CS5412 - Assignment 2: Using Amazon's EC2 and Elastic Beanstalk

Name: Prashanth Basappa NetID: pb476

Python Flask

Server:

I decided to code the server side in Python Flask. Flask is a small and powerful web framework for Python. I went with Flask because of its easy form generation and simple template rendering and the fact that it supports REST. Its is based on the WSGI toolkit and Jinja2 template engine.

For password protection, I included a secure hash algorithm - SHA224. In Python, there is one constructor method named for each type of *hash*. All return a hash object with the same simple interface. For example: we use **sha1()** to create a SHA1 hash object. At any point we can ask it for the *digest* of the concatenation of the strings fed to it so far using the **digest()** or **hexdigest()** methods.

To maintain a list of users and to make sure they are unique, the code hashes a password with a salt. It simply reads and writes usernames, hashed passwords, and salts to a json file. The 'render_template' function renders the given template as the response in the URL.

The goal is to handle user signups and logins in a safe and simple way.

The coding logic can be found in the application.py for the server and the templates are found in the templates folder.

Client:

I have written the client program in Python. Here it accepts the starting number and ending number as system arguments. The latency calculation is done for each get request which inherently hits the given URL(in my case it is my elastic beanstalk URL). After all the prime number calculation is done for all the odd numbers between the starting and ending number, the average, minimum, maximum and median latency are calculated to written on a test.txt file. The client program client.py can be found in the src folder.

Basic UI:

All of them are rendered as HTML pages by the Flask server. The UI coding can be found in the templates folder.

Design Decisions:

- It handles the common tasks of logging in, logging out, registering new users with netid and remembering your users' sessions over extended periods of time.
- The sessions are handled by login with the user name and session pops for every logout.
- To maintain a list of users and to make sure they are unique, the code hashes a password with a salt.

Potential Enhancements:

- Impose a particular database(say DynamoDB) for a user login and session management.
- Cache in the results of user logins and session.
- Using Miller Rabin primality testing algorithm to increase the speed of the execution.
- Building a more secure platform by creating a blacklist on AWS.

Experiment Results:

1. a) **From my personal computer: (all the latencies are measured in seconds)**

ken kpb3 passken
Number of queries fired: 25000
Latency Min: 0.290469884872
Latency Max: 6.31281709671
Latency Mean: 0.315687337017
Latency Median 0.302602052689

1. **b) A computer node in a different region in Amazon EC2 than the one your server runs.**

EC2 server region : Oregon || Client node region: N. Virginia

Number of queries fired: 25000
Latency Min: 0.185006141663
Latency Max: 2.24628686905
Latency Mean: 0.221360116148
Latency Median 0.219341516495

1. **c) A computer node in a different availability zone (but same region) in Amazon EC2 than the one your server runs.**

EC2 Server Region: Oregon

EC2 server availability zone : US-WEST 2a || Client node availability zone: US-WEST 2b

Number of queries fired: 25000
Latency Min: 0.00897598266602
Latency Max: 2.823127985
Latency Mean: 0.0128533650017
Latency Median 0.0119044780731

1. **d) A computer node in the same availability zone in Amazon EC2 than the one your server runs.**

EC2 Server Region: Oregon

EC2 server availability zone : US-WEST 2a || Client node availability zone: US-WEST 2a

Number of queries fired: 25000
Latency Min: 0.00738716125488
Latency Max: 2.88039278984
Latency Mean: 0.0100203877449
Latency Median 0.0080509185791

Explanation of observed results:

If we compare the **mean latency** values of each case considered, we can observe that when the client ran from a EC2 instance from the same availability zone(us-west-2a) and the same region(Oregon) the mean latency value was the lowest which means the interval between the time A (when the client fires the request) and the time B (when the client receives a response for this request) was the lowest, hence the responses were very fast. The elastic beanstalk did not scale because it was not configured to scale. The latency are the fastest in 1.4) because client and server are in the same zone in the Amazon data center in Oregon and there is very little delay between consecutive requests compared to others.

