Falkon/Animoto clone with Amazon Cloud Computing Services

Hina Garg

Rojin Babayan

Vivek Pabani

Illinois Institute of Technology

Abstract

The project aims to build a system similar to the Falkon framework with help of different amazon services. The system allows submission of multiple tasks to be performed by required number of workers. The jobs are performed in parallel to increase the throughput. We have also built a clone of Animoto with help of the built system. It performs multiple tasks of generating videos out of images in parallel. The system is scalable in terms of workers since the workers are created based on the pool of tasks.

Contents

1. Introduction

Many interesting computations can be expressed conveniently as data-driven tasks, in which individual tasks wait for input to be available, perform computation, and produce output. In master-slave model many tasks can be logically executed at once if they are dispatched in parallel to a group of workers by batch schedulers who manage these workers and notify client after their successful execution. One of the problems with such kind of implementation is that task dispatched by the schedulers maybe far more then the workers available to process them while sometimes workers available might be quite more than the required workers to complete small number of jobs. Therefore, a dynamic provisioner should be used to acquire/release of resources as per demand. We implemented one such system which works with varying number of workers as per the requirement. In addition, we also preserve uniqueness in the jobs executed by the workers by adding Dynamo DB which is a fast and flexible NoSQL database service of AWS.

2. System Architecture

2.1. Execution Model

The tasks are dispatched by the client to the front-end scheduler in batches. Once all tasks are sent, it waits for the responses from the scheduler. The client terminates upon receiving "last batch" message from the scheduler.
The scheduler receives all the tasks in batch and store locally. Once all tasks are received, it submits the tasks to the global request queue. After tasks are submitted, it starts polling for the responses from the global response queue. It waits till all the responses are received, and then sends back to the client in batches.
The workers pick the tasks from the global request queue one by one. They check in the dynamodb for the duplicate tasks, and perform accordingly. The responses are sent back to the global response queue.

2.2. Architecture

The system architecture consists of mainly a client, a front-end scheduler and zero or more local/remote back-end workers. There is one duplicate tasks handler, which keeps track of performed tasks. The dynamic provisioning module manages the number of active remote-workers based on the pending tasks in request queue.

2.2.1. The Client

The client is a program, which submits the task data to the system. It initiates the entire process by submitting the tasks to front-end scheduler. A local work-file contains all the tasks in predefined format separated by a new-line character. The client reads all the tasks from the file, generates a distinct task id for each task, and sends to the front-end scheduler in batches. Once all the tasks are submitted, it waits for the response messages from the scheduler. When all the responses are received, it compares the submitted and received tasks, and provides details of remaining tasks, if any. For animoto tasks, it provides extra information of the URL of the final video.

2.2.2. The Front-End Scheduler

The front-end scheduler offers a network interface to the client in order to allow client to send task data. It runs in two modes - local and remote, and works with the related worker. The server keeps running forever in one of the two mentioned modes, and waits for the client request. Once client starts sending the tasks, it stores the tasks locally till it receives "last-batch" message from the client. Once all the tasks are received, either the local workers are triggered, or the tasks are submitted to global request as per the working mode. After all tasks are submitted, the scheduler waits for the responses and start polling to the response queue. the responses are stored locally, and dispatched to the client in batches.

2.2.3. Local Back-End Worker

The local back-end workers stay on the same instance as the front-end scheduler. When the scheduler runs in local mode, as soon as the tasks are received, it places the tasks in an in-memory request queue for later scheduling. After all the tasks have been added to the queue, local workers start fetching the tasks from the queue and executing them in parallel as multiprocessing is applied in the program. The number of local workers (worker processes) are defined while starting the front-end scheduler. The results of the executed tasks are placed in an in memory response queue, which in turn are picked by the front-end scheduler.

2.2.4. Remote Back-End Worker

The remote back-end workers need to pop a task from the request queue, run it and store its result back to the response queue. Tasks and responses are stored in JSON formats. The response JSON can either indicate a success or failure of the task. Moreover, for Animoto tasks, the response contains the URL of the result video as well.

- **Multi-threading support**

  Our remote worker for "sleep" tasks has the capability of running multi-threaded. It pops a group of N number of tasks from the request queue (by default it is set to 5), runs them each in a separate thread parallel and concurrently, store the response for each of them in the response queue.

