Final Report Standing Cloud, Inc.

Submitted by: Byron Anderson, Kendall Aubertot, and Alec Westerman

Table of Contents

Abstract

Standing Cloud, Inc. seeks to provide a friendlier interface for end users of cloud computing servers. This product is a Java application with configuration details as input and a bootable server image as output. This image is configured up to, but not including, the point of final application installation.

This system is important because it saves the end users of Standing Clouds products time and money. This system lowers costs for end users because users pay for cloud computing services by time and bandwidth. The difficulty in this area comes from the need to interact with elements belonging to both Standing Cloud and the cloud computing services. This requires interacting with the Standing Cloud API, the APIs belonging to each individual cloud service provider, the cloud computing virtual machine controls, and the storage systems employed for server images. Standing Cloud is looking to make these interactions more efficient.

In general, using scripts to configure servers is a better practice than relying on frozen images because of the inherent flexibility. This process, however, takes longer than booting from images. Automating the creation of images from configured servers provides the best of both worlds.

The product was built using a three-phase approach. The first phase consisted of researching which tasks require interactions with which services. The second phase consisted of requesting updates to the Standing Cloud API so that it included any additional functions that might be necessary, as determined in step one. The final phase consisted of writing the code that accomplishes the described task.

Introduction

The use of cloud computing services generally requires a certain degree of specialized computer knowledge. Users are often forced to accomplish tasks through a terminal window and the Application Programming Interface (API). Standing Cloud makes cloud computing more accessible to the typical computer user by allowing users to install applications through a graphical user interface designed specifically for interactions with cloud computing services. These applications are dependent on a number of services (such as a web server and database). Such services are referred to as the application technology stack.

Currently, when a user installs an application using Standing Cloud's services, a virtual machine is created and booted. Elements of the application technology stack are installed through configuration scripts, and then the application is installed. The configuration scripts that install the stack take time and demand bandwidth, both of which cost money when using cloud computing services. Booting a virtual machine from an image configured to contain the desired stack combines the first two steps, thereby

saving time. These images, however, are static and therefore do not update with new releases.

In order to keep the images up to date, the product automates the image creation process. Automating image creation allows the images to be updated frequently, quickly, and easily. An automated image process provides the flexibility of running scripts with the configuration speed and bandwidth demand of booting from images.

Requirements

A number of project specifications were laid out. The system consists of a JAR file wrapped by a bash script. The JAR file needs to interface with the Standing Cloud API, the individual cloud service providers' APIs, and command a virtual machine residing on a cloud server to create a configured bootable image based on configuration inputs, upload that image to a cloud storage server, and return the identifier for the image created. The project shall be coded in Java, with a small bash-script wrapper. Because the imaging and storing process occurs on different cloud computing systems, and a significant number of cloud computing services are likely to be added in the future, the final application must be easily extended to support different cloud computing systems. The goal of the project consists not only of creating usable images, but also of creating a system that can be expanded to work on as many different systems as possible.

The images the application creates hold very specific data. Every bootable image needs to contain a functional operating system. The image should also hold a file system in a specific configured state, determined by the application technology stack the user requested. For example, an image on a server may contain Ubuntu's Hardy Heron release, and installations of Apache, MySql, and PHP. Clearly, all this data must be valid and functional in order for the images to be bootable and work properly.

Users of the final application will be knowledgeable in software engineering and cloud computing. Proper documentation and style has been dictated through a Java Code Conventions document. Examples of the type of conventions that must be followed include file names and locations, variable names, and the amount of white space between elements. The code is expected to be documented in the Javadoc style, with appropriate comments placed throughout the code. The code must be readable, maintainable, and extensible.

Because employees of Standing Cloud will extend the code for the final application after the end of the 6-week field session, the code is expected to meet certain quality standards. Beyond the documentation discussed above, the application has significant unit test coverage. Unit tests reduce worry when editing or adding to existing code. Additionally, the images created have been tested in order to make sure they boot correctly and have all of the necessary installations. Quality is an integral part of the final project to be delivered to the client.

