Jesi Merrick
Ron Pyka
CS553-01

Design Document

Program Design

For this assignment, we used previous python client-server code from CS550 (which was built on code from CS451). This gave us a simple base that we could then extend upon. The client code obviously stayed as client code, and the server code became what is now our scheduler program. This gave us the initial work of connecting the client and scheduler using sockets, and was customized to handle the data required for the assignment.

Our local workers were built into the scheduler program as processes, enabling them to run independently of each other and also independently of the server and scheduling components. Both the workers and schedulers have access to a work queue and a result queue that use the queue provided by the multiprocessing library, which is safe for multiple processes to use. The scheduler adds work to the work queue as it comes in, and the workers are constantly watching the queue to get work. The workers will then pull the work from the queue (removing it so no other workers do it) and perform the task, then place their results into the result queue. The result queue is being watched by another process, which pulls the results and sends them back to the client.

To implement the remote workers, the python package boto for AWS was used. The majority of the design for the remote workers follows that as described in the assignment description. Two SQS are used, one for requests and one for the responses. The scheduler adds requests from the client to the first queue. The remote workers then detect these. A remote worker will grab a message off of the request queue, add it a messages table in DynamoDB, process it, and then add the return message (success or failure) to the responses queue. The scheduler can then detect this, and pass on the information to the client.

The dynamic provisioning aspect is fairly simple. New workers are launched by the scheduler as required, using a private AMI. The AMI is a base Ubuntu AMI, with boto installed, the necessary credential files, and the worker.py file. Additionally, cron job is used so on reboot, the worker.py file is launched. This way, the scheduler starts a worker, it boots up, and immediately launches the necessary program on its own. On the scheduler side, no workers will be launched if the requests queue is empty. If there is one request, the one worker will be launched. After that, if there are less workers than requests, the scheduler will launch additional workers as necessary (up to the number requests or a max number of workers). The workers terminate themselves after a given period of idle time.

Design Tradeoffs

We used Python as our language of choice to make writing the code easier and utilize existing code, but were restricted by some of the libraries. We were able to get very good local queues and reliable workers from python, but the networking library (asyncore) we used to make servers that would accept incoming requests and call easily gave us unexpected problems. The loop within the asyncore library library would cut off our buffer on occasion and only send half of the job, resulting in errors in the scheduler.

Improvement and Extensions

One improvement would be to implement the extra credit of message batching, another would be to resolve the issues encountered with asyncore. We also could add a menu or enable a user to upload multiple files. Another improvement would be to have the client and scheduler (and local workers) automatically close after the final results were returned.