Web Services and Cloud-Based Systems

Assignment 3 - Virtual resources management on Public Clouds

Tom Peerdeman & Laurens Verspeek
10266186 & 10184465

May 8, 2015

1 Setup

In this assignment we started processing the twitter data with pumpkin on DAS4 resources (on 3 different VM's), just as in the previous assignment (2.2). The processing performance can be speed up by acquiring Amazon EC2 VMs (if the budget allows this). In figure 1 the different infrastructures are shown and the workflow between them.

The three DAS4 VMs run the injector, collector and filter worker-seeds with pumpkin. As the collector VM receives the twitter stats from the filter, this VM can measure the performance in tweets per second. This is done with the VM Controller python script. If the performance drops below a given treshold, the VM Controller will create new Amazon EC2 VMs which run filter workers. Which type of EC2 instances has to be created is determined by the VM Profiler python script, which measured the performance of different instance types of Amazon on different regions.

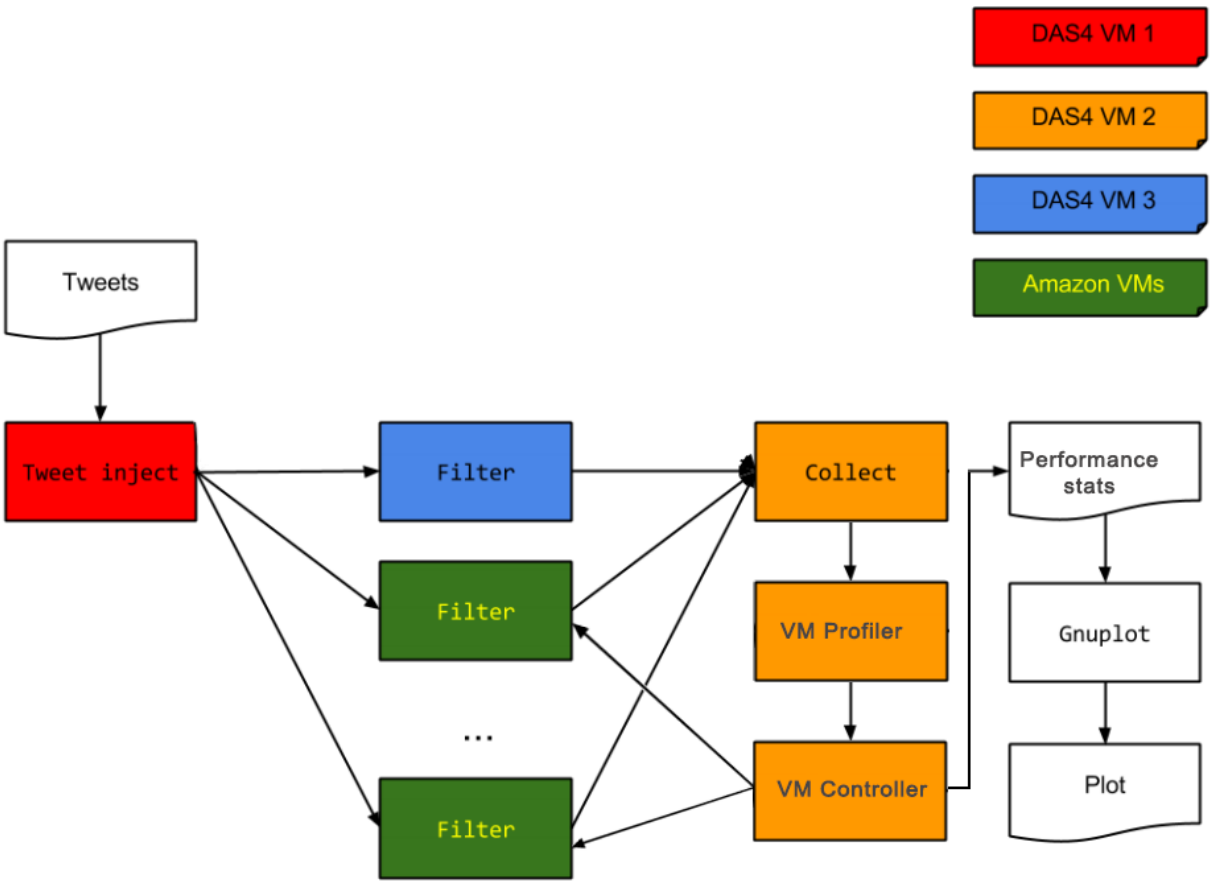


Figure 1: Infrastructure workflow

2 Amazon EC2 Cloud

2.1 Creating instances

Creating instances using the graphical web user interface is quite easy. The launch wizard fills in all the required variables, like security group and subnet. However to create an algorithm that can control the amount of running VM's this starting of instances also has to be done programmatically. To do this amazon has an API available. A simple binding of this API exists in python as the boto project. Using the boto library we created a class called *EC2Lib*. The goal of this class is to simplify the boto calls even more by hardcoding the parameters required, such as the AMI and key pair. This class also tracks the instances created by itself. It can thus also track the actual time the instances are running.

2.1.1 Contextualisation

We created a image which has pumpkin and the filter worker-seed preinstalled. This image was created by starting a VM from a image from the Ubuntu cloud site (http://cloud-images.ubuntu.com/locator/ec2/). We chose for Ubuntu server 14.10 as it has the most up to date software packages available. Ubuntu 15.04 has more up to date packages, but the 15.04 cloud images are still in development. Next we connected to this VM with SSH and installed all the necessary packages. With the web interface from Amazon EC2 we could easily create a new image (AMI) of this runnning VM. The image also has a *start.sh* shell script that runs pumpkin and the filter worker. The EC2Lib class uses the Amazon EC2 API to create VMs from the AMI we just created. The EC2 API also allows to execute commands on creation of the VM. So the only command we have to execute was the command to run the start shell script, and the pumpkin filter will start running.