The system is designed to handle Rackspace, Amazon, and GoGrid. Later on, Standing Cloud may expand it so that it works with all of the services that Standing Cloud works with. The other service providers that Standing Cloud currently supports are Flexiscale, and Slicehost.

Below are the use cases that the product satisfies.

Use Case 1: The user does not specify a number of fields

- 1. The user interacts with the application via a wrapper bash script. Through that interaction, the user does not enter some or all of the parameters.
- 2. The wrapper script feeds the JAR file the contents of a file containing default values for the fields the user left blank. These fields include the cloud computing services, stacks, and credentials to use.
- 3. The image is generated and the location of the image is returned.

Use Case 2: The user specifies a number of fields

- 1. Interaction occurs through the wrapper bash script. The user specifies some or all of the services, stacks, and/or credentials by entering each with the appropriate flag.
- 2. The wrapper feeds the JAR file the contents of the user's requests.
- 3. The specified image is generated and its location is returned.

Use Case 3: The script is run automatically

- 1. The wrapper script is executed on a schedule. All configuration input and credentials are provided via a file named `.image_profile`.
- 2. The wrapper feeds the JAR file the contents of the user's requests.
- 3. The specified images are generated and their respective locations are returned.

In order to accomplish the project goals, the final application interfaces with Standing Cloud's API. In the lone case where functionality was missing in the Standing Cloud API, changes were made to the API in order to facilitate image creation. The application interfaces directly with the cloud computing vendors' separate APIs in order to perform tasks not supported by the Standing Cloud API.

The project culminated in a presentation to demonstrate how the code works to Standing Cloud. Changes made to Stand Cloud's API were requested early so that the employees of Standing Cloud could implement the desired changes. All the code generated has been delivered to Standing Cloud, which will either use it to create images or use the code and process to expedite the creation of a new version of the application.

The project required a number of skills from the CSM Team. The entire team was already comfortable with Java programming and shell scripting. The project also demanded large amounts of coding using 3rd-party APIs. The project required the most

extensive use of 3rd party APIs that any of the team members had previously experienced. Communication with Standing Cloud personnel and individual research helped the CSM Team overcome the learning challenges.

The risks of this project dealt mainly with documentation. The cloud service providers' APIs occasionally suffered from poor error reporting. For example, the CSM Team experienced one instance where an error message requested that authentication information be renewed. Renewing this information did not produce results. The actual problem was that the wrong URL was being queried. Such eccentricities in the APIs led to time-consuming debugging.

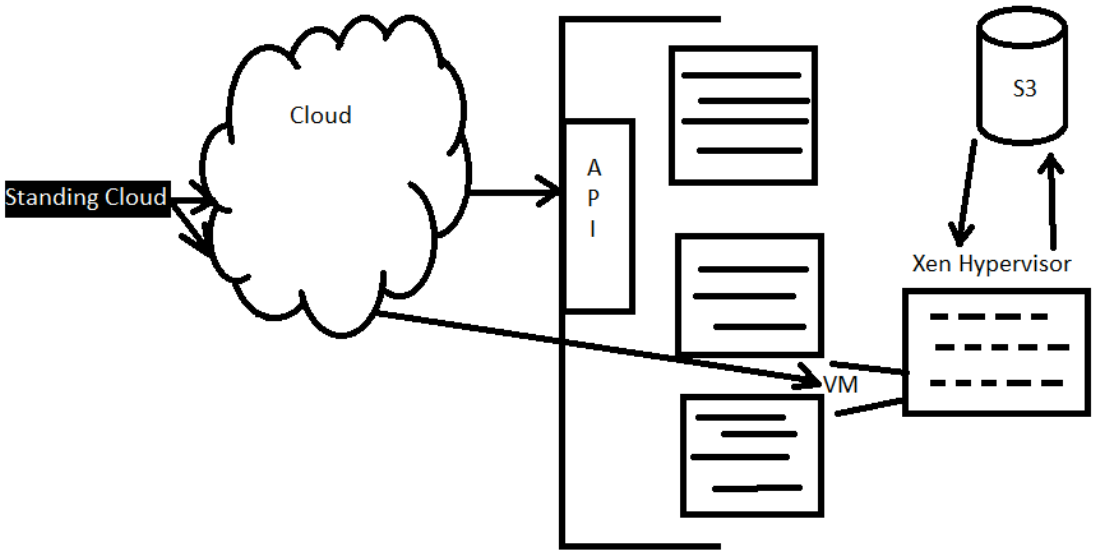
Finishing the project within the rigid time constraints of the CSM Field Session term was an important aspect of the project. The CSM Team learned to manage time effectively and ask for resources from the Standing Cloud team as soon as it became apparent they were needed.

