Cloud Computing

CS553

Programming Assignment 3

Distributed Task Execution Framework

CloudKon clone with Amazon EC2, S3, SQS and Dynamo DB

Student

Nikhil Nere

A20354957

nnere@hawk.iit.edu

Table of Contents

1. Introduction

1.1. Purpose

This document provides a detailed design of the distributed task execution framework on Amazon EC2 using the SQS. It provides an overview of different components in the framework. Document also contains manual to run the framework and the performance evaluation.

1.2. Overview of the Framework

A distributed task execution framework similar to CloudKon is implemented. The framework has two components, client (submits tasks to task queue) and workers (retrieves tasks from task queue and executes them).

The framework has two different implementations. One implemented of local client and local workers which read queue from memory. Second distributed design where client and workers are distributed over different nodes. The distributed frame work is run on AWS EC2 instances where client and workers reside on different instances. The distributed frame work makes use of AWS SQS (simple queuing service) and dynamo DB (The purpose of using Dynamo DB is explained later in the Document). Both the frameworks, local and remote are implemented in Java Language.

Client: Client reads the workload file and submits the tasks from the workload file to the task queue. For the local framework in memory queue is used. For distributed framework Amazon SQS is used. The client reads the responses of the tasks from the response queue and makes sure that responses for all the tasks are received.

In local framework the client is a thread running on the local system. In distributed framework the client is a process running on a separate EC2 instance.

Workers: Workers read the tasks from the task queue and execute them. The Result of executing the task is added to the response queue. In local system the tasks are read from the in memory queue. In distributed framework the tasks are read from Amazon SQS.

In local system the workers are different threads running on the same local system. In distributed framework the workers reside on different EC2 instance.

2. Design Overview

2.1. Local Task Execution Framework

The Local task execution framework is a single process where client and workers are different independent threads running on the single system.

