Texas A&M University

CSCE 689: Cloud Computing Project I

OpenStack v0.2

Aryan Sharma
UIN: 326006767 — aryans@tamu.edu

Implemented in C++

April 19, 2018

Contents

1 Project Information

This project implements a simple version of OpenStack in two steps: P0 and P1. **The github link for the code is:**

`https://github.tamu.edu/aryans/689-18-a/tree/master/P1`

The code has been written in C++ and it is contained within `AggieStack.cpp`, `AggieStack.h`, `Main.cpp`, `MakeFile`. Following are the overall functions of the files:

1. `AggieStack.h`: It is the header file of the AggieStack.cpp file and contains the class definitions, global variables and the inline functions used.
2. `AggieStack.cpp`: Contains the class function implementations.
3. `Main.cpp`: It is the main driver of the AggieStack v0.2. It contains the CLI implementations and the `main()` function.
4. `MakeFile`: Contains the direction as to how to run the program.
5. `aggiestack-log-date.txt`: Contains the log updates.

The time spent to complete this project was three days for v0.1 and about seven days for v0.2. It was done over a month in short bursts. Note that not all v0.1 commands are not suported in the v0.2 versions due to changes in the config files. To get the v0.1 the code has to be switched back to a prior git commit. However the CLI is missing in that git version.

2 Steps to run the code

1. `make clean`: to remove all the object files and previous logs.
2. `make`: to compile the program. This would generate the executable file *Main*.
3. `./Main`: to run the program. This is a helper mode where it prints how to use this binary.
4. `./Main command_file`: to run the program with a text files containing the commands. Any command that begins with `#` are treated as comments and are ignored.
5. `./Main -I`: to run the program in an interactive mode. Commands can be issued now one by one and the output can be observed in the command line.
6. `./Main -help`: to get the instructions to run the program.

3 Command Line Interface Design

A list of commands can be run using a file. The method to run it has been described above as: `./Main commands.txt`. The command line interface is reached by `./Main -I` or `./Main -i`. Some initial instructions can be fetched by `./Main -help`. A screenshot below explains this.

```
Apple@aryans:P1$ make clean
rm AggieStack.o Main.o Main
rm -rf aggiestack-log-*
Apple@aryans:P1$
Apple@aryans:P1$
Apple@aryans:P1$ make
g++ --std=c++11 -g    -c -o AggieStack.o AggieStack.cpp
g++ --std=c++11 -g    -c -o Main.o Main.cpp
g++ --std=c++11 -g -Wall  AggieStack.o Main.o  -o Main
Apple@aryans:P1$
Apple@aryans:P1$ ./Main
                                ---------------------- Usage ----------------------
                                    To read input file, type: ./Main <command_file>
                                To go to Interactive Environment, type: ./Main -I
                                          For help, type: ./Main -help
                                ---------------------------------------------------
Apple@aryans:P1$
Apple@aryans:P1$ ./Main -help
                                ---------------------- Usage ----------------------
                                    To read input file, type: ./Main <command_file>
                                To go to Interactive Environment, type: ./Main -I
                                ---------------------------------------------------
Apple@aryans:P1$
Apple@aryans:P1$
Apple@aryans:P1$ ./Main -I
                                -------- AggieStack CLI ----------
                                Version 0.2 for the AggieStack CLI
                                     Developed by Aryan Sharma
                                ----------------------------------

Entering Interactive Mode
Type command and hit ENTER to execute
Type "quit/q" to quit the program.

In [1]
COMMAND:
In [2]
COMMAND:
In [3]
COMMAND:
In [4] quit


---------- Summary ----------
Total number of Commands = 4
Total time elapsed = 0.353 ms
-----------------------------
Apple@aryans:P1$
Apple@aryans:P1$
```

Figure 1: How to run the program.

