CENG/ELEC/SENG 499 - Spring 2014

Improvement for Shoal: A Dynamic Web Cache Publishing Tool

Final Report

April 4th, 2014

Project Number : 09
Faculty Supervisor : Dr. Daniel German
External Supervisors : Dr. Ronald J. Desmarais, Mr. Michael Patterson
Team : Anita Katahoire (V00620051)
Erik Afable (V00692209)
Hao Lu (V00724916)
Mike Chester (V00711672)
Y Nguyen (V00705206)

Contents

List of Tables and Figures

1 Introduction

Shoal is an open source, web cache publishing tool developed by the UVic Department of High Energy Physics (HEP). In this section, we describe the scope and the overview of the project, explain the motivations behind improving the current Shoal implementation, and outline objectives for carrying out the improvement of Shoal.

1.1 Project Description

Shoal was developed to assist with the execution of high energy physics applications. Some of these applications require resources provided by CERN (European Organization for Nuclear Research). In order for these resources to be accessed more quickly and efficiently, they are cached on some proxy servers (Squid servers) that are geographically closer to UVic than CERN. Typically these physics applications run on static grid sites around the globe using static infrastructure. It has been shown that these physics application can be run on Infrastructure as a Service (IaaS) clouds. The issue with IaaS clouds for these physics applications is virtual machine creation and deletion is dynamic. The IaaS clouds in use have the ability to automatically create and remove virtual machines based on metrics such as network load. Having this ability to automatically create and delete virtual machines imposes a problem of figuring out the IP address of a Squid server to use, as at any moment in time a new one may be created, or an idle one may be removed. Shoal was developed to solve this issue by tracking Squid servers in a dynamically changing cloud.

The goal of our project is to take this existing application and expand on its development by making it more robust to allow better scalability when dealing with thousands of client requests per second. On top of these performance improvements we decided to include some minor GUI enhancements to the web facing front end of our application. These GUI enhancements include implementing a map to show the approximate location of each Squid, alongside a visual marker to show the load of each Squid; as well as implementing Ajax onto the main page to periodically update the list of Squids.

1.2 Motivation

Since its development, Shoal has gained some traction in the High Energy Physics community to be a potential candidate for Web Proxy Auto Detection. If adopted, this could mean thousands of worker nodes using Shoal every second to query their nearest Squid server. Previously, Shoal produced acceptable performance; however, in order for the software to be used by different organizations to support a heavier workload, the application needed to be performance tuned. Performance improvements and higher scalability will allow Shoal to be used by other research organizations besides the UVic HEP group.

1.3 Objectives

The previous implementation of Shoal employed a simple Python framework, Web.py. This implementation produced acceptable results; however, there was room for improvement in terms of performance and scalability. Specifically, this project aimed to increase the number of requests the server can handle per second and decrease the average time it takes to do geographical distance calculations. Aside from performance improvements, this 499 project also aimed to introduce new changes to the look and feel of the web interface to allow less knowledgeable clients to see what Shoal is doing.

In order to achieve the aforementioned goals, the following approach was taken:

1. Employed a new Python framework tailored for performance. The Tornado Python web server was considered as a replacement for the current Web.py web server.
2. Implemented a new algorithm and utilized caching to speed up computationally expensive operations, among which was the geographical distance calculation.
3. Implemented a new web interface for tracking proxy servers using Ajax and the JavaScript library Data-Driven Documents, D3.js.

2 Project Management and Teamwork

The Improvement of Shoal team was organized into three main groups: the Framework Improvement Team, the Graphical User Interface Improvement Team, and the Design Team. We define the roles of each team and their members in this section and have laid out our project milestones at the section's end.

2.1 Team Organization

Framework Team: Michael Chester, Y Nguyen
User Interface Team: Hao Lu, Anita Katahoire, Erik Afable
Design Team: Erik Afable

2.2 Task Division

Framework Team

- Responsible for the conversion of the web framework from Web.py to a new framework.
- Improve the speed of geographical distance calculation.
- Perform the appropriate testing and benchmarking steps.

GUI Team

- Convert the old web page templates to work with the new framework.
- Implement an additional feature on the website which can represent the current table of Squid data in a more visual form (dynamic maps, etc.).
- Perform the appropriate testing to interface changes.

Design Team

- Design and implement the Company Website.
- Design and create posters and pamphlets for the 499 Demo Presentation.

2.3 Project Milestones

ID	Task Name	Start Date	End Date	Duration	Assigned To	Percent Complete
1	Replace Web.py with the Tornado web server.	2014-01-28	2014-03-18	1 Month, 3 Weeks	M. Chester Y Nguyen	100%
2	*~~Implement a new algorithm to speed up computationally expensive operations.~~	~~2014-01-28~~	~~2014-03-18~~	~~1 Month, 3 Weeks~~	~~M. Chester, Y Nguyen~~	~~50%~~
3	Improve the web GUI	2014-02-04	2014-02-18	1 Month, 2 Weeks	E. Afable, H. Lu, A. Katahoire	100%
4	Idea Pitch	2014-01-27	2014-01-28	2 Days	Everyone	100%
5	Progress Report #1	2014-01-28	2014-01-31	4 Days	Everyone	100%
6	Progress Presentation	2014-02-18	2014-02-21	4 Days	Everyone	100%
7	Progress Report #2	2014-03-04	2014-03-07	4 Days	Everyone	100%
8	Public Demo	2014-03-20	2014-03-27	1 Week, 1 Day	Everyone	100%
9	Final Report	2014-03-25	2014-04-04	2 Weeks	Everyone	100%

* Due to limited remaining time for the project, we were not able to finish task #2. The team has decided to remove this task from our task list.

Table 1: Project milestones outlined for the Improvement of Shoal project.

3 Discussion

We will now report the tasks carried out and results from the Improvement of Shoal project. These tasks and results are outlined in the following three main sections: Performance Improvements, User Interface Improvements, and the Summary of Changes.

The Performance Improvements section breaks down the tasks, identifies performance bottlenecks, describes our solutions to those bottlenecks, and summarizes our tests plans and results. The User Interface Improvements section describes changes made to the Shoal web interface with use of Data-Driven Documents (D3) and Ajax. Lastly, the Summary of Changes section briefly overviews the changes our group accomplished in refactoring Shoal.

3.1 Performance Improvements

3.1.1 Task Breakdown

1. Performance profiling on existing system in order to identify bottlenecks.
2. Consider different alternatives to address these bottlenecks and select the best alternative based on the following criteria: performance and scalability boost, ease of use, good documentation, and open-source.
3. Implement the chosen alternative.
4. Perform unit tests and functional tests on new system.
5. Load tests and benchmark tests to compare performance between new and old system and develop proper capacity planning.

