15-719 / 18-847B, Fall 2013, Project 2: Auto Scaling with OpenStack (Part 2)

Assigned: November 18th, 2013

Due: December 6th, 2013

Free extension to: December 9th, 2013

In part 1 of project 2, you were asked to build a performance monitor and a performance collector for OpenStack. In this part of the project, you will transform your performance monitor and collector into a full-fledged auto-scaling service for a web site. Your auto-scaling service will be responsible for scaling the web site's VMs based on observed response times. As such, it will be similar in functionality to AWS's Elastic Load Balancer.

1 Collaboration & cheating policy

You **must** work with the same group that you worked with for part 1 of project 2. If this is not possible, please contact us via e-mail or in person. Do not share code or pseudo-code with members who are not in your group.

Cheating will not be tolerated. You are prohibited from using any solutions or code you find online. Please check the official 15-719 cheating policy for additional details.

2 Auto-scaler architecture

Figure 1 shows a diagram of the auto-scaler service you will develop. It is composed of monitors that reside within the web-site's VMs, a server that runs as a separate OpenStack process (i.e., as a process within the Marmot node), and a load splitter. A second Marmot node is used to drive load to the web site.

3 Auto-scaler server input

Your auto-scaler server must take as input a template describing the scaling properties of the web site. The template will describe the following:

1. The maximum number of VMs that can be allocated to the web site.
2. The minimum number of VMs that can be allocated to the web site.
3. The VM image to use when spawning new VMs.

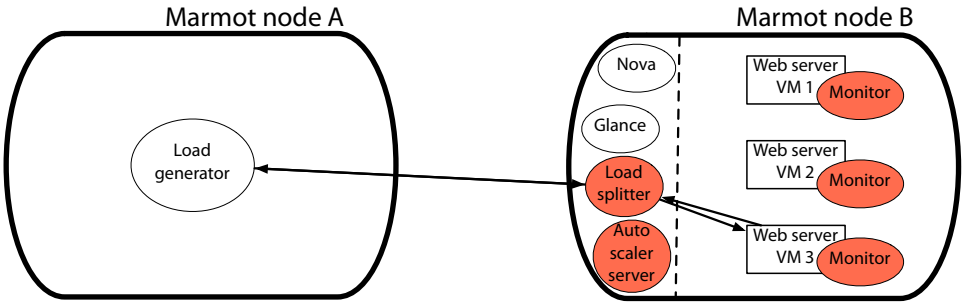

Figure 1: **Auto-scaler architecture**. The key elements of the auto-scaler service you will develop are shown. Note that a separate marmot node is used to run a load generator that drives load to the web site for which your are developing an auto-scaler service. This load generator should run as a process within the Marmot node, not within a VM. DevStack sould not be running in Marmot node used by the web site.

4. An average performance (response time) target for the entire web site. The units of this field is seconds.

An example of a template can be found in `/proj/OpenStackSys/proj2/scripts/auto_scaler_templates/tomcat.templ`

4 Auto-scaler server functionality

Your auto-scaler server must decide when to allocate new VMs or de-allocate existing ones. It must also execute these decisions. Your server should aim to satisfy the goals listed below.

- It should allocate the minimum number of VMs for the web site that will allow the web site to satisfy or exceed its performance target.
- It should not allocate or de-allocate VMs in response to transient changes in load (e.g., period of high loads that last less than 5 seconds). One way to accomplish this is to base allocation decisions on a sliding window of past throughput.
- It should allocate new VMs when existing ones fail or are killed.
- It should guarantee that no less than the minimum number of VMs specified in the template are running.
- It should guarantee that no more than the maximum number of VMs specified in the template are running.

5 Auto-scaler monitor functionality

Your auto-scaler monitor must run within individual VMs and transmit performance information to the auto-scaler server. It should also send periodic heartbeats to the sever to inform the server that the VM it is running on is still alive. You are free to transmit whatever information you desire at whatever granularity you desire. Performance data must be obtained from logs output by the load splitter (see

Appendix **??** for how to parse Apache Tomcat connector logs to obtain millisecond-level response times).

6 Load splitter functionality

Your load splitter must run within a VM and must split load evenly between each of the web site's VM instances. Your load splitter should satisfy the goals listed below. Note that to satisfy these goals, your load splitter must communicate with the auto scaler.

1. It should utilize new VM instances immediately by re-balancing load amongst all available instances (including the newly launched one).
2. It should allow VMs that have been marked for de-allocation to shut down gracefully. Specifically, the VM that is to be de-allocated should not be routed any new requests, but should be allowed to finish servicing requests already routed to it. **No requests should fail as a result of VM de-allocation**.