Design

The system takes configuration details as input and outputs image identification details. The application is run from the command line. It is easily extensible to support new cloud providers and stack configurations. The application design makes use of inheritance. Each of the cloud service providers has a dedicated class, which is inherited from a parent class named `DefaultImageCreator`. `DefaultImageCreator` implements the interface `ImageCreator`. This modular approach makes the addition of new providers easy.

Although each cloud computing service provider class varies in its design, there are a number of similarities between all of them. The system designed by the CSM Team interacts with the Standing Cloud API, the API for each of the cloud computing service providers, and the virtual machines located on the cloud computing services' servers through bash scripts executed via the Standing Cloud API. Figure 1 illustrates the interactions for Amazon EC2.

Figure 1. Inner workings of Amazon EC2



Interactions with the API of the appropriate provider and the API of Standing Cloud enable interactions with the server instance. These interactions can be used to create an image of the running instance. That image is then stored on Amazon's storage service known as S3. The result is that, during future uses, the instance is booted into a useful, configured state, rather than a state in which configuration scripts must be run.

The system consists of an `ImageAutomator` class, which contains the `main`, the `DefaultImageCreator` abstract class, and several `DefaultImageCreator` children. The `ImageCreator` classes implement the `ImageCreator` interface. `DefaultImageCreator` contains the method implementations for allocating and terminating server instances through the Standing Cloud API. The child classes (i.e. `AmazonImageCreator`, `GoGridImageCreator`, and `RackspaceImageCreator`) implement the image creation. Three cloud computing providers are supported: Amazon Elastic Compute Cloud (EC2), GoGrid, and Rackspace Cloud Servers. After the allocation of an instance, the program uses API calls to the appropriate cloud provider or runs scripts on the server instance in order to create an image. The image is then stored to the provider's servers so that it can be booted from in the future.

The `ImageAutomator` contains the `main` function of the program. It handles arguments passed to the application when called, and creates and starts the necessary `ImageCreators`. An Image Creator allocates a server instance using the Standing Cloud API's `allocate` call. Once the instance is initialized, as returned by the Standing Cloud API, the configuration script that prepares an instance is run automatically. Once the instance is configured, the image is created by either executing