3.1.2 Identifying Performance Bottlenecks

Two potential performance bottlenecks in the existing system were the inherent performance limitation from the framework (Web.py), the calculation of geographical distance and Python threading.

3.1.2.1 Framework

Web.py, under the hood, is a multi-threaded web server. Since Shoal was written in Python, the Python global interpreter lock imposes limitations of the concurrency level of the program. Though there were multiple threads being spawn, only one thread would be executing at any given time. As the number of concurrent clients increases, this could potentially become a performance bottleneck.

Additionally, on top of the threaded web server, Shoal implemented two additional threads for the RabbitMQ worker and a periodic cleanse function which would remove inactive Squids from the cache. Working with threads imposed some constraints to our program when it modified the shared resources, as it needed to attain a lock on the global resource, which could potentially impose performance limitations on the application.

3.1.2.2 Geographical Distance Calculation

For every active Squid, Shoal needs to calculate the geographical distance between the Squid server to the client using the Haversine algorithm every time the client requests the list of nearest Squids. As the number of active Squid servers gets larger, the computation becomes more expensive and could potentially slow down the response time.

In order to confirm that the calculation call was indeed a bottleneck, we ran the following experiment:

- Send 1000 requests to the server with 1, 10, 100, 200, 300, 500 and 1000 concurrent users. Record the performance metrics (time per requests and requests/second) for each run.
- Remove the computation from the function call and replace with a simple return None. Run the experiment again with this new system and record the same performance metrics.

Using Apache Benchmark, we were able to collect the following results for the experiment:

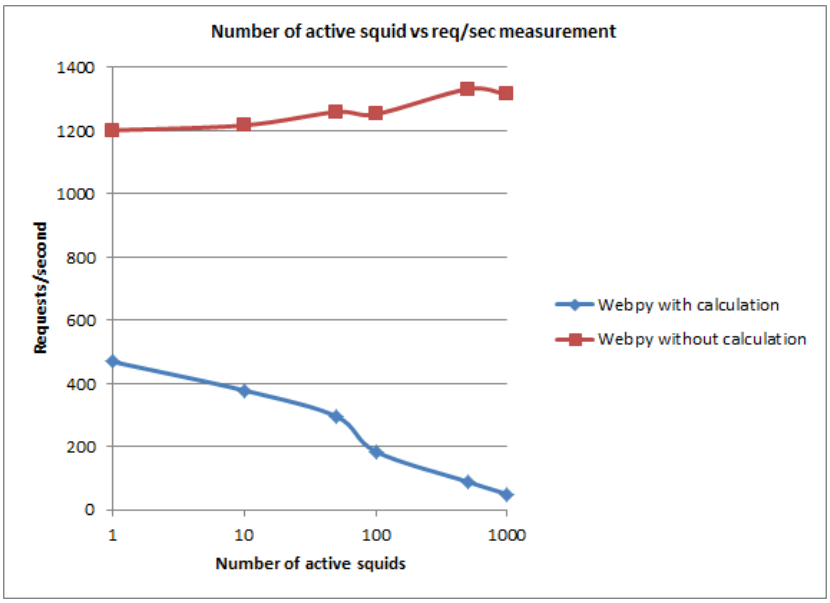

Figure 1: Impact of geographical distance calculation on number of requests/sec.

Figure 1 above demonstrates the impact of the Haversine function on the Web.py framework. As can be seen from the graph, the geographical distance calculation significantly reduces the number of concurrent requests the system can serve per second. With calculation enabled, as the number of active Squids increases, the number of requests served decreases for the system. This result was expected as the distance calculation was performed for every single Squid in the list. However, the slope quickly decreases and as seen from the graph, when the number of Squids is greater than 1000, the system reaches its limit and crashed, with the number of requests being served approaching 0. The system without calculation, however, gives consistent performance regardless of the number of active Squids.

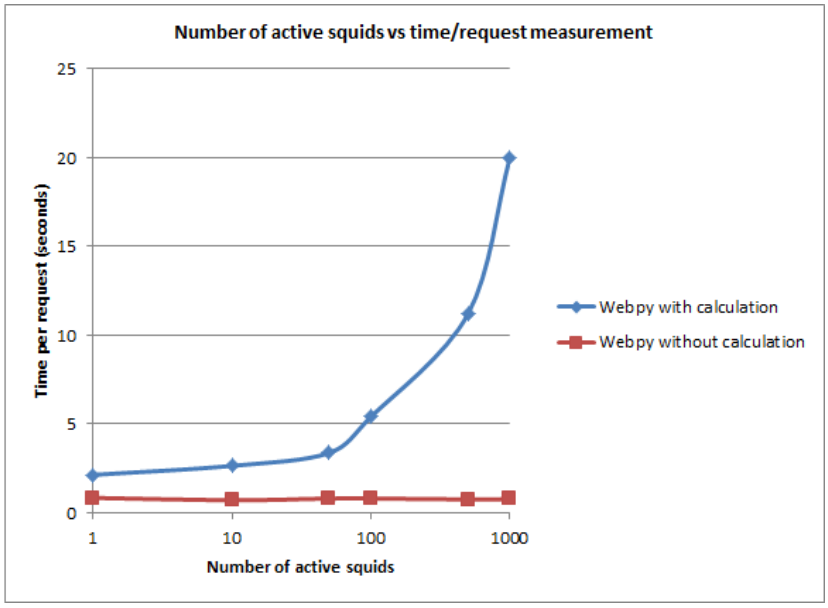

Figure 2: Impacts of calculation on time to serve 1 request.

Figure 2 demonstrates the result of the same test from a different perspective. This graph shows that the time it takes to perform the distance calculation is non-trivial and the system response time increases exponentially when the

number of Squids increases. The system without calculation remains stable regardless of the Squid count, as expected.

3.1.3 Framework Conversion

3.1.3.1 Alternative Considered

We investigated many alternative solutions to potentially replace the underlying web framework within Shoal. Some of these solutions included Django, CherryPy, and Tornado. Through our research we were able to layout the pros and cons of each framework and narrow the scope of our requirements in a web framework to these three key ideas:

1. Tuned for performance for serving RESTful API calls.
2. Provide a minimal web front end.
3. Experience with framework.

By narrowing the scope of our search we were able to eliminate Django as a possible candidate. Django provides a very robust web framework, but is mainly tailored to website development and not high throughput applications such as Shoal. CherryPy was also eliminated, as from our research when compared to Tornado, CherryPy fell behind in an application where requests per second was a key metric. In the end we chose to rework the Shoal backend onto the Tornado platform as it offered the highest throughput for our application and our team had experience in programming asynchronous web applications in the past.

