

Yogesh Palav | CSP553-Cloud Computing | May 1, 2016

Design-PA3

# Introduction

The assignment provides an implementation of a distributed task execution framework on Amazon EC2 using the SQS similar to CloudKon.

# Design

The framework contains two different components

1. Local task execution with Multi-Threading.
2. Client and multiple remote worker execution.

# Design Details

The overall functionality of the components is as below

1. Local task execution with Multi-Threading-
   1. The local program receives the input file and reads the task present in the file and adds in a Map data structure which enqueues it to Queue data structure of type Hashmap.
   2. The data is then dequeued and passed in the thread pool with number of threads given as an input.
   3. The task is then executed and a new response map is used to enqueue the all the executed tasks.
   4. The output to the program is number of time taken to execute the specific task
2. Client and multiple remote worker execution.
   1. This consists of following major components-
      1. Amazon Simple Queue Service
      2. Amazon Dynamo DB
      3. Client Program
      4. Worker Program
   2. Amazon Simple Queue Service
      1. A simple message queuing service which is fast, reliable, secure. It can be used to transmit any volume of data, at any level of throughput, without losing messages or requiring other services to be always available.
      2. With SQS, we can offload the administrative burden of operating and scaling a highly available messaging cluster, while paying a low price for only what you use.
      3. For more info refer to - <https://aws.amazon.com/sqs/>
   3. Amazon Dynamo DB
      1. Amazon DynamoDB is a fast and flexible [NoSQL database](https://aws.amazon.com/nosql/) service for all applications that need consistent, single-digit millisecond latency at any scale.
      2. It is a fully managed cloud database and supports both document and key-value store models.
      3. Its flexible data model and reliable performance make it a great fit for mobile, web, gaming, ad tech, IoT, and many other applications.
      4. For more info refer to - <https://aws.amazon.com/dynamodb/getting-started/>
   4. Client Program
      1. The client program accepts SQS Queue Name and Workload file as input parameters.
      2. The program reads the AWS credentials for specific user and creates a queue in the SQS console. Then it reads the messages in file, loops through them and inserts them in the queue. You can view the messages in the SQS queue console.
   5. Worker Program
      1. The worker is basically to be distributed to all of the EC2 instances. A bash script with parallel ssh will be used to start workers, as they need to instantialized parallely.
      2. The program takes queue name and number of workers as input params.
      3. It then gets users AWS credentials and searches for the queue and connects to it.
      4. Dynamo DB object is instantiated and new table is created.
      5. The messages are then retrieved from SQS queue and the tasks are added in DynamoDB table.
      6. The table is scanned for checking the primary key constraints violations.
      7. If the message is not present in the table it is inserted in table and value is retrieved to perform sleep task.
      8. After execution it is deleted from queue and added in response queue with value set as 0 or 1 depending upon whether the execution succeeded or failed.
      9. The total time to execute is provided as the output.

