Report for IN4392 Large Lab

Authors: Lipu Fei and Hans van den Bogert
Computer Science, EEMCS, Delft University of Technology
Emails: {l.fei, j.c.vandenbogert}@student.tudelft.nl
Course Instrutors: Alexandru Iosup and Dick Epema
Parallel and Distributed Systems Group, EEMCS, Delft University of Technology
Emails: {A.Iosup, D.H.J.Epema}@tudelft.nl

***Abstract*—In this report, we introduce a cloud management platform that provides an online video-transcoding application for users. The platform, which we call "WantCloudFrame", utilizes elastic cloud resource provisioning and allocation to make it scalable in performance.**

I. Introduction

Current situation: For WantCloud providing transcoding[1] facilities have been a great source of income. Due to the popularity of the current system, it is overloaded because it does not scale well during peak usage. In the existing solution there is only one machine which handles incoming jobs. The overloaded system therefor has a relatively high number of outstanding jobs causing it to not meet the deadlines which are guaranteed by WantCloud's Terms-of-Use. To circumvent the shortcomings of using only 1 physical machine we will look into using a IaaS as our platform.

Related Work: The pre-existing cloud environment used for the proposed system is the [**?**] cluster with the [**?**]-stack on top of DAS4. OpenNebula provides a low-level interface for spawning *Virtual Machines* [VM] on which our workload can then be placed.

For the actual conversion of the media files the [**?**] program is being used. For the sake of implementation feasability only the conversion from [**?**] to DVD-Video is considered. This software is freely available for everyone to use under the GPL-license.

As a method for inter-machine communication in the cluster, SSH is being used. SSH is available in every spawned machine by default and provides us the means for secure communication and file-transfer.

Proposed solution: To be able to cope with demand, a new system setup has been made which can tap into a pre-existing cloud environment, to scale during peak usage and thereby load-balance the workload over multiple machines. For the experiment we've looked at multiple methods for allocation of the machine resources. By keeping statistics in our implementation we track the total time it takes for a submitted media-file to be transcoded and sent back to the submitter. This metric will hereafter be called the *makespan* of a submitted job. Another metric is the cost for a job. Leasing a VM in the cloud costs money – we investigate the tradeoffs between leasing more VMs and the effect on the makespan.

@TODO more metrics

To experiment with the proposed solution - a benchmark has been created which uses a predefined sample from a exponentional distribution to simulate arrival times for jobs.

Overview: In the next section we will elobarate more on the application and provide more background information. In section III the system's design will be handled so the experiment in section IV can be understood. After the experiment we will discuss the findings in section IV and conclude in section VI.

II. Background on Application

For the experiment

III. System Design

A. Overview

From the big picture, our system is designed in a client/server way, where clients send job requests to the server using a simple customized language, which specifies things such as which application to use, which input files to use, etc., and the server parses the requests, creates a corresponding job and then put it into the pending job queue. The job is then waiting for dispatched by the scheduler.

The server part of the system consists of the following components:

Listener: The listener is responsible for accepting client requests through socket connection, parsing them, and put the jobs into the pending job queue.

Scheduler: The scheduler is the core thread in the system. It does scheduling according to the given allocation and provisioning policies and updates current system status. It will be explained in more details in the later part.

ResourceManager: This component maintains the VM instances on the cloud.

StatisticsManager: This module maintains all the statistics data of the system, including job performances, VM instance performances, etc. It also generates a final report when the system is shutting down.

Our system is currently only runnable on DAS-4 OpenNebula platform, but it can be easily modified and extended to support other cloud platforms because its good flexibility. It is easy to create new allocation and provisioning policies. All configurations are done through a configuration file.

B. Resource Management Architecture

Some features must be explained before we go to more details.

1) Jobs: There are three job queues in the system: *pending job queue*, *running job queue*, and *finished job queue*. It will be executed when it is assigned to a VM instance. Its execution procedure is as follows:

1) Downloading required files (in our case, executable tarball and input files).
2) Extracting the executable tarball.
3) Executing the job (converting H.264 video file into NTSC-DVD).
4) Uploading the resulting DVD file to the server.

[1]Transcoding: The process of converting a media file from one format to another

All these operations are done through SCP and SSH. The job thread is also responsible for collecting performance data, including downloading time, execution time, uploading time, etc., and once this job is done, it will be put into the *finished job queue* with its state set to FINISHED. Later scheduler will remove this job and update the statistics. If a job fails, its state is set to FAILED and also put into the *finished job queue*. The scheduler will handle these failed jobs according to the given policy.

