the client and the SQS. Your client should follow this interface:

client –s QNAME -w <WORKLOAD_FILE>

The QNAME is the name of the SQS queue and the name of the DynamoDB instance; you should create these ahead of time of starting the client; if QNAME does not exist, the client should terminate. The WORKLOAD_FILE is the local file (for the client) that will store the tasks that need to be submitted to the SQS. Here is a sample workload file:

sleep 1000
sleep 10000
sleep 0
sleep 500

Each task is separated by a new line character. Each line is a task description. For example, "sleep 1000" denotes that the task is to invoke the sleep library call for 1000 milliseconds. Remember that the client is simply reading these task descriptions and send 'em to SQS. For simplicity, you can submit each task one by one. To be able to keep track of task completion, each task should be given a unique task ID at submission time.

Once all tasks had been sent, the client should wait for results (success or failure) to be received at the client, and every received task ID result should be matched with the corresponding submitted task ID. You may need to have separate SQS queues for sending tasks to workers and receiving results from workers.

When running the client, it is important that the client also run in the cloud, in a VM.

2.2. Local Back-End Workers (10 points)

Your framework will only support sleep tasks, and therefore you can in run a large number of sleep tasks on the same resource where the client runs, as the tasks are extremely light-weight and do not require significant amount of resources. You will implement a configurable size pool of threads that will process tasks from the submit queue, and when complete will put results on the result queue. The pool of threads can be extremely simple, and only have to handle the sleep functionality, of a specified length in milliseconds. When the sleep task is completed, a success (e.g. value 0) should be returned. In case there is a failure (e.g. an exception, out of memory, etc), a failure should be returned (e.g. value 1). Note that the communication between the client and the workers is via the in-memory queue (e.g. insert(task) and task=remove()), and there is no need for any network communication or Amazon SQS at this point. At the completion of this section, you

should have a running framework, with a client, and local workers, and you should be able to run sleep workloads (as defined in the evaluation Section 3).

These local workers should run on the same VM as the client. You should start the local workers by:

client –s LOCAL -t N -w <WORKLOAD_FILE>

Where LOCAL denotes that it is the local worker version, N denotes the number of threads in the thread pool, and determines the number of concurrent tasks to execute.

2.3. Remote Back-End Workers (20 points)

You need to implement some back-end workers (that functionally do the same thing as the local workers from Section 2.2), but have the ability to run on different machines to allow large amounts of computing to scale. There are multiple ways to do this part of the assignment; we will focus on leveraging Amazon AWS services to implement this part of the project. In order to allow a general purpose cloud-enabled solution, you must use the SQS service to communicate between the client and the back-end workers, and vice versa. Your system should assume that there is only 1 client, and N back-end workers, where N could be 0 to 16; you are encouraged to use spot instance types.

For simplicity, the worker should only receive 1 task at a time, and process 1 task at a time. Your system must allow multiple tasks to run concurrently at the same time (e.g. through a pool of threads), and allow completed tasks to be returned back to the client (via SQS) as soon as they are complete.

The interface to start workers should be:

worker –s QNAME –t N

It is important that the remote workers run on different VMs than the client and other workers.

2.4. Duplicate Tasks (10 points)

SQS does not guarantee that messages from SQS are delivered exactly once. You are to use DynamoDB to keep track of what messages you have seen, so that if a duplicate is retrieved, it can be discarded.

2.5. Animoto Clone (10 EC points)

The assignment will be to implement a web application (one that converts pictures to video, see http://www.ffmpeg.org/faq.html#How-do-I-encode-single-pictures-into-movies_003f), which will involve both significant network I/O and significant amounts of processing. You are to use everything you have build up to now, but instead of running sleep tasks, you are to run a real application. S3 should be used to store the final results of the computation (e.g. a video file).

You should first make sure you can find some images on the web. Google Images (https://www.google.com/imghp?hl=en&tab=ii) has many images you can select. Your

implementation will be simpler if you use pictures from the web, as your workload will then consist simply of URLs to the images (as opposed to dealing with file upload options in dynamic pages).

Once you have the list of image URLs, you can use tools such as wget (http://www.gnu.org/software/wget/) as a simple command line tool that will download the image(s) to a local directory.

Once you download your images to a local folder, ffmpeg (http://www.ffmpeg.org/) can be used to convert images. The page at http://www.ffmpeg.org/faq.html#How-do-I-encode-single-pictures-into-movies_003f has some examples of how this can be done.

Once the video has been created to the local disk, it should be written to S3, and the client should be notified of the location of the video.

3. Performance Evaluation

3.1. Throughput and Efficiency (30 points)

You are to perform a similar performance evaluation such as Figure 4 and Figure 10 from the CloudKon CCGrid 2014 paper http://datasys.cs.iit.edu/publications/2014_CCGrid14_CloudKon.pdf.

To determine maximum throughput, we measured performance running "sleep 0" under a statically provisioned system. You are to measure throughput of 10K tasks, where each task is "sleep 0". If this experiment takes less than 10 seconds for any particular run, increase the number of tasks to 100K. You are to benchmark the system by varying the number of workers from 1, 2, 4, 8, and 16; keep the client fixed at 1. Plot the throughput achieved (number of tasks processed divided by the total time from submission of the first task to the completion of the last task).

To measure efficiency, in addition to varying the number of workers from 1 to 16 (in powers of 2), you are to vary the sleep time, from 10 ms, 1 second, and 10 sec. The number of tasks should be 1000 per worker for 10 ms tasks, 100 per worker for 1 sec tasks, 10 per worker for the 10 sec tasks. As you increase the number of workers, the aggregate number of tasks will also increase. For example, at 16 workers and 1 second tasks, you will have 100*16 = 1600 tasks. However, at 2 workers and 10 second tasks, you would have 2*10 = 20 tasks. You can compute the ideal time to run these experiments assuming zero cost to communicate and distribute the tasks. Plot efficiency of different task lengths against the number of workers. Recall that efficiency is the ideal time divided by the measured time.

3.2. Animoto (5 EC points)

Using a fixed workload of 160 jobs, where each job is a list of 60 pictures (1920*1080 resolution). You want to generate a 60 second video clip in HD 1080P resolution and store it on S3. Perform a

scalability experiment where you run this workload on 1 node, 2 nodes, 4, 8, and 16 nodes, and plot the running time of the experiments. Please use the static provisioning mechanism for this step.

4. What you will submit?

When you have finished implementing the complete assignment as described in section 1, 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. **Source code:** All of the source code, scripts, etc; in order to get full credit for the source code, your code must have in-line documents, and must compile. Also include executables in a sub-folder **(name it "executables")** of source code folder.
2. **Report (please submit this in pdf format):**
   i. **Design (5 points):** This section should be 1~2 page describing the overall program design, and design tradeoffs considered and made.
   ii. **Manual (5 points):** 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). If there are any manual steps that must be taken, for example to edit some configuration file, please include these steps in your instructions.
   iii. **Performance Evaluation:** 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
3. A sourcecode.txt file that includes your entire code in text format.
4. **Screenshots for your experiments and Outputs.**

Please put all of the above into one .zip or .tar file, and upload it to 'digital drop box' on blackboard'. The name of .zip or .tar should follow this format: *PROG3_SURNAME_FIRSTNAME.{zip|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.

Grades for late programs will be lowered 10% per day late.