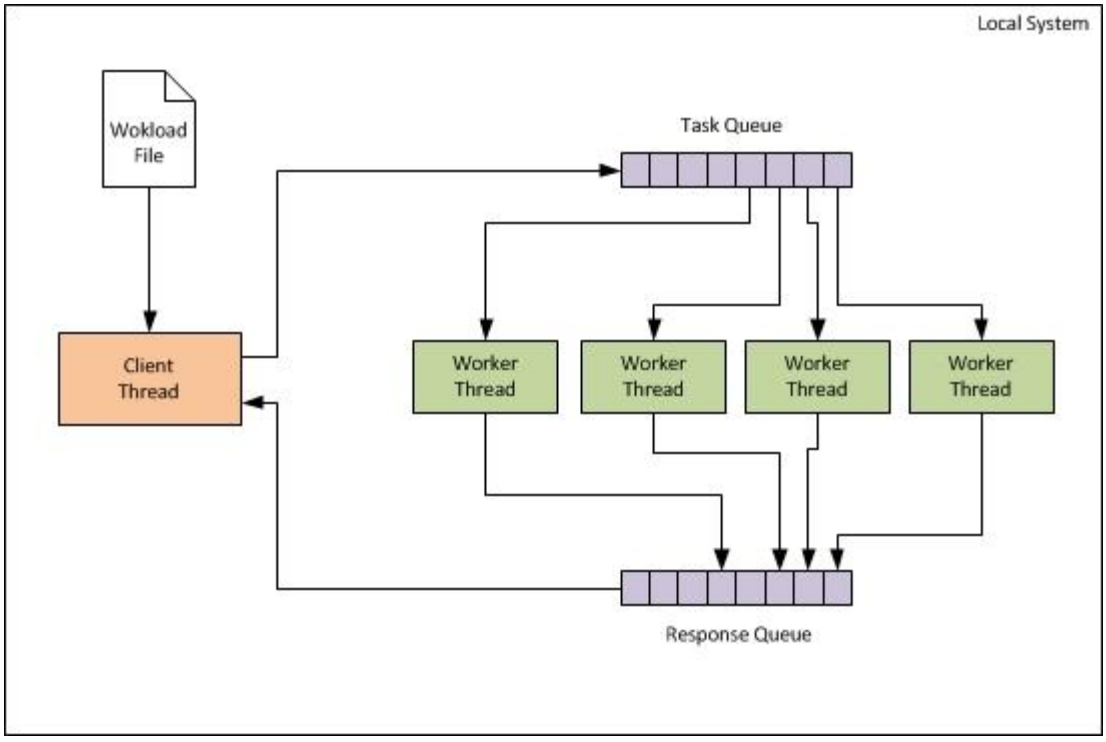


Fig 1: Local Task Execution Framework

- The client thread reads tasks from the workload file and submits them in to the queue. The queue here is java Concurrent Linked Queue.
- After submitting all the tasks, the client reads the responses from the response queue.
- The client makes sure that it received responses for all the tasks submitted. The response received contains status, 0 if successful and 1 if failure.
- The workers poll tasks from the queue and execute them. The responses are put on to the response queue.
- The actual concurrency of the workers depends on the hardware configuration i.e. the number of core available on the system.

2.2. Distributed Task Execution Framework

The distributed frame work is run on the AWS cloud where client and workers are separate EC2 instances. SQS (Simple Queuing Service) is used for task queue and response queue. The Dynamo DB is used to discard the duplicate tasks.

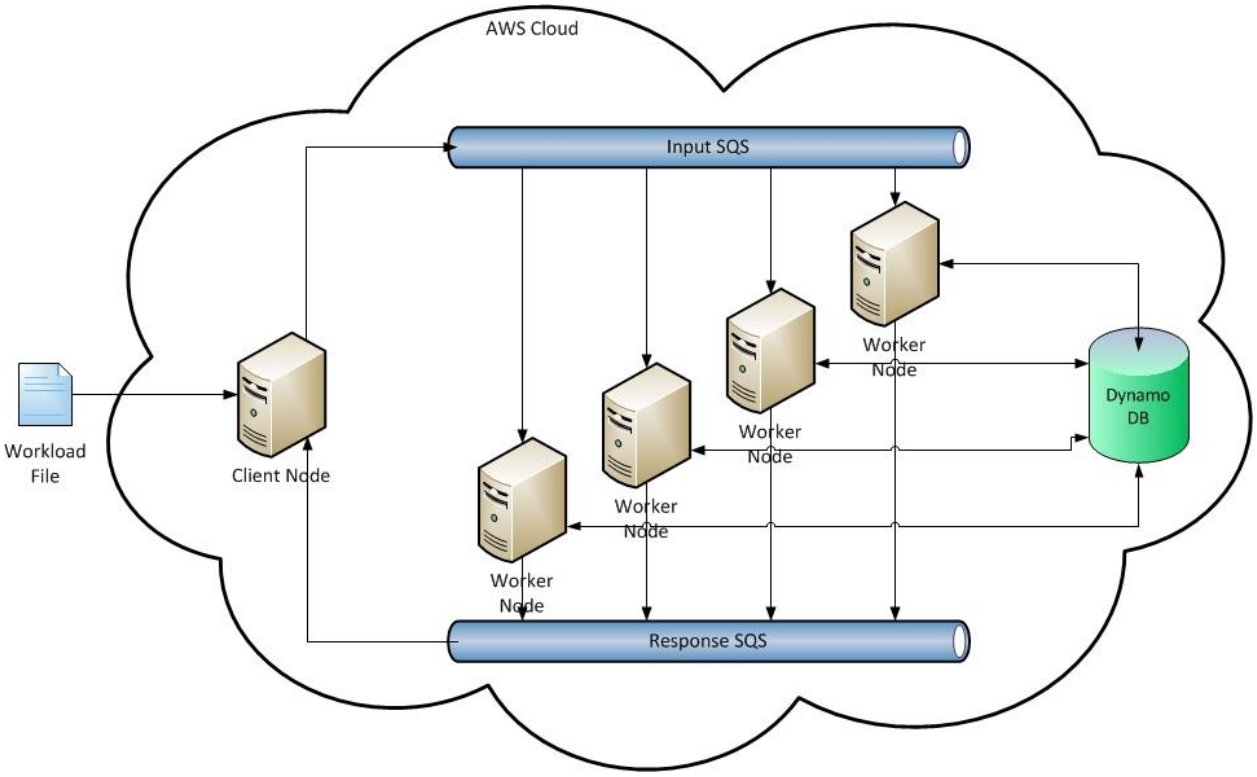


Fig2: Distributed Task Execution Framework

- The Client node reads tasks from the workload file and submits the tasks to the input SQS.
- After submitting all the tasks, the client starts polling the messages from the response queue. If the response SQS is empty, then it waits to get the responses.
- Client checks if responses for all the message are received. Client checks for the tasks ids in the response and discards duplicate response entries which can be caused by the duplication of messages in SQS.
- The worker nodes keep polling messages from the input SQS. If the SQS is empty they wait for the client to submit the tasks in the SQS.
- A worker before executing the task checks if an entry for that tasks is to find in the Dynamo DB. If an entry is found for a particular task that means the task has already been processed by some other worker hence the worker, simply discards that task. If there is no entry for the task in Dynamo DB, then the worker makes an entry for that tasks and executes it.