- **Python-Java Interprocess Communication**

  Our Animoto task runner is implemented in Python, while the worker itself is implemented using Java. We took advantage of socket programming in order to pass data between our Python and Java codes.

- **Removal of the Tasks from the Queue**

  When a task is popped out of the request queue by a worker, the worker will remove it from the queue. This will prevent other workers to pop repeated tasks as well as the dynamic provisioning module to take care of the number of workers.

2.2.5. Duplicate Tasks Handler

In our design, each task has a unique-id. It is the responsibility of workers not to run a task that is already executed (possibly by some other workers). In order for that to happen, when a worker wants to start executing a task, it puts the task-id into a table in DynamoDB to declare others

that the task is executed. Accordingly, every worker before executing a task, checks the table in DynamoDB to make sure that none else has executed (or has intended to execute) the task before. Figure 1 is the screenshot of the table.

Task_id
54.68.255.188_10124_141764524610
54.68.255.188_10124_141764540082
54.68.255.188_10124_141764524885
54.68.255.188_10124_141764524993
54.68.255.188_10124_141764540074
54.68.255.188_10124_14176452462
54.68.255.188_10124_141764524646
54.68.255.188_10124_141764524645

Figure 1: *Duplicate Tasks Handler : DynamoDB table for executed task-id values.*

2.2.6. Dynamic Provisioning

The Dynamic Provision (DP) module makes our whole system elastic in term of number of active workers. The number of active workers is dependent to the rate of incoming tasks in the queue. The more tasks come into the request queue, the more number of virtual machines for the workers will be launched by DP module. Besides, if a worker stays idle for a specific time (timeout) with no running task or incoming task in the queue, it will terminate itself. As a result, our system is able to scale up and down according to the demand.

In order to save the IDs of the running virtual machines (workers), we benefit from a table in DynamoDB. The table (shown in Fig 2) has two columns; the Instance ID and the timeout. When the DP module launches a new instance, after the instance is completely started, the DP module will store its ID in the DynamoDB table as well as a value for the timeout. When the worker's code is running on the worker virtual machine, before any thing it will fetch the "timeout" value from the DynamoDB. The "timeout" value is the number of attempts the worker makes to fetch tasks from the queue with no success before it terminates itslef.

Figure 2: *Dynamic Provisioning : DynamoDB table for timeout values.*

When starting the system for the first time, there is not any worker in the system. The DP module keeps monitoring the request queue and the moment it sees a task in the queue, it will launch the first virtual machine as the first worker. Afterwards, DP module continues to monitor

the queue to evaluate the rate of incoming tasks and calculate the density of tasks as :

Density = Number of tasks in queue / Number of active workers

If density becomes more than a threshold (by default the threshold is set to 50), needed number of virtual machines will be launched. Our DP module does not need to take care of the terminating instances, because when an instance decides to terminate itself, it removes its name from the DynamoDB table of running instances.

2.2.7. Animoto Clone

Animoto is a cloud-based video creation service that produces video from photos, video clips, and music into video slideshows. We have used our built application to perform a similar functionality, which can act as clone of Animoto. The Animoto clone uses all the components of the system described so far. The client submits a list of image URLs as a task to the front-end scheduler. The front-end scheduler adds the task to the global request queue, which in turn gets picked by one of the idle worker. Upon receiving the animoto task, the worker calls the animoto program on the list of URLs. The animoto program downloads the images from the URLs and generates a video out of the images. It uses services of FFmpeg to create a video. FFmpeg is a free software project that produces libraries and programs for handling multimedia data. The video is created with 1 frame/image per second. The generated video is then stored to an Amazon S3 bucket. The URL of the video is submitted back to the response queue, and the front-end scheduler delivers it to the client.

3. Performance Evaluation

3.1. Throughput and Efficiency

The performance is evaluated in terms of throughput and efficiency. For throughput, we submitted tasks of "sleep 0", which measure the time of performing any tasks. The task "sleep 0" does not have any overhead of itself. The efficiency is measured by "sleep 1" to "sleep 8" tasks. All these tasks are performed with 1 to 16 workers.

