Design Document

02/23/2018

Version 2.0

Team: Nimbus Technology

Sponsor: IBM

Faculty Mentor: Austin Sanders

Team Members:

Itreau Bigsby

Matthew Cocchi

Richard “Riley” Deen

Benjamin George

Table of Contents

1 Introduction

Our client is Daniel Boros, a senior software developer working on IBM's Spectrum Protect Server Development, which offers a suite of services to IBM's clients to help them manage cloud data storage that those clients have purchased from other vendors. One of the most important of those services is the ability to drastically reduce the data storage needs of its clients through various methods.

Of course, as the amount of data storage needed by a business continues to increase, so does the cost of that data storage. Per Figure 1 below, an example corporation using 6 petabytes of data storage on Amazon Web Services (AWS) incurs a monthly cost of $144,000.

Cost of Data Storage on AWS			
Storage Plan	Price per Month	Example Storage	Cost per Month
50 TB / month	$0.026 per GB	50 TB	$1,300
51-500 TB / month	$0.025 per GB	200 TB	$5,000
Over 500 TB / month	$0.024 per GB	6 PB	$144,000

Figure 1: AWS Storage Costs

It is important here to take a moment to discuss how Spectrum Protect interacts with AWS to cover a client's data storage. When a client pays for storage on AWS that storage takes the form of "objects," which are simply pieces of data of variable size. When that client pays for Spectrum Protect to manage their data, Spectrum Protect interacts with the data using "containers," which hold a variable amount of the client's data, up to 1 gigabyte.

Cost of Cloud Data Interactions on AWS	
PUT, COPY, or POST Requests	$0.01 per 1,000 requests
GET and all other Requests	$0.01 per 10,000 requests

Figure 2: AWS Request Costs

Each Spectrum Protect container maps to exactly one AWS object, and vice versa. Thus an example client with 50 terabytes of storage on AWS would have their data split into *at least* 50,000 containers by Spectrum Protect. Considering the prices shown in Figure 2, and the scale of storage needs of Spectrum Protect's clients, making requests for thousands upon thousands of AWS objects can become quite expensive--a metric we consider more in depth in this document.

To help clients manage these cloud storage costs, Spectrum Protect offers multiple ways to cut down the total storage needed by a client by as much as 90%, bringing the cost of the example 6 petabytes of data storage from $144,000 per month down to as little as $14,400 per month. This is a savings of nearly $130,000 per month, and over $1,500,000 per year, making Spectrum Protect's work quite valuable to clients in need of cloud data storage.

However, we at Nimbus Technology are offering another way in which Spectrum Protect could reduce data storage: "reclamation." This refers to taking expired data--that is, data that is not referred to anywhere else and thus is not necessary--and simply removing it. By removing expired data from a client's cloud storage, we can reduce the storage needs of a client while still maintaining the integrity of the client's data.

To truly capitalize on the removal of expired data, though, another step must be taken: reformatting. All of the spaces in a container that are empty after reclamation are still considered "occupied" due to being sandwiched between real data. To solve this, we will take all of the real, useful data and move it "up" in the container into the empty spaces. If, through the process of reclaiming and reformatting, a client's containers should be reduced enough then we will also merge containers, making data interactions easier and reducing costs.

However, all of this is will only be properly valuable to IBM's clients if displayed in a clean interface that reports useful metrics in an easily accessible manner. To that end, we will be creating a frontend display consisting of charts that show data storage and monetary savings, as well as useful information like (de)fragmentation percentage and network throughput. To give clients more useful data, the charts will be time-scalable, meaning the client will be able to select a range of dates over which to see reported metrics.

In developing the software to accomplish this, there are a several requirements we must adhere to, which we briefly describe here. First, and most importantly, our software must be reliable and not make *any* errors whatsoever with clients' data. This requires that we do extensive error checking and handling, as well as record any errors in daily log files.

Second, we must be cost effective in our decisions of when to pull from and push to AWS. Typically this decision will require weighing the amount of money that could be saved by culling and reformatting a customer's data against the cost of the many reads from and writes to AWS. To accomplish this we will have a module responsible for performing the cost/benefit analysis of culling and reformatting a container based on those factors. If the cost of interacting with AWS is more than what would be saved in storage, the culling and reformatting process must not take place.

Third, we ought to keep our software as modularized as possible. That is, functionalities of the software should be contained in modules that are not bound to each other, making them more easily reusable for other projects. Since each module will be a separate "program" we must use software communication methods that work between programs, rather than within.