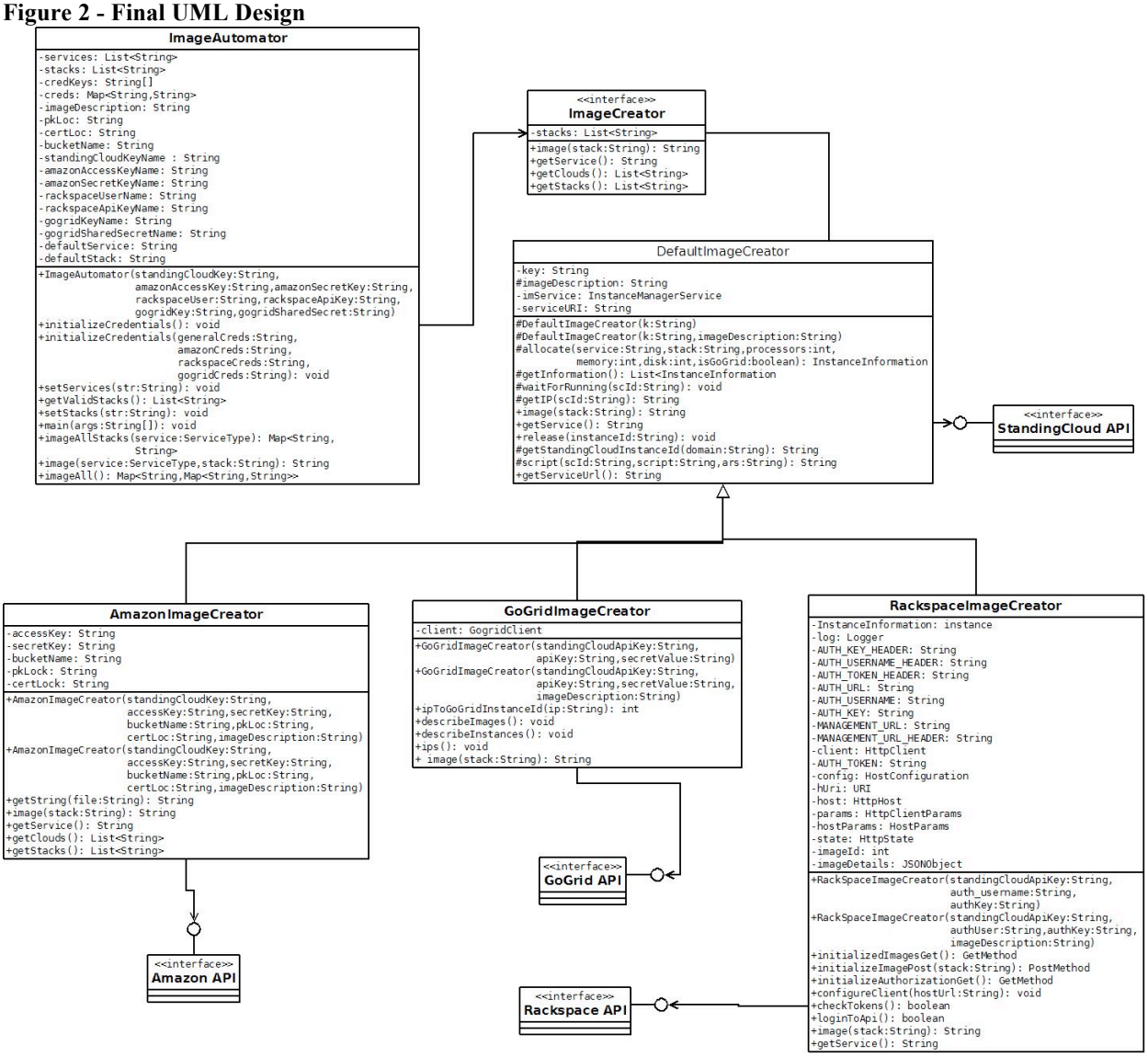
scripts on the instance (in the case of Amazon EC2) or through a series of API calls (in the cases of GoGrid and Rackspace). The image is then stored. The creation and storage process is specific to each cloud provider. In the case of Amazon EC2, API tools are installed on the server and the image is created through the script. In the case of Rackspace, image creation occurs through a function call in the Java application. GoGrid requires allocating an instance known as a Sandbox instance, running a python script on the server instance, shutting down the instance, and then making an API call. Once the storage process is completed, a call is made to the Standing Cloud API to terminate the instance created.

List of basic class concepts:

- `ImageAutomator`
  - Process input from tool wrapper
  - Creates instances of necessary classes
  - Calls appropriate functions
- `ImageCreator`
  - Introduces the benefits of an interface without supplanting those of an abstract class
  - Causes the end product to be more extendable
- `DefaultImageCreator`
  - Provides abstraction for cloud service providers
  - Contains redundant functionality between cloud provider classes including:
    - Instance allocation
    - Instance release
- Cloud Provider Image Creators
  - Interface with APIs to create and store the appropriate image from the stack requested
- Cloud Provider APIs
  - Handle calls to create an image specific to the cloud
  - Handle calls to store an image to the cloud
- Standing Cloud API
  - Handles calls to allocate instances
  - Handles calls to terminate instances

Figure 2 is a UML representing the CSM Team's design. The CSM Team did not implement the APIs or the `GoGridClient` class.

