CSE 546 — Project 1 Report

Chethan Kumar Kolar Ananda Kumar(1214234150)
Gurumurthy Raghuraman(1211150041)
Santoshkumar Amisagadda(1213177132)
Shashank Kapoor(1213181604)

1. Architecture

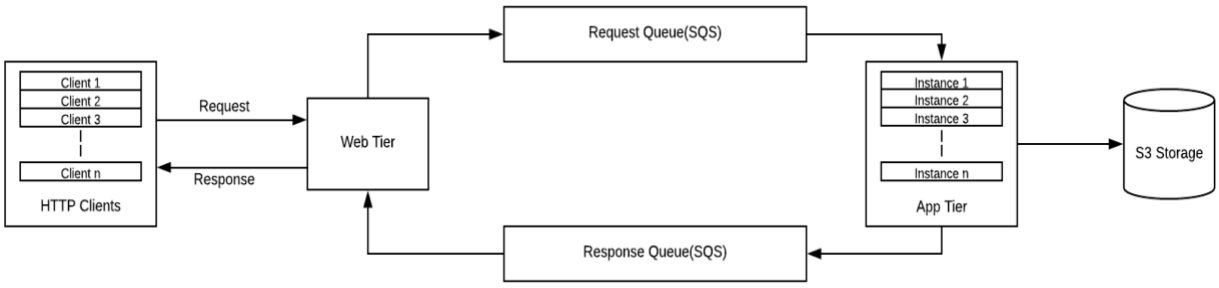

Fig 1: Architecture Diagram

General Architecture

- The architecture is straightforward and supports Auto Scaling. An Auto scaling supported architecture is used to maximize productivity and efficiency.

- In the application tier, we have multiple instances running depending on the requirement of the system, which in turn is established by the web tier. The instance in the web tier depending on the number of requests submitted by the clients starts instances of the application tier and these instances self-terminate when there are no requests.

Web Tier

- The HTTP request in this system has an URL of the image in the request. The instance in the web tier on getting the requests gives each request an unique id, adds the entry to the hashmap, extracts the image URL and inserts it into the SQS Request Queue along with the unique id. The web tier instance then waits for the response from application tier.

- The instance on the web tier then gets the message containing the result and the unique id from the SQS Response Queue. The instance then matches the unique identifier with the entries in the hashmap, generates an HTTP response with the result in it and sends it back to the client.

Application Tier

- The instances in the application tier, which were started by the instance in the web tier, gets the message from the SQS Request Queue. The URL, from the message, is then used as a parameter by the instance when running the deep learning python code. The instance on getting a result inserts it onto the SQS Response Queue along with the unique id. Also, on getting the result the instance stores it in S3 bucket as a key-value pair.

2. Auto Scaling

Auto Scaling Design:

The Auto Scaling design is such that it scales up instances based on the demand. In our auto-scaling algorithm, we estimate the number of requests that will be processed before the instances become active; this helps us in increasing the instances optimally, avoiding unnecessary start and shutdown of instances. So, for this we first check two things, the number of messages in the Request queue and then the number of instances active. After getting these two parameters, we calculate the number of instances required using the formula: Math.min((request queue size(Approx messages) – (instancesActive – 1)), (MAX_INSTANCES - instances active)). We have subtracted one in the first expression of Math.max because one is web-tier which will not process any requests. To avoid unnecessary communication over the network, we have divided our workers into two parts Active Worker and Dormant Workers. We only have one Active Worker which starts automatically and caters any immediate requests and remaining are Dormant Workers. Dormant workers terminate automatically if there is no request for them. This way we avoid network calls from the server/monitor to the worker.

The pseudo code or algorithm for auto scaling out is as follows:

1. Check the size of Request Queue and the number of active instances.
2. Decrease the number obtained in step 1 by (activeInstances – 1).
3. Check the number of available instances which is (20 – non-terminated instances).
4. Now take the minimum of the required and available instances.
5. Increase the number of instances by the number obtained in step 6.

The pseudo code for Active Worker

1. Check for message in request queue.
2. Process if not zero or sleep for half second if zero
3. Repeat from 1

The pseudo code for Dormant Worker:

1. Check the number of messages in the request queue.
2. If job is there process it or if the number of jobs is zero sleep for 1 seconds and repeat step 1 only once.
3. If still no job, shutdown and terminate.