- After executing the task, the workers put the response message in the response SQS

Duplicate Messages in SQS:

SQS does not guarantee that the messages from SQS are delivered exactly once. This is because of the duplication of the message on multiple SQS server. So there is a possibility that the two workers will get the same tasks. Dynamo DB is used to handle the duplicate tasks. A worker upon receiving a task checks if the task is already present in dynamo DB. If the task is present in dynamo DB, then the worker discards the task and if the task is not present in dynamo DB then the worker makes an entry for the task in dynamo DB and executes it.

2.3. Tradeoffs

- SQS does not guarantee that messages from SQS are delivered exactly once. The workers might get duplicate tasks from SQS. To discard the duplicate message a DHT (Distributed Hash Table) can be used where the workers will make entry for the tasks they picked up. I have used Dynamo DB (a NoSQL storage system provided by AWS)
- In distributed task execution frame work the network latency can be hidden by fetching the multiple tasks. There are two approaches to do this
    - Instead of fetching a task at a time, fetch multiple tasks at a time and store them in memory. Hence the worker will not for the message to come over the network for execution.
    - The second approach is - there will be a separate thread running on the worker to fetch the message. So once the worker start executing a task the separate thread will fetch next task to be executed. Hence the worker will not wait for the task to be fetched.

3. Performance Evaluation

The performance of the both the frameworks is measured on the t2.micro EC2 instance having below configuration.

Model	vCPU	CPU Credits / hour	Mem (GiB)	Storage
t2.micro	1	6	1	EBS-Only

In distributed task execution framework SQS is used for queuing service. The read and write capacity for dynamo DB table is set to 100.

3.1. Local Task Execution Framework

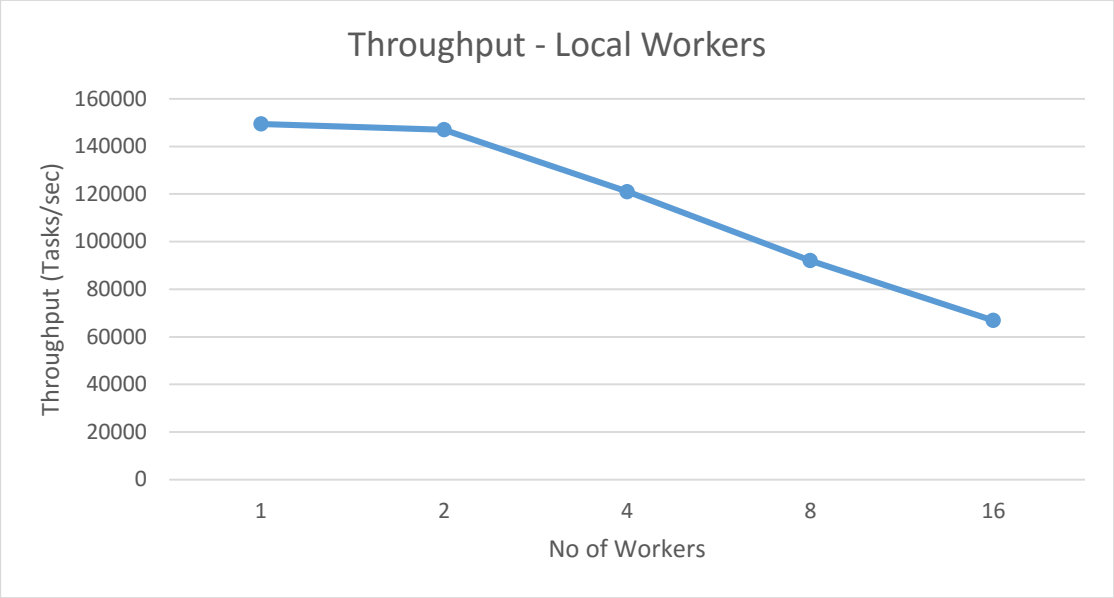
A Single t2.micro instance is created.