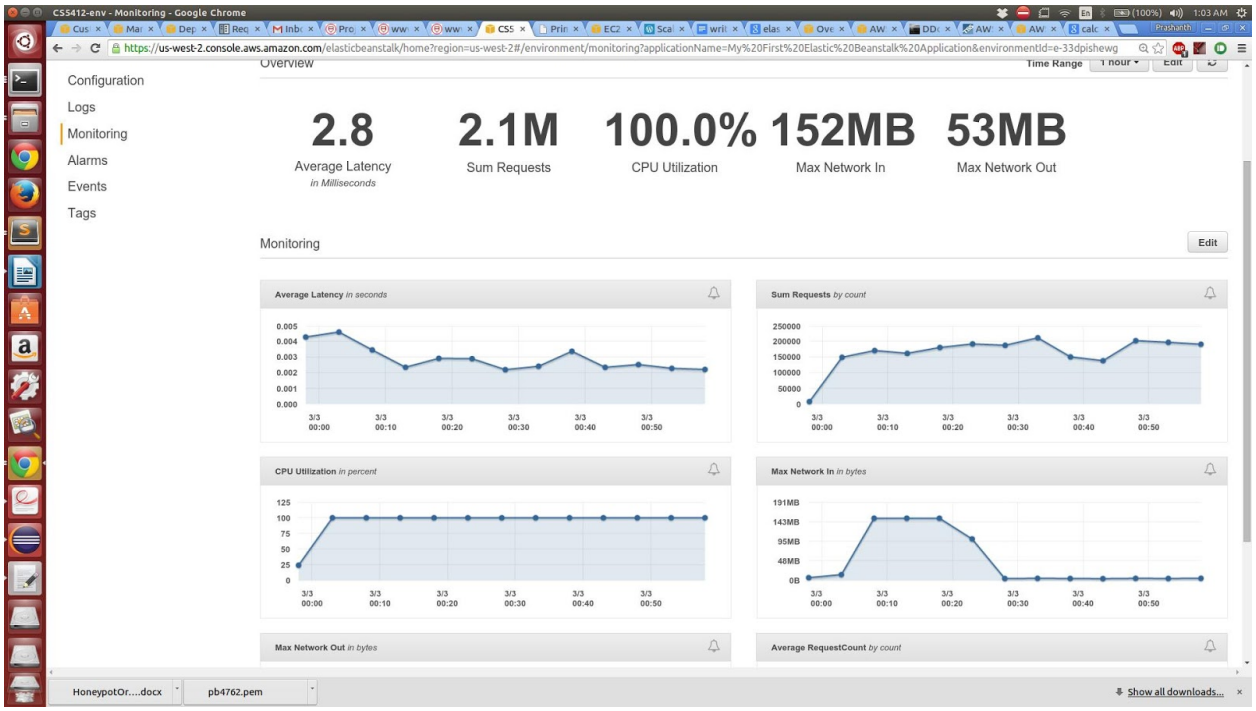
$\text{Mean Latency}_{\text{(EC2 instance in the same availability zone and region)}} > \text{Mean Latency}_{\text{(EC2 instance in a different availability zone but same region)}} > \text{Mean Latency}_{\text{(EC2 instance in a different region)}} > \text{Mean Latency}_{\text{(personal computer)}}$

