CS553 : Programming Assignment #4

Due on Monday, December 1, 2014

Ioan Raicu

Thomas Dubucq / Tony Forlini / Virgile Landeiro Dos Reis

Contents

1 Introduction

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 dynamic task provisionner. Our framework is divided into three parts: a client which 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. Finall 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.

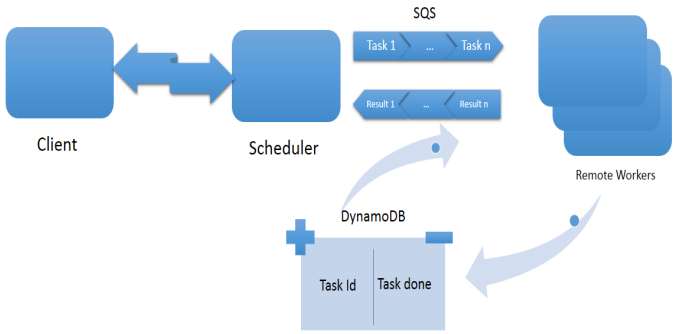

Figure 1: Framework layout

2 The Task Execution Framework

2.1 The Client

We implemented the client in the "Client.java" class. The client takes in argument the IP address/Port number of the front-end server used for scheduling using the following line command format:

client -s $< IP\_ADDRESS : PORT > -w < WORKLOAD\_FILE >$

The workload file is a file containing the different tasks that the workers have to launch. In our case these tasks are sleep tasks followed by the duration of the sleep.

We used Java sockets in order to make the client and the scheduler communicate.First we parse the command line arguments to create the socket.

We also created a Task object which contains attributes such as the Id of the task, the arguments which are the task nature (here sleep) and the argument (here duration), a boolean which indicates if the task has been completed and finally the result of the task (0 if worked, else 1). Thus we can parse the workload file

into a list of tasks which is sent to the scheduler.

The client also polls the scheduler every 2 seconds in order to keep track of the result of the jobs which has been submited from the workload file.

2.2 The Front-End Scheduler

The front end scheduler is implemented in the Scheduler.java class. The scheduler takes a command line agument which is the port number used to communicte with the client and an argument which sets the workers on "local" or "remote". In the local case, we have to define the number of local workers which corresponds to the numbers of threads which will run the sleep jobs. We first used a Java socket to retrieve the tasks from the client.

Then we put those tasks in a blocking queue in order to ba able to run the threads for the local workers.We also implemented a threadpool which contains and runs the number of threads provided by the client to execute the tasks.

In the remote worker case,We first create a SQS queue that we fill with the jobs from the workload. We also create a DynamoDB table in order to help workers to avoir to run a taks twice because of the replication of data inherent to the Amazon SQS queuing service.

In that database we store the taks Id and an integer initialized at zero and which is set at 1 iby a worker when he pulls the task from SQS. If the workers find a task which is already set to 1, the workers give up the task and pulls another one from the queue.

Finally the scheduler polls the results from the workers and sends it to the client through the socket

2.3 Local Back-End Workers

The local back end workers has been implemented within the WorkerThread class. These workers threads take in input the blocking queue filled with the tasks from the workload, and fills the result queue after running the sleep corresponding sleep tasks.

Those worker threads are used in the threadpool in the local back-end which is the scheduler. While the threads pull the tasks from the blocking queue, they add a "kill pill" to this queue to remove the tasks which have been pulled already.

As we previously said, the number of threads to run the tasks is set from the scheduler command line and used in the threadpool.

2.4 Remote Back-End Workers

We implemented the remote back-end workers in the SQSWorkers class. The workers first pull a message from the SQS queue corresponding to a task. Then they look into the DynamDB table implemented in the TaskTable class:

if the task_done field is set on 0 they set it on 1 and they run the task, otherwise they give up the task and pull another one from the SQS queue. When a worker pulls a message from the queue, it deletes it right after to avoid replication.

All the EC2 instances which are actually the remote workers will run using an AMI already configured to run the SQSworker class main method. These EC2 instances are working while they don't reach the idleness threshold which is set up to 30 by default. We are to speak about the dynamic privisionning of the workers in a further section.

2.5 Duplicate Tasks

This section just sums up what we already explained about the replication of tasks inherent to Amazon SQS service. In fact these queues are replicated for resilience purpose across diferent locations.

When a worker pulls a task and then removes the message from the queue, there is a chance that another worker has the time to pull a replicated message from the queue before it is actually removed.

That's why we need the use of a simple NoSQL database to keep track of the status of each task and make sure that each worker only launch a task once in order to make our framework scalable.

2.6 Dynamic Provisioning of Workers

The dynamic provisioning of workers is aimed at keeping a reasonable ammount of instances working given the workload size. If the queue is long, new workers will be created. On the contrary, we already explained that idle workers shut down themselves after reaching the idleness threshold if there is too much workers for a given workload.

We implemented the dynamic provisioning of the workers within the DynamicWorkers class. This object periodically watches the size of the SQS queue and runs new EC2 instances when the size of the queue increased from the last poll.

2.7 Animoto Clone

3 Performance Evaluation

3.1 Throughput and Efficiency

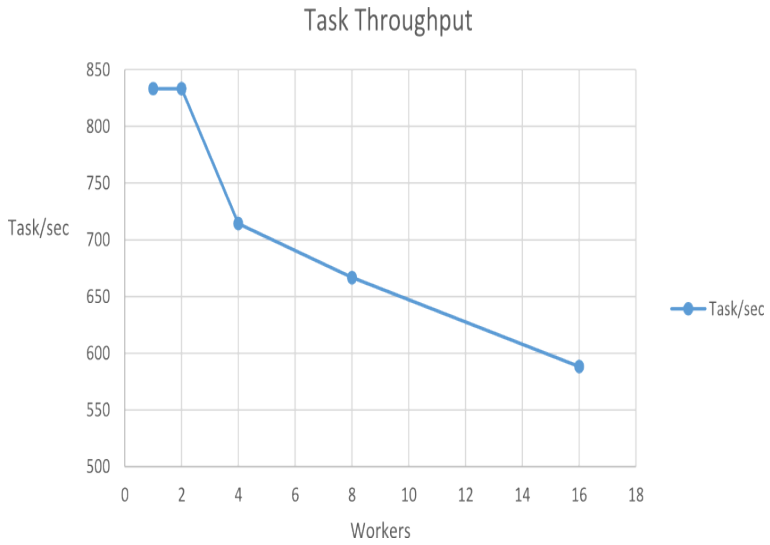

Figure 2: Throughput

Figure 3: Efficiency

3.2 Dynamic Provisioning

3.3 Animoto