Regionalization of an Asset Storage and Distribution Service

By: Cameron Dunn & Nate Morrison

California Polytechnic State University

Computer Engineering

Senior Project Report

Advisor: Zachary Peterson

May 28th, 2014

1. Overview

1.1. Background and Goal

The service we work on, *Asset Services*, is comprised of multiple independent services which work together to provide an easy interface for internal clients to allow end users to upload their own content, and then distribute that content to both internal and external clients, with a firewall between the service and most external clients. Although *Asset Services* handles millions of requests a day from all over the globe, all requests and data were sent through a single data center. The objective of this project was to create new endpoints for end users to upload and download assets to/from. By adding additional endpoints in new regions of the globe we hoped to increase download and upload speeds. Since *Asset Services* is already live and handling significant traffic, an extra challenge was rolling out our changes seamlessly and without causing any mandatory changes or downtime for internal or external customers.

1.2. Existing Service Architecture

Asset Services is primarily made up of three independent services, although there is some shared configuration and code between them. The *URL Vending Service* is an authenticated service only exposed to internal customers, which vends URLs to the *Streaming Service*. An internal service, such as a website which wants a customer to upload a PDF, asks the *URL Vending Service* to create a new asset and gets a URL back. The URL it receives is for the *Streaming Service* and allows a customer to upload a file or other content through a normal HTTP request. In this case the *Streaming Service* will receive the file and store it in an abstracted backend storage service. The *Streaming Service* also handles any verifications/derivations of files and sends messages to the *Daemon,* which performs various asynchronously tasks, such as virus scanning.

Downloading an asset, or deleting one, happens in the same manner; an internal service requests a URl from the *URL Vending Service*, and then the end user makes an appropriate HTTP request to the given URL.

1.3. Architectural Changes

In order to facilitate faster uploads and downloads to end users, it was decided that it was far more important to distribute the *Streaming Service* endpoints than the *URL Vending Service*. The requests to the *URL Vending Service* come from internal servers and are significantly smaller so additional latency doesn't have a noticeable end user impact. Also because the *URL Vending Service* uses a database to track asset information, the DB would need to be replicated to multiple regions (significant increase in complexity and introduces trade offs between performance and consistency) or it would have to continue to make calls to a single DB which would bottleneck any possible performance enhancements.

The *Streaming Service* has been modified to exist in two additional geographically distinct endpoints, with code structure allowing additional endpoints with very minor changes. The *Daemon* will continue to exist in just the original endpoint, although moderate benefits would be seen if it was located in additional regions. The decision to not add *Daemon* endpoints was due to its jobs already being asynchronous, limiting the benefit of latency reduction.

2. Terminology and Concepts

2.1. Original

Asset Type - A construct for defining rules and attributes for a group of similar assets. Includes a name, permissions, and verification & derivation rules.

Asset - A single file or piece of data identified by its *Asset Type*, ID, & index (essentially a secondary ID). An asset will have one or more versions.

Asset Version - A single instance of an asset, which is its own file/object and can be accessed separately from other versions of the same asset.

2.2. Introduced

Mastered In - The location that a particular *asset version* was uploaded to and will never be removed from (aside from deleting the asset).

3. Database Changes

3.1. Primary Database

The primary database contains various tables for tracking assets, versions, the status of long running operations, etc. It is a Amazon DynamoDB NoSQL database offering fantastic performance and scalability, but not relational queries.

3.1.1. Version Table

The version table contains information about every single *Asset Version*, that is every single unique object uploaded to *Asset Services*. Since we needed to start tracking which locations a version was present in, we needed to modify the schema to store additional information. Unfortunately the table was too large to update the existing rows. Instead we designed a schema to be used for new rows which contains additional information, and a set of rules for determining information about old versions.

To track a versions location(s) we added a set of new columns:

Mastered In - Contains the name of the region an *Asset Version* was originally uploaded to.

status<region> - A column for each region containing the *Asset Version's* status in that region. The possible statuses are:

null/unpopulated - The *Asset Version* is not present in this Region unless it was created prior to the update schema (decision described below)

'removed' - The *Asset Version* was once present in this region but has since been removed, either because the *Asset* was soft deleted or the caching algorithm removed it

'pending' - The *Asset Version* will be copied to this region but the copy operation has not completed yet.

other - All other values are used to describe a Version ID, which is used for retrieving the *Asset Version* from the backend storage system. In the original schema the Version ID was stored in its own column, but is no longer populated under the new schema.