What to use as the load splitter: You should use Apache's Tomcat Connector as your load splitter. More information about Apache's Tomcat Connector can be found in at this website: `http://tomcat.apache.org/connectors-doc/generic_howto/loadbalancers.html`. We recommend creating a custom private image with Apache installed and storing it with OpenStack. Please make sure your image cannot be used by other groups.

Configuring the VMs between which Tomcat should split load: The file `/etc/apache2/workers.properties` indicates the VMs (called *workers*) between which Tomcat should split load. Removing or adding a VM instance to the load balancer requires editing this file and reloading (**not restarting**) Apache. Appendix C describes this configuration file and further details about how to configure the Apache Tomcat connector.

7 Details about the web site

The web site your auto-scaler is responsible for should run Tomcat. It should serve a simple Javascript page, which can be found in `/proj/OpenStackSys/proj2/scripts/index.jsp` (Please be sure to copy this file to your VMs).

To create your website, we recommend installing Apache Tomcat on the provided Ubuntu image and storing this within OpenStack as a custom image. You can then specify this custom image in the template file the auto-scaler server takes as input. Appendix D provided details on how to setup Tomcat.

8 Launching & shutting down your auto-scaler service

Your auto-scaler service must be integrated with DevStack. It must start when the `stack.sh` script is run.

Your auto-scaler server must accept client-line commands, listed below. You must write the code to provide these client-line commands.

```
instantiate_template('<template_file_name>')
destroy_template(<template_file_name>)
```

`instantiate_template()` must load the template specified and automatically launch VMs for the load splitter and the minimum number of web servers specified. Each automatically launched VM should run a auto-scaler monitor. These monitors should start up automatically and communicate with the load-splitter without any manual configuration.

`destroy_template()` must gracefully shutdown all VMs associated with the given template (i.e., no requests should fail in the process).

Your auto scaler service needs to support only one template at a time.

9 Testing

We will test your auto-scaler service via load-test scripts. Each script will drive various amounts of load to your load splitter. The scripts will alternate between periods of high and low load and will output timestamps demarcating these periods. Your auto-scaler service should respond to these varying conditions by allocating and de-allocating VM instances as necessary. The scripts can be found in `/proj/OpenStackSys//proj2/scripts/load_tests`. You will need to install the `apache2-utils` package to use these scripts.

You can verify that your auto-scaler service is functioning correctly by plotting the following graphs and annotating them with periods of high load and low load. Figure 2 shows an mocked-up example of graphs generated for a correctly working auto-scaling service.

- **Number of concurrent requests issued vs. time**: This graph can be generated from each load-test script's output. Time periods where there are only a small number of concurrent requests correspond to periods of low load and periods where there are many concurrent requests correspond to periods of high load.
- **Avg. web-server response time vs. time**: This graph's values should increase at the start of high-load periods and stabilize to values near the performance target after new VMs have been launched. Response times should stabilize before the high-load period terminates. Similarly, this graph's values should decrease at the start of low-load periods and stabilize to values near the performance target after VMs are de-allocated. Your auto-scaler should always aim to exceed the performance target (i.e, stabilized response times should be slightly less than the target response time). Also, your auto-scaler service should not respond to transient load spikes (specifically, periods of high load that last less than 5 seconds).
- **Number of VMs vs. time**: The number of VMs allocated should increase during high-load periods and decrease during low-load periods.
- **Metric your auto-scaler server is using to make allocation/de-allocaton decisions vs. time** The choice of metric is up to you. But, plotting your custom metric and comparing it to the start and end of high-load periods and the graphs above can help with debugging and will help you fine tune your auto scaler.