2 Implementation Overview

To create software that will accomplish the goals of our project and meet the aforementioned requirements, there are a number of tools, libraries, and design patterns we will use. Here we give a rundown of those items and explain how each can be used and why we need them for this project.

First and foremost, we will be using Golang (Go) as the primary language for our backend development. This is largely due to the fact that Go handles threads in a simple and clean manner, which will help when servicing hundreds, if not thousands, of a client's containers simultaneously. Similarly, Go offers excellent data writing protection when multithreading, which aids greatly in satisfying our requirement of reliability.

In keeping with the requirement for modularity of this project, our software will use HTTP requests and responses to communicate between modules. To aid in this, we will use the "net/http" Go package which offers easy creation, formatting, and sending/receiving of HTTP messages.

Working alongside Go will be a MongoDB database, which will be used to hold metadata files that describe (but do not actually hold) the layout of clients' containers. These metadata files will take the form of JSON, per IBM requirement, motivating our choice to use MongoDB which stores its objects as JSON.

Another core piece of the backend will be the AWS Go SDK (Software Development Kit), which provides a simple means to perform the requests to read from and write to AWS' S3 data storage. By using this library, we can avoid having to handcraft the HTTP requests, and can instead have them written and sent like otherwise ordinary function/method calls.

For the frontend display which reports useful metrics to the client, we will be using a combination of ReactJS and D3JS. ReactJS is a Javascript library that was created with modularization in mind, better enabling us to create the components of our frontend in a way that makes them easily reusable for other projects.

D3JS is one of many graphing libraries, but what it offers more than any other library is dynamic graphing: the ability to, based on input parameters, make a new graph on the fly. This will make our frontend responsive, and enable clients to adjust certain factors like the range of dates over which to receive data.

With these tools providing their respective uses and advantages, we can create software that will satisfy the requirements of both IBM and its clients.

3 Architectural Overview

The components discussed in the previous section will be used to create software that adheres to the Representational State Transfer (REST) model, a key part of which is that data is handled and represented in three separate layers. These layers are as follows:

- Database Layer: where the data used by the software is stored
- Service Layer: where the data used by the software is manipulated
- Presentation Layer: where the data used by the software is displayed

This separation of responsibilities surrounding the data into separate layers ensures a level of security with data manipulation and makes sure that the data that is displayed is accurate.

This configuration of modules is shown in the diagram below:

Figure 3: High-Level Architectural Diagram

The Database Layer consists of two major components: S3 cloud storage and a MongoDB database. The S3 storage is where we will retrieve the cloud storage objects which need to be reclaimed, and where we will write reclaimed container files to. The MongoDB database will be used to store JSON files which describe the data in a container and provide certain metadata and metrics about that container.

These components communicate with the backend, which will be the core of this project, consisting of several modules written in Go that perform all of the data manipulation. The modules that compose the backend will primarily be responsible for making requests to AWS, performing a cost/benefit analysis on whether a container ought to be reclaimed, reclaiming containers, and merging containers.

The backend will create useful metrics based on the containers it has serviced and feed those metrics to the frontend in the Presentation Layer. In the frontend, those metrics will be displayed in a way that is easy for a user (who could potentially have little knowledge of the workings of cloud data storage) to understand, while still providing informative data. Examples of metrics that may be displayed to the user are data storage savings (in mega/gigabytes), monetary savings, percentage of a container's data that was expired before reclamation, and network throughput.

Note the arrows between each layer in Figure 3, depicting the direction of the flow of data between the layers. In keeping with the REST model, the Presentation Layer does not directly communicate with the Database Layer. If the frontend website requires data from either MongoDB or S3, it must first request that data from the backend, which will then talk to the appropriate component in the Database Layer. That component will give the corresponding data to the backend, which will forward the data to the frontend.

While this arrangement of layers and modules may seem cumbersome and slow, it is necessary to ensure the previously mentioned security of data storage and accuracy of data manipulation, both of which are key goals of this project.

4 Module and Interface Descriptions

The components described in the previous section and the modules which comprise them are further broken down in this section of the document. Here we explain the roles and responsibilities of all of the modules in this project, separated into a section for backend and a section for frontend.

4.1 Backend

To depict the functionality and data flow of the modules that compose the backend of this project we are using SysML, since we are not making use of Object Oriented Programming, and because a hierarchical view best fits the breakdown of our backend. (For the sake of readability, we present the branches of a hierarchical architecture tree separately, rather than showing it all in a single diagram.) Below is a diagram of the most front-facing portion of the backend:

Figure 4: Backend Top Level

The HTTP Listener module is at the top of the tree because it is responsible for triggering the chain of tasks that the backend performs. This module listens for HTTP requests which contain a JSON file that describes the layout of a container for which reclamation is being requested. The Listener extracts that JSON and passes it to the Core module, which is only responsible for acting as a driver for the other modules, which in turn will perform all of the data manipulation and interfacing with the other layers of the software.