3.1.1. Local Back-End Workers

For throughput, we performed 10k and 100k "sleep 0" tasks with 1 to 16 threads. As per the figure 1, the difference in performance with increasing number of threads is small. We believe this is due the fact that network performance dominates the entire execution. The time to send and receive 100k messages does not depend on the number of threads, and that results in similar throughput for 1 and 16 threads.

For efficiency, we performed "sleep 1" to "sleep 8" tasks. All tasks were performed with 1 to 16 threads. The length of tasks per worker is kept "80 seconds". The number of tasks per worker varies with task type. As per the figure 2, the difference in performance with increasing number of threads is small. We believe this is due the fact that the size of problem is small, and there is some overhead of creating the threads.

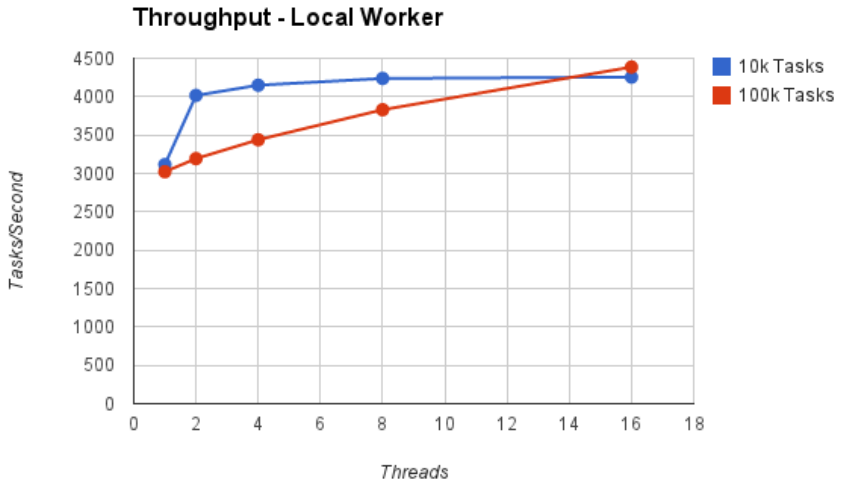

Figure 3: *Local Back-End Workers : Throughput with 10k and 100k tasks.*

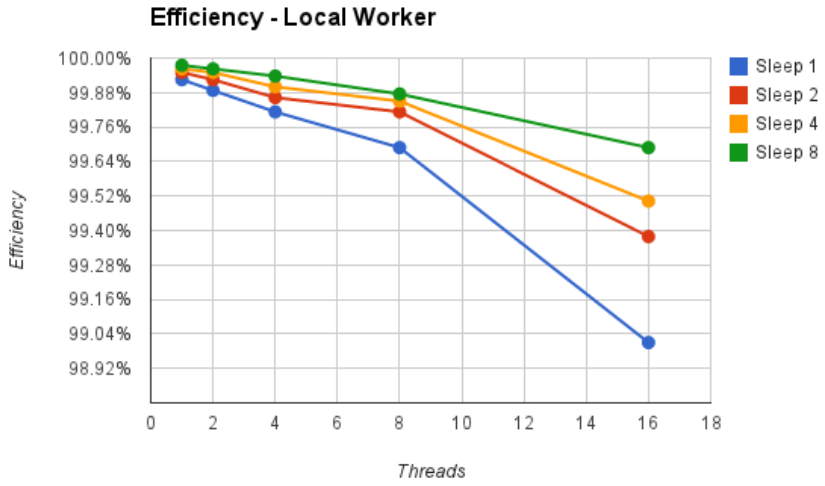

Figure 4: *Local Back-End Workers : Efficiency with different size of sleep tasks.*

3.1.2. Remote Back-End Workers

For throughput, we performed 10k "sleep 0" tasks with 1 to 16 workers. All the workers are created as static workers. As per the figure 1, the difference in performance with increasing number of workers is small. Also, the difference is not in accordance with the increased number of threads. The throughput is much lower than that of the local worker. We believe this is due two main reasons : submitting the tasks to SQS request queue and fetching the responses from SQS response

queue is done by one centralized scheduler, which dominates the entire execution time. Also, the dynamodb part implemented for duplicate avoidance takes more time as the number of tasks and number of workers increase, which affect the final execution time.