3.1.3.2 Tornado

Tornado offers a very unique type of framework that is not implicitly native to Python, and that is asynchronous program execution. Fortunately, Tornado provides a very robust framework to accomplish asynchronous programming efficiently, and is well suited for an application like Shoal. Unlike Web.py, Tornado runs on a single Input/Output Loop, which eliminates the overhead of Python threading. Much like the JavaScript programming language, every function call which could potentially block must provide a callback function for which the main task will be notified when the long running task completes. The benefit of this type of framework is that processes which take a large amount of processing power can be run in the concurrently, and allow the main IO Loop to continue serving additional requests, as well as reduce overhead of creating and managing threads.

Another reason Tornado was the ideal choice for an application such as Shoal was the fact each piece of the original framework could be easily integrated into an asynchronous application. Originally, Shoal contained three main application threads:

1. A periodic thread which would awake every set interval to cleanse inactive Squids from the cache.
2. A RabbitMQ thread which would process messages from the message queue.
3. A web server thread which would serve RESTful API requests.

These three main threads fit well into an asynchronous paradigm via the following conversion:

1. Periodic thread was changed to be a periodic callback in main IO Loop. (See Appendix A)
2. RabbitMQ thread was changed to hook into the main IO Loop. (See Appendix A/Appendix F)
3. Web Server thread was replaced by the natively built in Tornado HTTP webserver framework, so we simply had to convert the templates and handlers to return the same data as the original web.py server (See Appendix A/Appendix C)

Overall converting the application to Tornado was relatively straightforward, but some consideration had to be taken when writing the application to ensure nothing would block the main IO Loop and cause the system to crash. All functions which could potentially be an issue in an asynchronous environment were built using the asynchronous decorators provided in the Tornado framework, to ensure multiple clients could request data simultaneously.

3.1.3.3 Additional Changes

On top of converting the original framework to Tornado, we spent some time working on managing configuration settings. The original application used a series of global variables in the configuration python file. The downside of configuring an application with this method is managing new settings and naming variables. The original method of gathering configuration settings was made rather difficult when a new setting was introduced into the application, a total of three places needed to be changed in order for them to be loaded properly, this was changed for our project. The new method of gathering configuration settings allows developers to simply modify the global settings dictionary directly by including the name of the new setting within its respective section, along with its default value, and the type of the setting (See Appendix B). By changing the original configuration management to the new system we eliminated the possibility of forgetting to add one of the three sections in the original method, and allowed more flexibility for developers when it came to defining new settings.

3.1.4 Geographical Calculation Improvements

3.1.4.1 Alternatives Considered

As was identified earlier through our performance analysis, the geographical distance calculation was one of the system bottlenecks that needs to be tuned. Two alternatives were considered to improve the speed of this calculation. First of all, we considered improving the calculation itself, either by means of optimizing the Haversine algorithm or implementing a new algorithm that is less computationally expensive. However, upon close inspection, we observed that the calculation was time consuming was not due to the efficiency of the algorithm but was more because of the fact that it was being called for every single active Squid. Therefore, the option to optimize the algorithm was eliminated.

The second alternative being considered was the caching of the results (the distance calculation). Since most of the active Squids stay active for a long amount of time, it is likely that the next time the client requests the nearest Squids, the same distance calculation would again be performed. This redundant repeats of the calculation could be eliminated by storing the recent calculation results in a global cache on the Shoal server. Even though the cache could be implemented manually, there exist robust open source tools that solve similar problems. Redis was chosen for the implementation of the cache. The rationale behind this decision is discussed in the following section.

3.1.4.2 Redis

Redis was considered to replace memcached as the key-value store for Shoal. Although Memcached could also be used, we decided on Redis as it has a RESTful interface, and the ability of push notifications which we could take advantage of if we had to time to work on this area. By using Redis we were hoping to implement a caching layer to Shoal for calculating nearest Squid servers, but implementing this posed some issues for us, as theoretically this could overload one server if many clients were redirected to use a particular Squid while the cache became stale. Although this could be alleviated by implementing a shorter timeout on the cache, we decided to focus our efforts on improving other aspects of Shoal instead of fully implementing a nearest Squid cache.

3.1.5 Testing Plans

Our testing procedure included three main components:

1. Unit tests and functional tests need to be implemented in order to ensure the functionality of the system is correct and is consistent with the old implementation of Shoal. The components to be tested include the main server loop, the requests handlers, the utility functions, and the web templates.
2. Integration tests: Since the framework and the GUI are developed independently, after the development and

testing phases for both components are finished, the two will be combined and integration tests will be performed to ensure the end-to-end functionality of the system.

3. Load tests: Load testing is needed to ensure the system is providing the desired performance improvement and to compare the new system with the old one.

3.1.6 Test Results

Load testing was carried out upon successful conversion of Shoal server to the Tornado framework. Using the same machines running on the same network, the following tests were performed on both the old implementation and the new implementation of Shoal. Apache Bench was again employed as the main load testing tool.

Test 1: Comparing performance between Shoal running Web.py and Shoal running Tornado

For this set of test, parameters such as total number of requests and request URL ('/nearest/5') were kept fixed while the concurrency level was varied to test the ability of the system to handle high level of concurrent incoming requests. The same tests are run with the same parameters on both the new and the old servers. The results are depicted in Figures 3 and 4 below.

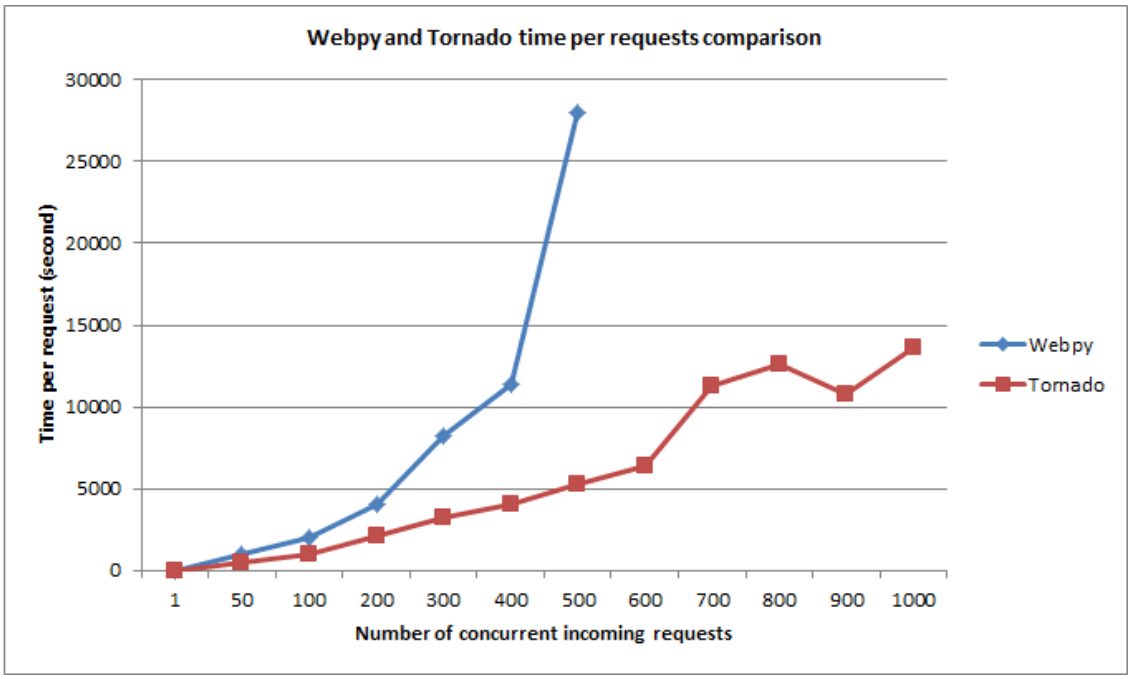

Figure 3: Time per request comparison between the Web.py and Tornado Shoal servers.