3. Coding

Web Tier Module in Node.js:

The Web tier is built in Node.js. It primarily performs the following three tasks:

- Receive http requests from client and send it to Request Queue.
- Based on the number of requests in request queue, it starts more number of dormant worker instances.
- Receive messages from Response Queue and send it back to the client.

In the next sections we discuss each of the above steps in detail.

- Receiving http requests from client and send it to Request Queue:

The server is setup in Node.js. Whenever an HTTP request is sent, the server first gets the input URL and then generates a unique UUID for it. We store the default response for this UUID in a global map. We then check whether the result is present in s3, if yes it adds it to the default response and sends it back to the user. If the result is not found in S3, it sends it to the SQS Request Queue. The algorithm can be explained as follows:

1. From the HTTP request get the input image URL.
2. Now create a UUID for each request and store it along with the default response in a map.
3. Check in S3 whether a result is present, if yes add it to default response and send response back to user. Delete the UUID from global map as response is sent.
4. If step 3 is not true, create a JSON string containing the UUID and the imageURL.
5. Send the JSON string in step 4 as a message to request queue SQS.

- Auto Scale Instances based on the demand:

In auto-scaling, we use the number of messages in SQS to scale the instances. Firstly, we keep one instance of the Active worker always on. For every 2 seconds, we check the number of messages in the SQS. Then based on that number we calculate the number of instances required for SQS by (number of messages in Request Queue -1). After we calculate the required number of instances, we check for the total available instances which are 20 – (numberOfActiveInstances) instances. We then startup the minimum (required instances, available instances). After the start of instances, we make the web tier wait 15 seconds before it again checks the queue size. Usually, the monitor runs every 0.5 second.

Pseudo Code:

1. Check the number of active instances.
2. Get the size of the queue.
3. Subtract (numberOfActiveInstances – 1) from the size of the queue.
4. Now check the number of available instances which is 20 – numberOfActiveInstances
5. Take the minimum of step 3 and step 4 and start those many instances.
6. Wait for 15 seconds before the instances get stabilized.
7. Again, start in 2 seconds (the monitor runs in 2 seconds interval).

- Send Responses back to user:

The web tier also reads messages from the response SQS queue and sends the user back responses. For this it gets the default global UUID map and then adds the results to the corresponding default response and sends it back to the user. The pseudo code of this is as follows:

1. Checks number of messages in response queue SQS.
2. If jobs are available, then receive messages in batches of 10.
3. Parse JSON to get request ID and result.
4. Check whether corresponding request ID is present in the map, if yes add effect to default response and send it back to the user.
5. If step 4 is successful, then delete the request Id from Map so that it is not processed again.
6. If two is not successful, then sleep for half a second.

Below we explain each file of the Web Tier and what its used for:

- In the folder EC2Functions, there is a file named EC2Utility.js. This file contains the start instances method which is used for spinning up more instances.
- Next, in the Globals folder, there are four files EC2Config.js, S3Config.js, SQSVars.js and credentials.js as the name suggests the files contain the configuration stuff and the AWS credentials.
- Monitor folder contains Monitor.js. It is responsible for continuously checking the number of messages in the queue and starting new instances if necessary.
- The folder RequestHandlers contains ImageURLRequest.js. This file contains the code for reading requests and then adds it to the request queue. It also reads messages from the response queue and sends the response. The Request Map is also updated in this file.
- The Routes folder contains routes.js, and it uses express JS framework to parse the request and the response.
- The folder S3Functions includes the S3Utility.js which contains all the necessary functions of S3 like retrieving a key-value pair from a bucket.
- The folder SQSTools consists of the SQSUtility.js which contains all the essential SQS Functions like sending and retrieving a response.
- The main file is server.js from where the execution starts. This starts the HTTP server and also creates child processes for faster performance.

Worker nodes (active/dormant) in java:

The entire code on the worker nodes is in java. The primary tasks of the Worker nodes are:

- Receive messages/requests from SQS Request queue.
- Process the request.
- Add the result in the S3 bucket as a key pair value. And send the result back to the web tier through the SQS Response Queue.
- Process the job if it is there and if not, then only Dormant Workers terminate after one more try to receive the message. The Active Worker continues the process.

- Receive messages/ requests from SQS Request Queue

The first task of the Worker node is to retrieve the message from the SQS request queue. The message it gets through the queue is a JSON string. So, once we have the JSON string we parse it to extract the UUID and the imageUrl. The pseudo code for this is as follows:

1. Get the JSON message from the SQS queue. If there is a message go to step 2 or else go to step 3.
2. Parse the JSON message into 2 parts the UUID and the image URL.
3. Break the while(continueloopexecutoin) loop.

