Project 4 – P4
AggieStack – version 0.2
Parts A, B, C Due: 12/7/17 11:59 pm CT
Part A - Early Bird Deadline: 12/01/17 11:59 pm CT
Part B - Early Bird Deadline: 12/04/17 11:59 pm CT

This project extends Project 0. You need the functionality from that project to be working in order to extend it.

1. Don't forget the GitHub requirement. Follow naming convention

Some of you seem to have forgotten to do your development using your GitHub tree. There will be no exception for P4. If you do not follow the rules (below), you won't get any credit for P4. The rules are:

- You need to use your github.tamu.edu repository during your development. The repository needs to be named as specified in Project 0 (489-17-c)
- As specified for Project 0, you need to share your repository with the following users:
    - dilma (instructor)
    - mahima.as (instructor)
    - rahul-bhagat (TA)
    - avinsaxe (grader)
- You need to create one new sub-directory (named P4) to put your work in.
- Make sure you commit your code as you change it. The project has several parts, and the GitHub history demonstrates how your implementation evolved to implement the several parts of this project incrementally. Students who fail to demonstrate through their GitHub activity how they incrementally solved the project will be interviewed by the instructors to explain their design choices and show their proficiency on how their code works.

2. Input and output

As you may recall from the Piazza discussion for Project 0, there were ambiguities in the specification regarding the input (your program could either read a file containing a list of commands or keep reading commands from the standard input until the program was terminated). For P4, you still have the freedom to have the input as you want, as long as you document in your report what input format your program expects. Be very specific so that the grading can be done correctly.

The file p4.zip (on eCampus) contains sample configuration files and one sample input file that you can use to test your code.

3. P4 overview

P4 extends P0 in three ways:

- Part A - You implement new commands that create virtual servers (also called virtual instances). Your code is responsible for doing the resource management for the data center. Your system needs to keep an inventory of which physical servers exist in the datacenter (which you implemented in P0), and how much space on these physical servers is available for fulfilling requests for new virtual servers. Your code will decide where to allocate the virtual instances. We are not asked to come up with optimal placements of new instances on the virtual server (an interesting topic that goes beyond the scope of this course.)
- Part B – preventive failure actions. If a physical server is showing signs of hardware or software problems, your code will move the virtual servers hosted on the "sick" physical server to other (healthy) physical servers.
- Part C - Racks have their own storage server to hold images. The hardware configuration specification is changed to allow for the description of how much storage there is local to each rack. You do not need to implement this part, only to design your solution.

4. P4 Part A – virtual server creation/instantiation

Command	**Description**
aggiestack server create --image IMAGE --flavor FLAVOR_NAME INSTANCE_NAME	Creates an instance named INSTANCE_NAME to be boot from IMAGE and configured as indicated in FLAVOR_NAME In P0, you implemented the functionality for “aggiestack admin can_host” that you can now reuse as you look for a machine (also called physical server) to host the virtual server being created.
aggiestack server delete INSTANCE_NAME	Deletes instance named INSTANCE_NAME
aggiestack server list	List all instances running (name, image, flavor)
aggiestack admin show hardware	For each physical machine in the cloud, it lists how much memory, disks, and vcpus are currently free. You already implemented this in P0, but

	before it always returned the configuration of the hardware, as we did not allocate instances yet. Now your code shows the resources available at the time the command is issued.
aggiestack admin show instances	For each instance currently running, it shows where (which physical server) the instance is running

As with the previous commands, you need to generate the appropriate output for errors and log information on each command.

We suggest you code your first version without any error checking, i.e., assuming that all configuration files and all commands adhere to the specification. Once you have things working, you can add error checking incrementally.

Your code for the "server create" command needs to determine the placement of the new instance, i.e., choose one physical machine for hosting the new virtual machine. If no physical machine has the required amount of ram, disks, and vcpu available, you indicate the error of no more available resources.

After you implement the placement algorithm, the actual creation or destruction of a virtual machine is carried out by invoking the appropriate hypervisor command. For example, if your physical machines are running the KVM hypervisor, for the creation of virtual machines, you would issue an ssh into the physical machine and execute something like:

```
virt-install --os-type=Linux ram=2048 --vcpus=2 --disk path=/var/lib/libvirt/images/Ubuntu.img
```

For "server create", all you need to do is to determine where to place the new instance. You can assume that all machines have access to the storage server so they can access the base image for the virtual machine directly. We are not giving you a set of hardware machines to use in our cloud, so you do not have to issue the hypervisor command. Your code keeps all the information about the cloud, so that "show instances" can be implemented easily.