2.2 Retrieving prices

A big section of the assignment is making sure not going over the budget of $5. One part of this puzzle we already explained: The tracking of running time of the instances. The second part is multiplying this runtime, rounded up to hours, by the instance price per hour. One way of getting these prices would be hardcoding them. This however not be very flexible. If Amazon would change their prices to be higher, our controlling script would estimate the cost lower, thus possibly going over budget.
The Amazon API does not have a way to fetch the current prices, therefore we had to hack our way around it. The actual Amazon pricing page
(http://aws.amazon.com/ec2/pricing/) uses javascript to load the prices from a JSON document. In the file *ec2pricing* a function called *get_ec2_pricing* is defined with a region as parameter. This function loads the same javasript page as the official page does, it then strips down the javascript to leave behind the bare JSON data. This data can then be parsed, and the prices for the selected region can be extracted.

3 Profiling amazon instances

In order to know which type of Amazon instances has to be created in which region, we ran a profiling script (*perfmonitor.py*) on the DAS4 collector VM. We terminated the DAS4 filter VM, so we could test the performance of different types of Amazon instances. The profiling script starts a timer every 10 seconds and checks the difference in number of tweets processed. This way, the script can determine the performance in tweets per second and stores this information to a logfile on the VM. We let the profiling script run for 4 different instance types in 2 different regions. The results can be seen in figure 2.

3.1 Choosing the right region

As the price per hour for the Amazon instances was a little bit cheaper in the US region than in Europe region, we decided to also test the performance of the US region. However, as can be seen in figure 2, the performance in the US is way worse then in Europe. This is probably caused by latency, because the DAS4 VMs are also located in Europe. So if you pay a little bit more for the instances in Europe, you get way better performance, so we decided to stick with the Europe region.

3.2 Choosing the right instance type

We decided to only look at instance types which have only 1 core, as Pumpkin doesn't have support for multiple cores. So buying instance types with multiple cores would