Although the parameters are configurable for doing other tasks, our design is not that flexible because this execution sequence is hard coded in the program. We will consider using script files for job execution.

2) VM instances: The VM instances are created asynchronously. In OpenNebula, after a VM instance is allocated, its state becomes PENDING, and you need to wait until it is RUNNING. At this point, the VM instance is actually running. However, it is not actually "ready" because when it becomes RUNNING, it still needs some time to boot the OS and initialize the system. Only after the system is fully booted can the VM instance be accessed using SSH to execute jobs.

In our system, we use an asynchronous way to create a VM instance: a thread called "VMAgent" is created every time the system allocates a VM instance. This thread first allocates the VM instance, and then does the following things:

1) VMAgent waits until this VM instance becomes RUNNING.
2) It waits until the VM can be reached by ping.
3) It waits until the VM can be reached by SSH.

After the VM is reachable, it is added into the VM list in the ResourceManager, and it becomes available for jobs to execute on.

Each VMAgent has a timeout of two minutes. If the operation times out, the VM instance will be terminated.

3) Scheduler: What the scheduler does is as follows:

1) It does some regular checks and updates.
2) It picks a job from the pending queue using the specified job allocation policy.
3) It picks a VM instance using the specified resource provisioning policy.
4) It executes this job on this VM instance.

In the regular check part, the scheduler does miscellaneous tasks, including:

- Updating job states, VM instance states, and system states.
- Checking the *finished job queue*. If a job is finished successfully, it is removed and its data is used to update the statistics. If it failed, it will be handled according to the given allocation policy. For now, the system just puts failed jobs into the *pending job queue* again regardless of how many times it has failed.
- Calling the provisioning policy to perform elastic provisioning.

In this way, the statistics of the system is continuously recorded, and flexible allocation policies and elastic provisioning policies can be implemented.

4) Reliability: Our system keeps tracking on every job and VM instance. During the shutdown sequence, it first stops all executing jobs. Then, it terminates all allocated VM instances, so that no VM instance would be running afterwards. After that, it checks jobs in all the queues, and put them into the statistics module, which finally creates a report and outputs to a file.

Besides that, our system also keeps separate log files for the system itself and each jobs respectively. So it is easy to debug the system.

C. System Policies

We have implemented one job allocation policy and two resource provisioning policies.

1) Allocation Policy: The job allocation policy we implemented is a First-Come-First-Serve (FCFS) policy. It always picks the job in the pending queue with the earliest arrival time.

2) Provisioning Policies: We implemented two provisioning policies: STATIC policy and SIMPLE ELASTIC policy.

STATIC policy allocated a specified number of VM instances when the system starts, and this number will not vary over time. This policy can not adapt itself to the changing environment.

SIMPLE ELASTIC policy is on-demand-like elastic policy, which has three parameters:

- *minvms*: The minimum number of VM instances the system must have.
- *maxmvs*: The maximum number of VM instances the system can have.
- *threshold*: A threshold value, which will be explained later.

Each time the system statuses have been updated by the scheduler, this provisioner is called, and it checks the number of pending jobs. If this number is larger than the *threshold*, it allocates one more VM instance until either the total number of VM instances in the system equals the sum of pending jobs and running jobs or it reaches the *maxvms*. However, when the pending job queue becomes empty, this policy will find one idle VM instance and release it.

It elastic policy is a simple on-demand policy is because it doesn't take into account how long a VM instance has been idle. It simply terminates one if it is currently idle. A main drawback of this policy can be illustrated in this scenario: Suppose there are 7 jobs running, no pending jobs, and 8 VMs in the system. So this policy will remove one idle VM. If a job arrives right after the policy removes a VM, this job has to wait for a long time to obtain an available VM.

IV. Experimental Results

A. Experimental Setups

We carried out our experiments on DAS-4 using OpenNebula platform. The VM image we use is CentOS-5.4 whose image ID is 35, and our VM setup is CPU=1, VCPU=2, with 1024MB memory.