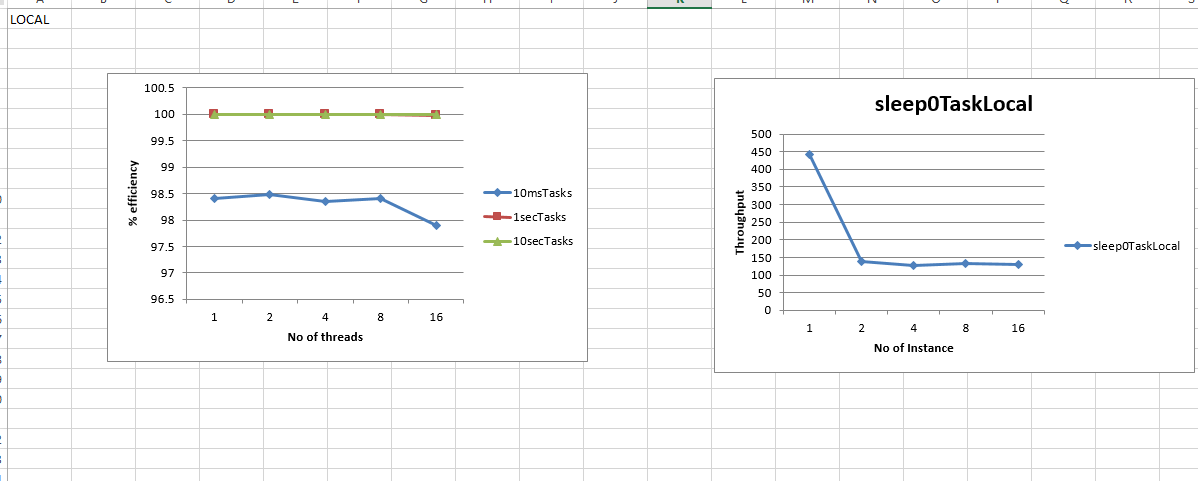
# Manual

1. User Instructions-
   * 1. Tools used- Eclipse Neon, Maven 3.1.2 build tool.
     2. Jar file for all the projects is created on the local itself by maven using “mvn clean package” command.
     3. The jar files are then moved to respective Ec2 instances along with the bash scripts if needed.
     4. The local execution is started by following command-
        1. Java –jar {jar file name.jar} client –s Local –t N –w <WORKLOAD\_FILE>
        2. The workload file will contain 1000 values for sleep 10 ms tasks, 100 values for sleep 1 secs and 10 values for sleep 10 secs tasks.
        3. Generate a file with above criterias.
        4. 1,2,4,8,16 threads variations are passed as the value of parameter N.
        5. This helps in calculating efficiency for the each task variations.
        6. Again, the same program is ran but the workload file will contain 100k rows for sleep 0 task so as to provide throughput for the program.
     5. The Client Program is started by following command-
        1. Java –jar { jar file name.jar} client –s QNAME –w <WORKLOAD\_FILE>
        2. The workload file will contain 1000 values for sleep 10 ms tasks, 100 values for sleep 1 secs and 10 values for sleep 10 secs tasks.
        3. Generate a file with above criterias.
        4. Pass the queue name, which will create a queue on SQS console.
        5. The program will create a queue and upload tasks in it.
     6. The Worker program is started by following command-
        1. Java –jar {jar file name.jar} worker –s QNAME –t N
        2. N is the number of workers.
        3. If running on multiple servers.
           1. Run the servers on x-shell or with parallel ssh script provided in deliverables.
           2. Start 2,4,8,16 AWS instances and run the above script on local machine.
           3. It will upload jar on all the hosts and also run the jar.
           4. Monitor the queue and the table consistently.
        4. Again the same program is ran but the workload file will contain 100k rows for sleep 0 tasks so as to provide throughput for the program.

# Performance Evaluation-

The following graph shows performance evaluation for both Remote and Local workers

Local Efficiency and Throughput-

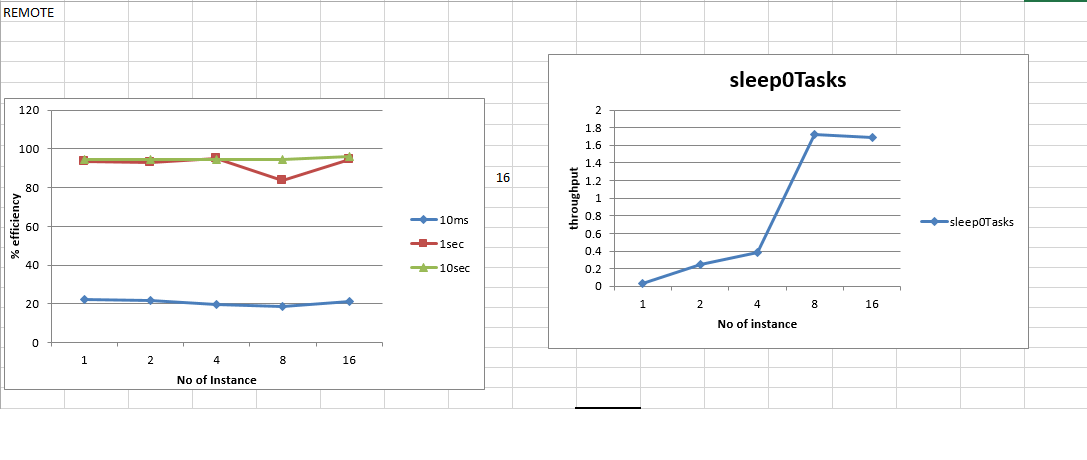


Evaluation-For efficiency, I have measured efficiency for 1, 2, 4, 8, 16 threads for 1000 tasks of 10ms each for each thread, 100 tasks of 1 sec for each worker and 10 tasks of 10 sec each for each worker.

The efficiency of my system is better considering the local worker scenario. Also, increase in task size improves efficiency gradually. So I can conclude that for local runs, the system performs like an ideal system with almost little or no overheads.

For Throughput, sleep 0 task is ran for different number of threads and with different time limits. The above graph shows the sleep 0 run for different number of threads for 100k tasks. As the program is running on t2.micro, it has only 1 core. The throughput is significantly higher for 1 thread than for other variations. So as the number of threads increases the throughput decreases as all the threads will access the same cache and memory.

Remote Efficiency and Throughput-



Remote Worker- For efficiency, 1000 tasks of 10ms each, 100 tasks for 1 sec each and 10 tasks for 10 sec each are measured for thread variations of 1,2,4,8, 16.

The efficiency is low for sleep tasks of 10 ms time. As we scale up the efficiency improves gradually. However, some tasks have low efficiency since they have to connect to Dynamo Db and heavy processing is observed. But as we move to tasks with higher sleep values this overhead decreases.

For throughput, Sleep 0 tasks is ran for different number of time variations and different number of workers. Throughput increases as the number of worker increases. Here throughput is calculated as number of tasks per total time taken to complete the tasks.