Large Lab Exercise Cloud Computing: Video Converter on AWS

13-11-2012

Author:
N. Singh
N.R.N. Timal
n.singh-2@student.tudelft.nl
n.r.n.timal@student.tudelft.nl

Support cast:
D.H.J. Epema
B.I. Ghit
A.Iosup
d.h.j.epema@tudelft.nl
b.i.ghit@tudelft.nl
a.iosup@tudelft.nl

Abstract

This report provides a use case for development of WantCloud BVs new cloud based application. For this, we develop a client-server based video converter application. The server of this application is deployed on Amazon Web Services(AWS) and uses its Elastic Compute (EC2) service. The system is developed as an IaaS based system which uses computing resources leased from the Amazon cloud (AWS). The developed IaaS system has a number of features like automation, elasticity, performance measuring, reliability, security and monitoring ability. These features are implemented using a python script which communicates with AWS using an interface called boto. These features are tested by performing several experiments which also run as python scripts, for our experiments we use m1.small instance type of AWS EC2. The application is not limited to this instance type and can be used with every other instance type. According to the experimental findings, we see that the cost and charge time of the application can be really high if it is used by 1 million user concurrently, as a suggestion we recommend using the cloud only if the WantCloud BV can afford it or is using CPU intensive specific instance types as this application has high CPU usage peaks.

1 Introduction

This document is constructed in response to WantCloud BVs interest in moving a new application to the cloud. To this end the CTO of WantCloud wants us to create a (relatively) simple application to identify what types of trade-offs exists when deploying an application on the cloud.

We chose Amazon Web Services (AWS)[1] as our cloud provider because it is one of the major companies providing resources to external parties to date. If WantCloud BV will follow through with their wishes to deploy an application in the cloud most likely AWS will be one to seriously consider. As part of AWSs policy it offers new users of the cloud a free usage tier[2]. For our system we solely used the accommodations provided by the free usage tier, in particular the free use of Elastic Cloud Computing(EC2) micro instances and Elastic Load Balancer.

We created an application which allows users to convert a video file to a desired format using a client-server model[3]. Although this may seem to be a simple application its workflow can contain very demanding operations. That is client(/user) needs to upload a video file to the server, the server in turn processes the uploaded file and converts and returns the converted file to the client. The client then stores the converted file to its local file system. We therefore find this application to be fitting for assessing the tradeoffs inherent in cloud based applications.

Our cloud based system will implement the following. Automation, which means that a servlet will automatically be assigned to a EC2 micro instance. Auto-scaling, that is depending on the amount of client requests addition (when CPU usage of existing instances are high) or removal (when CPU usage of an instance is below a low threshold and can be compensated by an existing instance) EC2 micro instances. Load balancing, equally dividing the work between instances using Round Robin scheduling[4]. Reliability is achieved attaching by every micro instance to a persistent storage volume. And last but not least monitoring which will provide a comprehensive view of e.g. the amount of resources used by each live micro instance. The implementation of the aforementioned features will be accomplished by interfacing AWS with Pythons boto framework. The code of this framework can be found at this location[5]. This interface extends among other components of AWS to EC2 and Simple Storage Service(S3) which will be used by our cloud based system.

The rest of the report is organized as follows. Section 2 gives a more in-depth view of the application that is implemented. Section 3 discusses how the features of the cloud based system are implemented (automation, auto-scaling, load balancing, reliability and monitoring). The experimental setup and the experiments conducted are discussed in Section 4. Any limitations of our cloud based system based on the experimental results from Section 4 appear in Section 5. We will also conclude in Section 5. This report also contains two appendices. Appendix A gives an overview of the amount of time spend for this project. Appendix B lists the repositories used for the project.

[1]http://aws.amazon.com/
[2]http://aws.amazon.com/free/
[3]http://en.wikipedia.org/wiki/Clientserver_model
[4]http://en.wikipedia.org/wiki/Roundrobin_scheduling
[5]http://code.google.com/p/boto

2 Application

We have constructed a video converter tool in Java[6] which allows an user to specify what file needs to be converted and to what format it will be converted. The conversion from one format to another is done in the cloud. Using a client-server model, the client uploads a file and additionally provides the output format. Whereas the server, which is provided by the cloud, deals with converting the uploaded file. When the conversion is done the converted file is written to the local file system of the client. We have used the Jersey[7]Framework to create a servlet which is constructed using a REST[8] based approach. This servlet contains the algorithm for converting a video and is deployed on a Tomcat server in the cloud. The algorithm uses several methods from the Xuggle[9] framework. This framework is free and open-source and is used for audio/video manipulation. As for the client we used Jersey Client Framework which provides a comprehensive interface for communicating with a server.

