

CS553 : Programming Assignment #3

Report

Hariprasad Ravi Kumar

A20348609

1. Design

The goal of this programming assignment is to enable us to gain experience with the Amazon Web Services such as EC2 Cloud, SQS queuing service and DynamoDB. The primary aim of this assignment is to implement a static task provisionner. Our framework is divided into three parts: a client who submits tasks, the front end scheduler which handles the jobs distribution and scheduling and eventually the workers which runs the tasks either locally or remotely.

We are to use both SQS and DynamoDB in order to handle the replication of jobs in the scheduler part of this project. Finally we will measure the performance of our framework for throughput and efficiency according to the number of workers and the duration of the sleep tasks of the workers. Our project is implemented using java language and helped with the AWS APIs.

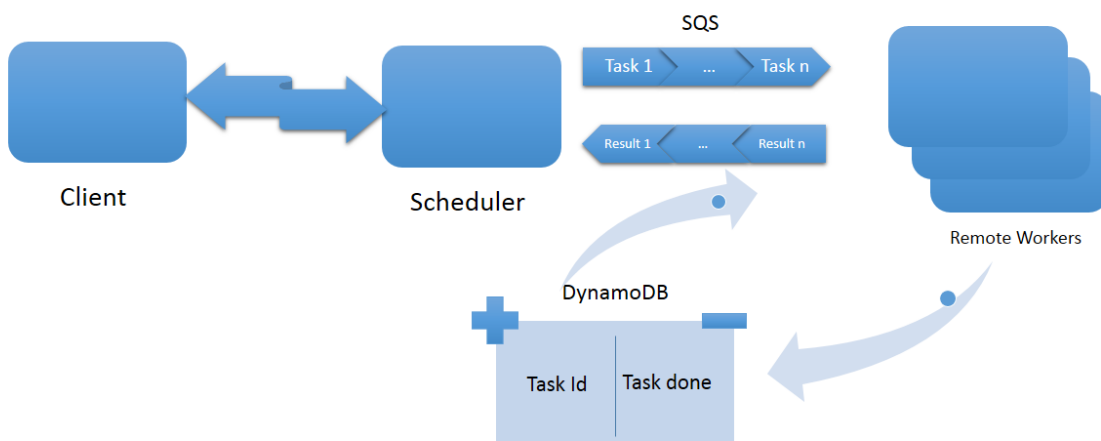


Figure 1: Framework layout

1.1 Components and their functionalities

This project focuses on developing a task execution framework that supports static scheduling. This framework is mainly composed of

- a client,
- a frontend,
- a number of local workers,
- a number of remote workers,
- a static provisioner.

The remaining contents of this section describe the functionalities of each component.

a. Client has three functionalities:

- reading from the workload file and generating the corresponding tasks
- sending all tasks to the frontend through a socket
- spawning another thread to serve a socket, in order to receive the result of each task execution from the frontend. Notice this thread starts at the same time as sending the tasks.

b. Frontend has three functionalities:

- serving a socket to receive the tasks from the client
- sending all tasks to the local workers through a implicit memory queue, or
- sending all tasks to the remote workers through a SQS queue
- spawning another thread to check task completion by polling from the implicit memory queue or SQS queue, and send all the completed tasks back to the client through a socket. Notice this thread starts at the same time as sending the tasks.

c. Local workers are in charge of:

- reading the task description from the received task
- spawn an OS process to execute the description, and check if it succeed
- return the task with the result record

d. Remote workers are in charge of:

- poll the task from the SQS queue
- check the dynamoDB to see if the task is duplicated. If it is duplicated, return to previous step
- reading the task description from the received task
- spawn an OS process to execute the description, and check if it succeed
- send the task back to another SQS queue (Note: the whole procedure is in a while loop.)

e. Static provisioner is in charge of:

- increasing or decreasing the number of worker according to the amount of incoming tasks statically in order to improve efficiency of the system.

1.2 Detailed Implementation of some components

- What language and tools we use for this project?

We use Java language and Eclipse IDE with AWS Toolkit.

- How do we generate the workload file?

We write a program to generate the workload file, corresponding to each benchmark request.

- How is each task represented?

Each task is encapsulated into a serializable object, which contains three attributes: task ID, task description, and result.

When the client generates each task, the task ID is a random string, the task description is the contents read from the workload file, and the result is set to “null”.

The result will be rewrite by either local or remote task.

- How does the frontend scheduler detect local worker’s completion?

Each local worker thread is encapsulated into a callable class.

An ExecutorCompletionService object is declared at front-end to submit each callable thread. Then, the thread which is in charge of send the result back to client only need to call the take() method of this ExecutorCompletionService object, to get a completed task object.

It is guaranteed that this take() method will return not only a completed task, but also the first completed task so far.

- How does the frontend detect remote worker's completion?

Since the front-end and remote worker communicate through SQS queue, the front end periodically spawn a new thread to poll from the SQS queue, and send them back to client.

However, sometimes the queue may be empty when the front-end poll it. In order to keep the stream between the client and front-end alive, the front-end will send a "fake task" instead.

- How does the remote worker terminate itself?

We have a line commented out during our experiments for convenience, which is:

```
(new ProcessBuilder("shutdown", "-h", "now")).start();
```

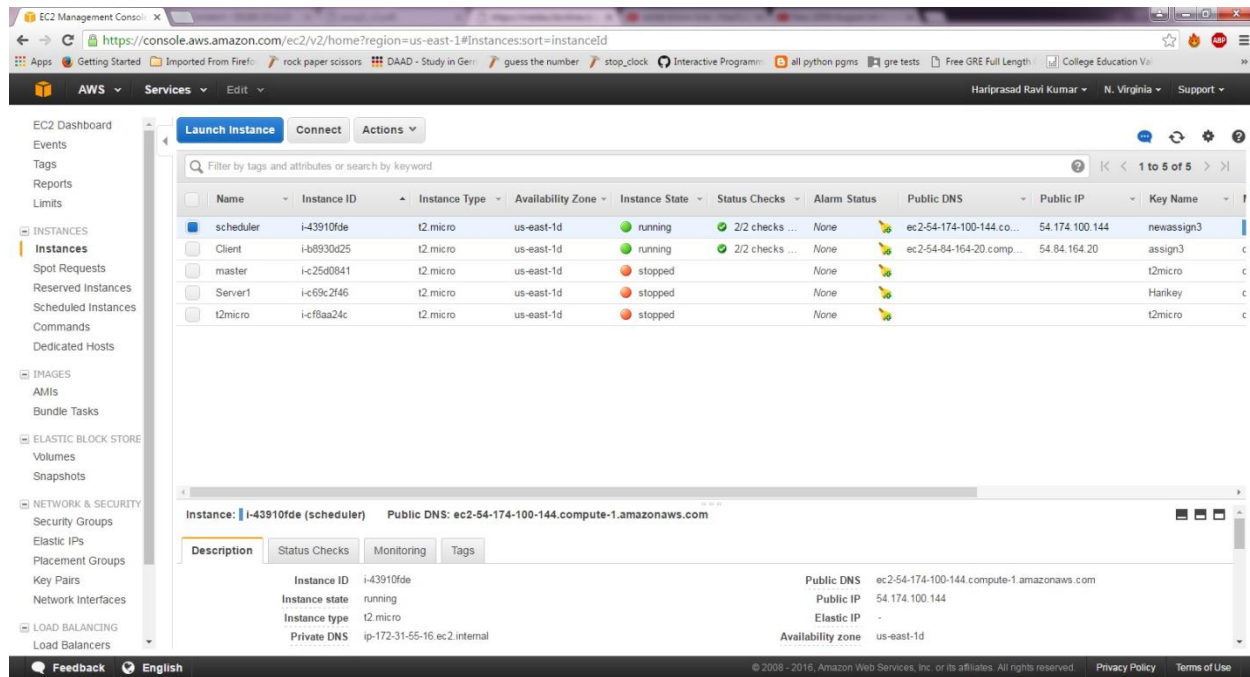
after a duration of idle time (fetch a message from task queue every half-second, until a non-empty message is obtained).

2. Manual

Client:

- Client runs on t2.micro instances of EC2.
- Client reads and takes up the local file and sent it to the server.
- Local file contains the list of task to be executed by server.
- Client runs on cloud instance using the interface, `client -s QNAME -w <WORKLOAD_FILE>`
- The QNAME is the name of the SQS queue and the name of the DynamoDB instance
- Workload file is the client's local file.
- Each task was separated by a new line and given a unique id for its identification.
- The client reads a file from the local and sends it to the server task by task.
- Client is run using the command `"java -cp pgm3.jar Client <<Server Name>>"`
- Server name is the name of server to which data is sent to. Localhost is given to say that the server is in the current machine.
- Port number specifies the port on which the server is listening to.
- A hash map is implemented in order to associate a task with ID.