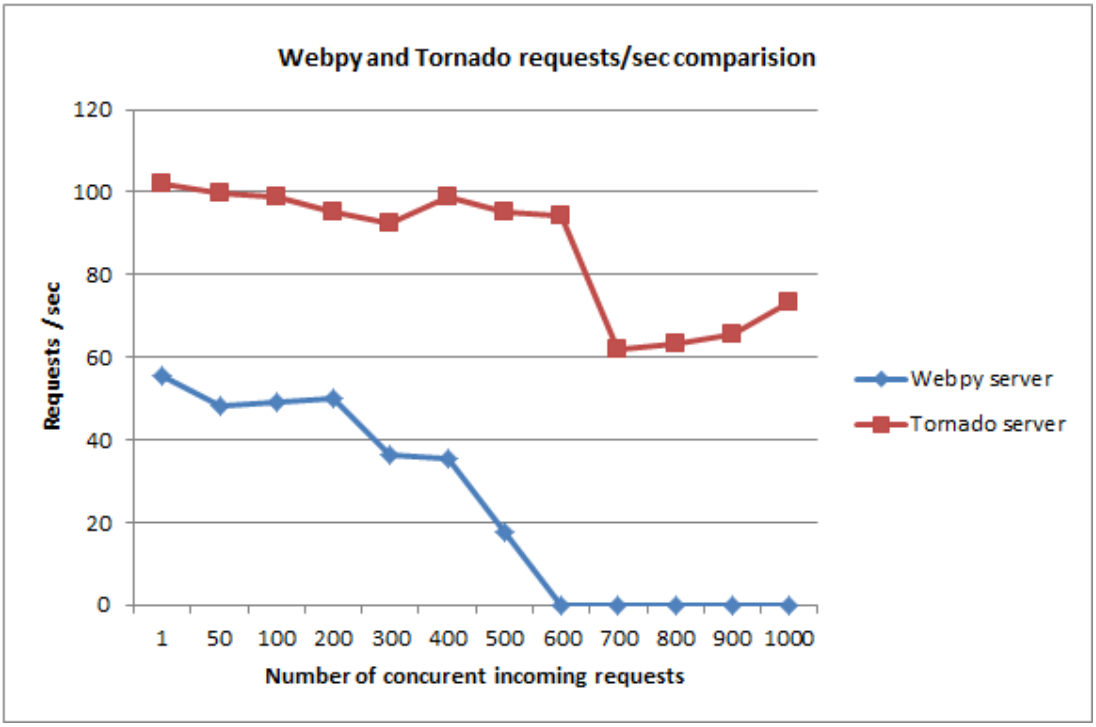

Figure 4: Requests/s comparison between the Web.py and Tornado Shoal servers.

As can be seen in Figures 3 and 4 above, Tornado outperforms Web.py in this current test case. These numbers will vary depending on the physical machine where the tests are run. However, the results are still sufficient to make meaningful comparison between the two. As the number of concurrent requests significantly impact the performance of the Web.py server. 600 concurrent requests appears to be the threshold after which Web.py becomes completely unresponsive. Results of the tests show that at 600 concurrent requests, the response time of Web.py goes to infinity and correspondingly, the number of requests served drops to 0. The Tornado server, on the other hand, is not affected as strongly. The time taken to serve one request does increase as the number of incoming requests increases. However, the slope of the increase is quite flat, meaning the server can handle a lot of concurrent requests before hitting its limit.

The drastic difference between the performances of the two servers can be explained by the underlying mechanisms of handling incoming requests by Web.py and Tornado. Web.py spawns a new thread per incoming requests and therefore, at 600 concurrent requests, it is likely that the particular machine that the tests are running on ran out of memory and was unable to allocate new threads. Tornado, on the other hand, uses epoll to handle multiple request per process. Epoll is an I/O event notification mechanism built into the Linux kernel. Using a single thread, Tornado can listen for events on different file descriptors at the same time and respond to each event as it happens. Tornado therefore can handle event-driven multitasking without requiring spawning new threads for every request. This significantly lowers the overhead and boosts the performance of the server.

Test 2: Comparing performance between Shoal running Web.py and Shoal running Tornado as the number of active Squids changes

For this test, we kept the number of incoming requests and level of concurrency as fixed parameters while varying the number of active Squids in the network. This set of tests aim to show the impact of the Squid calculation on the Tornado Shoal server versus the Web.py Shoal server. For this test, we sent 1000 incoming requests at the

concurrency level of 100 requests, while the number of Squids goes from 1 to 1000 in increments of 100.