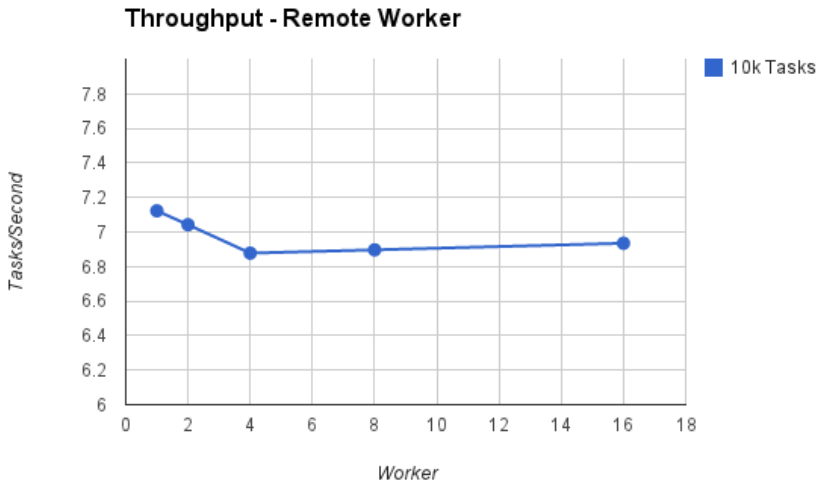

Figure 5: *Remote Back-End Workers : Throughput with 10k tasks.*

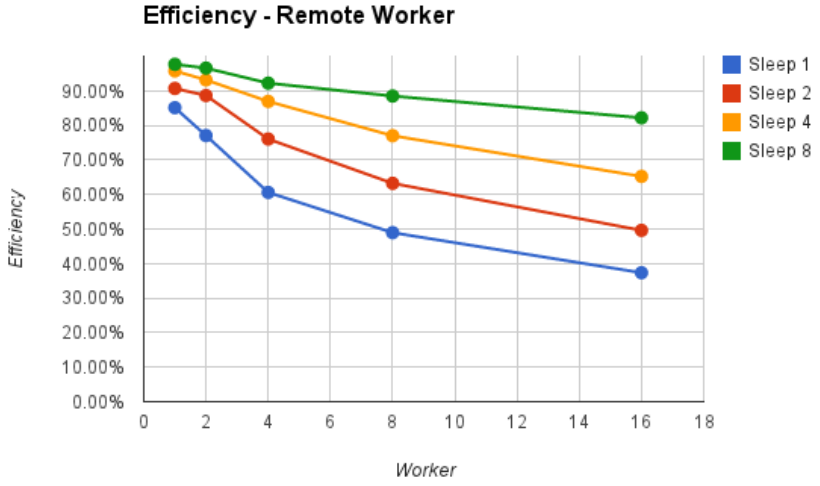

Figure 6: *Remote Back-End Workers : Efficiency with different size of sleep tasks.*

For efficiency, we performed "sleep 1" to "sleep 8" tasks. All tasks were performed with 1 to 16 threads. The length of tasks per worker is kept "80 seconds". The number of tasks per worker varies with task type. As per the figure 2, the difference in performance with increasing number

of threads is small. We believe this is due the fact that the size of problem is small, and there is some overhead of creating the threads.

3.2. Dynamic Provisioning

In this section we describe two experiments we conducted to measure the responsiveness of the dynamic provisioning module when a new task request comes. We measured the latency of the Dynamic Provisioning module in two experiments: dynamic allocation of the VM worker and static allocation of the VM worker.

3.2.1. Dynamic Allocation

In this experiment the Dynamic Provisioning module is set to allocate a maximum of 1 virtual machine as worker if a task comes to the queue. Otherwise, no virtual machine is needed. This made the Dynamic Provisioning module to allocate a module after a task comes to the queue. Resulting the added overhead of launching time of the virtual machine to the latency of doing the task. We repeated the experiment for 5 times. Table-1 illustrates the results:

Table 1: *Latency of running a "sleep 1" task with dynamic allocation of workers*

Test	Time(s)
1	142.545
2	158.113
3	146.972
4	121.589
5	133.092
Average	**140.462**

3.2.2. Static Allocation

In this experiment we set the Dynamic Provision module to always have at least one worker. Therefore, when a new tasks comes into the queue, it can immediately be popped by the worker without any additional launching time overhead. The results after repeating for 5 times are presented in Table-2.