When information about an *Asset Version* is needed the row is read from the Version table in Dynamo and converted into an object describing the version. In order to make the schema changes compatible with all existing code, we rewrote part of the accessor to interpret the different schemas into the same object. The basic logic is:

1) Determine the *Mastered In* region by getting the value of the *Mastered In* column if it's present. If the column is not populated, we know that the version was created before our schema changes, which happened before any assets were able to exist in other regions, therefore it must be *Mastered In* the original endpoint region.

2) Read all of the *status<region>* columns and set a map of regions and statuses which describe where the *Asset Version* is present in according to the new columns.

3) Check if the *Mastered In* region is present in the map of regions and statuses. If it is not, we add an entry to that map for the *Mastered In* region and populate it with the value from the Version ID column which we know will be populated because if the *Mastered In* region was not already in the map then this row must have been created using the old schema.

3.1.2. Request Count Table

The request count table is a new table populated by the *URL Vending Service* through a new job in the *Daemon*. Every time an *Asset Version* download URL is requested through the *URL Vending Service* for a specific region, it sends a message to the *Daemon* triggering an update to the table and possibly replicating the *Asset Version* (see section 6.2). The request count table contains the following columns:

Total - The total number of regional download URLs that have been requested across all regions. Note that this does not include requests where the client did not request a specific region.

<region> - The number of download URLs requested for that region.

lastRequested - The last time that the *Asset Version* download URL was requested

The *Total* and <region> columns are updated using a built in ADD operation in Dynamo to prevent race conditions.

3.2. Configuration Database

Asset Type data is stored in a separate configuration database, but required modification to add new concepts needed for regionalization.

3.2.1. Additional Fields

Default Region - The region new *Asset Versions* will be *Mastered In* if no other region is explicitly requested during creation.

Automatic Propagation List - A list of regions that an *asset version* will be automatically copied to upon upload.

Smart Replication Enabled - Simply enables or disables the behavior of replicating an *Asset Version* based on download count.

3.2.2. Schema

The database used for storing *Asset Type* configuration data unfortunately does not allow schema modification so in order to add the new fields we would have to create a new table and copy all existing information. Since we realized that additional changes will continue to be made to the *Asset Type* configuration table, we decided to move to a more flexible schema that will let us just do a single time and then be ready for all future changes.

The original table had identifier columns and then an additional column for each possible *Asset Type* option. Our new schema contains just the identifier columns and then a single

configuration column containing a JSON-like BLOB which represents a map of all of the configuration data.

4. URL Vending Service

The *URL Vending Service* is responsible for giving an appropriate URL to the regionalized *Streaming Service*. In order to direct a request to the correct *Streaming Service* endpoint, it uses information from the client, *Asset Type* configuration, and knowledge of where an *Asset Version* is stored (for download requests).

4.1. Client Changes

The client HTTP based API was modified to accept a region parameter which tells the *URL Vending Service* the clients preferred region. The parameter is used slightly differently based on the type of request.

4.2. Region Proximity

When a client requests to download an *Asset Version* from a particular region, it may not actually be available in that region. Rather than denying the request, we want to simply direct the request to the closest region that the *Asset Version* is present in. To facilitate this behavior we defined a mapping of each region to a corresponding ordered list of the nearest regions.

4.3. Behavior Based on Request Type

4.3.1. Download Requests

If a download request is received without the region parameter from the client set, it simply uses the *Mastered In* region for the requested *Asset Version* when selecting the *Streaming Service* endpoint. If a request has the region parameter then the *URL Vending Service* must evaluate the requested region against data from the version table using the region proximity ability described above; by doing so it can always choose an appropriate *Streaming Service* endpoint which has the requested *Asset Version* and is closest to the requested region.

4.3.2. Other Requests

Upload and delete requests do not need to have knowledge of where an *Asset Version* is located. Upload requests can't have location knowledge because the *Asset Version* does not exist anywhere yet. Delete requests don't need location knowledge because the actual deleting of data from the backend storage system happens asynchronously and the *Streaming Service* just needs to kick off that process.