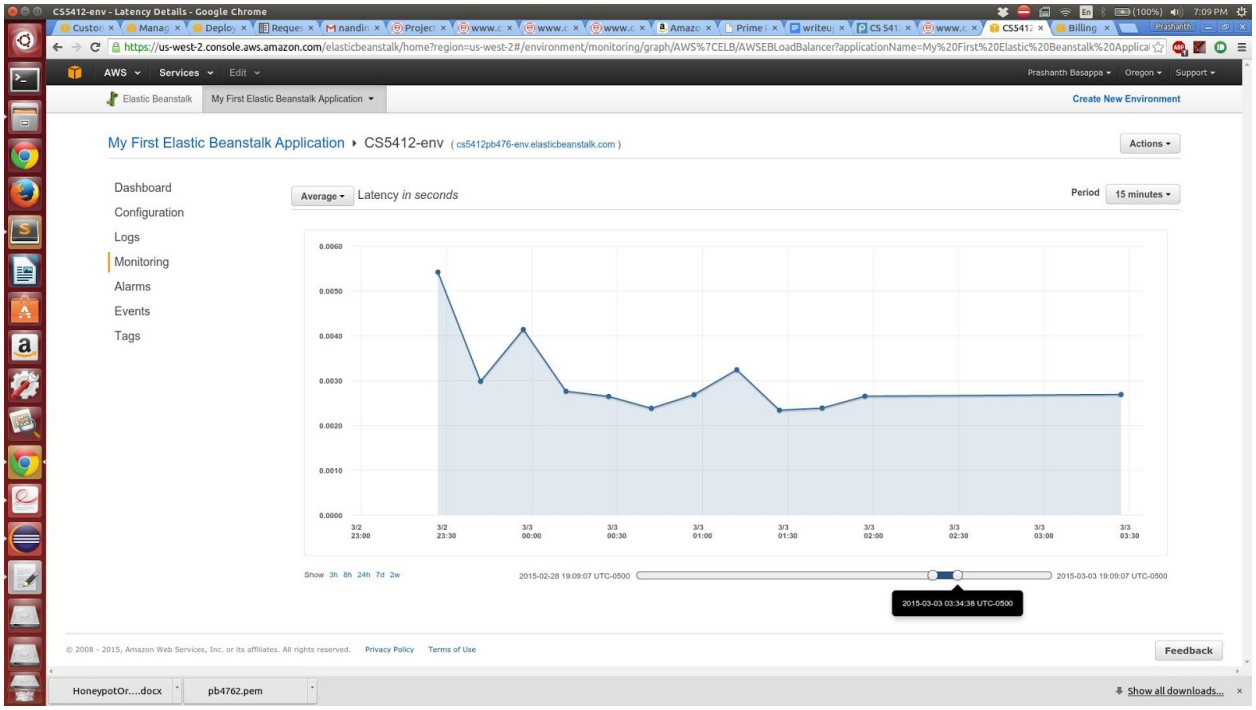
2. a) What is the latency and the throughput (requests/sec or requests/min) that you expect to get (ideally) compared to the 4th client scenario in the previous experiment?

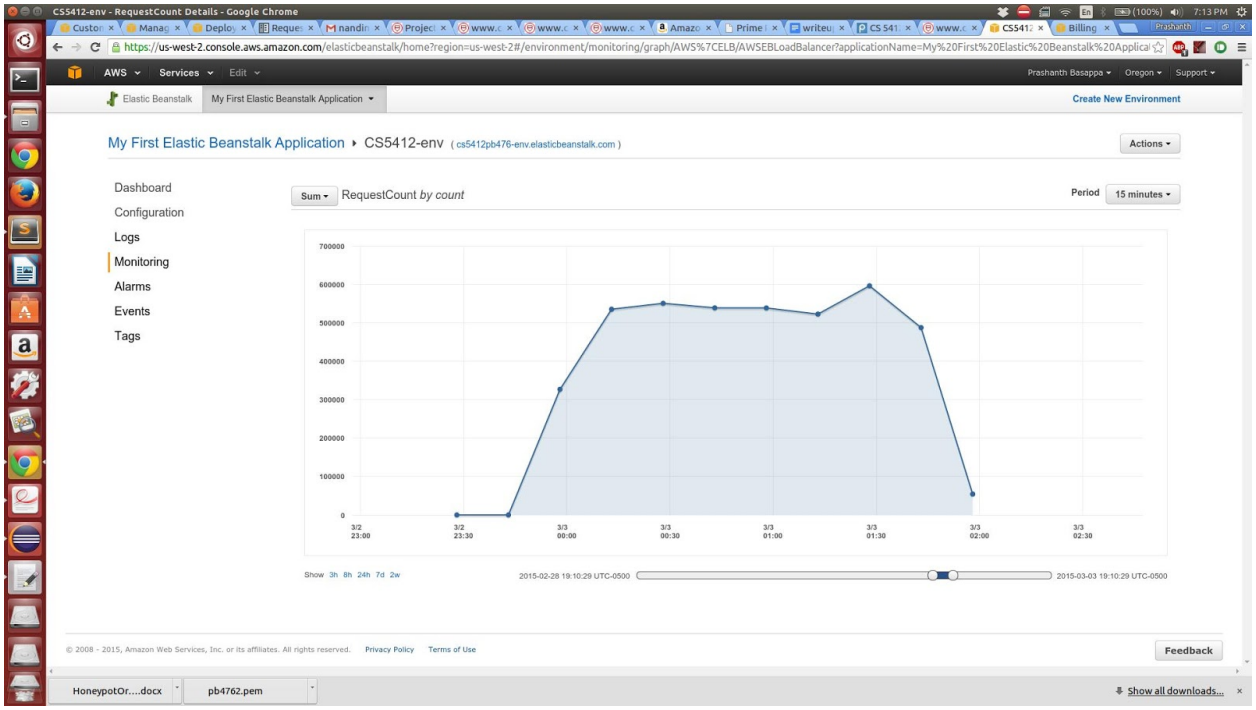
Since the client and server are in the same availability zone and region the expected latency should be approximately similar to the 4th client scenario(~10ms) and throughput should be 650 requests/min but in actuality it is slightly higher because of increased range of search(100000000003 to 100001000001) and also due to prolonged number of requests(=499999). As the range increases, the dependence on other external factors(like bandwidth, hardware) increases and we can expect a delayed response from the server.