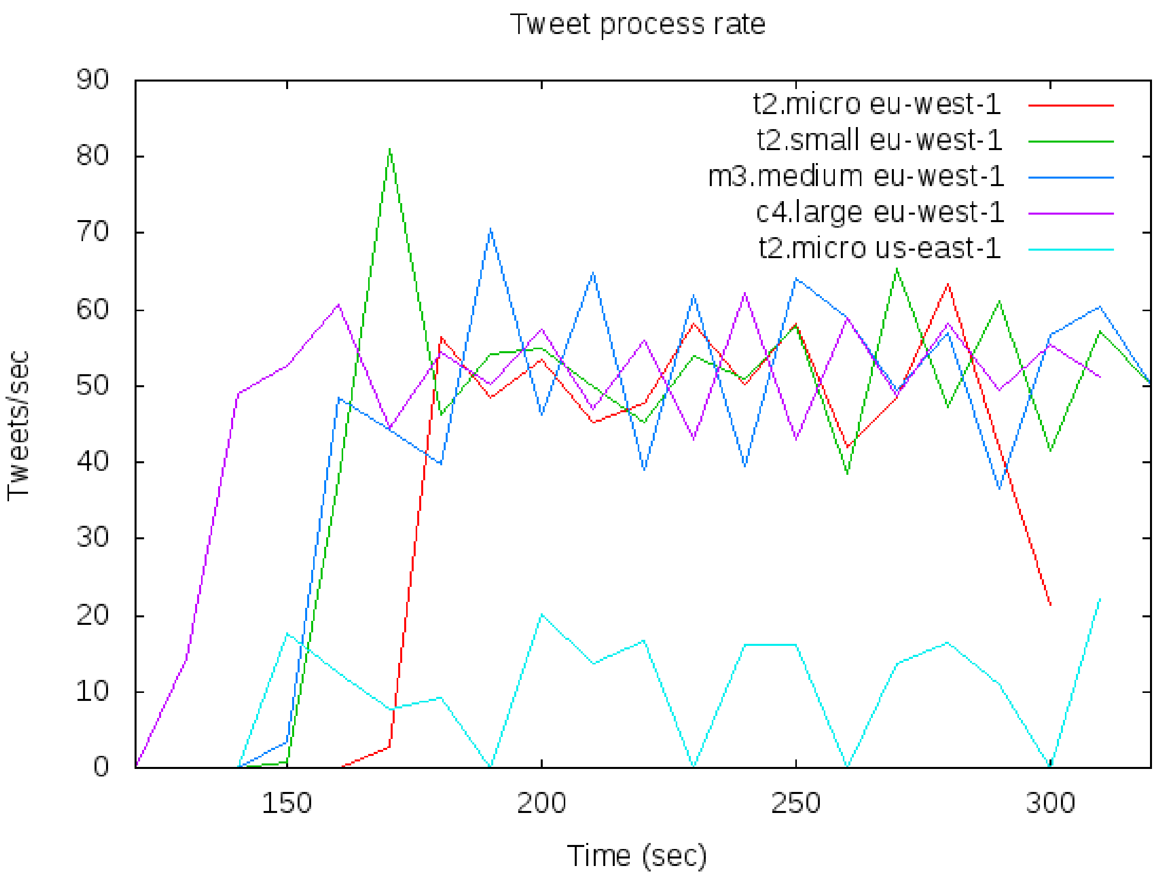


Figure 2: Process rate for different Amazon EC2 instance types

be a waste of money. However, As can be seen in figure 2, the performance of all the different tested instance types is roughly the same. This tells us that the bottleneck is probably not the CPU or the RAM of the VMs, but it is more likely that networking is the bottleneck. So it would be a waste of money to buy the more expensive instance types, as the cheap ones (t2) can reach the same performance.

4 VM Controller

Now that we know which instance types are best performing and most efficient in terms of money, the VM Controller script (*vmcontrol.py*) can be executed on the DAS4 collector VM. This VM Controller script is based on the Profiling script (*perfmonitor.py*) and also measures the average tweets per second over a time period of 10 seconds. You can configure the VM Controller by setting the budget, tweets per second (tps), Amazon access key and secrey key in environment variables. More information about how this works can be found in the next session: *Additional contextualisation on Das 4*. If the performance is too low compared to the tps set in the environment variable, you get a 'strike'. If you have three strikes in a row, the VM Controller will create a new Amazon EC2 instance of type *t2.micro* if the budget allows this. It can create a maximum of 20 instances per type. So if the limit is reached for *t2.micro*, it will switch to type *t2.small*. As the Amazon VM needs some time to startup, the VM Controller will not count any stikes in the 2 minutes right after a new Amazon instance was started. This gives the new VM the time to boost up the performance over the threshold limit. If the performance is 1.5 times higher than the set tps (three times in a row), this is a waste of money and it will stop a VM from Amazon to save on the budget. On each tick the script will also calculate the total costs of all the Amazon VMs with the help of the

EC2Pricing library we created (see section *Retrieving prices*). If the budget is almost reached, it will stop all VMs from Amazon. It will also stop all VMs from Amazon if all the tweets are analysed. All the information is printed and stored in a logfile on the DAS4 VM.