TABLE I
JOBS FOR TESTINGS

Job Name	File Name	Encoding	File Size
job1	cloudatlas-trailer1b_h1080p.mov	H.264 1080p	172MB
job2	skyfall-tlr2_h1080p.mov	H.264 1080p	180MB
job3	taken2-tlr1_h1080p.mov	H.264 1080p	175MB

TABLE II
WORKLOADS FOR TESTINGS

Workload Name	Number of Jobs	Job Arrival Time Distribution
wl-10	40	Exponential(10)
wl-20	40	Exponential(20)
wl-30	40	Exponential(30)

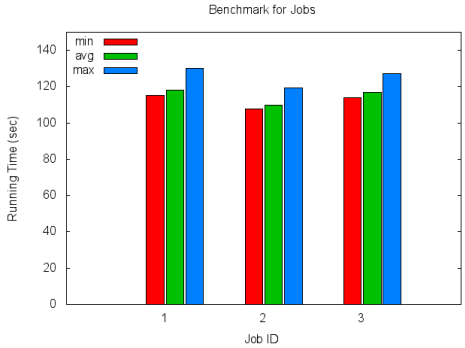

Fig. 1. Results of the benchmark tests for jobs

We prepared three H.264 1080p movie trailers downloaded from Apple Trailers[2], which are described in Table **??**:

For testing the provisioning policies, we created three different jobs with respect to these three files. Each job is to convert the H.264 1080p video file into an NTSC-DVD file. We then randomly generated three workloads, each of which comprises of 40 jobs. The job arrival times are generated using exponential distribution with three different parameters: 10, 20, and 30 seconds.

B. Experiments

1) Benchmark Tests: First, we did some benchmark tests to measure the performance of DAS-4 OpenNebula. The benchmark consists of four tests:

1) The first three tests are using ffmpeg to convert the three video files separately. Each test was done by 20 times, and we measure the job's running times (the total amount of time a job spent on its execution).
2) Another test is continuously allocating and releasing a VM instance, in order to measure the overhead of allocating a VM instance on DAS-4.

As illustrated in Figure **??**, it is obvious that the performances of all jobs are almost the same, with an average of 114.819 seconds. The maximum and minimum values are also close to the averages.

For the VM allocation overhead test, we continuous allocate and release a VM instance for 20 times, and we measure its total preparation time, which is from the time it is allocated until the time it is accessible through SSH. The results are shown in Table **??**.

[2]http://trailers.apple.com/trailers/

TABLE III
TOTAL PREPARATION TIME OF VMS

Average Total Preparation Time	60.309 seconds
Average Allocation Time	≈10 seconds
Average OS Booting Time	≈50 seconds

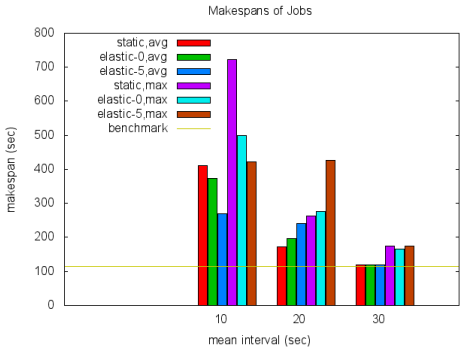

Fig. 2. Makespans of jobs in the three workloads

We only show the average values because the minimum and maximum values are close to it. The allocation time is from the VM instance is allocated until its state becomes RUNNING, and the OS booting time is the time after that and until this VM is accessible through SSH.

The results suggest that the DAS-4 OpenNebula is efficient in allocating a VM instance. It only takes approximately 10 seconds to become RUNNING after it is allocated. However, it still needs 50 seconds to be able to use.

C. Experiments on the Provisioning Policies

In this part, we use the three workloads to test the two policies. We will just call STATIC for static policy, and SE for simple elastic policy for short, and SE with threshold set to 0 will be denoted as SE-0. Figure **??** illustrates the job makespans using different policies in each workload.

The brown line there is the benchmark line indicating the average running time of a job. As expected, for wl-10, SE out performs the STATIC both in average case and worst case, and SE-5 has a much better performance than SE-0. For wl-30, the performances are almost the same. This suggests that five VMs are enough for wl-30.

What is worth noticing is that in wl-20, STATIC has the best performance. One explanation is that, our SE policy terminates an idle VM instance once the pending job queue becomes empty. In wl-20, the arriving of jobs is neither frequent, nor slow. So the pending job queue could get empty for several times. This scenario becomes a bottleneck of our SE policy, especially SE-5, because it only allocates more VMs when the number of pending jobs gets higher than 5.

Figure **??** shows the waiting times of jobs. In wl-20, ES-0 has a lower waiting time than STATIC, however the overall makespan is higher. The makespan may be dragged down by the uploading and download overheads.

