CS 553 Cloud Computing
Programming Assignment 3
Name: Kedar Kaushikkar (A20355218)

Design Document

The document includes the design and architecture for CloudKon clone with Amazon EC2, S3, SQS and DynamoDB

Language: Python
The basic aim of the program is to design an architecture and implement a distributed task execution framework on Amazon EC2 using the SQS. For this programming assignment I have used python language to connect to amazon SQS services.

Design:
The program takes up the tasks from text file that is provided during executing the python script and then providing the number of threads to get the total time for executing the tasks.

For my programs the number of threads used are 4 while varing the number of workers and the tasks.

The assignment consist of client and worker scripts that interact with the SQS service and output the time required to execute the tasks.

Client.py

- The client.py is the python script that takes up the task from the text file and sends it to the SQS messenging service to perform those tasks.
- Running the client script the arguments need to be provided . For ex :

is_local = QUEUE_NAME / LOCAL
This argu value takes whether the tasks are to be provided to the SQS service or it will stored in as in-memory queue and executed by the worker. The QUEUE_Name is the name of the queue defined in the SQS while the LOCAL simply takes up the tasks into the local in-memory

> -t = No_of_Threads.
> Defines the number of threads that our worker will use to perform the task parellely.
>
> -s Filename
> Defines the name of the file from where the client will pick up the task.

- The Client script takes up the filename and splits each task line by line. Each task is stored in queue .
- The program connects to the amazon SQS service using the api of boto which takes up the access_key_id and access_secret_key_id , the queue names and transfers the task from the task text file to the amazon SQS queue.
- The queues created in the amazon SQS is MY_QUEUE which copies all the task from the source task text file and , PROCESS_QUEUE which stores the completed tasks.

- Also we have implemented the Dynamo DB to avoid the same task being executed over by multiple workers . The configuration to Dynamo DB is also provided by the boto service a python api to use amazon Dynamo DB .
- The Dynamo DB requires a table and a hash key to be stored with defining the number of reads and writes .
- The readtask/readtaskSQS function reads the filename and puts in the local queue.
- The connection to the SQS on our amazon account is done in this function . Each task that is put into the SQS is given a task Id and it is incremented with incoming of each task.
- The connection to dynamoDB is also done which takes the hashkey to determine whether the task is duplicate or not. The number of read and write units in my script is taken as 20 which is constant throughout the number of workers to keep a constant environment throughout.
- Once the initialization is done on the tasks they are processed in the initializethread and sent for processing threads.
- The processlocalthread just appends thread to each task in the queue and joins it.
- In our experiment we perform sleep tasks for N number of miilliseconds on multiple threads and multiple workers.

Worker.py

- The worker.py is responsible for pulling up task from the queue on SQS executing it and report the processed task to the PROCESS_QUEUE.
- The worker.py scripts while execution takes the following argument.

  -s QUEUE NAME – defines the queue name from where the tasks to pick up.

- The PROCESS_QUEUE is also initialized where the completed tasks are being forwarded.
- The getandProcessTask gets the message from the queue via the boto api functuin queue.get.messages() .
- Each message is then checked with the hash key defined in the DynamoDB table to check if the task is duplicate.
- If the task is not duplicate it is then put on the threads to execute multiple task parellely.
- The processed task are written in the text file using function processTask.
- Exec function is used execute to the sleep task.

Animoto :

The task is to get 160 jobs with each job having 60 image url of 1920 * 1080 pixels and creating 160 videos of the 60 images using the animoto web application api.

Similar architecture is used where the consumer producer are the client and worker scripts and the text file of 9600 (160*60) image url is given.
Each task parses the url in the file and gets it using wget function, and using the animoto api function of ffmpeg function , creates a video of 60 images.

The videos are stored in the S3 storage with 160 links present for each video.