Imagely: A cloud environment for online image processing

Authors: SB Ramalingam Santhanakrishnan (4740270), K Kleeberger (???) and B Jain (???)
ICT Innovation, EEMCS, TU Delft
Emails: {S.B.RamalingamSanthanakrishnan, K.Kleeberger, B.Jain}@student.tudelft.nl

Course Instructors: DHJ Epema, A Iosup and A Kuzmanovska
PDS Group, EEMCS, TU Delft
Emails: {D.H.J.Epema, A.Iosup, A.Kuzmanovska}@tudelft.nl

Lab Assistant: BI Ghit
PDS Group, EEMCS, TU Delft
Email: B.I.Ghit@tudelft.nl

Teaching Assistant: A Parthasarathy
EEMCS, TU Delft
Email: A.Parthasarathy@student.tudelft.nl

***Abstract*—Imagely is a cloud service for batch image processing, designed for the specific requirements of WantCloud BV. Images are submitted by the end user, which are then processed by the service and the user is notified. It runs on Amazon Web Services (AWS) IaaS barebone compute resources to provide elastic scaling capabilities and its system design adapts the standard master-worker batch processing reference architecture pattern provided by AWS. The scheduler and provisioner are written in NodeJS in reactive style. We run three workloads each on three combinations of VM and scheduling policies with a hundred images. We show that it can achieve a speedup of up to ?? at peak load, compared to the baseline performance of a constant, pre-allocated compute resources. [ADD ONE MAIN RESULT]**

I. Introduction

WantCloud BV is a popular image processing company and are pioneers in converting raw satellite images to standard image formats consumable by users on the web browser. As the company is expanding into other domains and adding new customers rapidly, their existing deployment unable to cope up with peak traffic demands and maintain service level agreements (SLAs). Thus, WantCloud BV is evaluating an IaaS-based solution that can scale with the demand.

The proposed system utilizes Amazon Web Services (AWS) [1] IaaS cloud service provider's compute instance, EC2 (Elastic Cloud Compute) for providing on-demand provisioning of compute capacity to process the workloads. We design a custom provisioner and scheduler using the AWS low-level SDK EC2 API for requesting and releasing instances (VMs). We also utilize Amazon Machine Images (AMIs) for pre-packaging the application code, configuration information and the operating system for the instance to save time during its launch. For image processing and conversion, we use the ImageMagick [2] program. We wrap ImageMagick with a thin HTTP server written in NodeJS [3], which forks ImageMagick as a child process and performs read/write through stdin/stdout pipes, the instances which perform image processing are known as the workers. For analysis purposes, we restrict the scope of the worker to scaling down, rotating and converting the image from TIFF [4] to JPEG format [5].

The system design follows the AWS reference architecture for batch processing [6], except that we have implemented custom reactive scheduler and provisioner and have placed all the task queues and resource management responsibilities on the master node. The master node accepts the image processing task requests via HTTP and launches multiple worker nodes as per the number of tasks based on the given policy, balances and allocates tasks them and also monitors the progress. It also keeps building a report on completion of each task and launch/release of each instance during its lifetime. The policy targets the provisioner with parameters for minimum and maximum number of workers to launch and the scaling-up decision threshold.

The test workload is a collection of 100 satellite images from Landsat satellite, with file sizes ranginge from 40-60 MB. To determine the baseline (current) effort required, we run the workload on a static VM deployment and to analyse the scaling capability of the proposed system, we test the workload with two different thershold policies to evaluate the provisioner. The workload itself is varied with three exponential Poisson mean inter-arrival time distributions to simulate realistic request patterns.

In section II, additional background information on the application is provided. In section III we illustrate the design of the system in detail with focus on specific aspects pertaining to WantCloud's requirements and in section IV we present the experiments conducted along with the results and analysis. We conclude the report in section refconclusion with a short discussion on the findings and potential improvements.

II. Application

The web-based application receives tasks from end-users in JSON format [7], which point to a source image URL and defines a sequence of image manipulation operations such as scaling, rotation and finally file format conversion from TIFF to JPEG. This JSON request is forwarded to a worker in the resource pool by the master node. Internally, ImageMagick program is used for processing the image and it is invoked by a the worker's NodeJS web server. The end-user requests reach the master node, which queues the job, picks an idle VM and allocates the job to it, based on the allocation policy. In case an idle VM is not found, the provisioning policy requests AWS for a new instance and adds it to the worker pool when it becomes ready.