Client Instance:



The screenshot shows the AWS Management Console for the EC2 service. The left sidebar contains navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES, SPOT REQUESTS, RESERVED INSTANCES, SCHEDULED INSTANCES, COMMANDS, DEDICATED HOSTS, IMAGES, AMIs, Bundle Tasks, ELASTIC BLOCK STORE, Volumes, Snapshots, NETWORK & SECURITY, Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces, and LOAD BALANCING, Load Balancers. The main content area displays a list of instances. The 'scheduler' instance is selected, and its details are shown in the 'Description' tab.

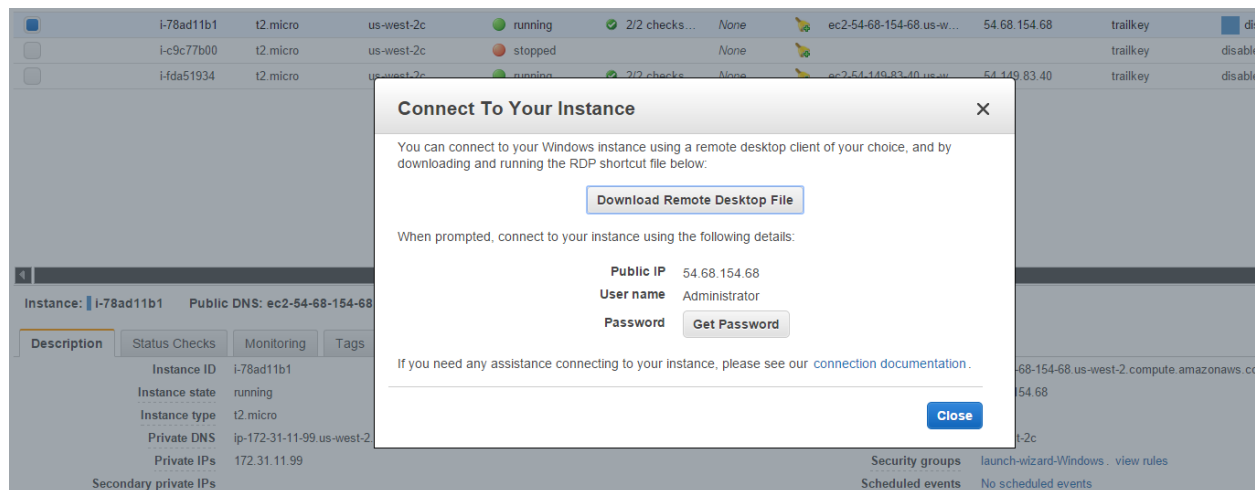
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS	Public IP	Key Name
scheduler	i-43910fde	t2.micro	us-east-1d	running	2/2 checks ...	None	ec2-54-174-100-144.compute-1.amazonaws.com	54.174.100.144	newassign3
Client	i-b8930d25	t2.micro	us-east-1d	running	2/2 checks ...	None	ec2-54-84-164-20.compute-1.amazonaws.com	54.84.164.20	assign3
master	i-c25d0841	t2.micro	us-east-1d	stopped	2/2 checks ...	None	ec2-54-84-164-20.compute-1.amazonaws.com	54.84.164.20	t2micro
Server1	i-c69c2f46	t2.micro	us-east-1d	stopped	2/2 checks ...	None	ec2-54-84-164-20.compute-1.amazonaws.com	54.84.164.20	Harkey
t2micro	i-cf8aa24c	t2.micro	us-east-1d	stopped	2/2 checks ...	None	ec2-54-84-164-20.compute-1.amazonaws.com	54.84.164.20	t2micro

Instance: **i-43910fde (scheduler)** Public DNS: **ec2-54-174-100-144.compute-1.amazonaws.com**

Description | Status Checks | Monitoring | Tags

Instance ID: i-43910fde
Instance state: running
Instance type: t2.micro
Private DNS: ip-172-31-55-16.ec2.internal
Public DNS: ec2-54-174-100-144.compute-1.amazonaws.com
Public IP: 54.174.100.144
Elastic IP: -
Availability zone: us-east-1d

Connecting Client Instance to Remote desktop:



The screenshot shows the AWS Management Console for the EC2 service. The 'i-78ad11b1' instance is selected, and its details are shown in the 'Description' tab. A 'Connect To Your Instance' dialog box is open, providing instructions on how to connect to a Windows instance using a remote desktop client.

Instance: **i-78ad11b1** Public DNS: **ec2-54-68-154-68.us-west-2.compute.amazonaws.com**

Description | Status Checks | Monitoring | Tags

Instance ID: i-78ad11b1
Instance state: running
Instance type: t2.micro
Private DNS: ip-172-31-11-99.us-west-2.compute.amazonaws.com
Private IPs: 172.31.11.99

Connect To Your Instance

You can connect to your Windows instance using a remote desktop client of your choice, and by downloading and running the RDP shortcut file below.

[Download Remote Desktop File](#)

When prompted, connect to your instance using the following details:

Public IP: 54.68.154.68
User name: Administrator
Password: [Get Password](#)

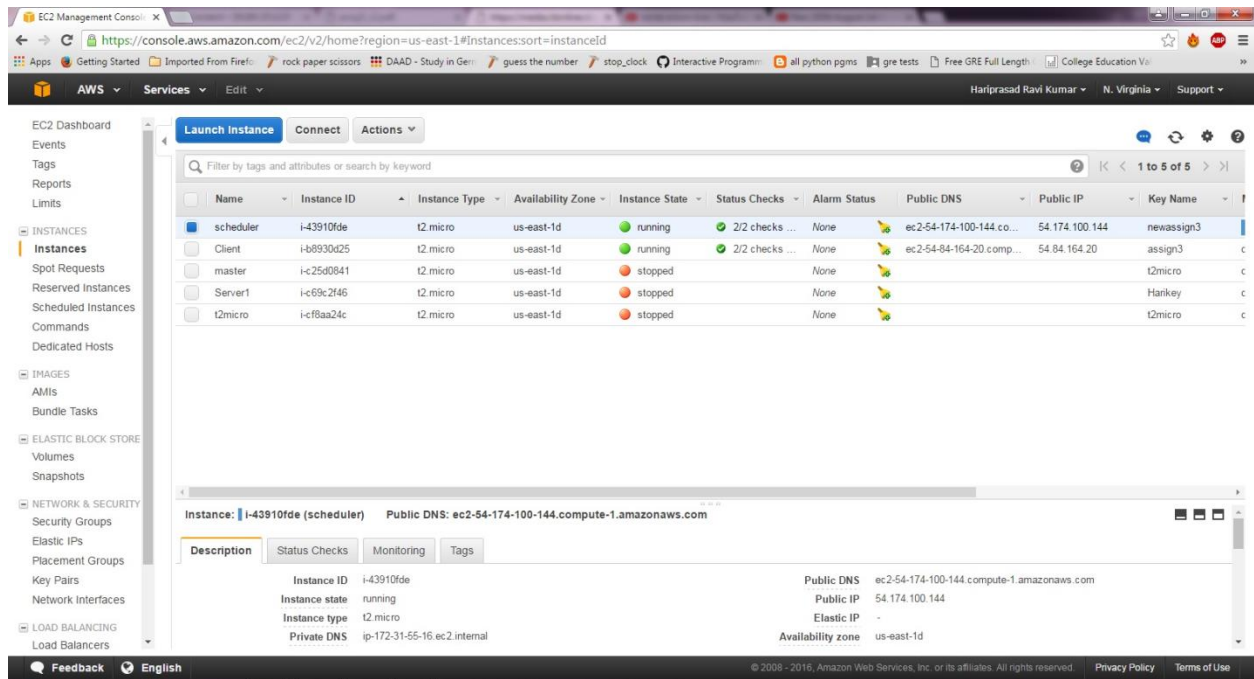
If you need any assistance connecting to your instance, please see our [connection documentation](#).

[Close](#)

Server:

- Server runs on t2.micro instances of EC2.
- Server listens on its specified port.
- The tasks arrive the server one by one or in batches
- Server runs with the interface, scheduler `-s QNAME -lw <NUM> -rw`
- `lw` usage specifies the local worker. Number of threads that it uses can also be specified.
- If `rw` switch is used, it uses the remote worker.
- Free thread picks a task and executes it.
- The tasks are either sent in batches or one-by-one.
- If the '`lw`' switch is used, the tasks are written to a in-memory queue which is implemented using a linked list.
- If the '`rw`' switch is used, the tasks are written to the SQS on AWS. The queue is called the '`taskQueue`'
- If the switch is '`lw`', then number of threads can be specified.
- A thread executor service is run which pops and item from the queue and submits it to a thread pool.
- The thread which is free takes up the task and executes it.