- Process the request

Once we have the image URL that needs to be processed we pass that as a parameter to the deep learning python code and execute it. We split the result to obtain the prediction of the image. The pseudo code for this is as follows:

1. Append the image URL into the command to execute the python code.
2. Execute the command with the appended URL using. exec, to get the result.
3. The result might have some unwanted data with it like multiple values with similar scores. By string manipulation, we eliminate this unwanted information.
4. Create a key-value pair of the image with the result. The key is the input image name parsed from the input URL and the value is the image recognition result.

- Add the key-pair value to S3 and sending it back to web tier:

After the image URL is processed and a key-value pair is created we need to add it to the S3 bucket as well as send it back to the web tier along with its UUID. The pseudo code for the same is as follows:

1. Put the key-value pair in the S3 bucket as an object.
2. Create a JSON string with the result and the UUID.
3. Insert the JSON string in the SQS Response queue.
4. Delete the message received from the SQS Request queue.

- Terminating the instance:

We need to terminate the instances if there are no requests in the queue. This is to support the auto scaling. The pseudo code to do the this is as follows:

1. If there are no requests in the queue, as discussed before we go to the else condition.
2. In the else condition a counter is checked to check if this is the first time or not. If is the first condition do not break the while(true) loop and increase the counter. If the counter's value is one, i.e. it is not the first time, break the while(true) loop.
3. Once the loop is broken, we execute the shutdown command. But at the start of an instance we have configured the shutdown command to actually terminate the instance. So essentially the shutdown command will terminate the instance.

4. Project Status

We were able to meet the following specifications:

- The application handles multiple HTTP requests in a stream or in a batch.
- The **application increases** the number of instances in the worker when there is an increase in the number of requests.
- The system is also able to reduce the number of instance to just 2 one for each the web tier and active worker node when there are minimal requests to be handled.
- The application accurately recognizes the image from the given URL and stores it in the S3 bucket as well as sends it back to the client who made the request.
- The application not only recognizes the image accurately but also does not miss any request from the clients.

Portfolio Submission for Santoshkumar Amisagadda

I worked on the code that needed to be executed on the application tier instances. The primary tasks were to extract the messages from the SQS Request queue, process these requests and store them in S3 bucket for persistency and send them back to the web tier via the SQS Response queue.

I wrote separate utility classes to access the SQS queues and S3 bucket. The functions from these utility classes were then called to facilitate various tasks required in the main class. The main class PredictImagesAndStoreInS3.java is where the entire processing takes place. The utility classes SQSJavaUtility.java and S3JavaUtility.java have their respective functions. The SQSJavaUtility.java has functions to get and put messages in a queue given an URL. The S3JavaUtility.java has a function to add a key-value pair to the bucket.

As mentioned above the entire processing is done in the main class. Once I get the message from SQS Request queue, using the utility function, the processing starts. If there is no message in the queue, the instance is terminated. The message I get from the queue is a JSON string, so we parse it to get the UUID and the image URL. The image URL is then appended to an already existing command, as the parameter. The command, a command to start the python deep learning code, is executed on the instance's terminal using '.exec'.

I then read the output of the python code from the terminal. The result might contain unwanted data like the score, so I apply string manipulation techniques to extract only the required value. Once we get this value, we create a key-value pair. The key is the name of the image while the value is the result. Using the utility functions from S3JavaUtility.java I insert it into the S3 bucket.

Later I create a JSON string with the UUID we initially extracted and the result I got from applying string manipulation on the output of the python code. This JSON string then with the help from utility functions from SQSJavaUtility.java is inserted into the SQS Response Queue. All this processing is done continuously in a while(true) loop. This loop is broken when there are no messages in the SQS Request queue. Once the while(true) loop is broken I have executed a command to shutdown the instance. But while creating the instance we mentioned that shut down be substituted with termination. So essentially the shutdown command will terminate the instance.

Conclusion and result:
I was able to handle all the requests extracted from the SQS queue and store the result in S3 bucket and send them back to the web tier. Also, I was able to terminate the instances at the right time.