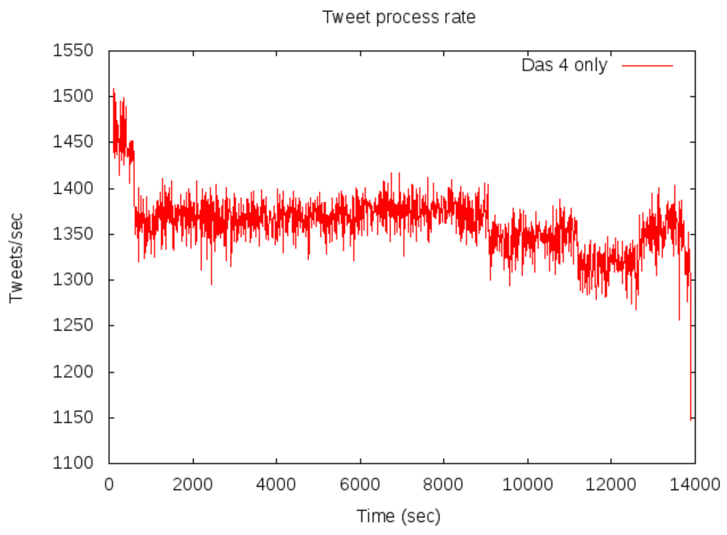
4.1 Additional contextualisation on Das 4

With this new VM controlling script we had to adapt our existing contextualisation on Das 4. Besides running pumpkin the collector now also had to run the control script. We did this by creating a new initialization script on top of the existing ones. In this script the control environment is set up and started using another tmux command. This command starts a new tmux session called mon. This session will then run in the background until all the tweets are processed, and thus the python control script stops. We chose to not create an image from the control python script itself, as we were developping it and had to make many changes. Using an image for the script we would have to remake the image for each small modification. Instead we pulled the latest version from our git repository.

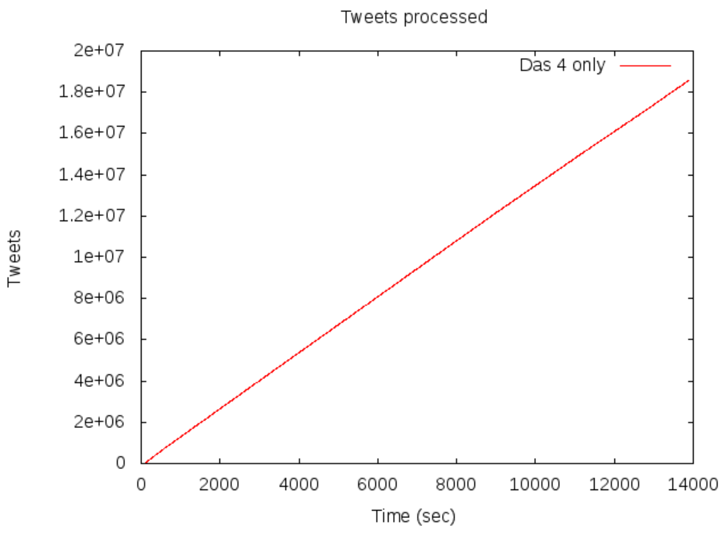
The second function of the new startup script is defining the configuration variables, which the VM controller uses, as described above. This way any user with access to our images and startup scripts (like the assistants) can change the parameters of the VM controller, without the need of write access to our repository.

5 Reaching the 15k barrier in 2 hours

As stated in the assignment text the minimum goal was to process 15000 tweets in 2 hours. This comes down to a bit over 2 tweets per second. As can be seen from fig 3a we clearly reached this target. Actually with an rough average of 1300 to 1400 tweets per second the 15000 mark is passed in about 11 to 12 seconds. In figure 3b we can see



(a) Process rate on Das 4 only using 3 VM's



(b) Number of tweets processed on Das 4 only using 3 VM's

that at the 2 hour mark about 10 million tweets are processed. These measurements were done on a day were no interference from other groups was noticeable. On the day of testing the actual controller script, the bare tweets per second count was around 250, still superseding the target tweets per second level. This shows us that the VM's on Das 4 are more than capable to reaching the target by themselves, and that the target of 15000 tweets in the assignment is very low.