A. Requirements

In order to replace WantCloud's current system, the following requirements are to be met by the new system.

- *Automation*: The system should be automated for creation, provisioning and other activities with minimal human intervention.

- *Auto-scaling*: The system should be able to detect fluctuations in demand and scale up or down the VM resource pool automatically.
- *Load Balancing*: The system should be able to best utilize the available pool of compute resources for the workload through allocation policies.
- *Reliability*: The system should be able to retry failed tasks, ensure availability through redundancy and recover from unexpected failures.
- *Monitoring*: The system should be able to keep track of resource usage and provide information on metrics to aid future decisions.

III. System Design

At a high level, the system follows a simple client-server model where end-users send job requests to the server in JSON, specifying the image source and the manipulations to be performed. The server accepts the requests, creates a job and adds it to the pending job queue. The job is then dispatched by the scheduler to an available worker at a later point in time. The workers finish the job and intimate the end-user via E-mail about the job completion and where the result could be found, simultaneously informing the master about the job completion and awaiting the next job.

A. Overview

The server consists of the following components:

- *Listener*: The NodeJS server, which accepts client requests through HTTP, parses it, creates a corresponding internal job representation and adds it to the pending job queue.
- *Scheduler*: The scheduler looks for a free VM in the resource pool and schedules a pending job onto the identified VM according to the given job allocation policy.
- *PoolManager*: This component which manages the provisioning and revocation of VMs in the resource pool according to the given scaling policy.
- *Monitor*: The monitor is responsible for monitoring the finished/failed jobs, health of the instances and maintain record, which is served at a HTTP endpoint, summarized for metrics.

The implementation of the scheduler and the pool manager is generic such that it is trivial to port it to other cloud providers by interfacing with the corresponding SDKs or APIs. The various policies are also defined in JSON format and the system is extensible such that the policies can be dynamically swapped through HTTP endpoints.

B. Resource Management Architecture

The system comprises of the following concepts which interact to achieve efficient resource management:

1) *Job*: As soon as the job moves from the pending queue to the active queue, it is submitted to the worker, which performs the following three steps,
   a) Downloading the source image.
   b) Execution of image manipulation and conversion.
   c) Writing the output image to data store.

   The above steps, in addition to the time taken in submitting and receiving the completion notification by master is counted into the job's makespan. When the job is marked as finished, it is removed from the active queue and reporting information such as the finishTime and the ID of the instance which performed the job is recorded and stored by the monitor module. In case a job fails or does not finish within the given timeout period, the job is again added to the pending queue with an incremented retry count, which has a policy-defined upper limit.
2) *Worker Nodes*: In AWS, after a request for a new VM instance is made, it enters the PENDING state and it is usable only when it reaches the RUNNING state. However, at the running state, additional setup needs to be performed in order to start executing our business logic, at which point the VM is said to be completely ready and usable. We reduce the time required for the VM to be ready by booting using pre-defined VM images, provided as a service by AWS, AMI [8]. The worker application implements a health check endpoint, which returns the cpu and memory stats. The successful response of this endpoint indicates that the worker is ready to accept job requests.
   The master node, asynchronously requests for a new worker instance and polls the health check endpoint of the VM until it responds, to determine its readiness. Once the worker is ready, it is added to the worker resource pool. If the worker is not able to reach the ready state within 1 minute, it is terminated.
3) *Scheduling*: The scheduler is invoked periodically to perform the following tasks,
   a) Count the available worker instances.
   b) Pick as many jobs from the pending queue as there are worker instances (this could be lesser), prioritized by earliest arrival time and retry count.
   c) If there are lesser jobs than workers, mark the surplus workers for termination, otherwise, request provisioner for an additional worker.
   d) Move matched jobs to active queue, asynchronously send job request to workers and monitor for completion.
   e) When a job completion notification is received, move the job out of the active queue and let monitor record the event. If a job fails, it is added back to the execution queue with incremented retry count.

   Note that the scheduler incrementally requests for new instances at each round. Also, the additional instances are not terminated until the next round by the provisioner, in case they are required again by the system. This avoid thrashing of VM provisioning activity.
4) *Reliability*: As mentioned in the previous point, scheduling already takes care of retrying failed tasks. As far as the master node is concerned, the entire system state, including the statistic is held in a global immutable container, which can be easily checkpointed. Although we

have not done it for feasibility reasons, the master itself could be replicated and deployed in different availability zones and the traffic could be evenly distributed.