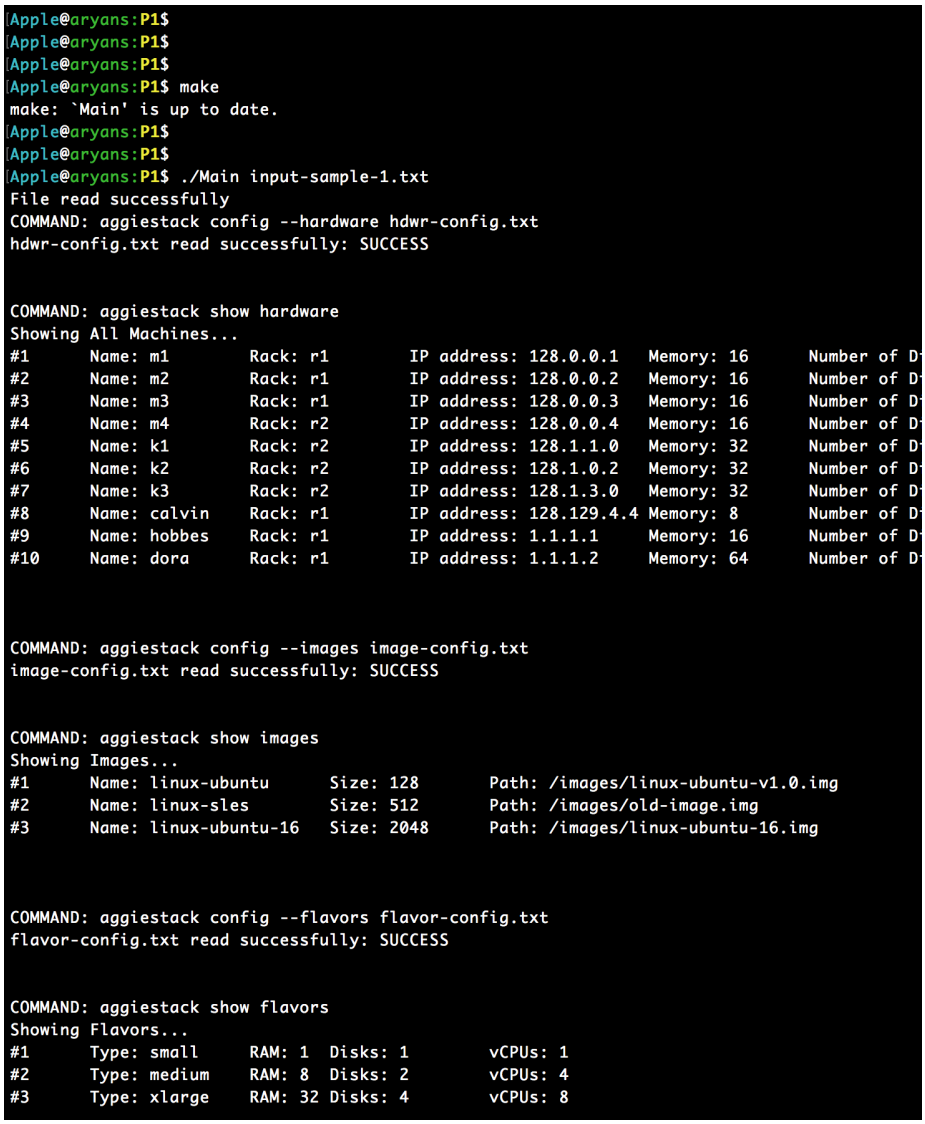

Figure 2: How to run the program using an input file

The commands supported has to be entered by the user in the CLI. Some commands that are run in interactive mode has an advanced suggestions mode. There have been some hardcoded commands other than the ones the user can enter based on the sugggested syntax which can be invoked by typing `help` in the CLI. They appear in the form of option like 1, 2 etc. and may promt the user to enter some specific detail to process their requests. This has been shown in Fig. 3. You can see that by typing `help`, a list was suggested. In the Fig.

