Project4 Part C

Nathaniel Leake, 424003778
Cloud Computing, Fall 2017
Time to complete Task: **3hrs**

Compiling & Executing

Compile by navigating to the directory containing the code and typing

- `javac AggieStack.java`

To run the compiled project, type

- `java AggieStack.java`

GitHub Link

Here is repo at its final stage. This version includes final submission zips, reports for parts A, B, and C, and final tweaks to the user interface and command help menu.

- https://github.tamu.edu/nateleake/489-17-c/tree/9c2faa859b402cea90586f3218567b133b2d6414

Main Ideas

My primary focus while designing this architecture was to maximize correctness & performance of the program. Secondarily, in the actual code involved (I accidentally implemented a decent bit of Part C) I spent a lot of time ensuring user-friendliness of the system. Ease-of-use and understandability are a tremendous aid to anyone diagnosing or upgrading a cloud system.

Introduction

The changes described in Part C impact the input of hardware, the organization of image storage, and the algorithm that determines placement of new virtual instances. In this report I will discuss how these changes will affect my design of the system from Part B, what methodologies should be employed to maximize effectiveness and improve performance of the new system, and how the final system will look from a structural perspective.

Design Considerations

Since my Part B system is able to assume that every machine has access to every image, images are stored in a "global scope" map that is queried each time a new machine is added. This accurately simulates a single storage server for the cloud cluster, but does not match the format of AggieStack 0.2 (Part C). Additionally, placement of machines in Part B is done using a function in the AggieStack that selects the host rack and machine based purely off of memory availability. For v0.2, it will become important to also consider whether the target rack has the requested images available in its storage server.

Architectural Strategies

Input format changes are straightforward and are comprised of simply seeding the global storage server with all available images for use in the cloud. Storage servers can be represented per rack by adding a hash set (or other fast-lookup structure) of images to each rack object. For maximal performance in allocating a virtual instance, racks currently keep track of the server with the most currently available memory (updating this is an O(1) operation). For Part C, it also becomes worthwhile to query for racks that already contain a copy of the images being used in order to minimize the number of racks that need to be search through to find a place for the instance. If no rack currently contains a copy of the image in its storage server, we can take the rack with the most available server space. Since links to the racks are sorted by available memory as mentioned above, simply

take the first rack and remove images from its storage server with the following strategy: while unable to hold the new image, remove the image that has not been used in the longest time. This way, as old images begin to phase out (Ubuntu14 to Ubuntu16, for example), the new images with higher demand will be easier to find and use. This strategy is better than just removing the largest image until there is space. The big issue with the strategy of removing the largest image is that if the removed large image is frequently used it is quite possible for another instance to come along and put it back into this rack's storage server, removing the image we just added in the process. This could result in a loop where the two largest image files are constantly being deleted and rewritten in turn to the storage server. Lastly, if there are multiple racks with free space and varying image cache sizes, it is better to write a new image to a rack whose cache will not require another image to be deleted. This reduces the number of image removals, which in turn helps increase the number of images available and reduce the need to rewrite images.

Detailed System Design

This paragraph describes the actual changes to the code needed to complete Part C, as well as listing some concerns in terms of physical limitations and reliability of data. As already stated, changes to hardware configuration format are straightforward and fall into Part B along with the structure of racks. Images should continue to be stored in the AggieStack HashMap to simulate the global storage server, but now also need to be stored in a HashMap structure within each rack. The image HashMap, by the way, maps an image's name to all other relevant data on an image for quick lookup. Another implementation tactic is for racks to store machines in a sorted data structure (such as a set or multiset) that automatically updates a machine's position when a new instance is allocated to that machine. This lets the rack holding the machines determine it's freest machine in O(1), and locate its machine with available memory closest to X in log(N) time, where N is the number of machines in the rack. This is useful for minimizing memory fragmentation. Total available space can also be kept track of in O(1) time by using a variable in the rack that gets updated with each new instance that comes to or leaves from that rack. One level higher in the architecture, the AggieStack structure used to hold references to each rack can be sorted based off of rack memory available, with lazily propagated updates to the sorting position of significantly modified racks at some specified time interval, probably set between 15 minutes and 1 day, depending on the size of the racks and the number of racks. The biggest limitation to the AggieStack simulation of a cloud cluster is that it is built and run in a synchronous system. Thus, when many virtual instances are being deployed at once, it may take the system a long time to fit each of them into the system, even when there is amply available space.

Closing Notes

Overall, from both implementation experience and careful estimations, I predict that the time required to fully implement Part C from my structure as defined in Part B would take no longer than 4 hours, which is the same upper limit I would place on each other phase of the AggieStack project. Thus, I believe requiring implementation of Part C would not render this project significantly more difficult. That being said, I feel this project (P4) might be better administrated as several separate homework assignments, or alternatively to make the early-bird deadlines into hard deadlines. If I had not worked ahead to complete the requirements for each part of the project by the associated early-bird deadline, I could foresee myself having gotten swamped on this project at the end of the semester.