You can implement any algorithm to place a new instance. For example, you go through the current information on resource availability in the physical machines and choose the first one with sufficient resources. This policy is not a good solution in practice, because it may miss opportunities for better load balance in the datacenter. But the approach was good enough for the first implementation of several real cloud managers, so it is good enough for us.

5. P4 – Part B – evacuating sick physical servers

Other team members in your company are implementing functions that monitor servers and storage units to identify signs of possible failure. They implemented a health service that, when needed to remove a "sick" physical server from the data center, will issue the new commands in the table below:

Command	**Description**
aggiestack admin evacuate RACK_NAME	The health monitor components detected an unusual number of error in the rack storage unit. To free the rack for preventive maintenance, the AggieStack admins want to evacuate the rack by invoking this new command. Your code should migrate all the virtual machines in a server residing in RACK_NAME to another rack.
aggiestack admin remove MACHINE	Your code removes the specified machine from its view of the datacenter. This implies that the machine does not appear in any output and that it does not host any new instance. It needs to make sure that no instance is created in physical server MACHINE
aggiestack admin add –-mem MEM –disk NUM_DISKS –vcpus VCPUs –ip IP –rack RACK_NAME MACHINE	Add the machine to the system so that it may receive new instances. This is usually invoked when a "sick" server was fixed, and it is ready to work again.

All commands generate output and update the cloud log as with previous commands

6. P4 – Part C - rack storage server

You will extend AggieStack 0.1 in many ways to reflect a more realistic view of the hardware platform. The P0 version assumed that every server in your datacenter was directly connected to the storage server hosting the images. This means that a hypervisor in any of the physical servers had access to any of the available images and could use the image to boot a new virtual machine (also referred to as instance or virtual server). In this datacenter architecture, the requirement of connecting every machine to the storage server limits the size of the datacenter (due to the limitation on how many connections can be made to a single storage server) . It also introduces a performance bottleneck: creation of virtual machines in different physical hosts may interfere with each other due to the read requests issued to read the image used to boot the virtual server.

In P4 Part C, you assume a design in which physical servers are hosted in racks, and each rack contains also a small storage server.

Figure 1: storage architecture defined for P0.