Scheduler Instance:



The screenshot shows the AWS Management Console for EC2 instances. The left sidebar contains navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES, Spot Requests, Reserved Instances, Scheduled Instances, Commands, Dedicated Hosts, IMAGES, AMIs, Bundle Tasks, ELASTIC BLOCK STORE, Volumes, Snapshots, NETWORK & SECURITY, Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces, and LOAD BALANCING, Load Balancers.

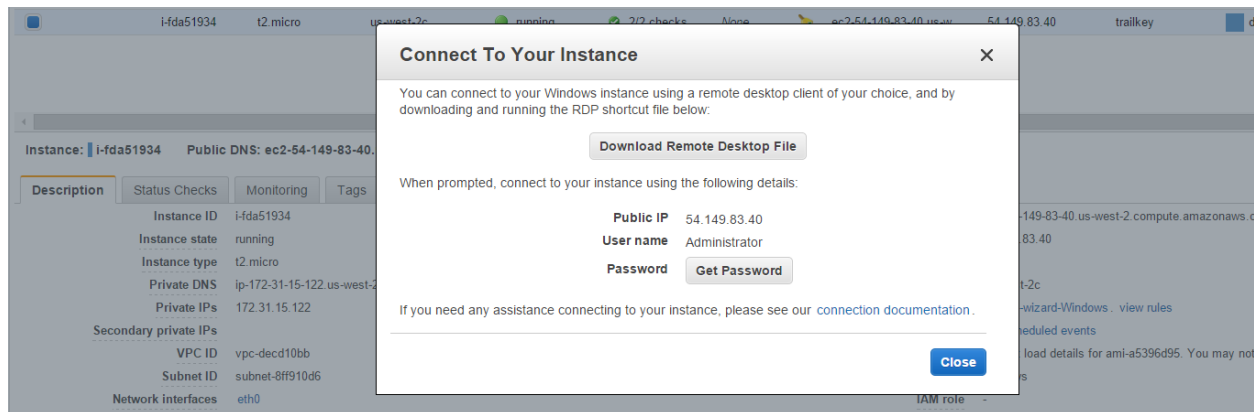
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS	Public IP	Key Name
scheduler	i-43910fde	t2.micro	us-east-1d	running	2/2 checks ...	None	ec2-54-174-100-144.co...	54.174.100.144	newassign3
Client	i-b8930d25	t2.micro	us-east-1d	running	2/2 checks ...	None	ec2-54-84-164-20.comp...	54.84.164.20	assign3
master	i-c25d0841	t2.micro	us-east-1d	stopped		None			t2micro
Server1	i-c69c2f46	t2.micro	us-east-1d	stopped		None			Harikay
t2micro	i-cf8aa24c	t2.micro	us-east-1d	stopped		None			t2micro

Instance: **i-43910fde (scheduler)** Public DNS: **ec2-54-174-100-144.compute-1.amazonaws.com**

Description | Status Checks | Monitoring | Tags

Instance ID	i-43910fde	Public DNS	ec2-54-174-100-144.compute-1.amazonaws.com
Instance state	running	Public IP	54.174.100.144
Instance type	t2.micro	Elastic IP	-
Private DNS	ip-172-31-55-16.ec2.internal	Availability zone	us-east-1d

Connecting to Remote desktop:



The screenshot shows the AWS Management Console for EC2 instances. The left sidebar contains navigation links for EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES, Spot Requests, Reserved Instances, Scheduled Instances, Commands, Dedicated Hosts, IMAGES, AMIs, Bundle Tasks, ELASTIC BLOCK STORE, Volumes, Snapshots, NETWORK & SECURITY, Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces, and LOAD BALANCING, Load Balancers.

Instance: **i-fda51934** Public DNS: **ec2-54-149-83-40**

Description | Status Checks | Monitoring | Tags

Instance ID: i-fda51934
Instance state: running
Instance type: t2.micro
Private DNS: ip-172-31-15-122.us-west-2...
Private IPs: 172.31.15.122
Secondary private IPs: -
VPC ID: vpc-decd10bb
Subnet ID: subnet-8ff910d6
Network interfaces: eth0

Connect To Your Instance

You can connect to your Windows instance using a remote desktop client of your choice, and by downloading and running the RDP shortcut file below:

Download Remote Desktop File

When prompted, connect to your instance using the following details:

Public IP	54.149.83.40
User name	Administrator
Password	Get Password

If you need any assistance connecting to your instance, please see our [connection documentation](#).

Close

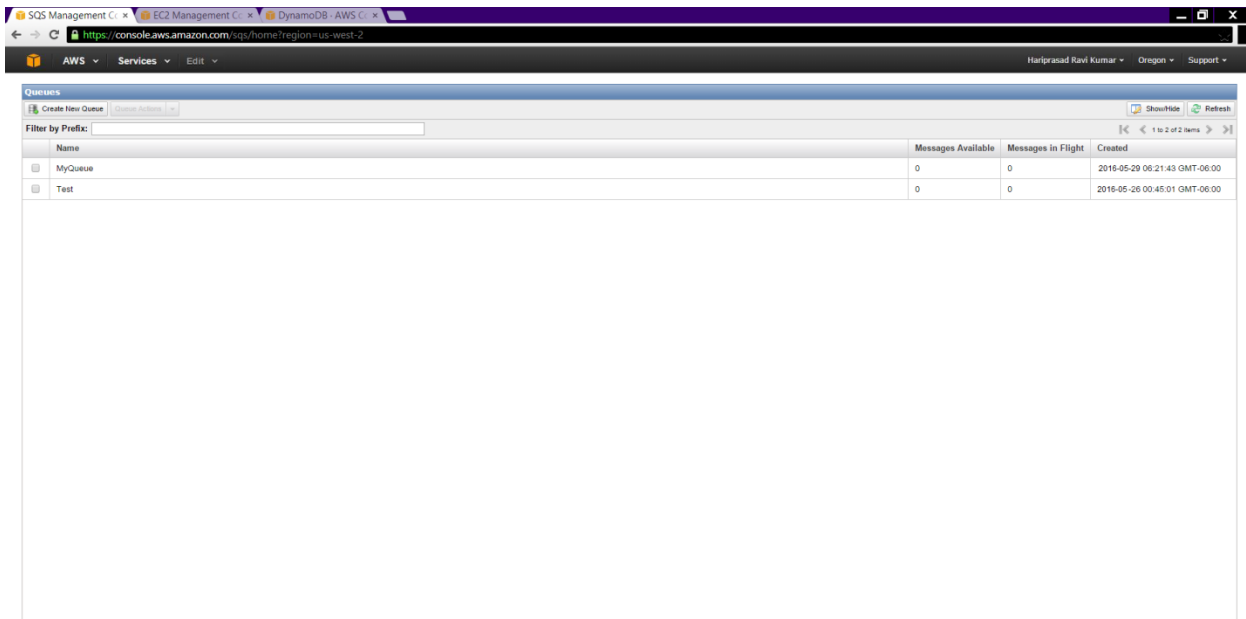
Local Backend Workers:

- The number of threads specified in the scheduler defines the number of local workers.
- A local worker method will take the task provided and sleeps for the specified amount of time.
- As the thread completes running the task, the result is written to another queue. If the thread fails an exception is raised and the return value is written into the result queue.
- Local Worker runs on t2.micro instances of EC2. It uses in-memory queue for storing the task and its results. The number of local workers is specified by the server.
- Local Worker picks up the task and sleeps for the specified amount of time. When the thread completes the task results are written into another queue and success is returned.
- When an exception occurs during thread execution, failure is returned.