3.1.1. Throughput

A workload of 100K, sleep 0 tasks is given to the framework. Below are the timings captured and the throughput. With each experiment the number of worker threads is increased.

No of Workers	No Of Tasks	Time (sec)	Throughput (Tasks/sec)
1	100000	0.669	149476.8311
2	100000	0.68	147058.8235
4	100000	0.826	121065.3753
8	100000	1.086	92081.03131
16	100000	1.495	66889.63211

The throughput and the corresponding number of worker threads is plotted –



Observations:

- The throughput decreased as the number of threads are increased.
- This task here is sleep 0 which takes time to execute an instruction and not more than that so the time we see is only the time taken by the scheduler and worker code to run.
- The t2.micro instance is used for the experiment which has only one virtual core. Hence with increased threads there is no performance improvement.
- The throughput in the local system depends on the number of cores available.

3.1.2. Efficiency

Three different experiments are run with different tasks. Below are the details of the experiments. The efficiency will tell the extra time taken by the task scheduler framework. The ideal time is the time when we have pure parallelism without adding any extra overheads.

1. 10ms sleep task

No of Workers	No of Tasks	Ideal time (ms)	Actual time (ms)	Efficiency
1	1000	10000	10153	98.49305624
2	2000	10000	10187	98.16432708
4	4000	10000	10208	97.96238245
8	8000	10000	10305	97.04027171
16	16000	10000	10282	97.25734293

2. 1sec sleep task

No of Workers	No of Tasks	Ideal time (ms)	Actual time (ms)	Efficiency
1	100	100000	100035	99.96501225
2	200	100000	100042	99.95801763
4	400	100000	100034	99.96601156
8	800	100000	100039	99.9610152
16	1600	100000	100070	99.93004897

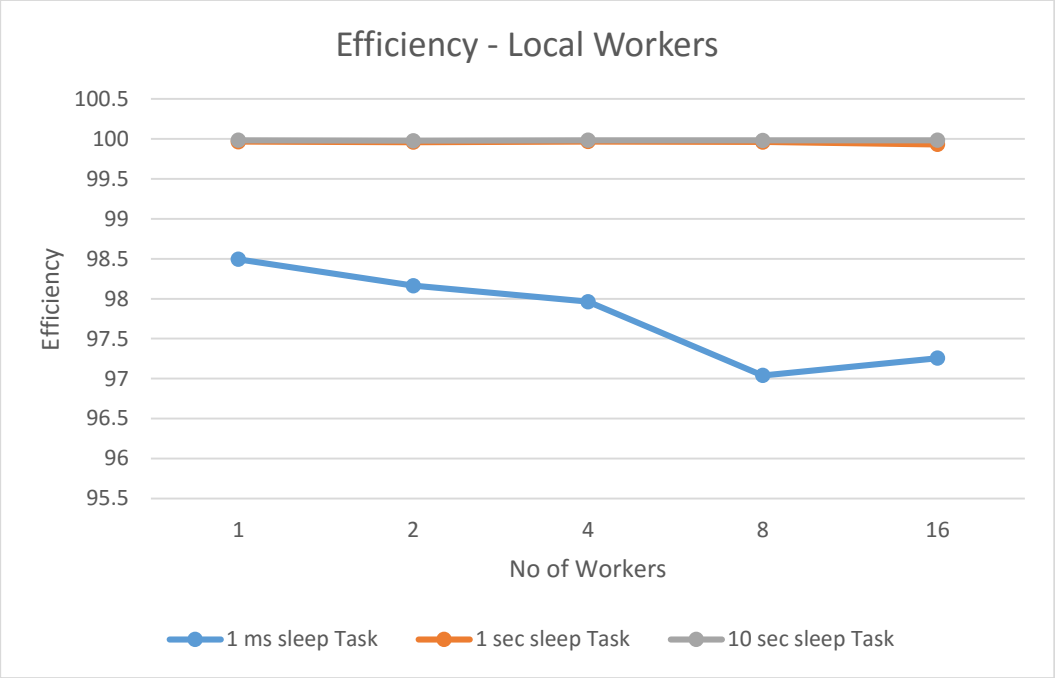
3. 10sec sleep task

No of Workers	No of Tasks	Ideal time (ms)	Actual time (ms)	Efficiency
1	10	100000	100017	99.98300289
2	20	100000	100023	99.97700529
4	40	100000	100018	99.98200324
8	80	100000	100020	99.980004
16	160	100000	100017	99.98300289

