Name,
class,
semester info

Cloud Render Farm

Project Goals

Describe your project goals in 1 - 4 paragraphs

What do we need it to do?

Instead of jumping right to class lists and methods. I'm going to break down the render farm project by problems to be solved. This will help organize my thinking and give me concrete goals to be met.

Present a web front end

The web front end will act as the primary interface for the application/farm. It will need to run on its own instance, with a web-server to deliver it. Most likely candidates currently are Apache with a Django backend, so I can stick to python.

Tasks for the front end

1. Collect job specifications. This will consist of job names, files to be included, number of instances to spin up per job, frame ranges to be rendered. Likely candidate database SQLite, due to integration with python.
2. File transfer. The front end will need to upload client files to S3 and organize them appropriately (using delimiters, etc.) Assuming files are large, it will need to deal with S3 multipart upload. Possible paths: client to web-server to s3 vs. client directly to s3 via HTML POST.
3. Job Management. Start, terminate and pause running jobs, on a job by job basis. Track job progress.

Running a job

The back end task of the render farm will consist of instances running a custom AMI loaded with the open source 3D application Blender and support python scripts which will launch and manage Blender subprocesses.

Environment Preparation

The first thing that will need to be done before any instances are spun up is collection of needed resources for the render. Blender doesn't understand AWS S3 object storage, so the files it uses will need to be staged on an EBS volume. Likely handled by the main application, the following tasks will be required:

1. Set up the workspace. An EBS volume will need to be spun up and attached to the

main app's instance. Alternatively a "temp" volume could start with the instance already attached. This might simplify setup.
2. Staging. The workspace would then need to be populated by the main application with all the production files required for the render.
3. Copy and Publish. Once this staging is complete, the EBS volume will need to be snapshotted so that copies of it can be created and attached to the rendering instances. Reduced redundancy volumes will be sufficient for this case as that the data is derived, temporary and already exists on S3.

Instance Startup

Jobs will have "number of instances" setup as part of their job settings. User will be able to specify On-Demand or Spot Instances (with a bid price). The frame ranges to be rendered will then be divided up. Instances will get started from a premade Blender AMI that I will create. A workspace copy will be attached, to give the render node access to the Blender project files.

Rendering

Renders will proceed in a node on a frame by frame basis. As a frame is completed, it will be uploaded to a receiving bucket on S3 by a Cloudwatch trigger (or custom python program, not decided yet). The main application will poll this receiving bucket to track job progress. When received frame match spec'd frames the job will be marked complete. When all frames for an instance complete, it will be terminated.

Download

At any time, the main web app should allow download of completed work from the received bucket. The download process should look something like this:

1. Client initiates a download request on a job.
2. Main app copies all frames from s3 to its local EBS store.
3. Main app gzips all frames.
4. Main app presents a link to the client to download the gzip'd file.

Classes Sketch

From the problem defined above, I can extrapolate the need for several different objects or classes.

Manager

Main class for the application.

1. Maintains job list; start, stops, clears completed jobs.
2. Checks for finished frames.

UI

Supplies HTML interface for client.

1. Django templates for HTML pages.
2. Direct upload of client files to S3.
3. Input of Job parameters.

Job

Holds necessaries for job:

1. Field for File to be rendered
2. Frames to be rendered.
3. Type of instances to use, On-Demand / Spot.
4. Number of instances to use.
5. Field for an Instance Pool if one has been assigned.

File

Holds the details of a Blender file uploaded for work.

1. S3 location of the file.
2. Original name of the file.
3. Job to which the file belongs (possibly not needed)

Instance-Pool

Holds a list of instances, assigned to a Job.

1. Methods for managing instances in the pool; start, stop, terminate.
2. Methods for assigning Tasks to an instance.

Instance

An in program object representing an EC2 instance.

1. Instance ID.
2. User-Data access.
3. Methods for start, stop and terminate.

Workspace

In program model for the EBS workspaces.

1. Field for snapshot used to create.
2. Field for type of EBS volume to use. (Reduced Redundancy)

Task

A model for tasks handed off by the Manager, for a Job, to an Instance.

1. Fields for frame ranges.
2. Methods to run Blender
3. Methods to capture the logs
4. Methods to capture frames rendered to S3

Frame

A model for each frame of a given job. Used to track completion and or the need to re-queue in the case of job interrupt on a spot instance.

Reflections on Scope

Not wanting to bite off more than I can chew, I can see a few ways in which I might change this initial design to simplify the farm.

1. Eliminate Job Specification in favor of a simple in/out queue. Users would upload their Blender file to the S3 and the main application would poll S3 (maybe using Cloud Watch) to start jobs.
2. AutoScaling Instance Pool. Instead of launching and managing instances on a per job basis, establish a single instance pool with some draconian scaling policies. When work was found in the queue, it could scale up to do the work, then scale down when it was complete. However, Autoscaling charges for full hour for each instance spun up, so Spot Instances would be a requirement to make this feasible for experimentation.

References