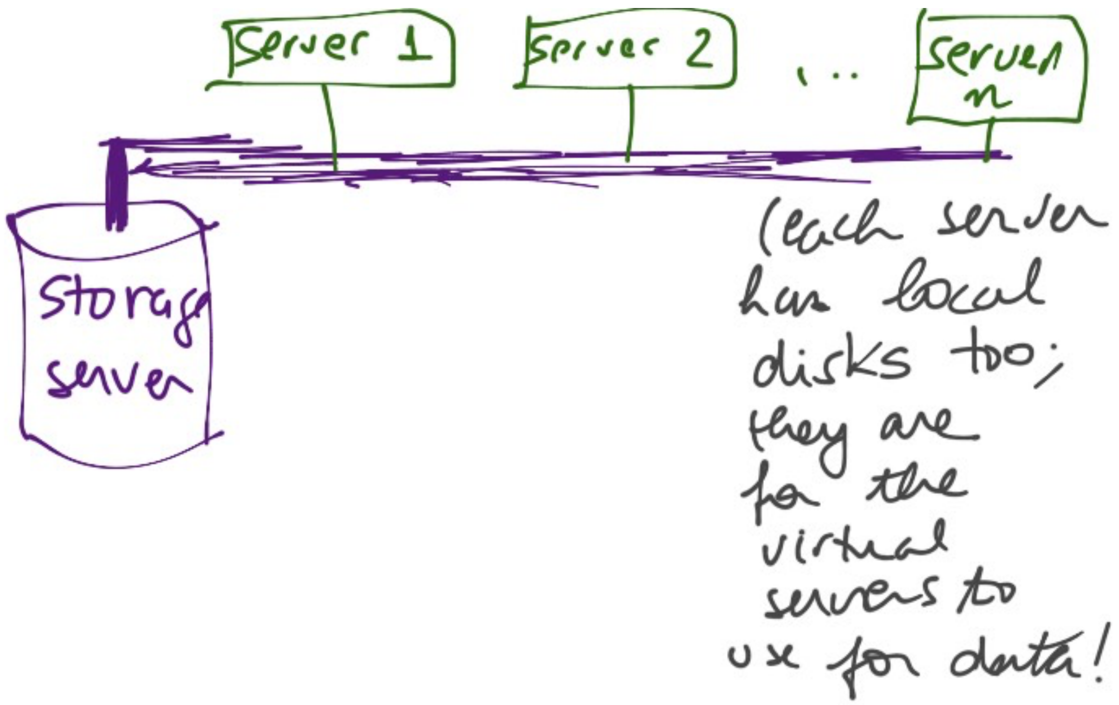
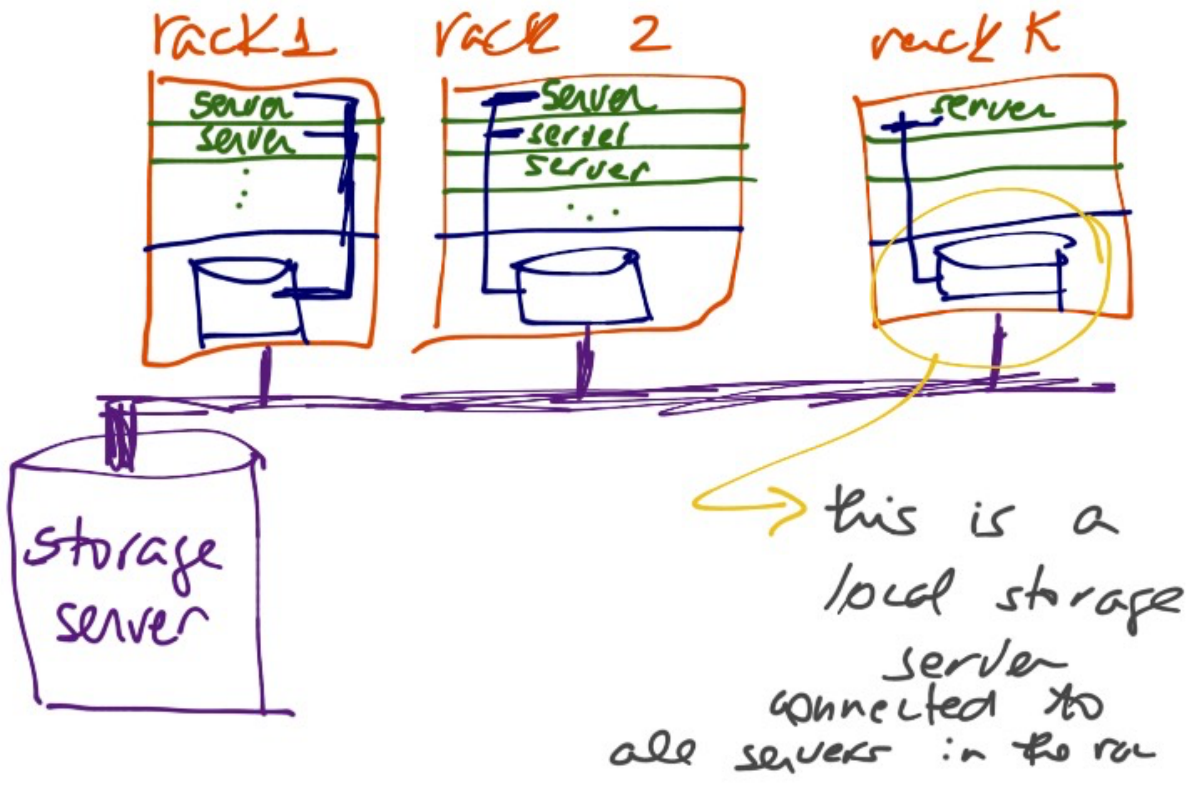

server 1
server 2
...
server n
storage server
(each server has local disks too; they are for the virtual servers to use for data!

Figure 2: new storage architecture for P4.



With the new data center architecture, image files stored in the main storage server can be copied to the storage unit for a rack, serving a number of servers. The rack storage server can be a cache of images, i.e., hold copies of the image files that are made available to the servers hosted in the rack. Once the image is copied to a rack, the servers in the rack have in-rack access to the images without having to go to the external storage server.