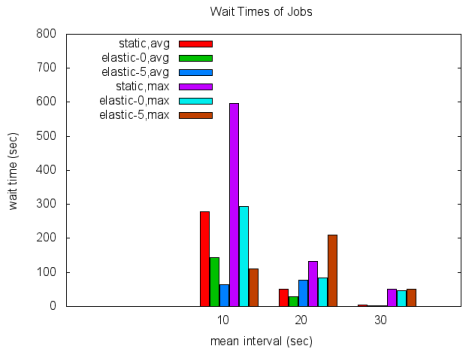

Fig. 3. Waiting times of jobs in the three workloads

Fig. 4. Total makespan of three workloads

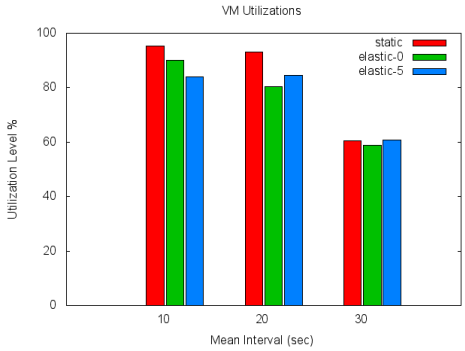

Fig. 5. VM Utilizations Levels

Looking at the total makespans of each workload in Figure **??**, we observe that in `wl-10` and `wl-20`, SE works better than STATIC.

D. VM Performances

We analyze the utilization of VMs in our system in this part. The utilization level is the fraction of time VMs spent on running jobs with respect to their total lifetime, and it can be described by Equation **??**.

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

The results are depicted in Figure **??**. Although in `wl-10` and `wl-20`, the utilization levels of SE are lower than

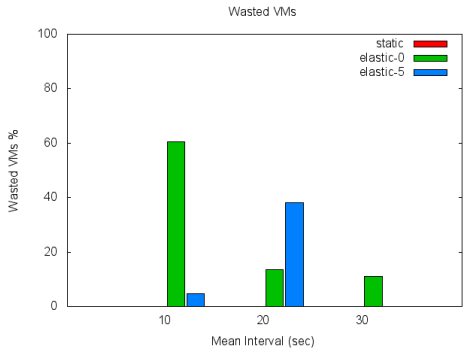

Fig. 6. Wasted VMs

TABLE IV
CHARGED-COSTS

Workload	STATIC	SE-0	SE-5
`wl-10`	1.54hrs ($0.246)	2.86hrs ($0.458)	2.73hrs ($0.437)
`wl-20`	1.44hrs ($0.230)	2.32hrs ($0.371)	2.13hrs ($0.341)
`wl-30`	2.13hrs ($0.341)	2.18hrs ($0.349)	2.13hrs ($0.341)

STATIC, the decrements are acceptable. This suggests that SE can utilize the VMs relatively efficiently.

Due to SE's feature that it terminates an idle VM as soon as there is no pending jobs, we would also like to analyze its drawback and overhead. For doing this, we measure the number of wasted VMs by using SE. A VM can be wasted in this scenario: suppose we are using SE-0, and there are 5 VMs in the system with 5 jobs running on them respectively. Then a new job comes, and SE-0 will allocate a new VM, say VM_{new}. However, before VM_{new} becomes available, one running job has been finished, which means a VM becomes idle, and this new job is then assigned to this free VM. So, when VM_{new} is ready, the pending job queue is empty, and according to our policy, VM_{new} is terminated. In this case, no job has been running on VM_{new} during its lifetime, and we say that VM_{new} is wasted.

In Figure **??**, we can see that in `wl-10`, SE-5 gets a surprisingly low number of wasted VMs, while the number of SE-0 is as high as 60%.

E. Speedup vs. Cost Tradeoff

In this part, we calculate the speedups of using our SE provisioning policy and the charged-costs. Because our VM setup has no corresponding machine on Amazon Web Service, we use the charged cost of m1.medium ($0.16 per hour for Linux), which has two compute units, for our charged-cost calculation. Table **??** shows the results.

We mainly focus on the results of `wl-10`, because this is the only case that SE outperforms STATIC. From Table **??**, we can see that only SE-5 is close to cost-speed proportional

TABLE V
SPEEDUPS AND COSTS FOR WL-10

	SE-0	SE-5
Speedup	1.28	1.52
Cost	1.86	1.78

TABLE VI
TIME SHEET TABLE FOR THE WHOLE LAB EXERCISE

Parts	Time
total-time	
think-time	
dev-time	
xp-time	
write-time	
wasted-time	

in `wl-10`. The results suggest that the cost is usually higher than the performance you can gain. It is not an easy task to achieve a cost-efficient cloud application.

V. DISCUSSION

This is discussion section.

VI. CONCLUSION

This is conclusion section.

APPENDIX
TIME SHEETS