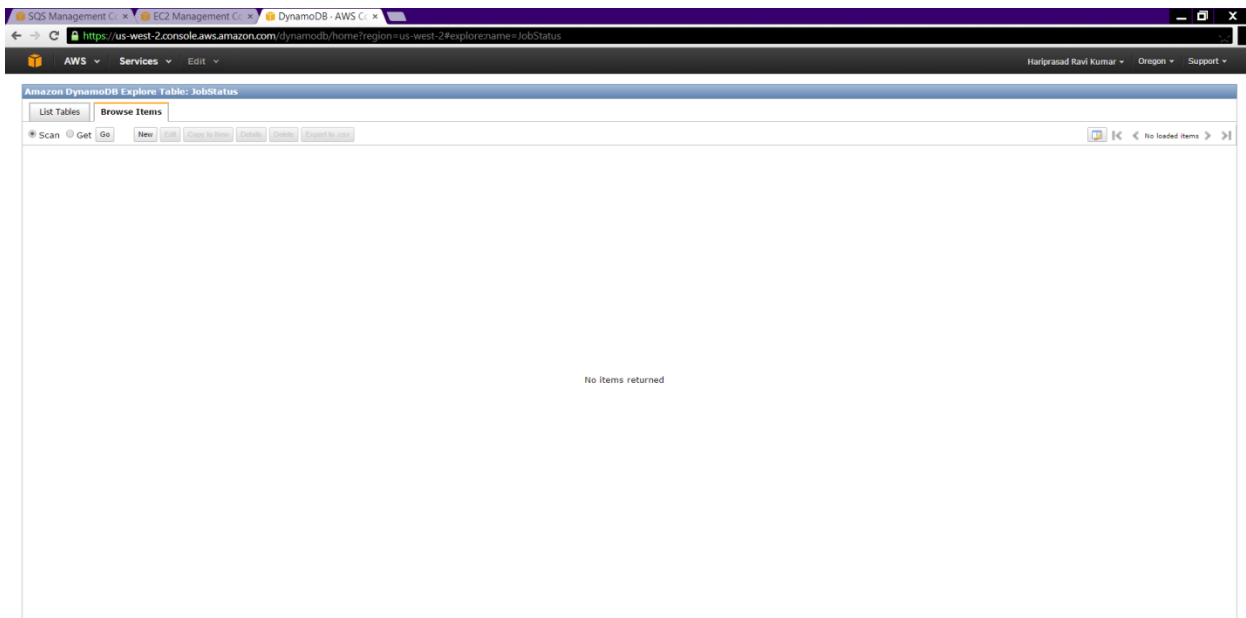
Remote Backend Workers:

- Remote Backend Workers runs on t2.micro instances of EC2.
- If rw switch is used then the tasks are written into SQS queue on AWS. The remote worker polls the queue at regular interval.
- If there exists a task it is picked up and executed.
- If there is no task, it is polled after specific time. If the queue is idle for long time, the worker is terminated along with VM. When the 'rw' switch is used, the tasks are written into SQS queue on AWS.
- A custom AMI is created holding the jar file and a crontab file. The crontab file runs the command on the system boot. The remote worker polls the SQS queue. If there is a task, it is retrieved and executed.
- If there are no tasks, it polls after a few minutes. If it is idle for the long time, the remote worker is terminated.

SQS



Dynamo DB:



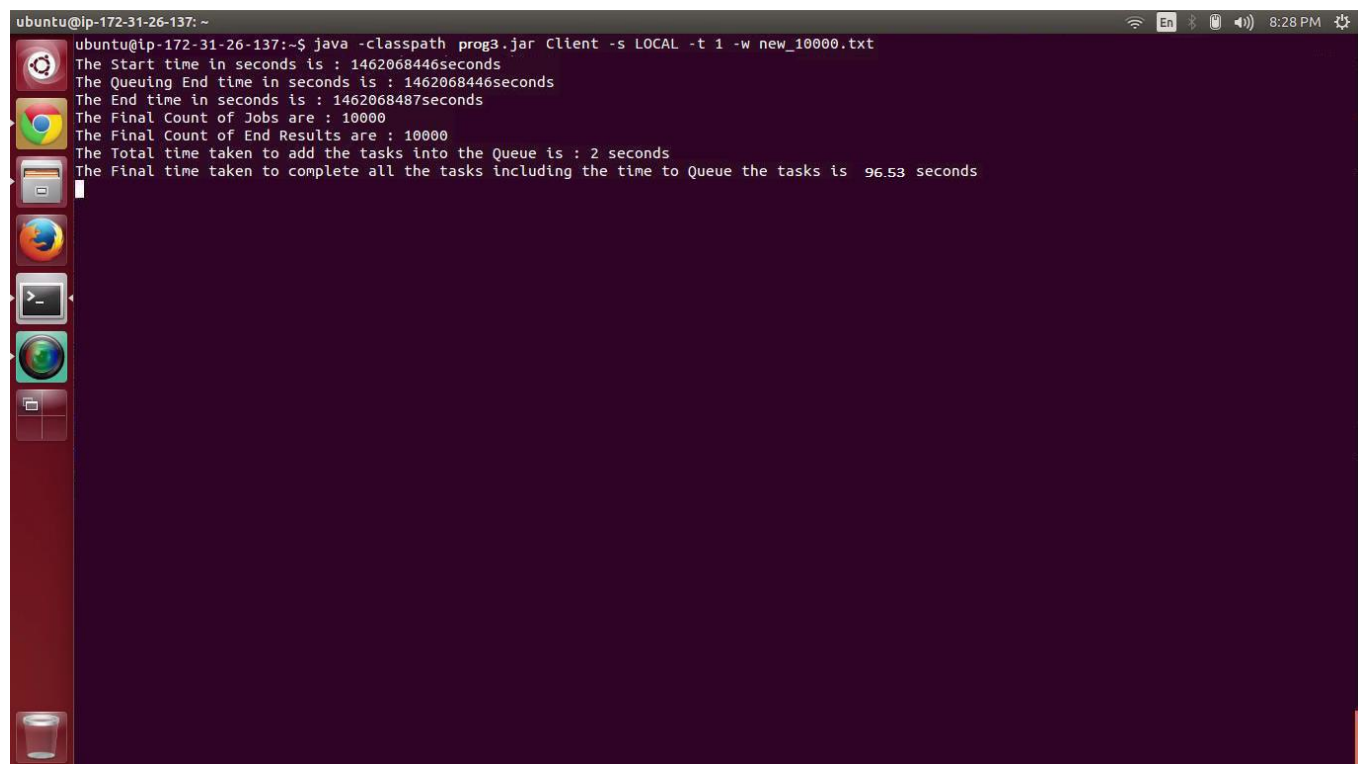
3. Performance Evaluation

3.1 Throughput Evaluation

The throughput is calculated by $10000 / \text{execution_time}$. For each experiment, a workload file contains 10k “sleep 0” is passed to the client. The client records the time first task object is sent to front-end, and the time it received the result of the last task.

Local:

1 worker



```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s LOCAL -t 1 -w new_10000.txt  
The Start time in seconds is : 1462068446seconds  
The Queuing End time in seconds is : 1462068446seconds  
The End time in seconds is : 1462068487seconds  
The Final Count of Jobs are : 10000  
The Final Count of End Results are : 10000  
The Total time taken to add the tasks into the Queue is : 2 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 96.53 seconds
```

2 workers

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s LOCAL -t 2 -w new_10000.txt  
The Start time in seconds is : 1462068597seconds  
The Queuing End time in seconds is : 1462068598seconds  
The End time in seconds is : 1462068619seconds  
The Final Count of Jobs are : 10000  
The Final Count of End Results are : 10000  
The Total time taken to add the tasks into the Queue is : 2 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 51.26 seconds
```

4 workers

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s LOCAL -t 4 -w new_10000.txt  
The Start time in seconds is : 1462068653seconds  
The Queuing End time in seconds is : 1462068653seconds  
The End time in seconds is : 1462068664seconds  
The Final Count of Jobs are : 10000  
The Final Count of End Results are : 10000  
The Total time taken to add the tasks into the Queue is : 2 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 39.98 seconds
```

8 workers

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s LOCAL -t 8 -w new_10000.txt  
The Start time in seconds is : 1462068693seconds  
The Queuing End time in seconds is : 1462068693seconds  
The End time in seconds is : 1462068702seconds  
The Final Count of Jobs are : 10000  
The Final Count of End Results are : 10000  
The Total time taken to add the tasks into the Queue is : 2 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 33.37 seconds
```

16 workers

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s LOCAL -t 16 -w new_10000.txt  
The Start time in seconds is : 1462068758seconds  
The Queuing End time in seconds is : 1462068758seconds  
The End time in seconds is : 1462068767seconds  
The Final Count of Jobs are : 10000  
The Final Count of End Results are : 10000  
The Total time taken to add the tasks into the Queue is : 2 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 33.34 seconds
```

Remote:

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s REMOTE -t 1 -w new_100.txt  
The Final Count of Jobs are : 100  
The Final Count of End Results : 100  
The Total time taken to add the tasks into the Queue is : 3 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 406.50 seconds  
|
```

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s REMOTE -t 2 -w new_200.txt  
The Final Count of Jobs are : 200  
The Final Count of End Results : 200  
The Total time taken to add the tasks into the Queue is : 5 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 222.22 seconds  
|
```

```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s REMOTE -t 4 -w new_400.txt  
The Final Count of Jobs are : 400  
The Final Count of End Results : 400  
The Total time taken to add the tasks into the Queue is : 6 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 148.25 seconds
```