Figure 2 - Final UML Design



As with all applications, quality assurance is an important part of this project. Furthermore, the client has requested that reasonable testing be done. Therefore, the java application includes as much unit test coverage as is possible. All images produced by the application have been tested manually to ensure all the necessary functionality is available. This functionality includes the capability to SSH, the application technology stack, and basic operating system operations. No automatic validation of bootable images is available. As a result, the images created are booted. An application is then installed on the resulting instance.

Results

The end product consists of a JAR file enclosed in a wrapper bash script that interacts with the APIs of Standing Cloud, Rackspace, Amazon, and GoGrid. Java was selected because the client specified it as a requirement. Java interfaces well with the Standing Cloud API and is the language of choice for programming throughout the company.

The project also uses bash scripts. There is a bash script that serves as a wrapper for the JAR. The Amazon image creation process uses bash scripts for both saving the image and sending it to Amazon's servers for future use. Bash scripting is useful for interacting with the running server instances. Bash scripting is also used to install an end application on an instance, which serves as the team's integration test.

The user runs `ImageAutomatorWrapper.sh`, the wrapper script, with a series of option flags. Each of the options must be followed by one string argument, except for -v and -h, which are flags signifying to run the application in verbose mode and bring up the help page, respectively. The -c and -s flags may be repeated. All flags, other than the -v, -h, -f, and -i options, are required. The wrapper script, however, can detect a configuration file named `.image_profile`, which holds default values for each option. Flags not given in the command line will be pulled from this file. In the case that neither the command line arguments nor the `.image_profile` file provide all the required flags, an error is returned. Below is a list of all of the flags:

- -c [service(s)]
- -s [stack(s)]
- -d [Standing Cloud key]
- -i [image name]
- -a [Amazon access key]
- -r [Amazon shared secret]
- -b [bucket name]
- -w [Amazon's Account ID]
- -x [Amazon's pkLoc]
- -y [Amazon's certLoc]
- -u [Rackspace user name]
- -p [Rackspace key]
- -k [Gogrid key]
- -e [Gogrid secret]
- -f [profile]
- -v
- -h

Multiple –c and –s options are allowed. Presence of the –v flag enables verbose mode. Presence of the –h flag outputs the commands.

The Java application makes numerous API calls. The application interfaces with the Standing Cloud API in order to allocate and release server instances. The application uses the cloud computing service APIs for the imaging process.

The CSM Team faced a number of issues. Aside from the previously mentioned authentication issue, the CSM Team encountered an issue involving inconsistent image creation with GoGrid. The GoGrid process requires that the instance to be imaged is shut down before imaging occurs. A support ticket filed with GoGrid helped the team realize that the instance was not being shut down correctly. It was determined that the Standing Cloud API was restarting images that had shut down.

One final challenge that the CSM Team had to overcome was the exclusion of the `www-data` directory from the imaging process when using Amazon EC2. At the point in time at which this mistake was discovered, an Amazon AWS command-line function call to image the instance was used with a flag to exclude certain directories, such as `proc` and the directory containing scripts executed to image the instance that should not be included in the image. Executing this call in verbose mode showed that the `www-data` directory was also being excluded. The problem was resolved by using a flag to include all directories in the image, omitting a few sensible directories, such as `proc` and `mnt`. The files to be excluded are now migrated to the `mnt` directory before imaging all sensible directories.

The end product takes configuration details as input, and outputs an identifier allowing the user to locate the image created. The Java code's execution is tested with using JUnit tests. The Eclipse plugin Emma helped the team determine which parts of the code were tested with JUnit tests. The bulk of the application's code lies in the `image` functions of the children of the `DefaultImageCreator` class. It is difficult to produce any meaningful unit tests for this function.