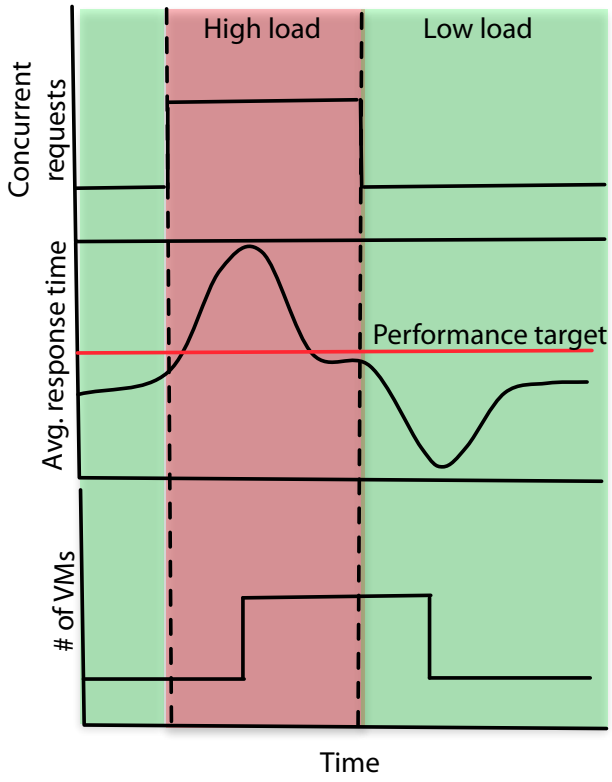

Figure 2: **Graphs that indicate a correctly functioning auto-scaler service.** The graphs show that in response to high load, response time increases until the auto scaler launches enough new VMs to reduce average response time and meet the performance target. Similarly, the graphs show that when a period of high load is followed by a period of extremely low load, response times decrease until the auto scaler de-allocates unnecessary VMs.

> You should also look at the output of the load-test scripts provided to verify that there were no failed requests. For grading purposes, you must develop an infrastructure to generate the graphs mentioned above automatically.

10 Grading

Grading will be done via in-person demos.

During the demo, we will grade your auto-scaler service according to the list below.

- **25 points**: Your auto scaler service successfully starts up and gracefully shuts down as per the requirements stated in Section 8.
- **25 points** Your auto-scaler service automatically spawns new VMs when previously-running instances are manually killed.

- **30 points** Your auto scaler service functions appropriately for a set of load-test scripts. We will provide you with the load-test scripts a priori. These scripts will be run with the template in `/proj/OpenStackSys/proj2/scripts/autoscaler_templates/tomcat.template_1`. We will determine appropriate functionality by examining the graphs and load-test script output described in Section 9. Please have in place an infrastructure to generate the graphs mentioned in Section 9 automatically.
- **20 points**: Your auto scaler service functions appropriately for load-test scripts that that will not be provided to you a priori. The template files we will use will also not be provided a priori.

A Enable client-server interactions

We recommend using the Thrift library for any client-server interactions between your auto-scaler components. A tutorial for thrift can be found here: http://diwakergupta.github.io/thrift-missing-guide/.

B Allow OpenStack VMs to communicate externally

Allow OpenStack VMs to communicate with external hosts, first, type the following commands as root in your Marmot node. (Note: this first part is already done for you in `setup_devstack_env.sh`.)

```
echo 1 > /proc/sys/net/ipv4/ip_forward
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
iptables -A FORWARD -i br100 -j ACCEPT
```

Second, type the following commands in each of your VMs.

```
sudo echo "Acquire::http::Proxy ̈http://ops.marmot.pdl.cmu.local:8888;" > /etc/apt/apt.conf
export http_proxy=http://ops.marmot.pdl.cmu.local:8888
export https_proxy=http://ops.marmot.pdl.cmu.local:8888
```

C Set up Apache Tomcat Connector on a VM

These steps will ensure a working load splitter.

1. type `sudo apt-get install apache2`.
2. type `sudo apt-get install apache2-mod-jk`.
3. Create a `/etc/apache2/workers.properties` file. A sample is provided in `/proj/OpenStackSys/proj2/scripts/w`
   You will need to dynamically edit this file to include information about the web site VMs currently in service.
4. Edit `/etc/apache2/mods-available/jk.conf` with as listed below (as sudo or root). You can also use the provided template in `/proj/OpenStackSys/proj2/scripts/jk.conf_sample`.
   - Change the `jkWorkersFile` property to `/etc/apache2/workers.properties`.
   - To obtain request response times, add the following two lines:
     `JkLogStampFormat "[%a %b %d %H:%M:%S %Y]`
     `JkRequestLogFormat "%w %V %T"`.
5. Edit `/etc/apache2/sites-enabled/000-default` (as sudo or root). Add the following lines. `<VirtualHost *:80>`
   `JkMount /tomcat-demo* loadbalancer`
   `JkMount /jkmanager/* jkstatus`
   `</VirtualHost >`
6. Reload Apache by typing `sudo /etc/init.d/apache2 reload`.

D Set up a Tomcat web server on a VM

These steps can be used to create a tomcat web server on a VM.

1. type `sudo apt-get install tomcat7`.
2. type `sudo apt-get install tomcat7-admin`.
3. cd to `/var/lib/tomcat7/webapps`.
4. type `sudo mkdir webserver`.
5. Copy the `index.jsp` file in `/proj/OpenStackSys/proj2/scripts` to the newly created `webserver` directory.
6. edit `/etc/tomcat7/server.xml` and uncomment the following line: `<Connector port="8009" protocol="AJP/13" redirectPort="8443" />`.

You should be able to test your web server by navigating to `http://localhost:8080/webserver/` on the VM in which you set up Tomcat.