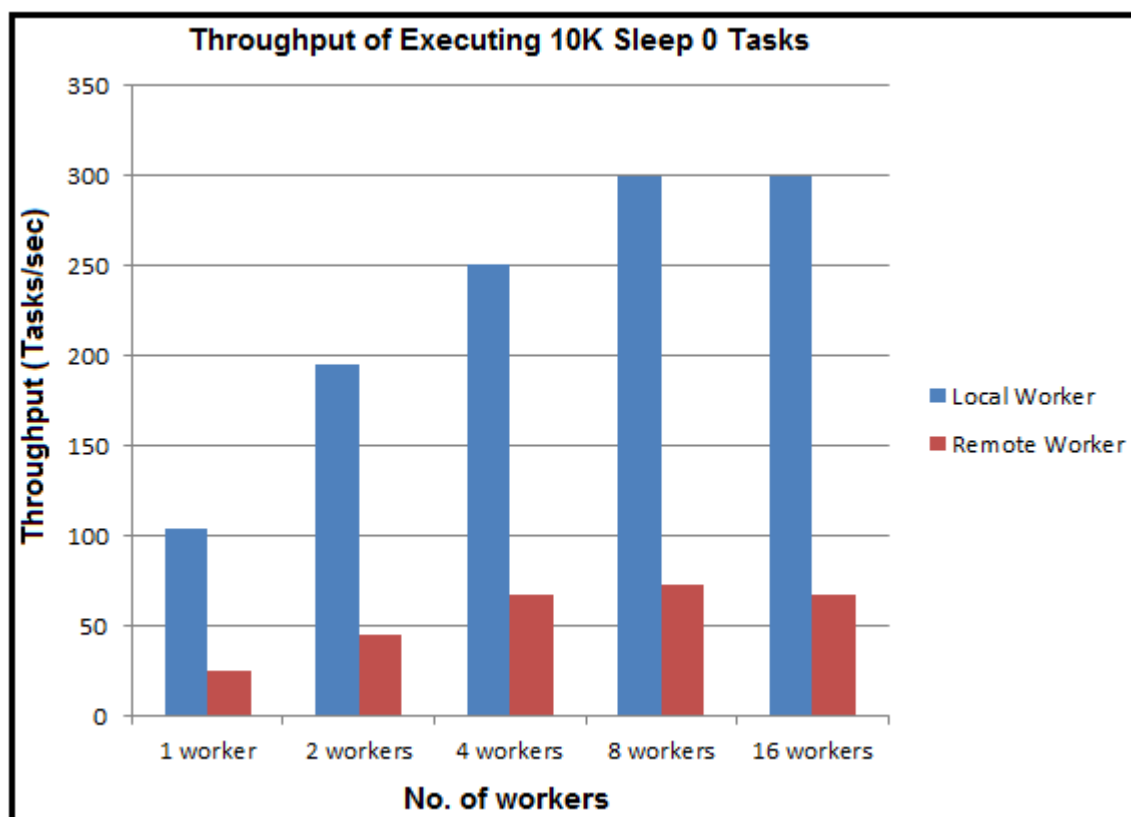
```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s REMOTE -t 8 -w new_800.txt  
The Final Count of Jobs are : 800  
The Final Count of End Results : 800  
The Total time taken to add the tasks into the Queue is : 6 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 138.46 seconds
```



```
ubuntu@ip-172-31-26-137: ~  
ubuntu@ip-172-31-26-137:~$ java -classpath prog3.jar Client -s REMOTE -t 16 -w new_1600.txt  
The Final Count of Jobs are : 1600  
The Final Count of End Results : 1600  
The Total time taken to add the tasks into the Queue is : 8 seconds  
The Final time taken to complete all the tasks including the time to Queue the tasks is 149.47 seconds
```

Throughput:

	1 worker	2 workers	4 workers	8 workers	16 workers
Local Worker	103.59	195.07	250.12	299.59	299.89
Remote Worker	24.6	45	67.45	72.22	66.9

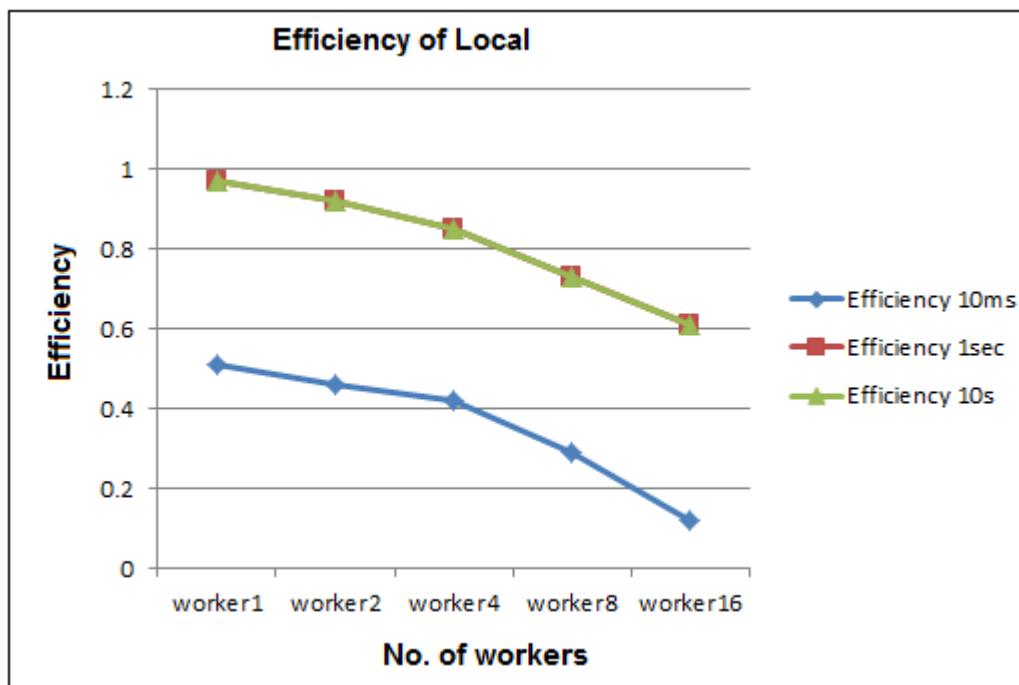


Both local worker and remote worker scale from 1-worker to 2-worker ideally. Their throughput improvements brought by more parallelism both stop working at 8-worker setting. For remote worker experiments, by observing dynamics of queue length at AWS SQS monitor, we find that starting at 8-worker setting, our frontend can no longer keep up with the pace the multiple workers are generating; e.g., 0 task in task queue, but more than a thousand results in result queue. Hence, the parallelism capacity of AWS services is impressive, and the throughput in our system is bottlenecked at frontend.