Comparing the efficiency from above experiments

No of Workers	1 ms sleep Task	1 sec sleep Task	10 sec sleep Task
1	98.49305624	99.96501225	99.98300289
2	98.16432708	99.95801763	99.97700529
4	97.96238245	99.96601156	99.98200324
8	97.04027171	99.9610152	99.980004
16	97.25734293	99.93004897	99.98300289

Plotting the efficiency against increased number of worker threads



Observations:

- For the 1 ms sleep tasks the efficiency is less because here we have more number of tasks.
- As the number of tasks increase the efficiency will decrease.
- In case of 1 sec sleep task and 10 sec sleep tasks the total number of tasks is very less as compared to 1ms sleep tasks.
- With every task the total amount of extra code running increases (scheduler code) which makes it less efficient (more time that the ideal time).
- Task granularity has effect on the efficiency of the framework

3.2. Distributed Task Execution Framework

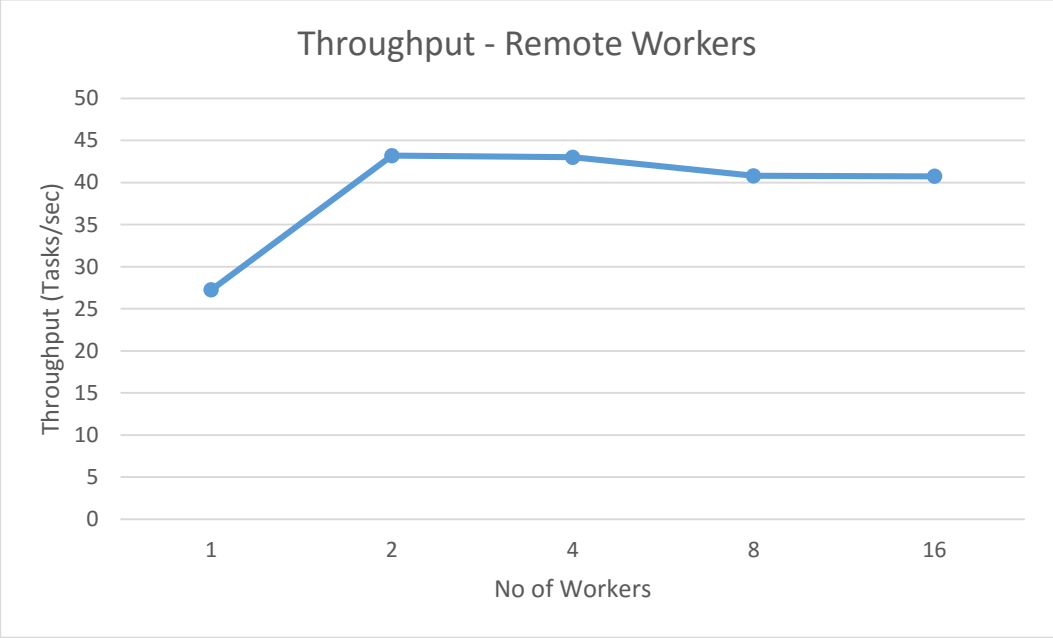
Client and workers are run on separate t2.micro instances. The read/write capacity of the Dynamo DB table is set to 100.
The number of threads on the single worker thread is set to be 1 for all the experiments carried out below because we are using t2.micro where only one thread can run at a time.

3.2.1. Throughput

A workload of 10K, sleep 0 tasks is given to the framework. Below are the timings captured and the throughput. With each experiment the number of worker nodes is increased.

No of Workers	No Of Tasks	Time (sec)	Throughput
1	10000	367.013	27.24699125
2	10000	231.511	43.19449184
4	10000	232.493	43.01204767
8	10000	245.046	40.8086645
16	10000	245.392	40.75112473

The throughput and the corresponding number of worker nodes is plotted –



Observations:

- Maximum throughput is achieved with 2 worker nodes.
- As the number of worker nodes increases the client becomes the bottle neck.
- The workers process the tasks simultaneously and populate the response queue really fast but the client now takes time to read the response queue.
- Hence even if we increase number of workers the performance will not increase since we have only one client to read from the response queue.
- After a certain point we will see a straight line i.e. almost the same throughput even if we go on increasing the worker threads.