2.1 Requirements

In this paragraph we will elaborate what requirements are essential for our application. The requirements can be subdivided into two categories: *functional* and *nonfunctional requirements*. A fitting *functional requirement* for our application is that it needs to convert to and from all commonly known codecs/video formats e.g. Windows Media Video(WMV)[10] and QuickTime MOV[11]. As a *nonfunctional requirement* the servlet needs to be deployed on the cloud, to achieve this goal we will be using the facilities provided by AWS. In particular Amazon Simple Storage (S3) for storing the WAR file created by the Jersey Servlet and Amazon Elastic Compute Cloud(EC2). An EC2 instance will be comprised of our servlets computational logic, that is once the servlet is launched on an EC2 instance, the instance will be able to execute a video conversion.

[6]http://www.java.com

[7]http://jersey.java.net/

[8]http://en.wikipedia.org/wiki/Representational_state_transfer

[9]http://www.xuggle.com/

[10]http://en.wikipedia.org/wiki/Windows_Media_Video

[11]http://en.wikipedia.org/wiki/QuickTime

3 System Design

The online video converter developed for WantCloudBV is deployed using Amazon Web Services (AWS). The video converter is build as a client-server model where the server part resides in the cloud and client is launched on the local machine. The application is developed using Java and we use AWS Java SDK to deploy our servers .war file to AWS Elastic Beanstalk Server (EBS) by uploading the application to Amazon S3. The Elastic Beanstalk environment is setup as Apache Tomcat Server 6 and deployed with a keypair to help in SSH-ing to the instances attached to EBS, the URL of this EBS environment is used by the applications client to send request to the server. The launch of the EBS is associated with the booting of a t1.small instance[12]. Once the server is deployed and running (state of instance), video converter client can be launched to make requests to the server.

3.1 System Objectives

For the video converter application we designed an IaaS based system with the desired properties namely automation, elasticity, performance, reliability, and monitoring. We have used boto(a python interface to Amazon Web Services) to write a python script which implements all the above stated features.

3.1.1 Automation

This python script[13] implements all the features of the IaaS based system without any human intervention. When the server is ready we run this script and then proceed with video conversion requests. The script checks for all the running instances, starts monitoring on each of these instances, automates the scaling of instances based on CPU Utilisation metric, and balances the load on these instances. All the monitoring statistics are gathered in a log file.

3.1.2 Reliability and Durability

The application launches with minimum one instance, and initially attaches an ephemeral EBS storage volume to it, which means if the instance terminates the attached storage volume will be deleted as well. In our system(script) we first find out all the instances and attach each one of them with a unique EBS storage volume which is permanent and is not deleted with termination of an instance. This same volume is then attached to the newly launched instance, thus providing reliability between instance restarts and re-boots.

3.1.3 Monitoring

In the first prototype of our monitoring module, every instance was monitored by first SSH=ing into the instance and running the script which calculated the cpu usage and network I/O metric and stored the results in a file on the instance. This script was run as a daemon on the system. The log file was retrieved from every instance and stored as unique files corresponding to every instance. This approach was however rejected since the command used for calculating cpu usage did not give the correct and reliable results. Find the discussion here [14] [15] If you run a command like top from within an EC2 instance, you will not see an accurate CPU utilization metric for your instance. Your EC2 instance runs inside a virtual machine. The only way to capture an accurate metric for CPU utilization is to get that data from the virtual machine parent. Our CloudWatch metrics capture data directly from the parent of the virtual machine so you can get metrics for your instances. According to the suggestions, we were forced to use Amazons CloudWatch for monitoring our instances. The final design queries the CloudWatch for metrics and stores the

[12]http://aws.amazon.com/ec2/instance-types
[13]https://github.com/nidhisingh88/CC_script
[14]https://forums.aws.amazon.com/thread.jspa?threadID=95288
[15]http://www.axibase.com/cloud/2010/07/22/ec2-monitoring-the-case-of-stolen-cpu/

results in a local file as log, the queries are launched at a gap of 60 seconds. The log file contains the metric with timestamp and instance id.

3.1.4 Load Balancing

The system script looks for the running EC2 instances and attaches them to a elastic load balancer which uses round robin technique to balance load between the instances. All newly launched instances are registered with the load balancer and all terminating instances are deregistered.

3.1.5 Security

Our system uses security groups to implement security of the instances. A new security group is created with traffic allowed only on ports 80(HTTP) and 22(SSH), all the running and newly spawned instances are associated with this security group.

3.1.6 AutoScale

The appication starts with one instance and our IaaS system is capable of autoscaling the number of instances. Every running instance is checked for the average CPU utilization in the past one minute, this value is checked against a minimum and a maximum threshold which are user configurable in the system script. If the CPU utilization exceeds the maximum configured threshold then a new instance is spawned which is a clone of the previous instance. The newly launched instance is configured for monitoring, a new permanent EBS volume is attached to it and it is also attached to the current load balancer. If the CPU usage of the instance is below the minimum configured then the instance is terminated and detached from the load balancer. Our system design also permits to specify the maximum number of instances that should be spawned and never terminates all the instances, there will always be one running instance. Autoscaling is automated except the configuration of initial threshold parameters. Once the system script is started it automatically checks for autoscaling implementation every 10 seconds.

4 Experimental Results

4.1 Experimental Setup

Our application is running on EC2 instances. For all our experiments we used t1.micro instances but found out that it cannot service more than 10 requests and cannot cope with bursts of traffic[16]. We were using t1.micro as it is part of AWS free tier group. So if the company anticipates say 100 users at a time, then they need 10 t1.micro instances and scale up or down accordingly. Hence, we use the next instance type m1.small for our application. The auto-scaling system implementation will only spawn m1.small instances as new instances. For testing each module of the system design, we developed simple python scripts which use threads. We use a 105KB file that will be converted to the desired format. The IaaS system also runs on a python script which uses boto to interface with the AWS.

4.2 Experiments

All the experiments test the automation and monitoring features of the system, we run the IaaS system scripts and then experiment scripts, thus automating the whole process. Monitoring is being done for every instance, by default we start detailed monitoring on each newly launched instance and stores the output in a log file.

4.2.1 Experiment 1

Objective For the first experiment we test our application with 100 concurrent client requests without our IaaS system. we launched 100 threads that individually made concurrent client requests.

Analysis We get a server error, which means that a single m1.small instance cannot handle 100 concurrent requests. Figure 1 displays the CPU utilization for a certain time span of the cloud based system without auto scaling. As can be seen from the figure the instance reaches 100 percent CPU-utilization and cannot service any additional client request.

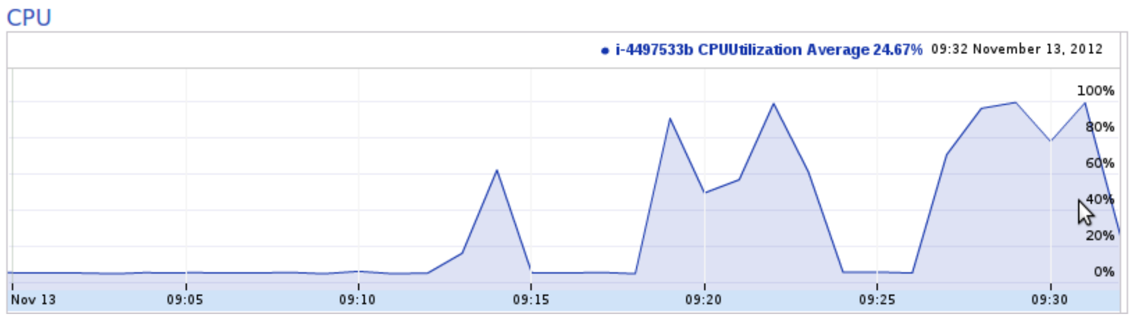

Figure 1: Cloud based system working without auto-scaling

4.2.2 Experiment 2

Objective We want to test the security of the application by implementing security groups. We create a security group through our system script with only 80(HTTP) port and launch our instances attached to that security group. Next we try to SSH to the instance using the keypair we generated from AWS.

[16]http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/concepts_micro_instances.html

Analysis We are not able to SSH into the instance and it shows an authentication error, this is because the instance is not deployed with a keypair and the security group doesnt allow us to use SSH port of the instance.

4.2.3 Experiment 3

Objective Our last experiment consists of testing our application with IaaS implementation of autoscaling and load balancing. For this we write a script that runs 60 threads at a gap of 30 seconds between the starting time of the batch. We set the upper CPU threshold parameter to 75 percent. The execution should cause the system to spawn a new instance when the average CPU utilization for the past one minute goes beyond 75 percent.

Analysis As this experiment can be seen as a scenario when the application is catering to 60 concurrent users every 30 seconds. We noted the response times for 20,40 and 60 concurrent users, and observed that if the number of concurrent users increase the server response time increases and users might have to wait for the converted video file. Figure 2, 3 and 4 shows the CPU utilization during a certain time span of the three created instances with the aid of auto-scaling and load-balancing

Charged-time and charged-cost for this experiment the load increases gradually new instances are spawned and application works with 3 instances for close to 15 mins, this can be extrapolated to an hour.

$$chargetimeperhour = 3 * m1.smallpricing = 3 * \$0.065 = \$0.195$$

The aforementioned equation is the charge time per hour for 60 users.

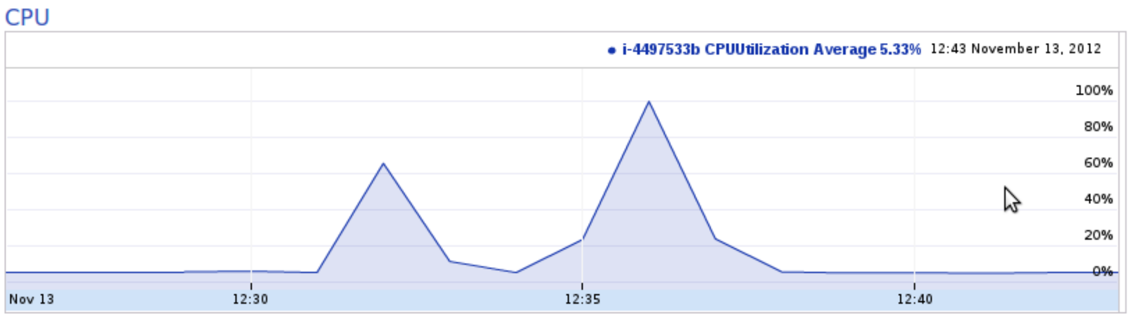

Figure 2: CPU utilization of original instance

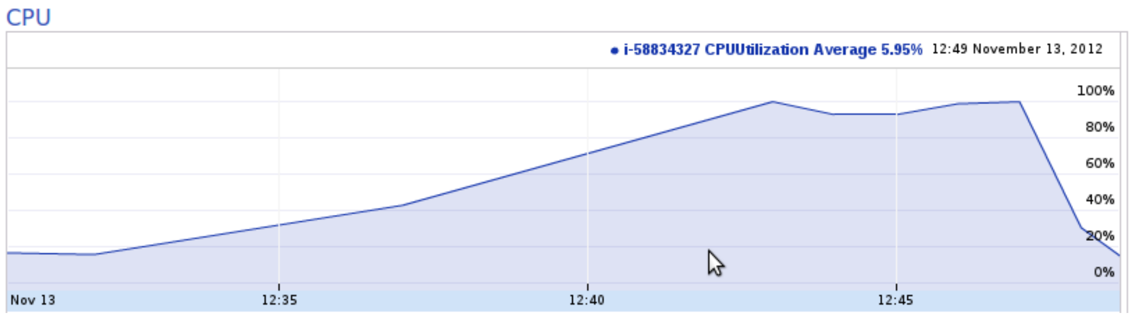

Figure 3: CPU utilization instance created by cloud based system using auto-scaling

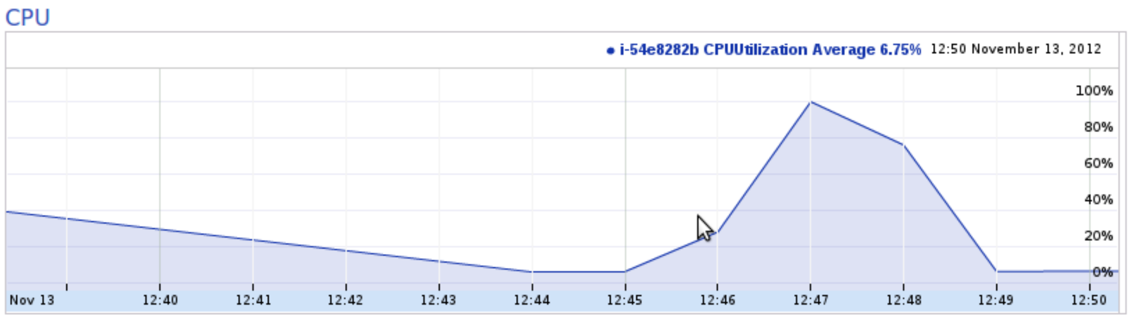

Figure 4: CPU utilization Second instance created by cloud based system using auto-scaling

5 Discussion and Conlusion

Our online video converter application is CPU intensive and since it temporarily stores data in the server before being outputted to the user it is also disk intensive. As an observation we found out that the application has high response times with increase in users, and also the CPU utilization increases many fold within few seconds, which leaves us with two choices either to autoscale and add more instances or change the instance types to higher types like m1.medium or m1.large, in our IaaS system we autoscale to more instances of the same instance types i.e. m1.small. Every instance takes close to 30 seconds to come to a running state, this maybe a downtime for your server when it cannot handle any client requests. One solution to this problem is to provision instances based on your average number of requests per hour metric or to adjust the upper threshold values for scaling up of the instances based on the average traffic.

By extrapolating the results obtained from our experiment we can determine the charged time and charge cost. If the application is used by 100000 users concurrently for an hour then approximately 6000 m1.small instances are needed. This will amount to 6000 * 24 * 0.065, $whichis$9360 per day, 280800$peryearandapproximately$3.4M per year. These results can be varied if other CPU intensive instance type are used and the cost can be reduced to a large extent. Thus, we do not recommend WantCloud BV to use cloud based services for online video converter application if their average number of users are very high and they use m1.small instance types.

Metric	**Hour**
total-time	133
think-time	8
dev-time	65
xp-time	32
analysis-time	12
write-time	12
wasted-time	4

Table 1: Time spend on project

Metric	**Hour**
total-time 14	
dev-time	6
setup-time 8	

Table 2: Time spend on experiment 1

Metric	**Hour**
total-time	7
dev-time	0.5
setup-time	2

Table 3: Time spend on experiment 2

Metric	**Hour**
total-time	11
dev-time	6
setup-time	5

Table 4: Time spend on experiment 3

6 Appendix A: Time Sheet

In this appendix we evaluate how much time we have spent on realizing this project. For this we have defined 7 metrics. *Total-time* , determines the amount of time for completing this project. *Think-time*, reflects the time it took too figure out how to solve WantCloud BVs problem. *Dev-time*, rates the amount of time it took to code and setup the system. *Xp-time* is the time taken by conducting experiments. Evaluating the results gathered from the experiments is reflected with the metric *analysis-time*. The time taken by writing this report is represented by *write-time*. The time it took for every activity that cannot be accounted for in the aforementioned 6 metrics and which does facilitate progress of the project is reflected by the *wasted-time*.

Furthermore we define three metrics for each experiment. *Total-time* is the time it took to conduct the experiments. The amount of time to create the experiment is reflected by *dev-time*. *Setup-time* accounts for making the environment suitable to conduct the experiment.

Table 1 presents the time spend on the project. Whereas Table 2, Table 3 and Table 4 represent the time spend on the individual experiment, respectively Experiment 1, Experiment 2 and Experiment 3 conducted in Section 4. Taken into account that at the start of the project we lacked sufficient knowledge of Python the overall (*total-time*) of Table 1 is higher than anticipated. When conducting experiment 1 we ran in to the limititation that the micro instances have. We wrongfully assumed that there was something wrong with our cloud based system. Which in turn resulted in debugging the system and caused the excessive amount of time spend on Experiment 1.

Code	**Repository**
Client	https://github.com/ntimal/Cloud_Computing_Client
Report	https://github.com/ntimal/CC_Report
Servlet	https://github.com/nidhisingh88/CC_LargeLabAWS
Script	https://github.com/nidhisingh88/CC_script

Table 5: Public repositories created and used during this project

7 Appendix B: Repositories

To promote open-source we have created several public repositories on GitHub[17]. In Table 5 you will find the url of the repositories corresponding to the client, servlet, script and report code.

[17] https://github.com/