3 option 2 was selected and the file entered was `hdwr-config.txt`. As we can see the command executed was:

```
aggiestack config --hardware hdwr-config.txt
```

In the end, it generates the summary of number of commands executed in the interactive mode and the time taken to execute them. The summary can be seen in Fig. 1.

```
Apple@aryans:P1$ ./Main -I
                              -------- AggieStack CLI ----------
                              Version 0.2 for the AggieStack CLI
                                   Developed by Aryan Sharma
                              ---------------------------------

Entering Interactive Mode
Type command and hit ENTER to execute
Type "quit/q" to quit the program.

In [1] help
COMMAND: help
---------------------- Commands ---------------------
1.  aggiestack read all config
2.  aggiestack config --hardware <hardware_config_file.txt>
3.  aggiestack config --flavors <flavor_config_file.txt>
4.  aggiestack config --images <image_config_file.txt>
5.  aggiestack show hardware
6.  aggiestack show images
7.  aggiestack show flavors
8.  aggiestack show racks
9.  aggiestack server create --image IMAGE --flavor FLAVOR_NAME INSTANCE_NAME
10. aggiestack server delete INSTANCE_NAME
11. aggiestack server list
12. aggiestack server show imagecaches RACK_NAME
13. aggiestack server show hardware
14. aggiestack admin show instances
15. aggiestack admin show hardware
16. aggiestack admin evacuate RACK_NAME
17. aggiestack admin remove MACHINE
18. aggiestack admin add --mem MEMORY --disk NUM_DISKS --vcpus VCPUs --ip IP --rack RACK_NAME MACHINE
-----------------------------------------------
In [2]
COMMAND:
In [3]
COMMAND:
In [4] 2
Enter hardware config file (relative/absolute path): hdwr-config.txt
COMMAND: aggiestack config --hardware hdwr-config.txt
hdwr-config.txt read successfully: SUCCESS


In [5] COMMAND:
In [6]
COMMAND:
In [7] q


---------- Summary ----------
```

Figure 3: Suggestion help mode in CLI

4 AggieStack v0.2 Design

The `AggieStack` has been designed as a **singleton class** which maintains a **private** list of the following:

1. Machine information
2. Available Machines while in process
3. Images that are available
4. Racks that are available for use

It has some categories of APIs which include `set` and `get`. Besides it has `read` comands to read the files, `show` comands to show the states, and some special functions like create, delete and evacuate etc. A single instance of class `AggieStack` is created and is maintained throughout the sessions to avoid any extra copy or corruption of any internal data. Also it makes the maintainance of the states quite easy.

Other classes are that of `Machine, Image, Flavor, Rack,` and `Instance`. The hierarchy in which they are connected is as follows. The singleton instance of `AggieStack` maintains the racks and also pointers to machines that are running. The `Rack` inturn has cached images information and the list of machines that are hosted on that rack. The `Machine` on the other hand has an id to the rack on which it is hosted and can provide that information through the `AggieStack`.

Besides, when the image, hardware and flavor config files are read, they create the `Image`, `Flavor` and as per the requirement the `Machine` objects from the class interface provided. They are maintained thorought the sessions except in the situation where they are either forced killed or have to be removed from the system. Each machines have the capability to host `Instance` class objects which are the user requested instances that they need to run. The information about the running instances are with the `Machine` themselves.

All of the classes have in addition some special functions that help them to update their states and interact with each other. Everything has been designed such that the `AggieStack`, `Racks`, `Machine`, `Instance` hierarchy is maintained.

4.1 Part A design

The part A design required the commands that create virtual servers. This was achieved by `AggieStack` by calling the Machines, deciding if they can host and then allocating the ressources to the instance. The states were internally updated and were reflected through `show` commands. If no physical machine has the required amount of ram, disks, and vcpu available, it indicates the error that it can't add the instance. The policy that was used here was to provide the instance that Machine which is found first to have available space. This was changed in part C.

4.2 Part B design

In this part the AggieStack admins evacuate the rack by invoking evacuate command. It migrates all the virtual machines in a server residing in the faulty rack to another rack. In case no rack has free space, it kills those instances. Also it removes the specified machine from its view of the datacenter though the remove command and add a new machine to the system so that it may receive new instances.