Figure 5: Cost/Benefit Modules

Above is a diagram that shows the modules that will handle the calculation of cost-related metrics and make decisions based on those metrics. Considering the costs of reading from and writing to AWS shown in Figure 2 earlier in this document, it is possible that the monetary savings from reclaiming a container will be outweighed by the cost of the numerous HTTP requests necessary to pull all of a container's data from cloud storage and write updated data back to cloud storage. The Change Metrics module will calculate those costs and determine whether it is worthwhile to reclaim a container, passing that decision back to the Core module, which will respond accordingly.

Similarly, the Statistics Analysis module will gather statistics on a container, such as its fragmentation percentage (percentage of the data stored that is expired) and how much data storage can be, or has already been, saved through reclamation. These statistics will be given to the Core module which will pass them to the frontend to be displayed to the user.

Figure 6: Reclamation Modules

Figure 6 depicts the breakdown of the modules pertaining to the reclamation of cloud data storage. Once a container has been pulled from cloud storage, the Core module will pass its contents to the Reclaiming Space module. That module will then, based on the

corresponding container layout file, remove all chunks of expired data and move all of the non-expired data into the vacant space. The resulting container and its new layout will then be passed back to the Core, which will send the container to AWS and the layout to the database.

The Reclamation Statistics module, much like the Statistics Analysis module previously discussed, will gather metadata about the containers being reclaimed and send them to the Core, for the purpose of displaying to the user.

Figure 7: “Talking” Modules

The final piece of the backend, shown in Figure 7, is the “Talking” Modules, which handle communicating with the other layers of the software. This consists of a module to talk to each of: AWS, MongoDB, and the frontend, wherein each module will handle making requests to (and receiving responses from, where appropriate) another component of the software.

The AWS Talker module will take one of two things: 1) a container name or ID, or 2) a container file (note that this is the actual container file, not a layout file). In the former case, this module will request that container from S3 and, upon receiving the response, send the container file to the Core. In the latter case, this module will request to write the new container contents to the corresponding S3 object. If a response is received for this request, it will be interpreted by this module and, if there is any error, it will be relayed to the Core module.

The Database Talker module is responsible for storing items to and retrieving items from the MongoDB database. It will take as input one of two things: 1) a container name, or 2) a JSON container layout file. In the former case, this module will retrieve the container layout file from the database that corresponds to the named container and send that file to the Core module. In the latter case, this module will simply store the file in the database and send an appropriate response to the Core module.

The Front End Talker module handles sending any necessary data to the frontend web page via HTTP messages. These messages can contain various things, but will predominantly hold metadata about reclaimed containers and the amount of data/money saved by reclaiming.

Note that the bottom row of nodes in Figure 7 are not actually a part of the backend, but are shown to illustrate the connection between the backend and the other layers of the software.

4.2 Frontend

This project is primarily focused on the backend, and the reclamation process and analytics that surround it. With that in mind, the frontend is relatively small and lightweight. With this in mind, we have broken the frontend into a set of small modules represented in the UML diagram below.

Figure 8: Frontend UML

Each of the modules depicted in Figure 8 will consist of Javascript/JSX, HTML, and CSS (the CSS will, of course, be the same across all modules).

The DataService module will serve as a connection point between the backend and frontend, similar to the HTTP Listener module seen in the backend. A key piece of this module is the ability to send and receive HTTPS messages, which will be necessary for the frontend to communicate with the backend and vice versa. In keeping with the practices of RESTful software, we will implement a custom library of HTTPS response codes, which will help us log the results of any input received from the backend.

The Dashboard module, similar to the Core module in the backend, will act as the driver for the other frontend modules, dispatching calls to those modules and supplying them with the data necessary to fill out displays for the user.

The Aggregate Dashboard module will contain all of the data used by the frontend regarding aggregate container metrics. Most of the aggregate data will be displayed as simple text, and thus doesn't require any sort of graphic display. This component will simply handle updating the aggregate text values via the data source.

The Activity Dashboard module will contain all objects corresponding to the data-over-time variables. This module will contain a dropdown list that will allow cycling between line graphs of different data over time, such as network throughput, monetary savings, and other metrics. There will also be two selectors that will allow the ability to choose a beginning date and ending date over which to display data. Lastly, the Activity Dashboard will contain a reference to a line graph that will be populated with the data passed to this component.