In part because of the difficulties in creating meaningful JUnit tests, an integration test to ensure the validity of the images created is used. The images created are tested by installing an application on an instance allocated by booting from one of the images created using the java application.

Scope

At the onset, the minimum functionality specified by Standing Cloud consisted of Java to automate the creation and storage of images using one cloud computing service. The client expressed a certain expectation that the project would support at least two cloud computing services. The application was to take configuration details as input, and output an image identifier. The configuration details are given by the name of the application technology stack that should be installed on a running instance booted from the image. The final system supports Amazon EC2, Rackspace, and GoGrid. The application can take multiple configuration details as input, in which case it outputs

identifiers for all images created. The application also supports creating instances on multiple services at one time. A bash wrapper script has been written for convenience.

As previously stated, the project was executed in a 3-phase approach. The first phase consisted of researching the tasks that needed to be accomplished and determining where the implementation for these tasks lay. The second phase involved determining what changes needed to be made to the Standing Cloud API and requesting that those changes be made. The third phase was to implement the application. Although there were no rigid lines between the phases, we found the project closely adhered to these 3 phases. The research and initial, non-automated, creation of images took slightly over a week. Only one change to the Standing Cloud API was necessary. The change was made a few days after the appropriate request to Standing Cloud personnel. The rest of the project consisted of implementation.

Amazon Process

As the CSM Team started work with Amazon, initial research pointed to using the command line imaging tools `ec2-bundle-vol` and `ec2-bundle-image`. The team played around with these two until they had a functional imaging script, made of calls to Standing Cloud's instance manager command line tool, scp and ssh, and assumptions of what would be exist on a running instance. This initial product resulting from this route turned out poorly because it required much in terms of interaction with a user. Attention was shifted to Amazon's SOAP API.

The CSM Team learned more about Amazon's API and tried to use a function called `CreateImageRequest` to have the API do the imaging brute work for them. However, the team quickly ran into trouble with this because the API call does not work with non-Windows instances. Knowing this, focus was shifted to investigating the use of scripts for imaging instance.

Moving back to the use of a script, previous failures were remedied with some ingenuity. To copy files non-interactively, as well as just running commands, the Standing Cloud function for running scripts on instances was employed. Assumptions of existing packages were removed, and tool installation functionality was added. Another challenge that presented itself was having the image just include what is on the server before the scripts begin. The CSM Team overcame this obstacle by placing its installed files into locations not imaged by `ec2-bundle-vol`. A final, finishing touch challenge that presented itself was proper error handling: having the tool fail as early as failure was detected, and fail in a descriptive way when exceptions arose.

GoGrid Process

The basic framework of the GoGrid imaging process presented itself quickly: first request an instance from Standing Cloud, wait for it to be ready, map it to a GoGrid instance ID, prepare the instance for imaging, send the imaging request, and release it. Finding out how to use GoGrid's RESTful API was a bit of a hurdle, but did not turn out

to be the most time consuming effort. As is often the case, debugging the process turned out to be the largest hurdle for working with GoGrid. The initial process was extremely inconsistent – it would sometimes work fine, and sometimes fail. There were no obvious difference between the two cases from a debugging standpoint, and after much help from Standing Cloud engineers and GoGrid support, it was determined that the imaging that failed was due to the server being prepared improperly for imaging. It turns out that GoGrid doesn't automatically replace an instance's kernel with every boot, so Standing Cloud's configuration files reboot the server to replace it manually. This individual problem took several hours of troubleshooting, and the solution makes the tool (and GoGrid instance allocation in general) very slow.

Rackspace Process

The creation of the `RackspaceImageCreator` class presented many learning opportunities. The process to create a Rackspace image is straightforward. First, and instance must be allocated. Once that allocated instance is configured, make an API call to image the desired instance. The `image` API call is asynchronous, and returns an identifier. The application then acquires the identifier and checks the status of the image through another API call. The image progresses through the `QUEUED`, `PREPARING`, `SAVED`, and `ACTIVE` states. The image status is set to `FAILED` if the image creation process is unsuccessful. Upon ensuring that the image has been completed successfully, the image identifier is returned.