An interesting thing about this design was that many functions were re-used internally to make the code modular and reusable. This was kept in mind while designing the `remove_machine, can_host` commands.

4.3 Part C design

In this part the image files stored in the main storage server is copied to the storage unit for a rack, serving a number of servers in its cache. It holds 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. The list of cached images is maintained by every rack and is sorted based in their available memory.

When an instance is requested, the image of the instance is searched across all racks. In case one of the rack has the image, it checks if any of it machines has the capability to host it. In case it cannot, it looks on other racks, otherwise it creates an instance there and returns. In the case that none of the racks have the cached image, it copies the image from the central storage, and creates an instance in a rack which has maximum amount of storage. The caches list is updated based on the storage of the particular rack. It's a queue kind of data structure where the oldest used image is dropped from the cache to accomodate a newer image. When all such search is exhausted, `AggieStack` return an error that it cannot host the instance due to memory issues. Please refer to the code and git commits for more details. Few of the examples have been shown in Fig. 4.

5 Other design decisions

The error check has been quite strict in the command line and therfore it promts user to use the suggestion mode in the interactive mode. It logs the error in the log and prints on the console when an error happens. The `logger` function is responsible to log the results in the file `aggiestack-log.txt`. The file contains the timeline of the execution. An example is in Fig 5.