For these other types of requests the requested region from the client is always used. If it is not passed in the default region for the *Asset Type* is used.

5. Streaming Service

Interestingly, the *Streaming Service,* despite being the system that was actually duplicated in additional regions, required the smallest and simplest changes.

5.1. Database Use

The *Streaming Service* had to be modified to use the *status<region>* column from the Version Table in order to find the correct location and version of an *Asset Version* in the backend storage. The *Locator Service* guarantees that an *Asset Version* will be present in the chosen *Streaming Service* region, so there does not need to be any additional redirect logic outside of that modification.

5.2. Starting Replication

After an *Asset Version* is uploaded through the *Streaming Service,* it must send messages to the *Daemon* telling it to replicate the version based on the *Automatic Propagation List*. These messages are sent using SQS messages to the *Replicate Job*. The messages identify the *Asset Version* and a separate message is sent for each region the version must be copied into. Before sending a message to the *Replicate Job* the appropriate *status<region>* column is set to pending.

5.3. Deletion

Originally the *Streaming Service*, upon receiving a delete request for a particular *Asset Version* would immediately mark the version as deleted and then delete the actual object from the backend storage solution. Since *Asset Versions* can now be in multiple regions across the globe, continuing to do that would be slow due to the additional requests and potential for some of those requests to have to be made across the world. Instead the *Streaming Service* was modified to send a message to the *Deletion Job* in the *Daemon* which would handle the actual deletion of objects.

6. Daemon

6.1. Replicate Job

The *Replicate Job* was a new job added which is responsible for copying an *Asset Version* from one region to another. To do so it uses a copy call from the backend system, if necessary it does the copy in multiple parts. Once the copy is completed successfully, the *status<region>* is set to the new version ID.

6.2. Request Count Job

The *Request Count Job* receives messages from the *URL Vending Service* after every request for an *Asset Version* in a region. The job does a DynamoDB add operation to the *Total* and appropriate <*region*> columns, and sets the *lastRequested* field to the actual time the request for the *Asset Version* was received.

When updating the two counter fields, DynamoDB returns the newly updated values, which the job uses to determine if an asset needs to be replicated. If the number of downloads in a region modulus a small constant is zero then we evaluate further whether or not to replicate the *Asset Version* based on the percentage of total downloads taking place in that region, the size, the last time it was downloaded at, and whether or not this *Asset Version* is the latest version of the asset. If the *Asset Version* should be replicated, it is done using the same methods described as in the *Replicate Job*.

6.3. Cross Region Replication Estimation Report Job

In order to fine tune the logic behind deciding if an *Asset Version* should be replicated, an additional reporting job was added. It allows a developer to manually send it a message containing a set of constants and options, which it uses to produce a report describing how many *Asset Versions* and bytes would be copied to each region if those attributes were to be used in the *Request Count Job*.

6.4. Deletion Job

The *Deletion Job* is simply responsible for deleting an *Asset Version* from the backend storage and then marking it as *Removed* in the *Version Table*.

7. Evaluation & Future Works

The regionalization of *Content Services* is largely complete, and we believe we met all of our goals. The largest missing piece is just actual use of the new features. We haven't had time to educate current customers about the new features and give them time to take advantage of them. Since no users are making requests asking for a specific region, we're not propagating any assets to additional regions. We have tested the systems using artificial workloads to verify that they are working as expected. Future work will need to take place in order to refine the automatic propagation algorithm once we have significant data to model how much data will be copied to which regions, and what the costs of that copying is.

8. Conclusion

Working on such a large-scale system that is already operational and cannot tolerate any downtime provided numerous interesting challenges. Without the constraints of scale and reliability, this project may have been somewhat trivial. With these constraints in place we had to constantly search for workarounds and be extremely careful with each choice we made. The result is a system that will provide vastly improved performance for global customers while still meeting all of their existing needs.

9. Special Thanks

This project received much patience, guidance, and resources, and would not have been possible without:

Dr. Zachary Peterson
Cal Poly Computer Science Department

Kevin Watson

Josh Tang