The Activity module will display data corresponding to each individual data reclamation. It will retrieve data from the Activity Dashboard based on which time range has been selected in the graph (the user will have the ability to select a day to focus on). This module will retrieve this data and display averages via a status bar at the top of the module. The center of the module will display data over time pertaining to this individual reclamation such as network throughput over time and data throughput over time.

The Line Chart module will act as a reusable chart component that will be rendered in D3. This module will be fed information regarding its statistical data (axis values, data points) as well as graphical configuration (size, labels, colors). This module is only responsible for displaying data, and performs no data manipulation. This module will be invoked by the Activity Dashboard and Activity Module modules.

Not depicted in the Figure 8 is nimbus.CSS, a file which be solely responsible for managing all of the stylistic elements of the frontend. These elements will not be altered anywhere else, and this file will will be used by all frontend modules responsible for displaying anything to the user.

5 Implementation Plan

Now that we've made blueprints for each of the modules that will go into this project, and how those modules are reliant on each other, we can create a rough schedule for how and when we will develop each of those modules.

While this project is primarily focused on the backend, we find it to be a better use of our resources to work on backend and frontend development simultaneously, since a member of our team (Itreau) is already proficient in frontend work. With this in mind we have split our development into concurrent backend and frontend phases, shown in the Gantt chart below:

Figure 9: Implementation Plan Gantt Chart

Figure 9 shows the timetable for concurrent development of backend (blue) and frontend (purple). The initial tasks are the creation of a Git repository for managing our code base, and a website for tracking the progress of our project. After these are done, the first proper development milestone is a module (currently not planned to be connected to the project, but still to be used) that will handle converting CSV container layout files to JSON, since some IBM systems use CSV instead of JSON. Tackling this early on will ensure that our software can handle either input type.

Following this, development of the actual project will begin, with frontend development focusing on using D3JS and ReactJS in tandem, and backend development focusing on the AWS Talker module and ensuring that we can safely and reliably send data to and receive data from AWS. This phase will take an estimated two weeks.

As that phase is ending, backend development will move to implementing multithreading necessary for handling the reclamation of many containers simultaneously. This leads into developing the Database Talker module, as well as the ability to store JSON to and retrieve it from the database. At the same time, we will be developing the functionality responsible for reclaiming and merging containers, and gathering statistical data on reclamations.

While backend development focuses on AWS and MongoDB interaction, the frontend will begin a three week phase that constitutes most of the frontend work: taking data from messages received from the backend, and using it to populate charts to display to the user. This will can be tested with hardcoded dummy data initially, but will eventually require joining the frontend and backend together to use real data generated by the backend.

Towards the end of frontend development, backend will switch to working on error checking, handling, and logging. This will include creating a library of HTTP response codes, creating a module for logging events daily, and throwing/handling exceptions for any variety of input or error.

Finally, with all of the development completed, the project will enter a testing phase that will take an estimated three weeks. In this time, we intend to rigorously test the built software against many varieties of bad input and under different conditions to ensure that, above all else, the project’s goal of reliability is upheld and no improper data operations occur. Once this is done, the project will be considered finished.

6 Conclusion

Cloud technology is a growing industry, with cloud data storage being a vital part of that industry. Our client, Daniel Boros at IBM, works on tools and services which help customers more easily--and more cheaply--manage their cloud stored data, which makes his work considerably valuable. While IBM offers many ways to reduce a customer's cloud storage costs, we are prepared to offer IBM another way in which this can be accomplished: the reclamation of storage space used by expired (that is, unused and unnecessary) data.

Our goal with this project is to create software that can examine the contents of cloud data storage containers and, based on the percentage of the held data that is expired, make a decision of whether or not to reclaim the storage in that container. Through balancing the cost metrics of storing data and making requests to AWS, we hope to save both Spectrum Protect and its clients a considerable deal of money and time.

We will need a variety of tools, primarily Golang and Javascript libraries, to create software that can accomplish this data reclamation. Considering those tools, we now have an architectural breakdown and formalized plan for exactly how we intend to develop software capable of cloud data storage reclamation and metrics analysis. This plan centers on the REST model which stipulates that software should be developed in a layered manner, in which the modules responsible for representing data should not manipulate it, and vice versa. Finally, with an architectural blueprint in place, and a sense for how the modules of this project will connect, we have a development schedule that will see the project finished in a few months' time.

With all of this done, we at Nimbus Technology are more than prepared to begin development of this project, and we are eager to show Mr. Boros and IBM the results of our work.