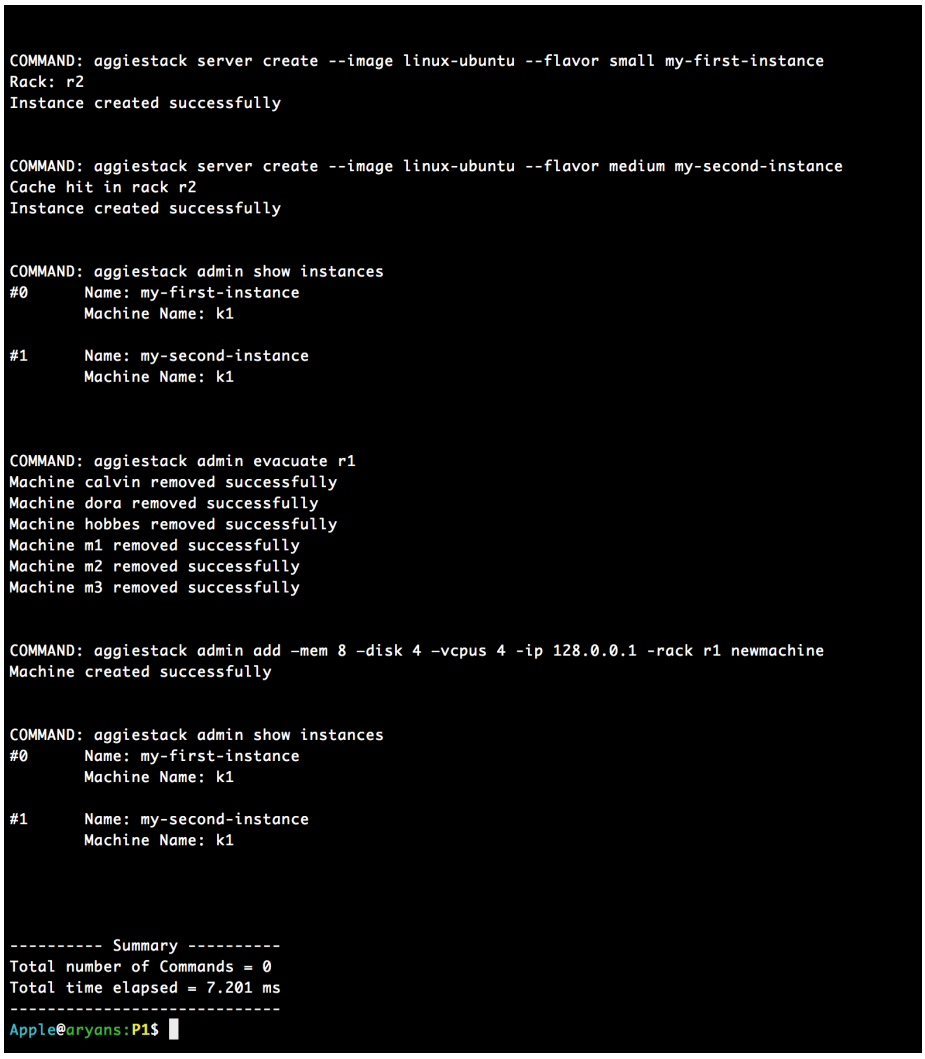

Figure 4: Examples

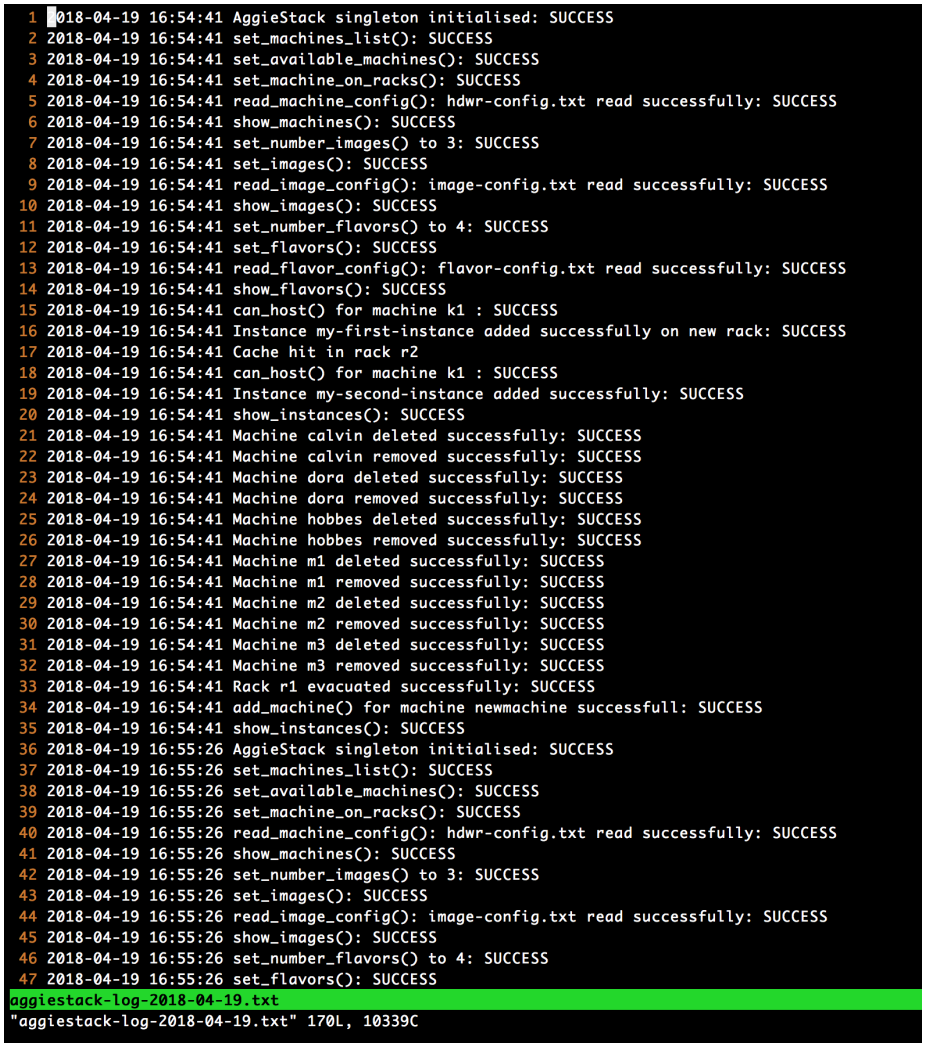

Figure 5: Log file entry