3.2.2. Efficiency

Three different experiments are run with different tasks. Below are the details of the experiments. The efficiency will tell the extra time taken by the task scheduler framework to schedule the tasks. The ideal time is the time when we have pure parallelism without adding any extra overheads.

1. 10ms sleep task

No of Workers	No of Tasks	Ideal time (ms)	Actual time (ms)	Efficiency
1	1000	10000	65615	15.24041759
2	2000	10000	65140	15.35155051
4	4000	10000	108788	9.192190315
8	8000	10000	184718	5.413657575
16	16000	10000	364405	2.744199448

2. 1 sec sleep task

No of Workers	No of Tasks	Ideal time (ms)	Actual time (ms)	Efficiency
1	100	100000	116194	86.06296366
2	200	100000	112014	89.27455497
4	400	100000	119466	83.70582425
8	800	100000	112446	88.93157605
16	1600	100000	112787	88.66270049

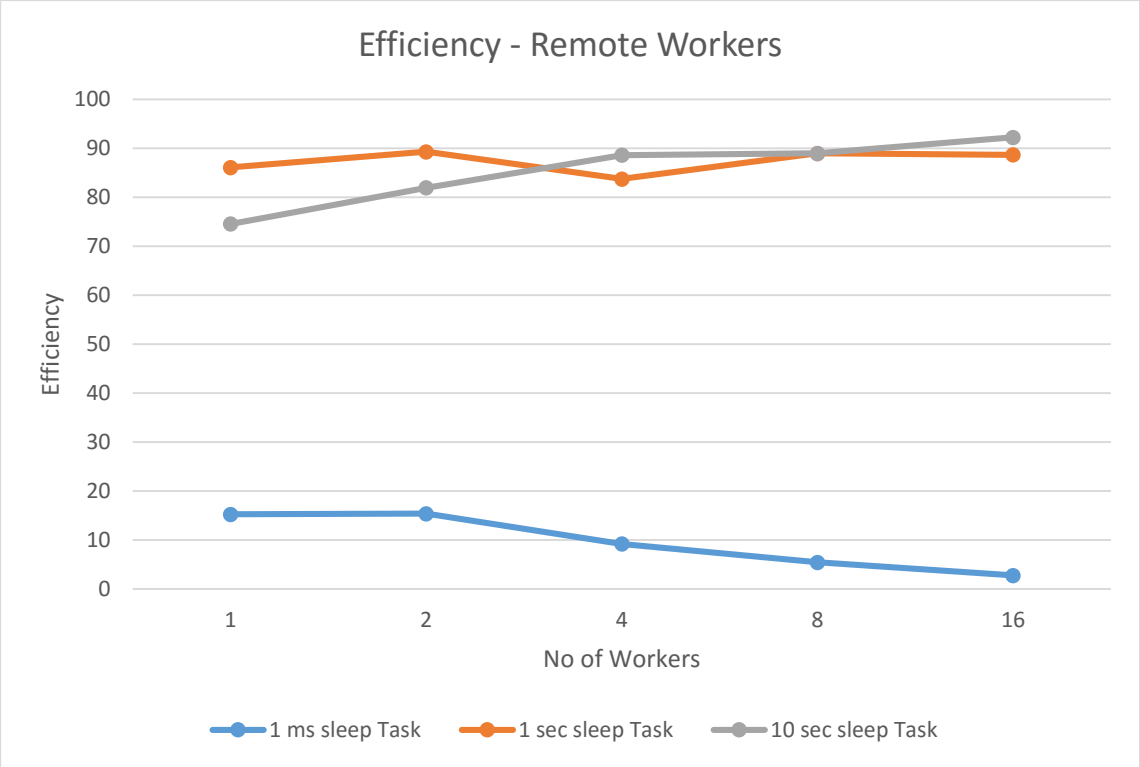
3. 10 sec sleep task

No of Workers	No of Tasks	Ideal time (ms)	Actual time (ms)	Efficiency
1	10	100000	134186	74.52342271
2	20	100000	122091	81.90611921
4	40	100000	112886	88.5849441
8	80	100000	112433	88.94185871
16	160	100000	108430	92.22539887