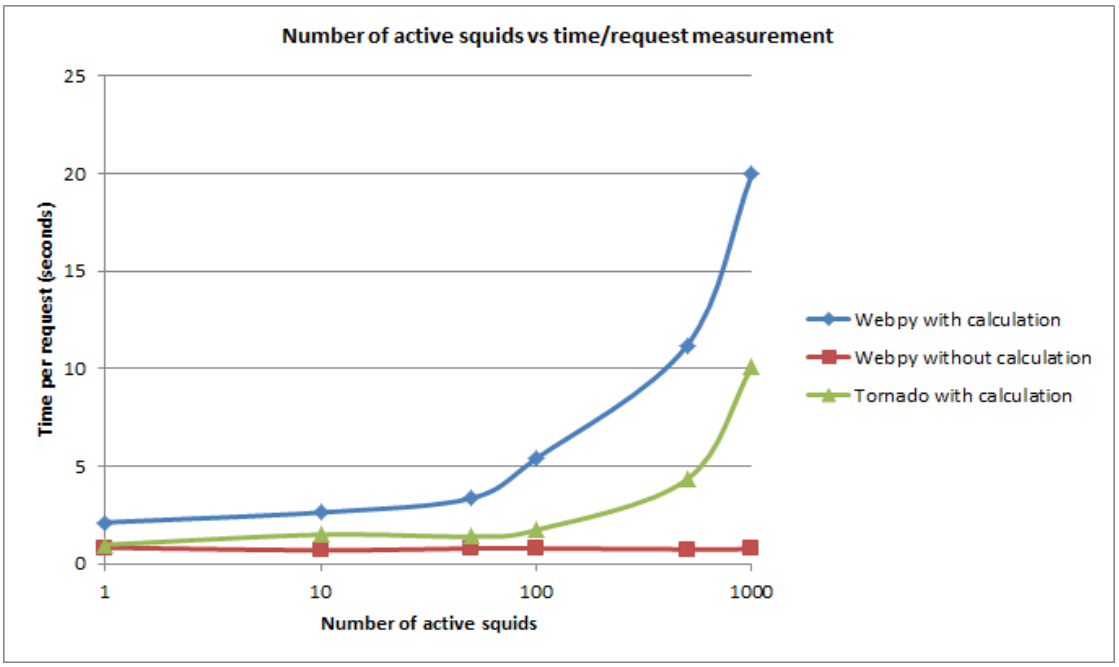

Figure 5: Time per request comparison.

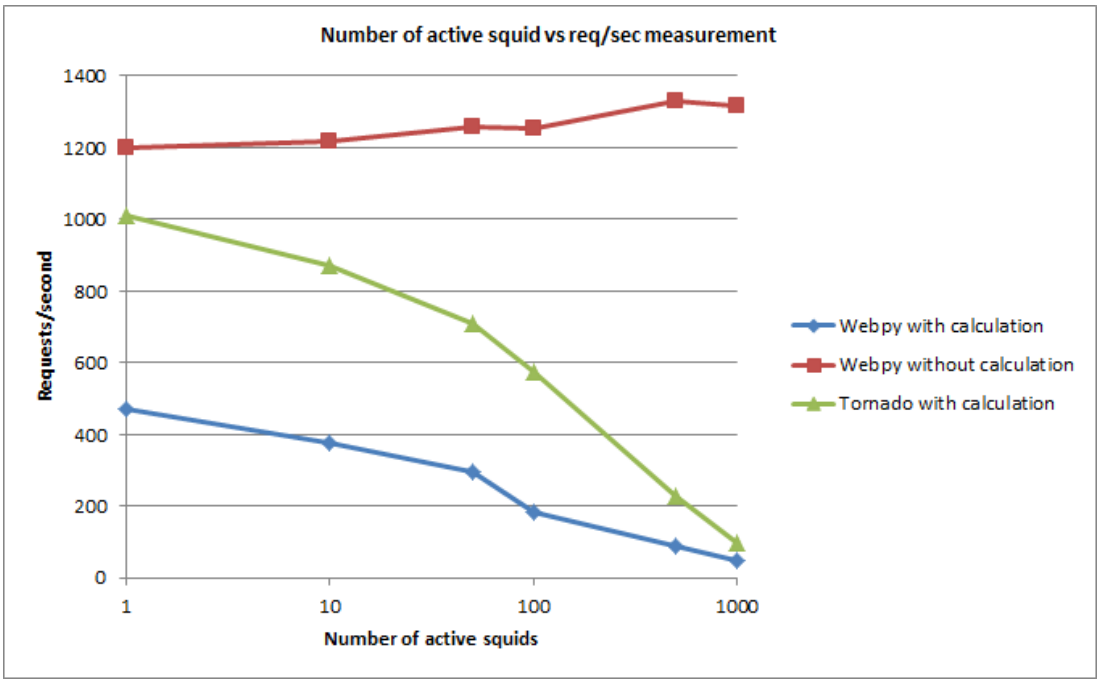

Figure 6: Requests/s comparison.

Shown in Figure 5 is the number of active Squids versus the time taken to complete the calculation to find the nearest Squids. As can be seen, as the number of Squid servers being tracked increases, the time to complete an individual request increases nearly exponentially. Although both frameworks show this exponential increase in their

behavior, Web.py increases at a much higher rate than the Tornado web server. Figure 6 shows similar data as Figure 5, but versus requests served per second. Although Shoal shows a higher degradation when comparing requests per second, this could be accounted for in our tests as Tornado may show a faster decrease when the number of concurrent requests increases, although not ideal, it still shows Tornado outperforming the original by about a 50 requests per second at 1000 tracked Squids.

3.2 User Interface Improvements

The existing user interface for the Shoal project, although informative, could be restructured to display the data in a more meaningful way and help provide a better user experience. In particular, data visualization was noted as a key lacking component. It was thus agreed upon that the new page be enhanced to meet this criteria. D3, a JavaScript library and Ajax, a web development technique, were chosen as the main resources of choice to achieve these goals.

3.2.1 Data-Driven Documents (D3)

D3.js is a JavaScript library for manipulating documents based on data. D3 uses digital data to drive the creation and control of dynamic and interactive graphical forms in web browsers by making use of widely implemented Scalable Vector Graphics (SVG), JavaScript, HTML and Cascading Style Sheets 3 (CSS3) standards. With these characteristics, D3 was strong choice to address the issue of data visualization on the Shoal web interface.

For the Shoal project, it was thought that a map would be a welcome addition to the main page since it would display the geographic locations of the different proxy servers, and other useful information such as the load at each server. The original Shoal web page displayed, amongst other things, the longitude and latitude of each proxy server. Google maps provides an API to project longitude and latitude on its own map widget. Using D3, a customized projection that wraps the Google Maps API was used to add a map to the Shoal page. A Google map was used as it is commonly used worldwide and as such is easily recognisable and usable by a variety of users. The new UI is displayed in Figure 7 below.

