CS 553 Programming Assignment #4

Falkon/Animoto clone with Amazon EC2, S3, SQS, and DynamoDB

Instructions:

- ***Due date: 11:59PM on Monday, 12/01/14***
- ***Maximum Points including extra credit: 100%+40%EC***
- *You should work in teams of 3 for this assignment.*
- *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#_LASTNAME1_LASTNAME2_LASTNAME3.{zip|tar|pdf}". E.g. "Prog1_Raicu_Li_Wang.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 dynamic resource provisioning, and how applications can be made autonomic and adaptive

Sign up for an account on Amazon Web Services. We have secured sufficient funds for each student to use $100; instructions on how to use this credit will follow from the TAs. In the meantime, you can start with using free Micro Instances, to familiarize yourselves with the virtual machine images, and learn more about EC2, SQS, S3, and DynamoDB.

This assignment will involve implementing dynamic provisioning for a task execution framework on Amazon EC2 using the SQS. The assignment will be to implement a task execution framework (similar to Falkon, http://datasys.cs.iit.edu/publications/2007_SC07_Falkon.pdf; if you are adventurous, you could try out the Falkon system http://dev.globus.org/wiki/Incubator/Falkon, although the project has been retired and is not under active development); you are highly encouraged to read the Falkon paper to better understand the context of this assignment. You are to use EC2 to run your framework; the framework should be separated three components, one client (e.g. a command line tool), one front-end (e.g. the scheduler) and multiple back-ends (e.g. workers which communicate with the front-end to execute tasks). The SQS service is used to handle the queue of requests to load balance across multiple back-ends. You can use the SQS length to automatically increase or decrease the size of the allocated EC2 instances, essentially implementing the dynamic resource provisioning logic. The framework should dynamically scale as more work is submitted in order to keep the work round-trip-time short. Under idle conditions, the resources should be released.

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

2. The Task Execution Framework

2.1. The Client (10 points + 5 EC points)

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

client –s <IP_ADDRESS:PORT> -w <WORKLOAD_FILE>

The IP_ADDRESS:PORT are the location of the front-end scheduler, and should be known ahead of time of starting the client. The WORKLOAD_FILE is the local file (for the client) that will store the tasks that need to be submitted to the scheduler. 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 sending them to the front-end scheduler. For simplicity, you can submit each task one by one. ***You can optimize the submission with batching tasks together.*** 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. For simplicity, you can implement a polling interval from the client to the front-end scheduler that asks if any tasks have completed (since the last poll), and if yes, to retrieve the results of the tasks one by one. ***You can optimize the receipt of results with batching results together (5% extra credit). For additional extra credit, you can also implement a notification handler, to eliminate the need for polling.***

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

2.2. The Front-End Scheduler (10 points)

Your front-end scheduler should be as simple as possible. It should offer a network interface to the client, in order to allow tasks to be transferred from the client to the scheduler. The scheduler should follow the following interface:

scheduler –s <PORT> -lw <NUM> -rw

The PORT is the server port where the clients will connect to submit tasks and to retrieve results.

When tasks are received, they should be placed in an in-memory submit queue for later scheduling. You will implement two types of back-end workers to process tasks from this queue: 1) a Local Worker, and 2) a Back-End Worker.

The –lw switch denotes the local worker, along with the size (NUM) of pool of threads. The –rw switch denotes the remote worker. Note that the remote worker does not specify the number of workers, as that logic is implemented in a separate component; more on this in Section 2.5.

It is important to run the scheduler on a different VM than the client.

2.3. 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 front-end scheduler 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 scheduler 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, scheduler, 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 scheduler.

2.4. Remote Back-End Workers (20 points + 5 EC points)

You need to implement some back-end workers (that functionally do the same thing as the local workers from Section 2.3), 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 front-end scheduler and the back-end workers, and vice versa. Your system should assume that there is only 1 client, 1 front-end scheduler, and N back-end workers, where N could be 0 to 32 (you must use spot instances to get up to 32 instances; on-demand instance will be capped at 20 instances, and should not be used; you can use small instances for this experiment).

For simplicity, the worker should only receive 1 task at a time, and process 1 task at a time. ***For extra credit, allow multiple tasks to be retrieved at the same time, and 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 scheduler (via SQS) as soon as they are complete.***

Also, for simplicity, workers should keep track of the time they are idle without any task received or processing, and upon a threshold limit being reached, the workers should terminate the VM running the remote worker. This will have the effect that the number of workers will decrease under idle conditions. This decrease could also be done in a centralized manner from your dynamic provisioning component (this would be extra credit, see Section 2.5).

Remote workers can be statically provisioned (via manual intervention from user) for evaluation purpose. The interface to start workers should be:

worker –i <TIME_SEC>

The <TIME_SEC> denotes the idle time in seconds that the worker will stay active. An idle time of 0 means never terminate worker for idleness. Upon worker termination, also terminate VM.

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

2.5. 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.

2.6. Dynamic Provisioning of Workers (10 points)

You should implement a program whose sole job is to monitor SQS and decide whether to allocate EC2 instances. Under idle conditions (the SQS service is empty), 0 workers should be allocated. Workers should be started programmatically, based on the SQS length. For example, if the SQS has entries, it should have at least 1 worker. Once the worker is ready to process and start working, if the SQS length is stable or decreasing, there is no need for any change to the number of workers. If the SQS length is still increasing, additional workers should be added. Since workers will de-allocate themselves under idle conditions, the dynamic provisioning mechanisms does not need to explicitly de-allocated any EC2 instances. This logic should continue forever, to continuously increase the number of workers based on the SQS state.

2.7. Animoto Clone (20 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 (or perhaps it happens through a poll.

3. Performance Evaluation

3.1. Throughput and Efficiency (25 points)

You are to perform a similar performance evaluation per section 4.1 and 4.4 from the Falkon SC07 paper (http://datasys.cs.iit.edu/publications/2007_SC07_Falkon.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. 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). See Figure 3 in the Falkon SC07 paper for an example.

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 1 second, 2 sec, 4 sec, and 8 sec. The number of tasks should be 80 per worker for 1 sec tasks, 40 per worker for 2 sec tasks, 20 per worker for 4 sec tasks and 10 per worker for 8 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 80*16 = 1280 tasks. However, at 2 workers and 8 second tasks, you would have 10*2 = 20 tasks. All these experiments should take 80 seconds ideally to complete, but in reality it will likely take longer (because of various overheads). Plot efficiency of different task lengths against the number of workers (see Figure 6 in the Falkon SC07 paper for an example). Recall that efficiency is the ideal time divided by the measured time.

3.2. Dynamic Provisioning (5 points)

You are to measure how responsive your dynamic provisioner is when additional work is placed on SQS. Set the minimum number of workers to 0 and the maximum number of workers to 1, and measure the latency of running a 1 second task. Repeat the experiment 5 times and present the average latency. Now configure your provisioner to statically allocate 1 worker. Measure again the latency of running the 1 second task, and compare it to the dynamic provisioner approach.

3.3. Animoto (10 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.
2. **Report:**
   i. **Design (5%):** This section should be 1~2 page describing the overall program design, and design tradeoffs considered and made.
   ii. **Manual (5%):** 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.

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: *PROG4_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.