This new storage organization implies a change to the hardware configuration file. Now the description of the hardware includes the number of racks and the storage capacity of each rack storage unit. The description of physical server machines specifies which rack houses the server. Appendix 1 describes the new format.

Your code now needs to manage the image cache at each rack. When creating a new instance, your placement algorithm should prioritize servers in a rack where the required image is already cached. If no rack caches the image, then you should choose the ones with more free space in the rack storage unit. If no rack has enough space, you should choose a rack to remove an image from the cache such as there is space for the new cached image. You are free to choose whatever scheme you want to decide which image to evict from the cache.

The table below describes the API for AggieStack 0.2. Your job is to describe in a design document the changes you would need to do to your P4 code to accommodate this new data center architecture. Your design document should also identify the primary implementation changes/tasks, and estimate how much time a good programmer will need to implement the changes or new functionality. You should also estimate how much time should be allocated for testing the new system.

API number	**Command**	**Description**
1	aggiestack config --hardware hdwr-config.txt	As before, but with the new file configuration format as defined in Appendix 1 below.
2	aggiestack config --images image-config.txt	As before, but with the new file configuration format as defined in Appendix w below.
3	aggiestack config --flavors flavor-config.txt	As before
4	aggiestack show hardware	As before, but with the new file configuration format as defined in Appendix 1.
5	aggiestack show images	As before, but it also prints for each rack which images are cached in its rack storage server.
6	aggiestack server create --image IMAGE --flavor FLAVOR_NAME INSTANCE_NAME	Extends your virtual server placement algorithm in Part A to take into consideration the image cache.
7	aggiestack server delete INSTANCE_NAME	Same as before

8	aggiestack server list	Same as before
9	aggiestack admin show imagecaches RACK_NAME	New API: lists the names of the images present in the specified rack and the amount available in the rack storage to host more image files
10	aggiestack admin show hardware	Same as before
11	aggiestack admin show instances	Same as before

All commands generate output and update the cloud log as before.

4. What to turn in – IT IS DIFFERENT FROM PREVIOUS PROJECTS – please read!

There are submission buttons for Part A, Part B, and Part C. (Also, specific ones for the early bird deadline). For each part, you need to submit a zip file containing the following:

- Your code
- A short write-up with the following information:
  - Any information needed to be able to run your code;
  - how much time you spent implementing the part
  - The URL for your GitHub tree and the timestamp of your part solution. You are still providing the code (to make it easier on the grader), but we can also retrieve that specific version based on the timestamp you specify. This analysis will be carried out if there is the need for any additional analysis of your development history (usually when there is suspicion of plagiarism; hopefully it won't happen this term. Students sometimes make unwise choices while under pressure.) You can continue to change your tree as you work on other parts of the project, but the tree used for grading a specific part continues to be the one with the timestamp you specified in your report;
  - what are the main ideas in your design (and for Part C, the detailed design since no implementation is required).

Rubric

If you turn in Part A by its early bird deadline, you get 7 extra points.

If you turn in Part B by its early bird deadline, you get 5 extra points.

The parts have the following point distribution: Part A 35 points, Part B 45 points, Part C 20 points.

The corresponding points for a Part are graded as follows

- If failed to adhere to GitHub way of working: all points in the part are lost
- If new /changed APIs have the expected functionality, but with significant errors/failures: -20
- If new /changed APIs have the expected functionality, but with minor errors/failures: -10
- If error checking is very well done: +5 (subjective metric)

Appendix 1 – New format for hardware configuration with racks

The list of racks and their physical machines is provided in a configuration file in the following format:

```
<number of racks>
<list of <rack names, storage capacity> one per line>
<number of machines>
<list of machine spec one per line>
```

The specification of each machine has the following format:

```
<name> <rack name> <ip> <mem> <num-disks> <num-cores>
```

The file hdwr-config.txt in the provided files is an example of the expected format.

Appendix 2 – New format for image configuration file

The image configuration file specifies the number of images available to be used when starting virtual machines, providing for each one the name of the image, the size of the image in MB, and the path in the storage server to the file containing the image. The file image-config.txt in the provided files is an example of the expected format.