3.2 Efficiency Evaluation

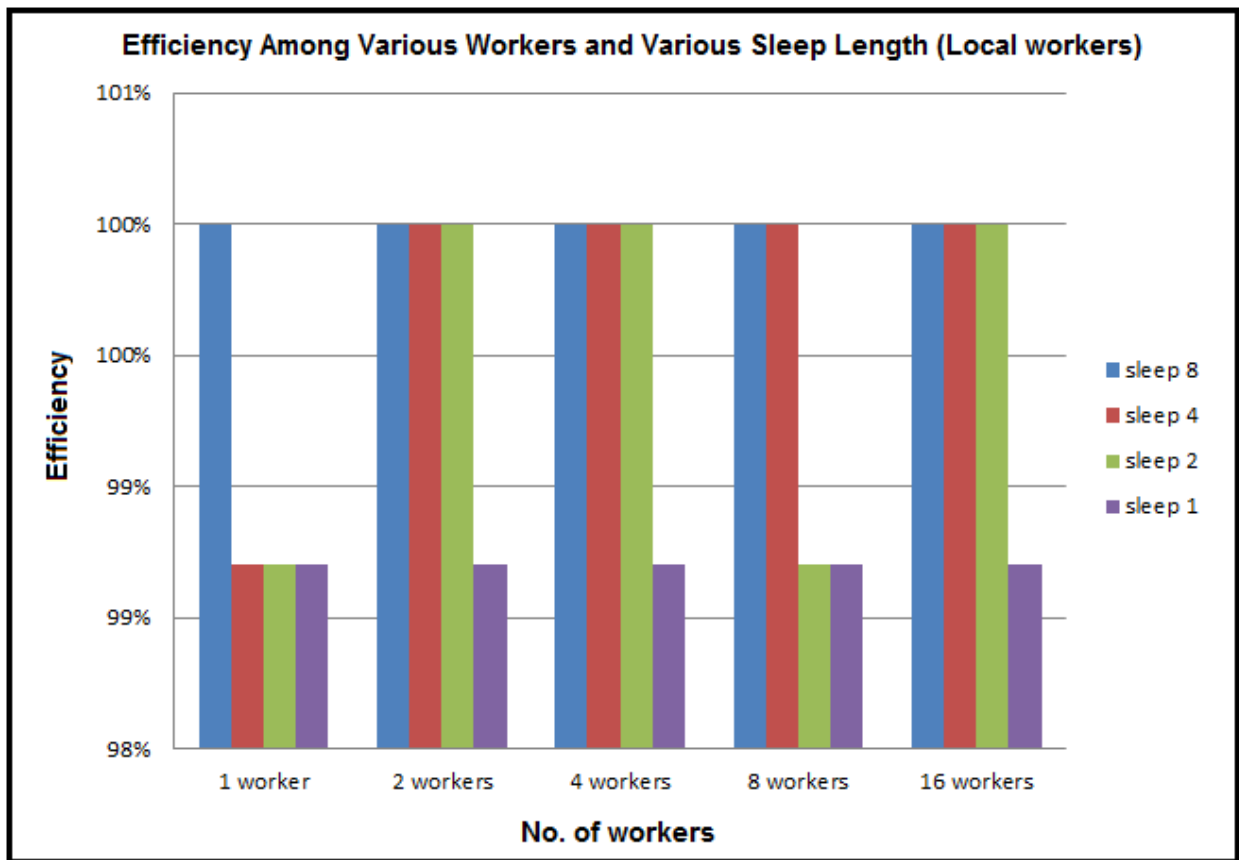
3.2.1 Local workers

	worker1	worker2	worker4	worker8	worker16
Efficiency 10ms	0.51	0.46	0.42	0.29	0.12
Efficiency 1sec	0.97	0.92	0.85	0.73	0.61
Efficiency 10s	0.97	0.92	0.85	0.73	0.61



The efficiency is calculated by $80 / \text{execution_time}$. For each experiment, a workload file is generated in a way such that every worker will have the average sleep time to be 80 seconds. The client records the time first task object is sent to front-end, and the time it received the result of the last task.

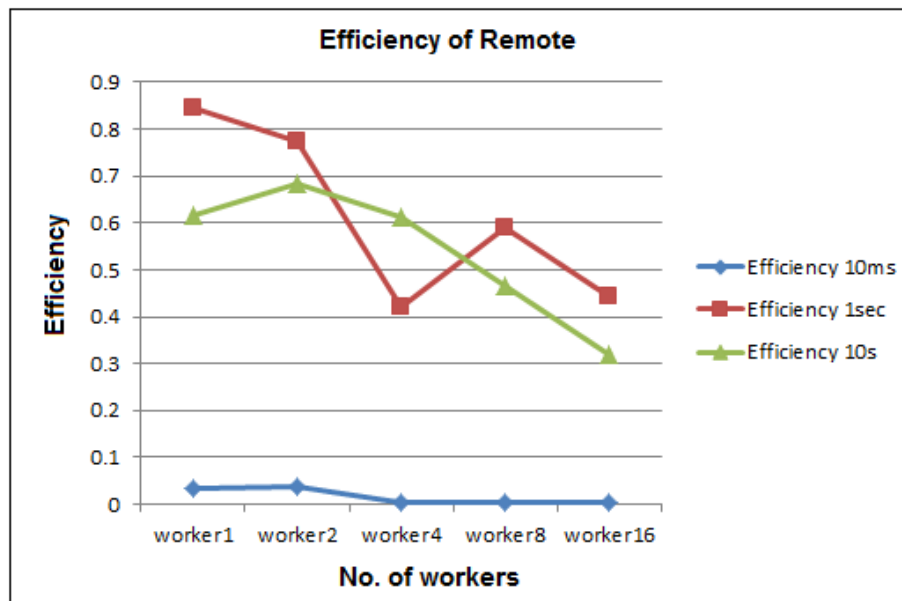
	1 worker	2 workers	4 workers	8 workers	16 workers
sleep 8	100%	100%	100%	100%	100%
sleep 4	98.70%	100%	100%	100%	100%
sleep 2	98.70%	100%	100%	98.70%	100%
sleep 1	98.70%	98.70%	98.70%	98.70%	98.70%



Running local workers show great efficiency in all experiments. sleep 1 experiments do show lower efficiency due to much more overheads with the same amount of real work.

3.2.1 Remote workers

	worker1	worker2	worker4	worker8	worker16
Efficiency 10ms	0.03421	0.03808	0.0057	0.00386	0.00254
Efficiency 1sec	0.84316	0.77367	0.41952	0.58834	0.44491
Efficiency 10s	0.61613	0.68436	0.6117	0.46555	0.31925



Let's first take a detailed look at a simplest case where there is only 1 worker, and 1 task, which is sleep 1:

- a task is read from disk at the client's machine, wrapped into a task object, and sent to the frontend over a TCP connection;
- the frontend then wrap the received task object into a SQS message, and sends it off to the task queue;
- 1 worker fetches messages in every half-second while idle, until a non-empty message is retrieved. Once the message is retrieved, worker first sends its task id off to task table (using DynamoDB) for consistency check. If the task id is proved to be valid, worker passes the task to local process to execute; if invalid, ignore and idle;
- Local process sleeps for 1 second, and informs worker the success of the run;
- Worker wraps updated task object (with result) in a SQS message, and sends it off to result queue, from which frontend fetches once every 10ms;
 - When the frontend retrieves a nonempty message from result queue, it immediately carves out the task object within, and sends it back to client over a TCP connection;
 - The client reads the resulting *task* object from TCP connection and displays it content on screen; and the user is notified that the task has been finished.

The entire process above consists of 8 transfers between 6 entities:

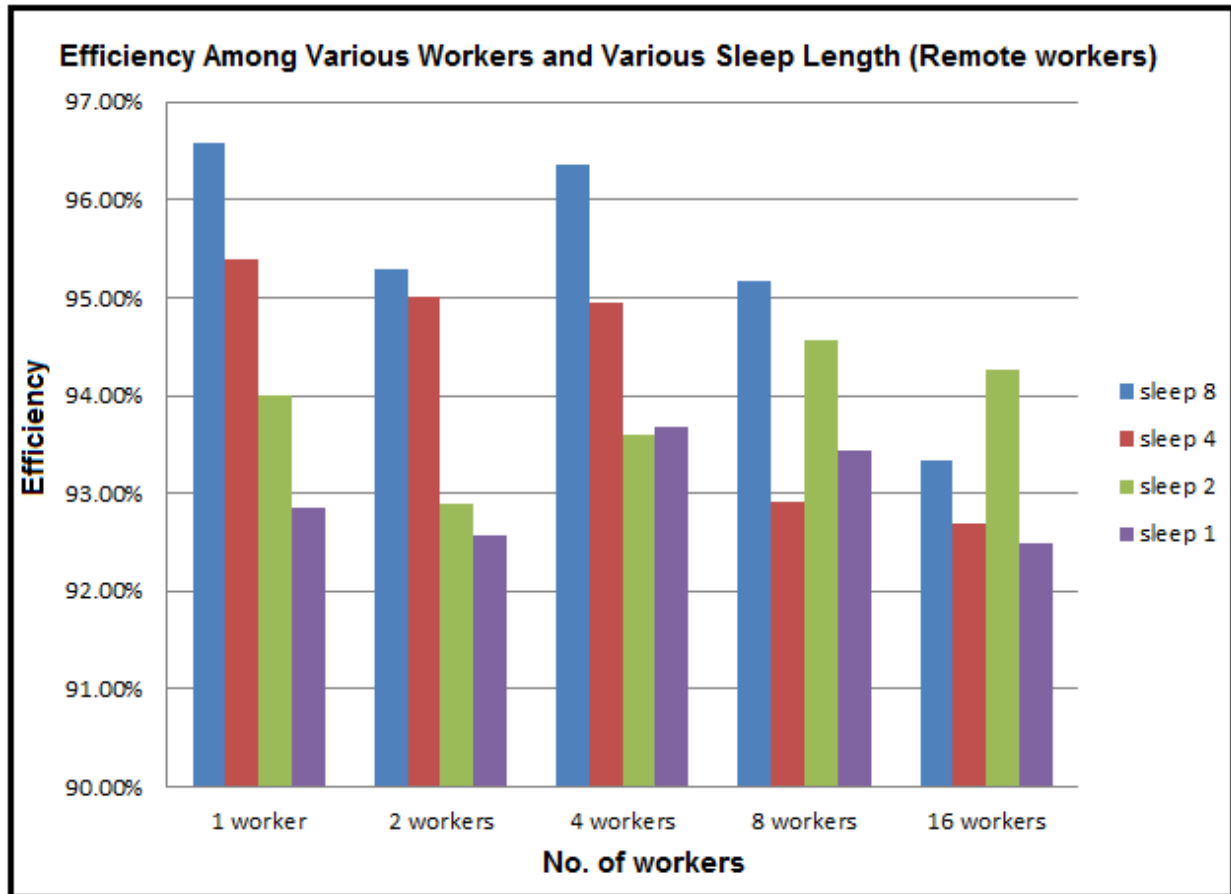
- client *to* frontend *to* task queue *to* worker *to* task table *back to* worker *to* result queue *back to* frontend *back to* client.