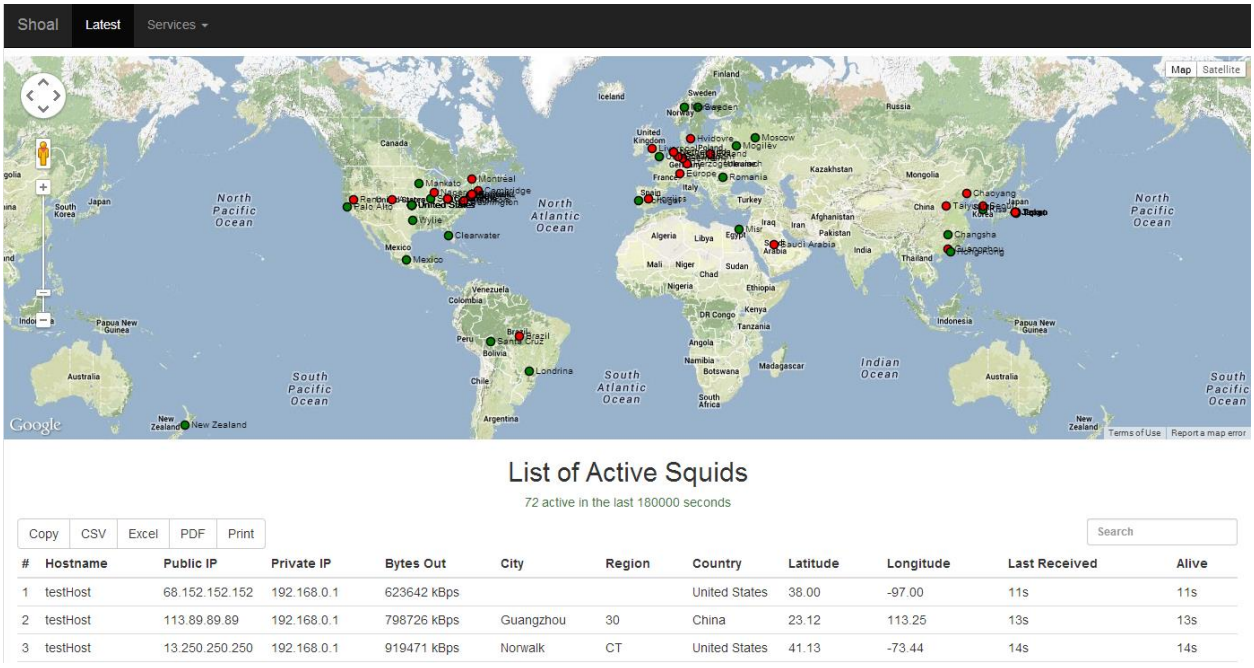

Figure 7: New UI implementation improving the Shoal web facing front.

3.2.2 Ajax

The original site updates itself every 30 seconds, via a very simple HTML tag, to display the latest proxy servers. However, this became an unreasonable behaviour when the map component is added in, as it would cause the map to refresh as well, losing any exploration a user may have been doing at that time. This behaviour disturbs the user experience and reduces the usability of the UI. The solution the team implemented was instead to use Ajax to only update the data table at a set interval, and keep a consistent user experience. The added benefit of using Ajax is reduced load on the server as well as network bandwidth, as the page now only partially refreshes, instead of a full refresh. Additionally, by using a new format for the data table, we were able to move the majority of the server computations that were used to sort the table onto the client via the use of JavaScript.

3.3 Summary of Changes

This section outlines the changes our group accomplished in refactoring Shoal below:

- Refactored entire backend Shoal code base to use the new Tornado framework.
- Refactored the configuration management of the Shoal server application, to be more developer friendly.
- Added a map to the main index page to allow users to quickly see where Squid servers are located. The map also displays graphically via different colours as to what the load of a particular server is.
- Removed the periodic refresh of the entire index page, and now use Ajax to dynamically refresh the table of Squids, thus reducing overall network bandwidth when viewing the index page. This also allows users to use the map without it being periodically reset by an entire page reload.

4 Conclusion

Overall, the project was a success. The team was able to accomplish most of our initial goals with the exception of improving the speed for geographical calculation. Otherwise, we were able to successfully convert the entire code base for Shoal server into a new framework, and implemented a richer web user interface which features a dynamic map showing location of the Squids. Due to limited time, our objective to speed up the distance calculation was not accomplished. Different approaches were taken; however, none yielded meaningful results.

For the objectives that were accomplished, we achieved promising results with an approximately 100% improvement in terms of time per request when handling 100 concurrent request compared to the old server. With this change of framework, we also had the opportunity to clean up the code base, and reduce the size of the project from having 984 lines of code to 776 lines of code.

The new user interface with the dynamic world map showing the locations and loads of the Squids was an effective retouch on the web GUI. The client can now visually inspect the active Squids from the dynamic map in conjunction with the existing table listing of Squids in order to find the most optimal proxy for connection.

5 Recommendations

Although Shoal in its current stage is reasonably high performing, there is still room for improvement in a few aspects of the application. In future implementation of the Shoal, the following improvements could be made:

1. During the development cycle, do performance analysis incrementally as changes are added to identify bottlenecks.
2. Replace geographical distance with network distance (the number of network hops) between the Squid

servers and the client, since network distance is more representative of the true distance between two machines.

3. Implement caching in order to reuse calculation result where possible. Currently, Shoal recalculates the distance for every single Squid and clients even though there is a strong possibility that the same server coordinates are repeated. The application could save time on computation by keeping a cache of precompiled values.
4. The distance calculation could be performed by the individual Squid server, instead of by the Shoal server. Every time a client requests a list of nearest Squids, the Shoal server informs all active Squids of the client location and the individual Squid server can calculate their distance to this client, then returns the result to Shoal. Shoal is only responsible for compiling the list of distances received from the Squid servers, sort this list and return the appropriate portion back to the requesting client. This would help parallelize the calculation and speed up the process. However, further testing and investigation need to be performed to determine the cost benefit ratio of this approach since there might be networking overhead involved.

6 Resources

A list of resources for the Improvement for Shoal project is listed below:

- Shoal Poster: https://particle.phys.uvic.ca/~igable/chep2013/shoal-chep2013-poster.pdf
- Shoal GitHub Repository: https://github.com/499/shoal
- 499 Group Website : http://499.github.io/shoal/
- Improvement of Shoal Company Website: http://web.uvic.ca/~eafable/shoal/index.html

7 Glossary of Terms

Term	Description
Cache	A component that stores data for quicker future retrieval.
GUI	Graphical User Interface
Haversine Algorithm	Calculating great-circle distances between two points on a sphere from their longitudes and latitudes
HTTP Proxy Server	A proxy server in a computer network that acts as an intermediary for requests from clients seeking resources from other servers.
IaaS	Infrastructure as a Service is a basic cloud-service model. Typically IaaS providers offer virtual machines and other resources to users. Cloud users install operating-system images and their application software on the cloud infrastructure.
RabbitMQ	An open source message broker software that implements the Advanced Message Queuing Protocol (AMQP).
Shoal	Custom open source software developed to provide a solution for tracking dynamic Squid cache servers.
Squid	A proxy server and web cache service.
WPAD	Web Proxy Auto-Discovery Protocol. A method used by client to locate the URL of a configuration file using DCHP and/or DNS discovery methods.