The Rackspace API is RESTful. Representational State Transfer (REST) uses the client-server model. An application is said to be either in transition or at rest. When the application is in the rest state, the user can interact with it, but there is no load on the network. When the application is waiting for the response from an HTTP request, the application is said to be in transition. None of the members of the CSM Team had previously used REST.

While REST removes a certain amount of abstraction from the application, it allows low-level control of the HTTP requests that are made over the network connection. Rackspace's API uses REST to allow applications to send requests to specific URLs. Each API call sends a request to a different URL. The application can send `GET`, `POST`, and `DELETE` requests to the server. For example, the "/images" path at the Rackspace API server's URL returns a list of images if it receives a `GET` request, creates an image if it receives a `POST` request, and deletes and image if it receives a `DELETE` request.

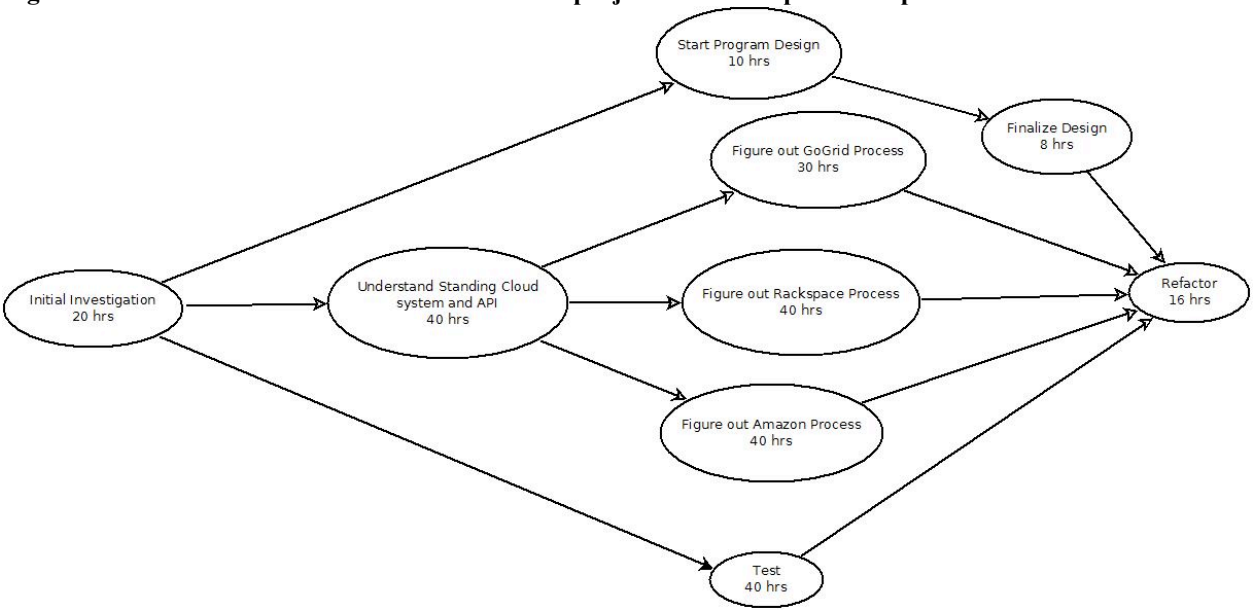
In order to use the Rackspace's RESTful API with Java, the CSM Team created the HTTP request objects. Each request must detail the URL to query and provide the appropriate HTTP headers for authentication and describing the response encoding that the application accepts. `POST` requests must also include a request body, whose encoding must also be specified in the headers. The system created by the CSM Team uses JSON encoding.

Currently, Rackspace does not support storing images with their cloud storage service Cloud Files. Instead, images are stored on the server and are associated with an instance. As a result, the image is deleted once the instance is deleted. The next version of the Rackspace API is expected to support storing images with Cloud Files.