Table 2: *Latency of running a "sleep 1" task with static allocation of workers*

Test	Time(s)
1	2.221
2	2.319
3	1.981
4	2.191
5	2.019
Average	**2.146**

The dynamic allocation and de-allocation of workers in the system provides elasticity in the system that can help very much to cost-efficiency and resource utilization. However, this adds

an unwanted overhead to the running time in the system because when a new worker wants to be added, it takes time for it to launch and become ready for running tasks. The results show that this unwanted overhead (instance launching time) is about 2 to 2.5 minutes. Moreover, another interesting observation in this experiment is that the system itself even when the worker is idle and ready, has an overhead. This overhead probably involves working with the queue and network. However this overhead is very low, especially if we increase the size of the task, it will be inconsiderable.

3.3. Animoto

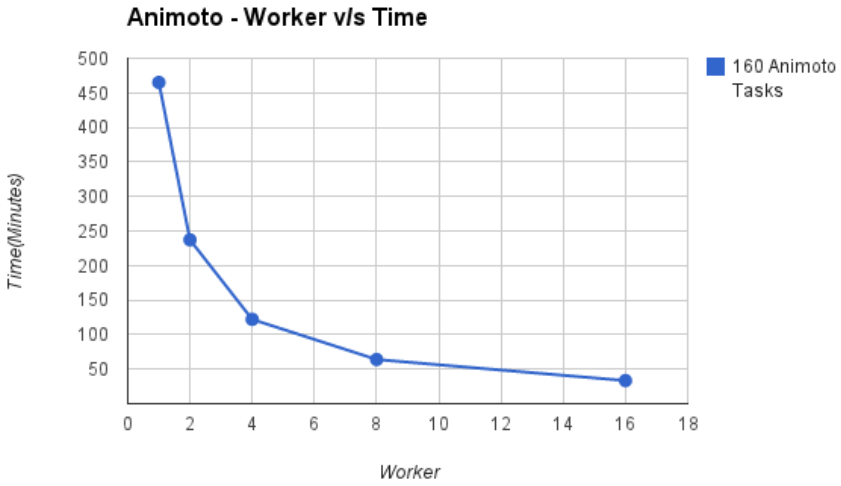

Figure 7: *A*

One animoto task is to create a 60 seconds long video clip with HD 1080P resolution out of 60 images. All the images are of 1920x1080 resolution. To make the task simple, we stored all the images under on S3 bucket, and those URLs were passed with the task-data. We performed 160 such tasks with 1 to 16 workers. The time decreases with increasing number of workers as expected. The time of downloading the images and creating a video dominates the execution. The overhead of dynamodb and performance of image-downloading deviates the graph from the normal by some margin.

4. Conclusion

The aim of the project is fulfilled since it replicates the main functions of the Falkon architecture. At present, the system works for one client, but it can be modified with minor changes to support multiple clients in the front-end scheduler design. The dynamic provisioning of the workers makes the system effective in terms of resource utilization. Since this is a framework, it be integrated with different type of tasks like Animoto to be performed in parallel.

5. Extra Credit

The following parts/functionalities are implemented for extra credit.

- The client sends the tasks and receives the results in batches.
- The remote workers fetch multiple tasks and run in parallel with multi threading.
- Animoto clone is implemented with help of the build system as part of extra credit task, and performance is evaluated.

6. Contribution

1. **Hina Garg** [A20342375] [hgarg@hawk.iit.edu]
    - Design and implementation of the Front-End Scheduler.
    - Design and implementation of the Local Back-End Workers.
    - Performance analysis of all modules.
2. **Rojin Babayan** [A20334414] [rbabayan@hawk.iit.edu]
    - Design and implementation of the Remote Back-End Workers.
    - Design and implementation of the Duplicate Tasks Handler.
    - Design, implementation and performance analysis of the Dynamic Provisioning.
3. **Vivek Pabani** [A20332117] [vpabani@hawk.iit.edu]
    - Design and implementation of the Client.
    - Design and implementation of the Animoto Clone.
    - Workloaf generation for performance analysis of all modules.
    - Performance analysis of all modules.