C. System Policies

In the system, we have implemented one allocation policy and two provisioning policies.

1) *Allocation*: The scheduler implements the first come, first serve (FCFS) policy. It picks the job with the earliest arrival time and maximum retry count for allocation.
2) *Provisioning*: Provisioning policies fall under two classes, STATIC and DYNAMIC. We implement the CONSTANT-5 static policy where the entire workload is handled by a constant number of workers (count 5) and under the dynamic class, we implement ON-DEMAND SINGLE VM (OD-S), which allocates one VM per job and releases the VM after job completion plus a wait time (10 seconds in our case).
   The configurable parameters of the OD-S policy are,
   - *min-vms*: The minimum number of VMs the system must have at any point.
   - *max-vms*: The total number of VMs the system can provision at most.
   - *threshold*: The trigger value for provisioning activity, in pending queue length.

   When the pending queue length falls below the threshold, the scheduler marks instances to be terminated and the provisioner executes the termination after the wait period.

IV. EXPERIMENTAL RESULTS

We perform all the experiments on the AWS cloud platform, restricted to a single region North Virgina (US-East-1) with default availability zone and the default virtual private cloud settings. A custom security group is used for exposing ports 8000 (master server), 3000 (worker server) and 3001 (worker health check) for inbound TCP (HTTP) traffic.

A. Experimental Setup

We have extended our custom AMI from the base public Ubuntu image provided by Amazon (id = ami-cd0f5cb6). Workloads have been created from the publicly available image files of NASA's Landsat-I satellite (x images, total y MB). The workloads are varied by arrival time distribution in order to evaluate the provisioning policies. Each job in the workload requests the given image to be scaled down to 25% of the original size, rotated by 90 degrees clockwise and converted to JPEG. We generate and send the workloads to the system using custom scripts. The instance type we use for both master and worker is m3.medium (general purpose compute, vcpu=1 and memory=3.75Gb).

B. Experiments

The goal of our experiments is to measure various workload execution and VM metrics on running the static policy and the dynamic policies and analyse the results to understand the inherent tradeoffs in the elasticity of cloud environments. [1]

1) Baseline: We determine the makespan of the three workloads on a single unit (SU) instance to analyze the relative speedups and cost-tradeoff in later experiments.

2) Overheads: We identify a major overhead in the system due to the time taken for a newly launched VM to reach the ready state. To measure this, we repeatedly request, wait till ready and release a single VM instance for 25 times.

3) Provisioning Policy: We exercise three workload variants, linear, bursty and poisson across the three allocation policies,

- CONSTANT-5, constantly 5 VMs.
- OD-S-0, threshold set to 0, 2 to 20 VMs.
- OD-S-5, threshold set to 5, 2 to 20 VMs.

4) Utilization: Utilization is the ratio of the total time spent by the workers in executing the jobs to the total lifetime (including boot time) of the workers, described by equation 1.

$$Utilization = \frac{\sum T_{execution}}{\sum T_{lifetime}} \quad (1)$$

5) Speedup vs Cost Tradeoff: We summarize the results obtained from the previous experiments and compare the cost and speedup of the policies on the workloads.

V. CONCLUSION

REFERENCES

[1] Aws.
[2] Imagemagick.
[3] Nodejs.
[4] G Parsons and J Rafferty. Tag image file format (tiff). RFC 3302, September 2002.
[5] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, pages 30–44, 1991.
[6] Amazon Web Services. Batch processing.
[7] T. Bray. The javascript object notation (json) data interchange format. RFC 7159, March 2014.
[8] Aws ami.

APPENDIX A: TIME SHEETS

TABLE I
TIME SPENT ON THE PROJECT PER ACTIVITY.

Activity	Time (hours)
Total	
Think	
Dev	
XP	
Analysis	
Write	
Wasted	

[1] TODO: for each experiment, list *Charged-time*, *Charged-cost* and *Service metrics of the experiment*

TABLE II
TIME SPENT PER EXPERIMENT.

	Time (hours)		
	Dev	Setup	Total
Baseline			
Overheads			
Provisioning			
Utilization			
Speedup			