Denote the overhead in this process as Δt seconds, for 1worker experiments, we have their total run time (in seconds) presented as follows:

- 8 sec, 10 tasks: $(8 + \Delta t) \times 10 = 80 + 10\Delta t$;
- 4 sec, 20 tasks: $(4 + \Delta t) \times 20 = 80 + 20\Delta t$;
- 2 sec, 40 tasks: $(2 + \Delta t) \times 40 = 80 + 40\Delta t$;
- 1 sec, 80 tasks: $(1 + \Delta t) \times 80 = 80 + 80\Delta t$.

After running the 4 × 5 Experiments, each with 3 runs, and get average run time, we can plot the following chart:

	1 worker	2 workers	4 workers	8 workers	16 workers
sleep 8	96.57%	95.28%	96.36%	95.17%	93.33%
sleep 4	95.40%	95.00%	94.94%	92.92%	92.69%
sleep 2	94.00%	92.90%	93.60%	94.57%	94.26%
sleep 1	92.85%	92.57%	93.67%	93.43%	92.49%



Aside from these final results in chart, we have 2 major observations during experiments:

- in all experiments, frontend feeds task queue much faster than workers consume;
- in all experiments, client receives many “fake” results (utilized as heartbeat in order to keep TCP connections alive) from frontend, which means that the frontend retrieves results much faster than workers produces.

According to the two observations, we conclude that frontend overheads Δt_{front}

doesn't play big role in total overheads. And the total run time is largely dominated by 80 sec sleep time plus backend overheads.

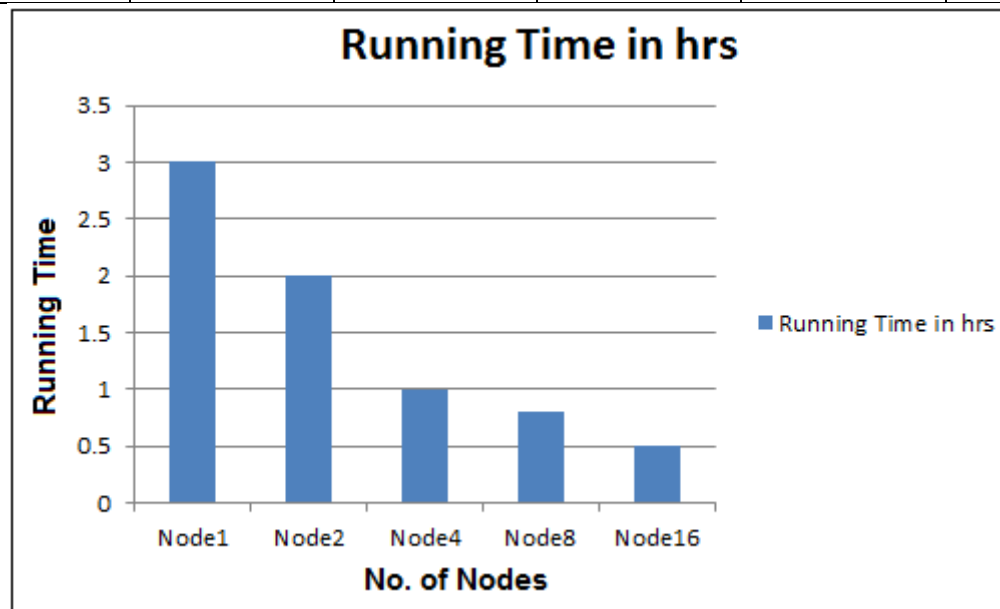
The chart confirms this, to certain degree. Experiment results within one cluster show a trend that efficiency decreases as sleep time shrinks (or, the total run time increases as sleep time shrinks). While experiment results across clusters for a given sleep time show a degree of evenness, suggesting that the elimination of front-end overheads in our analysis above is reasonable, and demonstrates scalability of AWS SQS service and DynamoDB service.

3.2. Animoto Clone:

- It runs on m1.small instances of EC2.
- Real time web applications (conversion of pictures to video) are done instead of sleeping application.
- Pictures from various web sites are collected and converted to video.
- Video is written to S3 and its location is specified to the user.

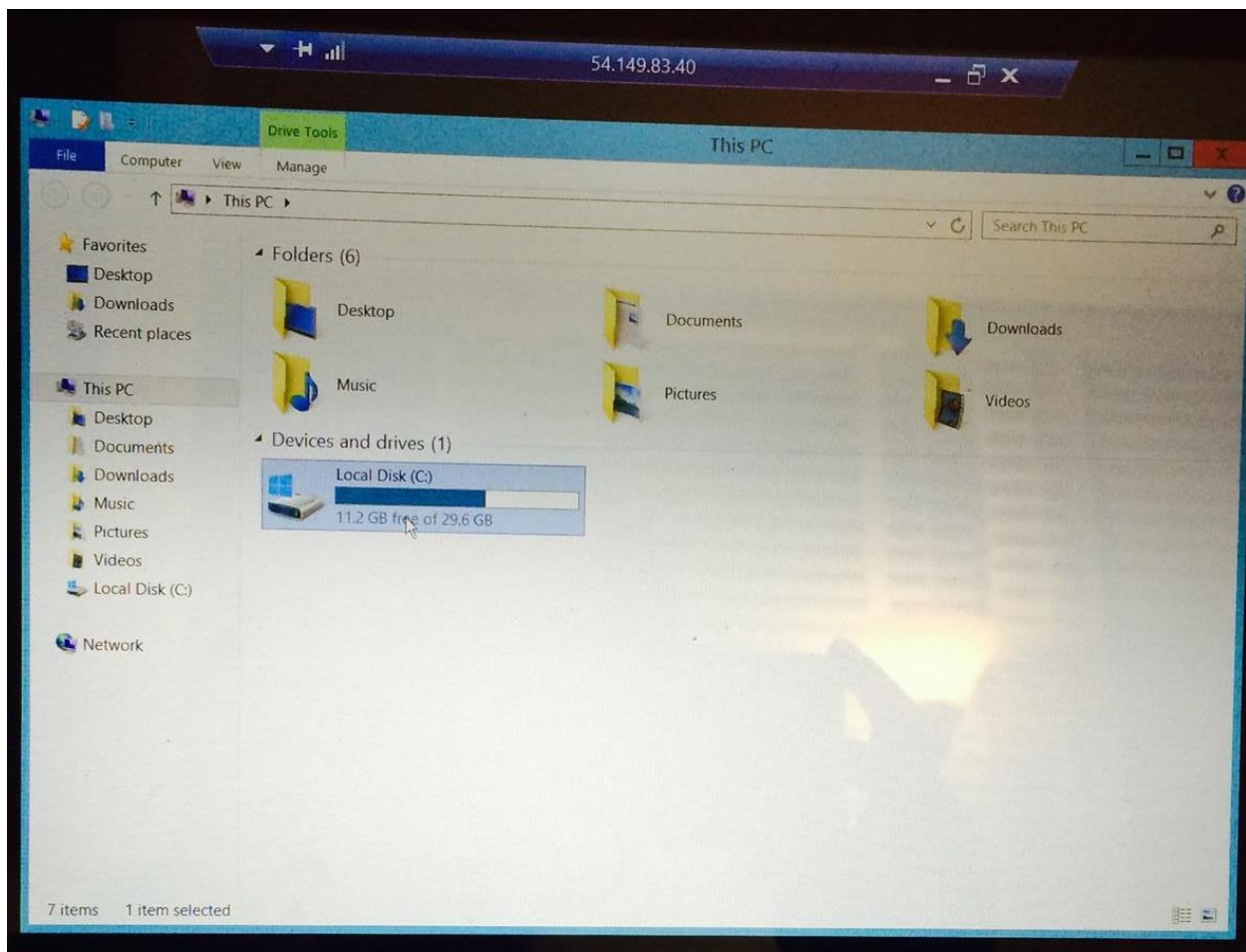
Fixed workload of 160 jobs were used and each job consists of 60 pictures (1920*1080 resolution). The running time for 1,2,4,8 and 16 nodes were observed and plotted as follows

