Imagely: An elastic 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, B.Jain, K.Kleeberger}@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: Bogdan Ghit
PDS Group, EEMCS, TU Delft
Email: B.I.Ghit@tudelft.nl

***Abstract*—In this report we introduce *Imagely*, a cloud service for image processing. It runs on top of Amazon Web Services' barebone compute resources to provide elastic scaling capabilities and through benchmarks and experiments, we show that it can achieve a speedup of up to ?? at peak load, compared to the baseline performance of a static single compute resource. (TODO: Add more points on design and analysis)**

I. Introduction

WantCloud BV is an image processing company and are pioneers in processing raw satellite images to a standard image formats, consumable by the end-user on the web browser. As WantCloud BV is expanding into other domains and adding new customers rapidly, their existing solution is not able to cope up with peak traffic demands and maintain the service level agreements (SLAs). Thus, WantCloud BV is evaluating a 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. AWS provides low-level SDKs for provisioning and revoking virtual machines (VMs) on which we place the workloads. We also utilize Amazon Machine Images (AMIs) for pre-packaging the application code, configuration information and the operating system for the virtual machine. For image processing and conversion, we use the ImageMagick [2] program. We wrap ImageMagick with a thin HTTP server written in NodeJS [3], which invokes ImageMagick as a child process and uses stdin/stdout pipes for input/output. The VM which performs image processing is known as the worker node. For analysis purposes, we restrict the scope of worker to scaling down, rotating and converting the image from TIFF [4] to JPEG format [5].

The proposed solution consists of a master node, which accepts the image processing task requests via HTTP and spawns multiple workers as per the policy, balances and allocates tasks them and also monitors the overall system statistics. The total time it takes to perform tasks on an image by the worker is known as the *makespan* of a given job. VMs are leased on an hourly basis, thus we also measure the cost for a given job and analyse the tradeoffs in adding more VMs to the system.

To determine a baseline, we run a test workload on a single VM and to analyse the scaling capability of the system, we run workloads with varied arrival time distributions to simulate realistic scenarios.

In section II, additional background information on the application is provided. In section III we illustrate the design of the system 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 which we build takes jobs defined in JSON format [6], which point to a image URL and defines a sequence of basic image manipulation operations such as scaling, rotation and finally file format conversion from TIFF to JPEG. We internally use the ImageMagick program for processing the image and it is invoked by a 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 guides the master to provision a new VM and add it to the worker pool.

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:

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 [7]. 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 to be provisioned at each scheduling round. Also, the additional instances are not terminated until the next round by the provisioner, in case they are required again by the system.

C. System Policies

D. Additional System Features [OPTIONAL]

IV. Experimental Results

A. Experimental Setup

B. Experiments

1) *Charged-time*:
2) *Charged-cost*:
3) *Service metrics of the experiment*:
4) *Usage metrics of the experiment (OPTIONAL)*:

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] T. Bray. The javascript object notation (json) data interchange format. RFC 7159, March 2014.
[7] Aws ami.

APPENDIX A: TIME SHEETS

[TODO: Restructure this section onto a table]

Project:

1) *Total time*:
2) *Think time*:
3) *Dev time*:
4) *XP time*:
5) *Analysis time*:
6) *Write time*:
7) *Wasted time*:

Per Experiment:

1) *Total time*:
2) *Dev time*:
3) *Setup time*: