Auto-scaling and cloud bursting service for Eucalyptus

CS290c Virtualization and Cloud Computing Technologies Fall 2011

David Lloyd Johnson, Daniel Vucci
Department of Computer Science
University of California, Santa Barbara
davidj@cs.ucsb.edu,djvucci@umail.ucsb.edu

1 Introduction

Rural regions in the world often have very limited access to broadband Internet and local powerful computing resources. Internet is usually provided by asymmetric satellite links with typically download speeds ranging from 256kbps to 512kbps. The link is often shared between many users in a village or Internet cafe and at peak-usage times the link becomes highly congested and often unusable. This causes many services, only available on the Internet, to become unavailable or too slow to use. However locally installed private cloud computing infrastructure could provide some valuable services to local residents even when the Internet link is congested or unavailable.

We envisage a number of cloud computing-based applications which could benefit rural communities. For example, speech synthesis could be provided to people in rural regions who could use this technology to convert web-pages or photographed documents to localized speech in cases where they are illiterate or semi-literate. Other possible local cloud services are music streaming and health information systems. Locally installed gateway services such as virus scanning, proxy caching and email hosting could all also run on backed up virtual machine instances which are easily re-instantiated on new hardware in the event of hardware failure.

This project sought to solve the problem of optimally using local computing resources by providing a mechanism to auto scale services to demand on a local cloud computing platform. We provide a flexible auto scaling framework for Eucalyptus which is able to start and stop virtual machine instances based on CPU, memory or disk demand. We also provide a mechanism to allow traffic to be offloaded to a public cloud (Amazon EC2) in the case where local resources are exhausted or not available.

2 Related work

Auto scaling features are available in Amazon EC2 [1]. These services allow you to use the monitoring features built into amazon EC2 to trigger instance creation and termination when thresholds are crossed. However, no auto scaling features are currently available in Eucalyptus and this project provides these features. Auto scaling has also been achieved using advance reservation which pre-empts known peak periods of usage [3]. Providing local cloud computing for cases where their is not substantial bandwidth at an Internet gateway has been carried out for mobile devices with access to local resource rich computers or clusters for mobile devices dubbed “cloudlets” [2].

3 Architecture

3.1 Overview

The overall architecture of our system is as shown in Figure 1. The village has a local network, which includes a private Eucalyptus cloud. The village network is also connected to the Internet, and can make use of Amazon EC2. Traffic from clients in the village first goes to an HAProxy load balancer, which then forwards the traffic to any available cloud machine instance . How many and which cloud machines are available to forward traffic to depends on current usage. There is always at least one machine running in the private Eucalyptus cloud, with additional instances being started as necessary. Because running EC2 instances can become costly, especially considering the budget constraints of a rural network, EC2 instances are started only after all local Eucalyptus resources have been exhausted, and are the first to be stopped when utilization decreases.

Our Auto Scaler orchestrates this system. It sends commands to EC2 and Eucalyptus to start and stop instances as necessary. To determine this necessity, it monitors all running instances for over- and under-utilization. When newly started machine instances are responsive or instances are terminated, HAProxy is updated regarding which instances are available to send traffic to.

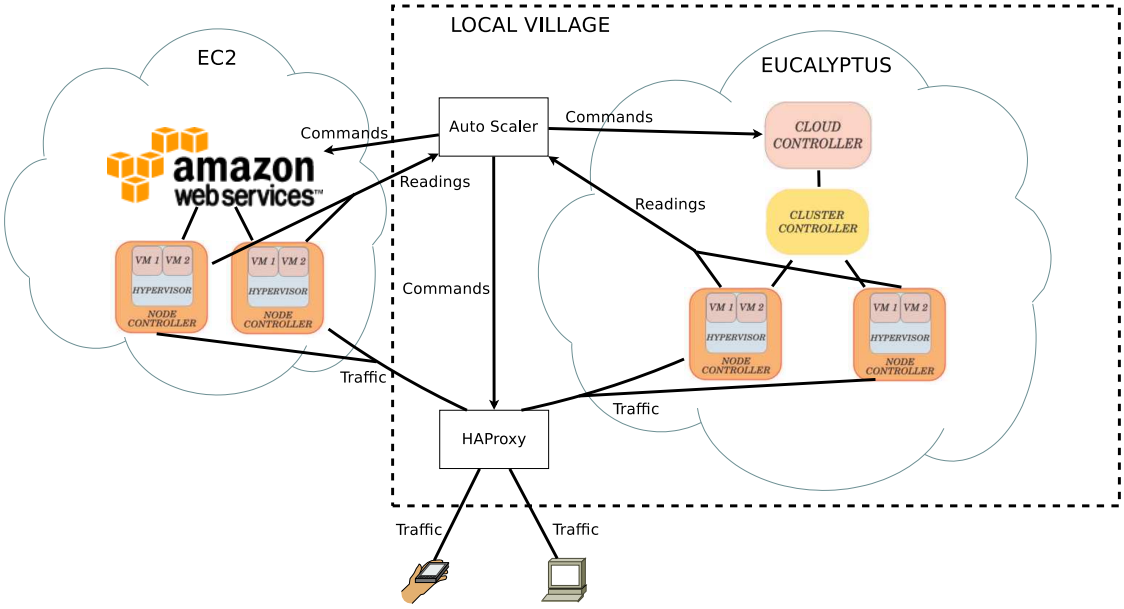


Figure 1: *Architecture Overview*

3.2 Auto Scaler

Our Auto Scaler was written in Java as shown in Figure 2.

Currently, there are three Listeners which monitor CPU, MEM, and DISK utilization, although the system's architecture is such that additional Listeners can easily be added. Each Listener gets its own well-defined UDP port on which it listens for readings from instances. The readings are generated and sent by our custom monitoring software which leverages the Linux sysstat tools. At the present time, this software must be running on each instance; however, it may be possible to integrate more directly with EC2 and Eucalyptus components and avoid the need for modifying the instances. The Listeners report the received readings to the system. The readings are always reported to the Statistics class, which stores readings by instance machine. The readings can also be given directly to Calculators, if necessary (see below).

Each Listener has a corresponding Calculator, which works with its respective readings and decides when to start and stop instances. For our case, this means that there is a CPU Calculator, a MEM Calculator, and a DISK Calculator. There are currently two different base types of Calculators. The first type does work at regular intervals, checking the accumulated readings in the Statistics class. The second type receives readings directly from Listeners as they come in. The Calculators behaviour is dependent on its algorithm and the algorithm's inputs. We implemented three different algorithms, while making it easy to add new ones to the system. The currently implemented algorithms are described in Section 3.3. It should also be noted that all running Calculators do not need to use the same algorithm or the same inputs to those algorithms. When a Calculator decides that an instance needs to be started or stopped, it invokes methods in the Cloud class, which then carry out the appropriate operations on EC2 and Eucalyptus.

It is the Monitor's job to keep track of all available cloud instances. This information is used to update HAProxy's table of machines available to forward traffic to. The Monitor is kept up to date both by the Listeners and the Cloud class. Listeners give it all seen readings, which act as a heartbeat. Currently, the heartbeats are only used for discovering newly available cloud machines. To learn of a machine stop, the Monitor receives notifications from the Cloud class after the Calculators make their decision to shut down an instance.

The Auto Scaler can be placed on any available machine. For our testing we placed the Auto Scaler and HAProxy on the same machine. This eliminates the possibility of the Monitor and HAProxy getting out of sync with respect to available cloud machines.

3.3 Algorithms

We have implemented three different algorithms in the Calculators for deciding when instances are to be started and stopped. The first algorithm, AVG, takes as inputs a window period and upper and lower thresholds. AVG makes decisions at the end of every window period. It will start a new instance if the average of all readings within the window period is above the upper threshold, and will stop an instance if the average is below the lower threshold. The second algorithm, CPTT (Constant Passed Threshold Time), has upper and lower thresholds like AVG, but does not use a window period or do any averaging. Instead, it has a time to action parameter, and it looks at readings as they are coming in. As readings come in, it keeps track of when readings begin to go beyond the upper or lower thresholds. If readings are continuously beyond either threshold for the time to action time, then an instance is started or stopped. If a reading drops below the upper threshold or goes above the lower threshold, the count is reset. The third algorithm,

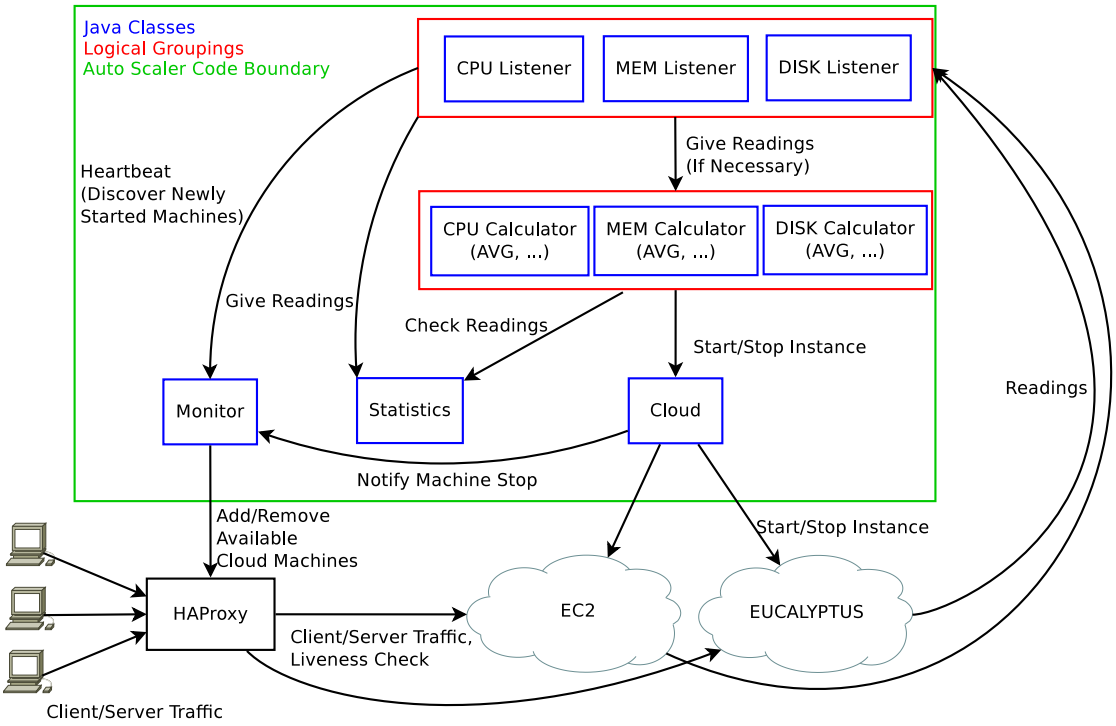


Figure 2: *Auto Scaler Architecture*

PTTWI (Passed Threshold Time Within Interval), takes as input a window period, upper and lower thresholds as well as a time threshold. This algorithm will start a new instance if the total amount of time spent above the upper threshold within the window period is greater than the time threshold, with the analogous process for determining when to stop an instance. This algorithm differs from the AVG algorithm in that if there are many high readings but some low readings, the low readings will pull the average down, perhaps such that the average is not above the threshold even though many high readings were obtained. In contrast, PTTWI considers the fraction of time spent above the threshold.

Now that the algorithms we used have been explained, a few notes are in order. Firsly, there are a couple of ways in which oscillation of starting and stopping instances is avoided. To start, there is a configurable period of time that starts at the instant a machine is started or stopped, during which requests for more starts and stops are ignored. Also, the upper and lower thresholds should configured such that they are differ by a large degree. As a second note, it should be understood that new algorithm implementations do not have to be created when new measurements other than CPU, MEM, and DISK are needed. As long as what is being measured fits within the context of an available algorithm (for example, if taking an average of readings within a window period (AVG) makes sense), then the available algorithms can be reused. Finally, for our tests, the EC2 instances were more powerful than our Eucalyptus nodes. When running the algorithms over EC2 instances in addition to Eucalyptus instances, we found that the EC2 instances would stop too soon because the workload was such that the utilization was under the threshold, which is currently the same for both EC2 and Eucalyptus instances. To get around this, we only monitored Eucalyptus instances.

4 Evaluation

Our evaluation was done using a web-based text to speech application. This application allows you to enter any length text string and a mp3 audio speech file is posted back on the web interface once the conversion is complete. The process follows 3 steps: (a) A text string is posted to the php-based application using an HTTP POST (b) the php program extracts the text string and converts this to a raw WAV audio file using the flight text to speech application (c) The raw WAV audio file is converted to mp3 compressed audio using the LAME encoder and a link to the mp3 file is posted on the web site.

The following were typical response times for the text to speech application on Eucalyptus and EC2 instances. On Eucalyptus: (a) 22 words responded in 368ms, (b) 202 words responded in 1.5 s and (c) 2121 words responded in 29s. On Amazon EC2: (a) 22 words responded in 491ms, (b) 202 words responded in 1.7s and (c) 2121 words responded in 20s. Amazon EC2 had longer response times for the short and medium length text strings due to the network delay playing a significant factor, however it performed about 30% better for long text strings due to EC2s better CPU and memory resources and network delay not playing a significant factor.

4.1 Experimental method

The local eucalyptus infrastructure was installed on three Intel dual core 3GHz 64-bit blades with VT capability, 4GB of ram and 150GB disk. One machine was used as a Cloud controller (ClC), Cluster Controller (CC) and a instance storage system (Walrus) and the other two machines where used as node controllers (NCs). We made use of a local instances using the m1.large type which was configured as 1 CPU, 768MB ram and 10 GB disk. On Amazon EC2 the smallest 64-bit instance type we could use was m1.large which has 2 CPUs (4 ECUs) and 7.5GB ram. For the evaluation, 1 Eucalyptus instance was always kept active and the maximum number of Eucalyptus instances was set to 2 to force an early use of the Amazon EC2 cloud infrastructure. We only allow 1 EC2 instance to be started up as peak demand is expected only for a short period of time.

We made use of the *httperf* traffic generation tool to test our system. A steady set of HTTP POSTs was sent to HAproxy using a medium length (202 word) string. The experiment starts with a 100 POSTs at 1 request per second and then is increased to 2 requests per second for a further 200 POST messages. The upper threshold was set to 75% and the lower threshold was set to 15% with a window size of 10 seconds.

4.2 Results

In Figure 3 the CPU, memory and disk utilization is shown as the VMs are loaded and as the load is removed. A vertical block line is shown when a new instance is asked to start, a vertical blue line is shown when an instance becomes active and a red vertical line is shown when an instance is terminated. For this experiment, the PTTWII algorithm was used to calculate when instances should start and stop. Instances are started or terminated when they are above or below the upper or lower threshold for more than 7 seconds in the 10 second window.

As shown in Figure 3(a), the CPU utilization rapidly increases as requests enter the system. At 40 seconds, the auto scaler requests a new instance and a Eucalyptus instance is available and hence is immediately started. At 100 seconds the 2nd instance is available and the auto scaler sends this address to HAproxy which begins to load balance the requests. As expected the CPU utilization drops by almost half on both local instances. At 250 seconds the rate is increased to 2 requests per second and shortly after the CPU begins to increase, a new instance is requested. As there are no more Eucalyptus instances, a public instance is requested from EC2. The EC2 instance boots faster then Eucalyptus as the EC2 infrastructure is more capable than the local Eucalyptus environment. Once the EC2 instance is active, the load is shared equally between both local machines and the EC2 machine. The CPU utilization is substantially lower for EC2 due to its more powerful CPU specifications. Once the load is removed at approximately 450 seconds, the EC2 instance is stopped and then the local eucalyptus instance is terminated after a "minimum time between instance terminations" window has expired.

The memory utilization shown in Figure 3(b) only increases slightly in the case of a single machine becoming heavily congested, otherwise it remains fairly constant over the course of the experiment. Should the application have been more memory than CPU intensive, the system would have trigged on memory usage before CPU utilization as it is designed to trigger on any metric crossing a threshold value. The disk usage, shown in Figure 3(c),was also only significantly effected in the case of the single machine being overloaded. This spike was most likely caused by cache thrashing as more and more text to speech processes overloaded the system.

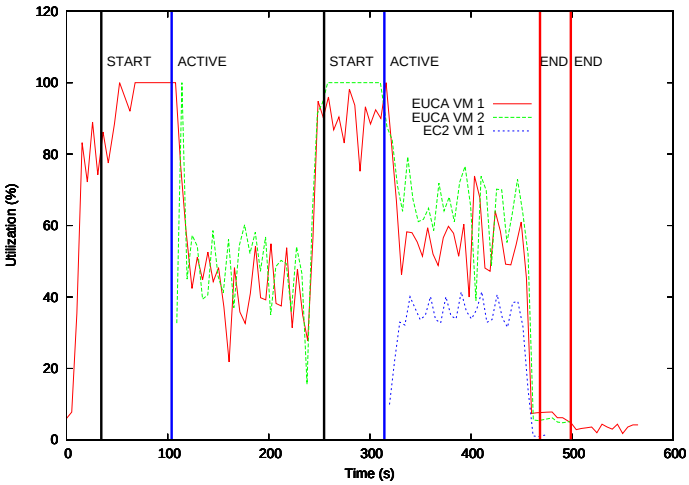
5 Lessons learnt

A key challenge in auto scaling is knowing whether a CPU/Mem/disk increase/decrease over/below a threshold momentary is a short or longer term trend. Two opposing user needs have to be balanced: responsiveness and maximum utilization of the local computing resource. Hysteresis can protect against momentary peaks and dips and is implemented in this system by averaging results over a long time window. We also allow the user to set the minimum time that must pass for an instance is started or stopped. Windows should never be too short as the system needs to first stabilize as old processes are flushed out.

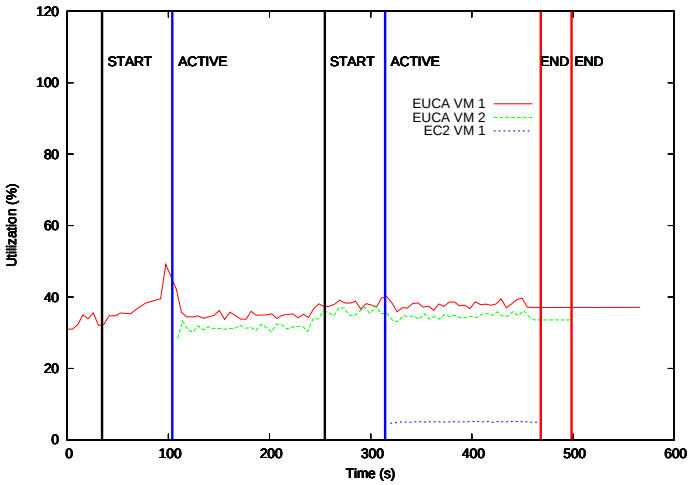
Machine instances oscillation is caused by a system with not enough of a gap between the upper and lower threshold. EC2 was substantially more capable than local Eucalyptus instances and hence thresholds were only checked on local cloud. For booting an instance from the cache, EC2 does slightly better than Eucalyptus and takes 60 seconds whereas EC2 only takes 50 seconds. However to boot an instance from the local Walrus data store that is not in the cache for Eucalyptus can take up to 10 minutes whereas EC2 can offload a 5GB instance from S3 in approximately 50 seconds.

6 Future work

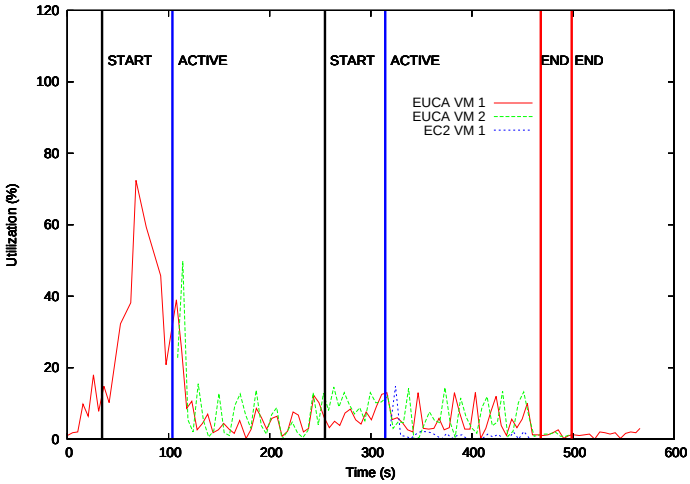
Our current system can only respond to threshold values within a fixed window period. However, making use of history trend data to quantify patterns that usually lead to a congested resource, could provide an auto scaler with better tools to start an instance early and avoid a period of slow response time while a new instance is starting. One of the key challenges in wireless networks in developing regions is the lack of capacity on the Internet gateway. Sensing available network capacity on Internet link and only adding public cloud instances to HAproxy if enough capacity is available



(a) CPU utilization



(b) Memory utilization



(c) Disk utilization

Figure 3: CPU, Memory and Disk usage on local Eucalyptus and Amazon EC2 instances as load is increased from 1 to 2 requests per second. Diagram also shows virtual machines instances starting and stopping on local and public cloud

will avoid wasting unnecessary costs on public cloud infrastructure. If spare resources are available on the local cloud environment, a spare instance could always be made available to avoid instance start up delay. Due to EC2 being better provisioned than the local cloud, EC2 is always underutilized as HAproxy uses a simple round robin strategy to schedule requests. However, it is possible to provide weights for the instances in HAproxy and weights could be set that take the capability of instances into account. Ultimately it would be best to avoid installing custom monitoring software on each instance. This could be achieved by instrumenting the hypervisor in Eucalyptus to report back CPU, memory, network and disk usage statistics. These monitoring tools are already available Amazon EC2. In order simplify the setup of the auto scaler, ultimately a new set of *euca* commands should be built to set up an auto scaling environment as well as start the auto scaling process.

References

[1] Autoscaling in Amazon EC2. http://aws.amazon.com/autoscaling.

[2] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[3] B. Sotomayor, R.S. Montero, I.M. Llorente, and I. Foster. Capacity leasing in cloud systems using the opennebula engine. *Cloud Computing and Applications*, 2008:1–5, 2008.