Number of nodes	1	2	4	8	16
Running time	3 h	2 h	1h	0.8 h	0.5 h

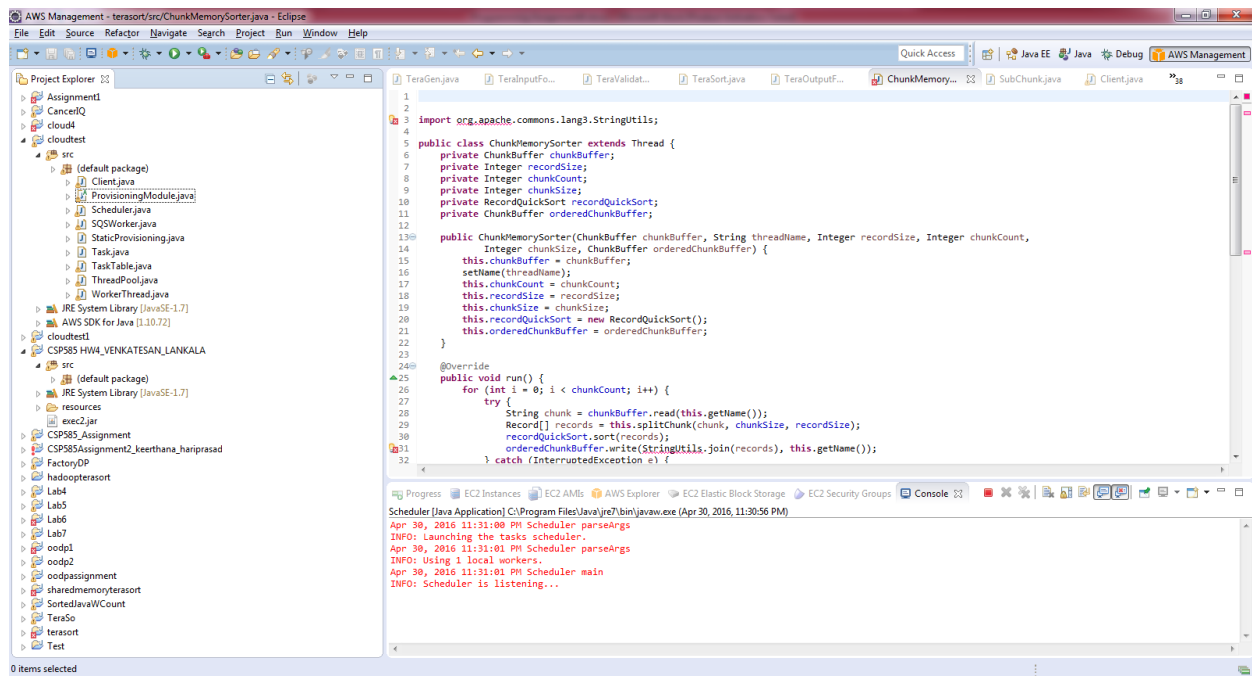


Steps:-

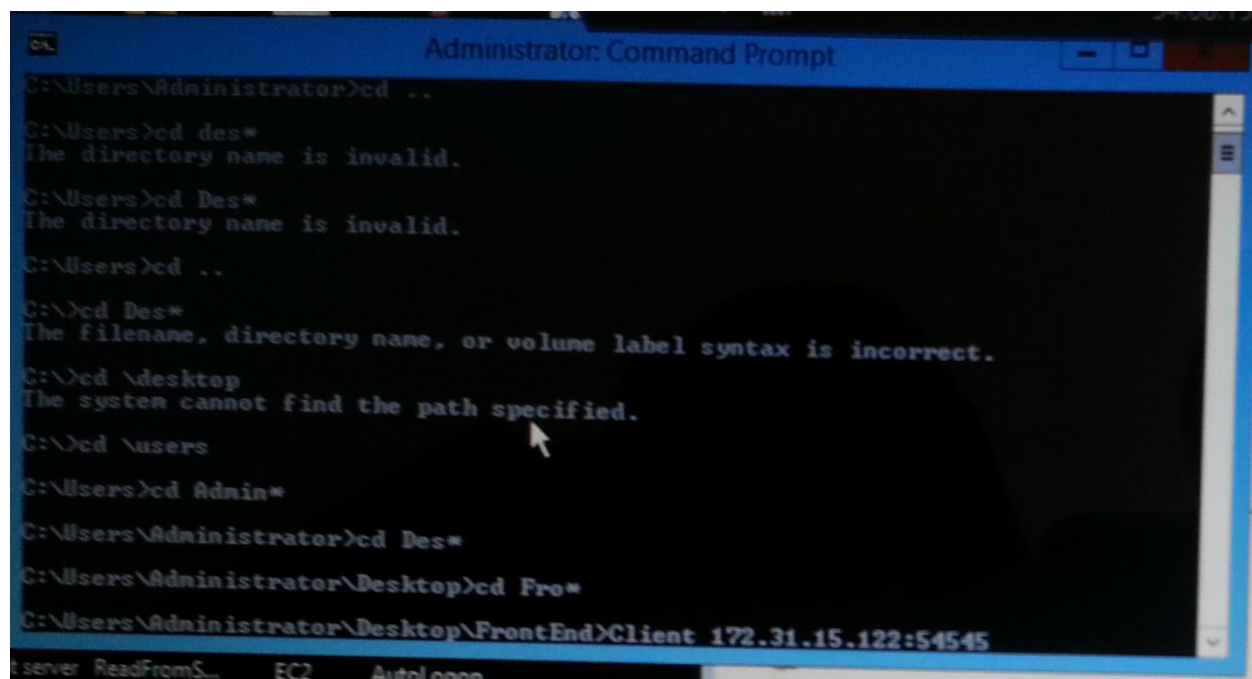
1. Will create Instances 'CLIENT' and 'SCHEDULER' in AWS (aws.amazon.com)
2. SQS have no messages this point, but during running it will contain the messages consists of JOB from client.
3. DynamoDB now it contains 0 rows, at runtime time it will contain the job status like success.
4. Scheduler will connect and download remote desktop IP and start that remote IP



5. In Remote Desktop go to Local Desktop-> workout-> main->Scheduler.java
6. Build the Scheduler java, and Run it to make the ports listen.



7. Build the Image Scheduler, Run it.
8. Connecting Client Instance to Remote Desktop.
In remote desktop run in the command prompt,
Cd desktop
Cd Frontend
Client 172.31.15.122:54545 workload



After Running Front End DynamoDB and SQS:

← → ↺ ↻ <https://us-west-2.console.aws.amazon.com/dynamodb/home?region=us-west-2#explore:name=JobStatus> ☆ » ≡

Apps Maa Paapalu Tola... Introduction to Data... Illinois Institute of T... CMC IIT Blackboard Learn Hair Care : Human N... SolarMovie: Action

AWS Services Edit Hariprasad Ravi Kumar Oregon Support

Amazon DynamoDB Explore Table: JobStatus

List Tables Browse Items

Scan Get Go New Edit Copy to New Details Delete Export to csv

1 to 100 of 100 loaded items

<input type="checkbox"/>	Id	Command	Request	Status
<input type="checkbox"/>	7	sleep	1000	success
<input type="checkbox"/>	47	sleep	1000	success
<input type="checkbox"/>	79	sleep	1000	success
<input type="checkbox"/>	69	sleep	1000	success
<input type="checkbox"/>	8	sleep	1000	success
<input type="checkbox"/>	62	sleep	1000	success
<input type="checkbox"/>	54	sleep	1000	success
<input type="checkbox"/>	75	sleep	1000	success
<input type="checkbox"/>	89	sleep	1000	success
<input type="checkbox"/>	32	sleep	1000	success

← → ↺ ↻ <https://us-west-2.console.aws.amazon.com/dynamodb/home?region=us-west-2#explore:name=JobStatus> ☆ » ≡

Apps Maa Paapalu Tola... Introduction to Data... Illinois Institute of T... CMC IIT Blackboard Learn Hair Care : Human N... SolarMovie: Action

AWS Services Edit Hariprasad Ravi Kumar Oregon Support

Queues

Create New Queue Queue Actions Show/Hide Refresh

Filter by Prefix:

<input type="checkbox"/>	Name	Messages Available	Messages In Flight	Created
<input type="checkbox"/>	MyQueue	1	0	2016-05-29 06:21:43 GMT-06:00
<input type="checkbox"/>	Test	0	0	2016-05-26 00:45:01 GMT-06:00

References:-

1. <http://aws.amazon.com/ec2/>
2. <http://aws.amazon.com/sqs/>
3. <http://aws.amazon.com/s3/>
4. <http://aws.amazon.com/dynamodb/>
5. http://www.ffmpeg.org/faq.html#How-do-I-encode-single-pictures-into-movies_003f
6. <https://www.google.com/imghp?hl=en&tab=ii>
7. <http://www.gnu.org/software/wget/>
8. <http://www.ffmpeg.org/>
9. http://www.ffmpeg.org/faq.html#How-do-I-encode-single-pictures-into-movies_003f