2. b) What do you actually get from the experiment? Report the diagrams that Elastic Beanstalk produces for Average Latency and Sum Requests and the output of your four clients (latencies). Try to explain the results.

Screenshot of my Elastic Beanstalk Monitoring page showing the Average Latency and Sum Requests







Output of my four clients

2. a) sean spo38 passsean 100000000003 100001000001
Number of queries fired: 499999
Latency Min: 0.00729513168335

Latency Max: 31.1281809807
Latency Mean: 0.0131311335015
Latency Median 0.00961804389954

2. b) theo tg294 passtheo 100001000003 100002000001
Number of queries fired: 499999
Latency Min: 0.0072717666626
Latency Max: 15.05488801
Latency Mean: 0.013116561459
Latency Median 0.00971817970276

2. c) z zt27 passz 100002000003 100003000001
Number of queries fired: 499999
Latency Min: 0.00740098953247
Latency Max: 15.0544629097
Latency Mean: 0.0128502532234
Latency Median 0.00965690612793

2. d) edward ejt64 passedward 100003000003 100004000001
Number of queries fired: 499999
Latency Min: 0.00723695755005
Latency Max: 31.0707571507
Latency Mean: 0.0123633741401
Latency Median 0.00934910774231

Explanation of observed results:

When we run four different clients on EC2 concurrently, all of them on the same availability zone and region, the mean latencies were found to be very similar(13 ms,13ms,12 ms,12 ms) and much faster compared to 1a(from my personal computer). All the four median latency values among them very found to be similar as well(9ms).

The elastic beanstalk spawned multiple instances(3 more and reached the maximum number 4) and the CPU utilization reached 100% after the sum requests crossed 1.2 million requests. This can seen in the screenshot provided. The average latency in elastic beanstalk server was 2.8 ms

c) Is the server protected against a Distributed Denial of Service attack (DDOS attack) that tries to exhaust the resources of the server by making queries? If yes (no), why (why not)?

The Elastic Beanstalk server is not protected against DDOS attacks. A smart attacker can overwhelm a vulnerable service if they can exploit a resource intensive operation. Eg : A computationally expensive search query which cause multiple full table scans on backend DBs will take too many requests to stack up and start denying access to Service!

We can reduce the risk of DDOS attacks by EC2 INSTANCES ARE NOT OPEN TO PUBLIC or provision a DDOS mitigation service like Cloudfare or Neustar Cloudprotect or Reduce DNS Time to Live value (TTL).

d) Is the server protected against a Distributed Denial of Service attack (DDOS attack) that tries to exhaust the resources of the server by trying to make new connections? If yes (why), no (why not)?

The server is not protected against DDOS attacks which try to make new connections. DDOS can be achieved with a SYN flood which can occur when a host sends a flood of TCP/SYN packets, often with a forged sender address. Each of these packets is handled like a connection request, causing the server to spawn a half-open connection, by sending back a TCP/SYN-ACK packet (Acknowledge), and waiting for a packet in response from the sender address (response to the ACK Packet). However, because the sender address is forged, the response never comes. These half-open connections saturate the number of available connections the server can make, keeping it from responding to legitimate requests until after the attack ends. It can also be achieved by peer-to-peer attacks where a site could potentially be hit with up to 750,000 connections in short order. The targeted web server will be plugged up by the incoming connections.

e) Is this approach satisfying when it comes to handling bursts of queries that last a few minutes? If yes (Why), no (why not)?

This approach can be satisfied with the availability of T2 instances for EC2. T2 instances are a new low-cost, general purpose type that are designed to provide a baseline level of CPU performance with the ability to burst above the baseline. T2 instances are based on a “processing allocation model” that provides a baseline amount of processing power

combined with the ability to scale up to a full core when needed. The “burst” capability is based on the number of “CPU credits” customers accumulate during quiet periods and how many are spent while running heavy workloads.

Refer http://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/