8 References

[1] I. Gable, M. Chester, P. Armstrong, F. Berghaus, A. Charbonneau, C. Leavett-Brown, M. Paterson, R. Prior, R. Sobie, R, Taylor, *"Dynamic web cache publishing for IaaS clouds using Shoal". 2013.* University of Victoria

[2] M. Bostock. (2014). *D3 API References* [Internet resource] Available: https://github.com/mbostock/d3/wiki/API-Reference

Appendix A - application.py

```
import connections
import logging
import time
import tornado.web
import tornado.ioloop
import utilities

from handlers import IndexHandler, NearestHandler, AllSquidHandler, WPADHandler


class Application(tornado.web.Application):
    def __init__(self, settings, io_loop):
        handlers = [
            (r"/", IndexHandler),
            (r"/nearest", NearestHandler),
            (r"/nearest/?(\d+)?/?", NearestHandler),
            (r"/wpad.dat", WPADHandler),
            (r"/all", AllSquidHandler),
        ]
        self.global_settings = settings
        self.shoal = {}

        # Setup Redis Connection
        logging.info("Setting up Redis connection...")
        self.redis = connections.setup_redis(self.global_settings)
        logging.info("complete.")

        # setup rabbitmq connection
        logging.info("Setting up RabbitMQ Consumer...")
        self.rabbitmq = connections.setup_rabbitmq(self.global_settings, self.shoal)
        logging.info("complete.")

        # setup periodic squid cleanse (configurable).
        logging.info("Setting up Periodic Callback to remove inactive Squids ( %ds ) every
%ds...",
            settings["squid"]["inactive_time"],
            settings["squid"]["cleanse_interval"])
        tornado.ioloop.PeriodicCallback(self.cleanse,
                settings["squid"]["cleanse_interval"]*1000,
                io_loop=io_loop).start()
        logging.info("complete.")

        # GeoIP database initialize
        if utilities.check_geolitecity_need_update(settings['general']['geolitecity_path'],
                settings['general']['geolitecity_update']):
            utilities.download_geolitecity(settings['general']['geolitecity_url'],
                settings['general']['geolitecity_path'],
settings['general']['geolitecity_update'])
        tornado.web.Application.__init__(self, handlers, **self.global_settings['tornado'])
```

```
    # Cleans up the inactive squids. Runs periodically.
    def cleanse(self):
        """updates and pops squid from shoal if it's inactive"""
        curr = time.time()
        for squid in self.shoal.values():
            if curr - squid["last_active"] > self.global_settings["squid"]["inactive_time"]:
                logging.info("Removing inactive squid (%s) %s from shoal.", squid["uuid"],
squid["hostname"])
                self.shoal.pop(squid["uuid"])
```

Appendix B - config.py

```
from os.path import join, expanduser, exists, abspath
import sys
import ConfigParser
import logging

"""Setup shoal using config file.
   setup will look for a configuration file specified in the following order:
     directory of shoal-server
     /etc/shoal/shoal_server.conf
     ~/.shoal/shoal_server.conf
   The first one found will be used.
"""

SHOAL_DIR = '/var/shoal/'

# set default values dictionary,
# add new keys here, and they will automatically be populated
settings = {
    # Tornado Specific Section
    'tornado': {
        'static_path':        { 'default_value': join(SHOAL_DIR, 'static'),
                                'type': 'string' },
        'template_path':     { 'default_value': join(SHOAL_DIR, 'templates'),
                                'type': 'string' },
        'port':               { 'default_value': 80,
                                'type': 'int' },
    },
    # General Section
    'general': {
        'geolitecity_path':   { 'default_value': join(SHOAL_DIR, 'bin'),
                                'type': 'string' },
        'geolitecity_url':    { 'default_value':
'http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz',
                                'type': 'string' },
        'geolitecity_update': { 'default_value':2592000,
                                'type': 'int' },
    },
    # Squid Section
    'squid': {
        'cleanse_interval': { 'default_value': 15,
                              'type': 'int' },
        'inactive_time':    { 'default_value': 180,
                              'type': 'int' },
    },
    # Redis Section
    'redis': {
        'port': { 'default_value': 6379,
                  'type': 'int' },
```

```
        'host': { 'default_value': 'localhost',
                  'type': 'string' },
        'db':   { 'default_value': 0,
                  'type': 'int' },
    },
    # RabbitMQ Section
    'rabbitmq': {
        'host':          { 'default_value': 'localhost',
                           'type': 'string' },
        'port':          { 'default_value': 5672,
                           'type': 'int' },
        'virtual_host':  { 'default_value': '/',
                           'type': 'string' },
        'exchange':      { 'default_value': 'shoal',
                           'type': 'string' },
        'exchange_type': { 'default_value': 'topic',
                           'type': 'string' },
        'use_ssl':       { 'default_value': False,
                           'type': 'bool' },
        'ca_cert':       { 'default_value': '',
                           'type': 'string' },
        'client_cert':   { 'default_value': '',
                           'type': 'string' },
        'client_key':    { 'default_value': '',
                           'type': 'string' },
        'reconnection_attempts': { 'default_value': 10,
                                   'type': 'int' },
    },
    'logging': {
        'config_file': { 'default_value': 'logging.conf',
                         'type': 'string' }
    },
}

def setup():
    homedir = expanduser('~')

    # find config file by checking the directory of the calling script and sets path
    if  exists(abspath(sys.path[0]+"/shoal_server.conf")):
        path = abspath(sys.path[0]+"/shoal_server.conf")
    elif exists("/etc/shoal/shoal_server.conf"):
        path = "/etc/shoal/shoal_server.conf"
    elif exists(abspath(homedir + "/.shoal/shoal_server.conf")):
        path = abspath(homedir + "/.shoal/shoal_server.conf")
    else:
        print >> sys.stderr, "Configuration file problem: There doesn't " \
                             "seem to be a configuration file. " \
                             "You can specify one in /etc/shoal/shoal_server.conf"
        sys.exit(1)

    # Read config file from the given path above
    config_file = ConfigParser.ConfigParser()
    try:
```

```
        config_file.read(path)
    except IOError:
        print >> sys.stderr, "Configuration file problem: There was a " \
                             "problem reading %s. Check that it is readable," \
                             "and that it exists. " % path
        raise
    except ConfigParser.ParsingError:
        print >> sys.stderr, "Configuration file problem: Couldn't " \
                             "parse your file. Check for spaces before or after variables."
        raise
    except:
        print "Configuration file problem: There is something wrong with " \
              "your config file."
        raise

    # Get values set in config file and update settings dictionary
    for section in settings.keys():
        for key in settings[section]:
            try:
                if config_file.has_option(section, key):
                    if settings[section][key]['type'] == 'int':
                        try:
                            settings[section][key] = config_file.getint(section, key)
                        except ValueError:
                            print "Configuration file problem: %s must be an " \
                                    "int value." % key
                            exit(1)
                    elif settings[section][key]['type'] == 'bool':
                        try:
                            settings[section][key] = config_file.getboolean(section, key)
                        except ValueError:
                            print "Configuration file problem: %s must be an " \
                                    "boolean value." % key
                            exit(1)
                    else:
                        settings[section][key] = config_file.get(section, key)
                else:
                    settings[section][key] = settings[section][key]['default_value']
            except Exception as e:
                pass

def setup_ssl():
    try:
        settings['rabbitmq']['ca_cert'] = abspath(config_file.get("rabbitmq", "ca_cert"))
        settings['rabbitmq']['client_cert'] = abspath(config_file.get("rabbitmq",
"client_cert"))
        settings['rabbitmq']['client_key']  = abspath(config_file.get("rabbitmq",
"client_key"))
    except Exception as e:
        print "Configuration file problem: could not load SSL certs"
        print e
        sys.exit(1)
```

```
def setup_logging():
    import logging.config
    import json
    path = settings['logging']['config_file']

    try:
        with open(path, 'rt') as f:
            conf = json.loads(f.read())
    except IOError as e:
        logging.error("Unable to open logging configuration file, please specify in
configuration file.")
        sys.exit(1)

    logging.config.dictConfig(conf)


setup()
setup_logging()
if settings['rabbitmq']['use_ssl']:
    setup_ssl()
```