Gurumurthy Raghuraman-Portfolio Report

In this project I have worked on the auto-scaling module and the configuration of AWS instances. This module consists of increasing the number of instances according to the number of requests. To perform this, I first implemented the starting of AWS instances using the AWS Javascript SDK. Using this a number of instances were started and each of them were configured with shutdown behaviour as terminate. To check whether we are correctly scaling the instances we use two parameters: the number of messages in queue and the number instances active. To calculate the approximate number of messages in the queue I used theruounted the number of non-terminated instances using Ec2 API. First, I found out the number of active instances. For counting the instances we check for the number of instances in non-terminated state. Once we get this the number of instances required is given by the formula: (number of messages in request queue - (numberOfActiveInstances - 1)). I subtracted one here because we do not consider the web tier instance which does not process any image recognition requests. Next, I calculated the number of instances available using the formula (20 - number of active Instances). I then took the minimum of the number of instances needed and the number of available instances so that the application only starts at most 20 instances and never tries to start more than 20 instances. This piece of code has to work repeatedly and also after starting every instance we should allow those instances to be stabilized before we recalculate the number of instances. So, after starting a required number of instances we wait 15 seconds and then we repeatedly check the number of messages in queue every 2 seconds. Checking the number of messages every 2 seconds helps us to start instances quickly once the requests come in. We also kept a repeat time of 2 seconds as the web server has to take in requests, process responses and also start instances when necessary. This parameter of 2 seconds helps us to keep the load on the web server balanced.

I have also worked on the configuration and setting up of the Web Tier and the application tier. For configuring the web tier, I first started an ubuntu 16.04 ec2 instance and transferred the files onto the instance. For file transfer we directly downloaded the files from our git repository. Next, I installed node js in this instance and then we moved into our project directory and did npm install. The package.json already has all the required dependencies listed. Next to run the server continuously I installed nohup forever and then put the nohup command in the /etc/rc.local file. This starts the server as a background process and it runs continuously. Also, so that we don't have to type in the port number as part of the request I configured the default port as 80.

For setting up the app tier which was built completely in java I had to first import the dependencies. For importing the dependencies, I used maven and created a pom.xml. We had two dependencies in the pom.xml file: one for importing aws-java-sdk and the other for importing simple json for converting json strings to object and vice versa. For building a maven project we install maven first and then do a maven clean install to create a jar file. To make this jar run we go to /etc/rc.local and then change the header from bin/sh to bin/bash and we first add the source command to activate tensorflow. After this I ran the nohup command to run the jar file continuously as a background process. Diving deep into these modules and implementing them sums up my work as part of this project.

Portfolio Submission (Shashank Kapoor)

Building the right framework for our application was important; this was going to the face of our app. So how I proceeded with the structure has a Requirement and Solution logic.

Requirements and Solutions
1) Framework to choose

Requirement:
The First thing I noticed was that our app was decoupled among different workers and a job had to be sent to an SQS queue and collected from another SQS queue, and also, we needed a mechanism such that once a task is completed, we need to send back the response to the client. That requirement there, for sure we needed something that could store our request map. For such thing market prefers to use In-Memory Databases like Memcache or Redis but that was out of the scope of this project. Instead of that, I decided to go with a global request map in the server itself which could also act as an In-memory request map

Solution:
First, I tried with PHP, a language used by millions of web developers. It has all the right things a web backend developer wants, but it has a Shared Nothing Architecture which was a dominant blocker of our project. If ideas are Shared Nothing, I cannot create a global map. Rather than that, I choose the coolest kid in the language world, Node.js. It's lightweight, and we can build global variables in it.

2) Lightweight and minimal development

Requirement:
We were informed earlier that we should be able to handle a lot of concurrent requests and it was also mentioned that web tier should control the logic for the monitor. So lightweight was required and something else which could incubate monitor logic as well. Also, respecting the time fast development was also needed.

Solution:
Node Js is a very lightweight language; things can go fast if Node Js is used a backend for REST which we had. To make things minimal, I used Express.Js framework which made the development easy. Also to incubate read SQS to send the response and Monitor logic I used a fantastic framework called Async; this can run behind your on your servers and has a lot of functions for asynchronous programming which we again had. Some of the functionalities I used from async was Forever, Waterfall, forEach (parallel execution), forEachOfSeries(Sequential Execution). The added advantage I got was my server was all asynchronous, so things moved whenever they were available even though my system was message based (somewhat similar to what Akka does).

Conclusion and Results
That there I implemented the entire backend web server which easily incubated monitor logic and receiving the message from the client and sending messages back to the client. The benchmarks of my front are useful, and none of the requests is ever missed; this gives high availability to the system.