Comparing the efficiency from above experiments

No of Workers	1 ms sleep Task	1 sec sleep Task	10 sec sleep Task
1	15.24041759	86.06296366	74.52342271
2	15.35155051	89.27455497	81.90611921
4	9.192190315	83.70582425	88.5849441
8	5.413657575	88.93157605	88.94185871
16	2.744199448	88.66270049	92.22539887

Plotting the efficiency against increased number of worker threads –



Observations:

- The efficiency in case of 10ms tasks is less.
- For 1 ms sec tasks we have more number of tasks. As the number of tasks increase we add extra overhead to schedule that task (Overhead is the scheduler scheduling the tasks). Hence as number of tasks will increase the efficiency will decrease.
- Also as number of tasks increase, the messages exchanged also increases which adds up the network latency with every tasks scheduled this is because we only have one client and it reads the responses sequentially.
- In case of 1sec and 10sec tasks the number of tasks is very less, so the efficiency more as compared to 10ms tasks.

Key point –

In the distributed task scheduler if the time required to execute the task is less than the network latency then the performance will degrade significantly. Scheduling a task over the distributed network will take more time than actually executing the task.

4. Manual

4.1. Running Local Task Execution Framework

- Follow below steps to run the experiments mentioned in the performance evaluation
    - Copy all the code and scripts (TaskSchedulerLocal folder) on the System where you want to run the local task executer framework.
    - In case of the EC2 instance use scp command to copy all the code and scripts.

      scp –i 'filename.pem' –r TaskSchedulerLocal ubuntu@IPADDRESS:/home/ubuntu

      Enter the pem file name and the ip address of the instance
    - Login to the instance and invoke the below script. The script will run all the experiments mentioned above in the performance evaluations. The script will also installs all the required packages. The workloads are generated for different experiments

      ./home/ubuntu/TaskSchedulerLocal/runLocalScheduler.sh

- To generate the sleep tasks workload run the utility as below
    - *java WorkloadGenerationUtil 1000 "sleep 10" Workload*

      WorkloadGenerationUtil is the utility name
      1000 is the number of sleep tasks
      sleep 10 is the task
      Workload is the workload filename

- To run the frame work for a specific case use below command
    - *ant -Dt=1 -Dw=Workload run*

      -Dt is the number of worker threads
      -Dw is the workload filename

4.2. Running Remote Task Execution Framework

Before running the client, the workers should be started

4.2.1. Running Remote Back-End Worker

- Open *taskSchedulerProp.properties* file and enter AWS access key and secret key.

- Open *runWorker.sh* script and update the below command as per your need
    - *ant -DinputQ=inputQ -DoutputQ=outputQ -Dt=1 runworker*

      This command will start the worker process on the remote worker node.
      -DinputQ is the input SQS name (The input SQS will be created if does not exist)
      -DoutputQ is the output SQS name (The output SQS will be created if does not exist)

-Dt is the number of threads that will run within a single worker process

- Open *workers.txt* and enter IP addresses of all the workers. The number of workers will be decided on the ip addresses mentioned in this file.

- Run below command which will copy all the files, libraries and scripts on all the worker nodes mentioned in the workers.txt file and will start the worker process on all the worker nodes simultaneously
    - *./runWorkerRemotely.sh*

- You can also use cluster ssh to run commands mentioned in *./runWorkerRemotely.sh* file. Cluster ssh will execute the commands on all the workers simultaneously same as pssh used in *./runWorkerRemotely.sh* file.
- Now you are good to start you client node.

4.2.2. Running Remote Client

- Open *taskSchedulerProp.properties* file and enter AWS access key and secret key. This you have already done while starting the workers

- Open *runClient.sh* script and make changes to below entries as per your need
    - *java WorkloadGenerationUtil 10000 "sleep 10" Workload*
      This command is to generate the workload file on the client instance
      WorkloadGenerationUtil is the utility name
      1000is the number of sleep tasks
      sleep 10 is the task
      Workload is the workload filename

    - *ant -DinputQ=inputQ -DoutputQ=outputQ -Dw=Workload runclient*
      -DinputQ is the input SQS name
      -DoutputQ is the output SQS name
      -Dw is the workload filename

- Open *runClientRemotely.sh* script to update the ip address of the client instance
    - *CLIENT_IP=54.172.170.204*

- Invoke the below script which will copy all the required files, libraries and scripts on the client instance and will start the client from your local system
    - *./runClientRemotely.sh*