Schedule Summary

The efforts by the CSM Team are detailed in the PERT Chart in Figure 3.

Figure 3 - PERT Chart. Much of the work for the project was accomplished in parallel.



Conclusions

The client was very clear about a number of lessons regarding working in industry. The first lesson was to list all time constraints when setting up meeting times in order to keep the number of emails sent to accomplish this task to a minimum. The second lesson was that just because an individual had said they knew they were responsible for a certain task doesn't mean they remembered the next day or were actively working on it. The client recommended checking in with members to ensure work was being done and also pointed out that being on location helped with this a lot.

The final project could be extended in a number of ways. Currently, the Rackspace API stores images created with the server instance. Once the instance is destroyed, the image is lost. This could make keeping images from which to boot cost-

prohibitive. The next release of the Rackspace Cloud Servers API will allow the images to be uploaded to their cloud storage service known as Cloud Files.

Also, In order to minimize the total time the application takes to make several images, the program could make use of threading. Most of the time spent creating the images is currently spent waiting for the server instances to be configured. Furthermore, the next greatest amount of time spent running the application involves using compute cycles on the remote server. The project would see significant performance increases when imaging multiple instances by taking advantage of threading.

The project has been very useful as a learning tool for the members of the CSM Team. The team members got the opportunity to hone skills developed in courses they'd taken at the Colorado School of Mines. The members also gained valuable experience working with 3rd-party APIs. The members learned how to delegate tasks amongst themselves and hold each other accountable for the functionality of specific components. Maintaining communication with the client helped the team ensure it was on the right path, and Standing Cloud's employees were a valuable resource to the team.

Glossary

API - Application Programming Interface

REST - Representational State Transfer.

SOAP - Simple Object Access Protocol

Bash – a scripting language for UNIX machines

Cloud Computing - Internet-based computing in which resources, software, and information are provided on-demand and in a scalable and elastic manner following the client-server model.

Image - A single file containing the complete contents and structure representing a data storage medium or device.

JAR file – Java Archive, an archived and consolidated file that contains what was earlier a project.

S3 – Simple Storage Service, the storage space provided by Amazon Elastic Comput Cloud (EC2)

Standing Cloud - the company contracting work from a group of students in the Colorado School of Mines field session that provides a user-friendly interface to end users.

Wrapper Script – a script that serves as the medium between the terminal and a program.

Xen Hypervisor – a virtualization machine monitor

Bibliography

Amazon EC2 API Tools [a website]. Amazon Web Services, 2010. [cited May-June 2010]. Available at http://developer.amazonwebservices.com/connect/entry.jspa?externalID=351.

API [a website]. GoGrid, 2010. [cited May-June 2010]. Available at http://wiki.gogrid.com/wiki/index.php/API.

APIs and Documentation on SDN [online]. Oracle, 2010. [cited May-June 2010]. Available at http://java.sun.com/j2se/1.5.0/docs/api/.

Cloud Servers Developer Guide [a pdf document]. Rackspace Hosting Inc, 2009. [cited May-June 2010]. Available at http://www.rackspacecloud.com/cloud_hosting_products/servers/api.

Flangagan, David. JAVA IN A NUTSHELL [a desktop quick reference]. Jamie Peppard. 5. Beijing: O'Reilly, March 2005. [cited May-June 2010].

Haggar, Peter. Practical Java [paperback]. Kerninghan, Brian. 1. Boston: Addison-Wesley, May 2000. [cited May-June 2010].

Hornstmann, Cay. Big Java [paperback]. Fowley, Don. 3. USA: John Wiley & Sons, Inc, 2008. [cited May-June 2010].

Instance Manager [a pdf document]. Standing Cloud. [cited May-June 2010]. Available via direct communication with Standing Cloud.

Interfaces vs Abstract Class [a website]. Java Glossary, 2008. [cited May-June 2010] . Available at http://mindprod.com/jgloss/interfacevsabstract.html.