Appendix C - handler.py

```
import json
import tornado.web
import time
from utilities import get_nearest_squids, generate_wpad
from tornado import gen


class IndexHandler(tornado.web.RequestHandler):
    def get(self):
        shoal = self.application.shoal
        sorted_shoal = [shoal[k] for k in sorted(shoal.keys(), key=lambda key:
shoal[key]['last_active'], reverse=True)]
        inactive_time = self.application.global_settings['squid']['inactive_time']
        self.render("index.html", shoal=sorted_shoal, inactive_time=inactive_time,
now=time.time())


class NearestHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self, count=10):
        squids = yield gen.Task(
            get_nearest_squids,
            self.application.shoal,
            self.application.global_settings['general']['geolitecity_path'],
            self.request.remote_ip,
            count=count
        )

        if squids:
            self.write(json.dumps(squids))
        else:
            json.dumps({})
        self.finish()

class AllSquidHandler(tornado.web.RequestHandler):
    def get(self):
        shoal = self.application.shoal
        sorted_shoal = [shoal[k] for k in sorted(shoal.keys(), key=lambda key:
shoal[key]['last_active'], reverse=True)]

        self.write(json.dumps(sorted_shoal))

class WPADHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    @tornado.gen.engine
    def get(self):
        ip = self.request.remote_ip
```

```
db_path = self.application.global_settings['general']['geolitecity_path']
shoal = self.application.shoal
proxy = yield gen.Task(generate_wpad, shoal, ip, db_path)
self.set_header('Content-Type', 'application/x-ns-proxy-autoconfig')
if proxy:
    wpad = "function FindProxyForURL(url, host) { return \"%s DIRECT\";}" % proxy
    self.set_header('Content-Length', len(wpad))
    self.write(wpad)
else:
    wpad = "function FindProxyForURL(url, host) { return \"DIRECT\";})"
    self.set_header('Content-Length', len(wpad))
    self.write(wpad)
self.finish()
```

Appendix D - utilities.py

```
import sys
import os
import subprocess
import pygeoip
import logging
import operator
import gzip
from time import time, sleep
from math import radians, cos, sin, asin, sqrt
from urllib import urlretrieve
from tornado import gen


def get_geolocation(db_path, ip):
    """
        Given an IP return all its geographical information (using GeoLiteCity.dat)
    """
    try:
        gi = pygeoip.GeoIP(db_path)
        return gi.record_by_addr(ip)
    except Exception as e:
        logging.error(e)
        return None

@gen.engine
def calculate_distance(db_path, ip, shoal, callback=None):
    """
        Given an IP return a sorted list of nearest squids up to a given count
    """
    request_data = get_geolocation(db_path, ip)
    #request_data = get_geolocation(db_path, '142.142.0.0')
    if not request_data:
        callback(None)
    try:
        r_lat = request_data['latitude']
        r_long = request_data['longitude']
    except KeyError as e:
        logging.error("Could not read request data:")
        logging.error(e)
        callback(None)

    nearest_squids = []

    ## computes the distance between each squid and the given ip address
    ## and sorts them in a list of squids
    for squid in shoal.values():
        s_lat = float(squid['geo_data']['latitude'])
        s_long = float(squid['geo_data']['longitude'])
```

```
        distance = haversine(r_lat, r_long, s_lat, s_long)

        nearest_squids.append((squid, distance))
    callback(nearest_squids)


@gen.engine
def get_nearest_squids(shoal, db_path, ip, count=10, callback=None):
    """
        Given an IP return a sorted list of nearest squids up to a given count
    """
    nearest_squids = yield gen.Task(calculate_distance, db_path, ip, shoal)
    squids = sorted(nearest_squids, key=lambda k: (k[1], k[0]['load']))
    callback(squids[:int(count)])


def haversine(lat1,lon1,lat2,lon2):
    """
        Calculate distance between two points using Haversine Formula.
    """
    # radius of earth
    r = 6371

    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))

    return round((r * c),2)

def check_geolitecity_need_update(db_path, db_update):
    """
        Checks if the geolite database is outdated
    """
    curr = time()

    if os.path.exists(db_path):
        if curr - os.path.getmtime(db_path) < db_update:
            logging.info('GeoLiteCity is up-to-date')
            return False
        else:
            logging.warning('GeoLiteCity database needs updating.')
            return True
    else:
        logging.warning('GeoLiteCity database needs updating.')
        return True

def download_geolitecity(db_url, db_path, db_update):
    """
        Downloads a new geolite database
    """
```

```
    try:
        urlretrieve(db_url, db_path + '.gz')
    except Exception as e:
        logging.error("Could not download the database. - {0}".format(e))
        sys.exit(1)
    try:
        content = gzip.open(db_path + '.gz').read()
    except Exception as e:
        logging.error("GeoLiteCity.dat file was not properly downloaded. Check contents of {0}
for possible errors.".format(db_path + '.gz'))
        sys.exit(1)

    with open(db_path,'w') as f:
        f.write(content)

    if check_geolitecity_need_update(db_url, db_update):
        logging.error('GeoLiteCity database failed to update.')

@gen.engine
def generate_wpad(shoal, ip, db_path, callback=None):
    """
        Parses the JSON of nearest squids and provides the data as a wpad
    """
    squids = yield gen.Task(get_nearest_squids, shoal, db_path, ip, count=5)
    if squids:
        proxy_str = ''
        for squid in squids:
            try:
                proxy_str += "PROXY http://{0}:{1};".format(squid[0]['hostname'],
squid[0]['squid_port'])
            except TypeError as e:
                continue
        callback(proxy_str)
    else:
        callback({})
```

Appendix E - __init__.py

```
import logging
import config
import signal
import time
import tornado.ioloop

from application import Application

def shutdown():
    max_wait = 3

    io_loop = tornado.ioloop.IOLoop.instance()
    logging.info("Shutting down Tornado IOLoop in %s seconds...", max_wait)

    deadline = time.time() + max_wait

    def stop_loop():
        now = time.time()
        if now < deadline and (io_loop._callbacks or io_loop._timeouts):
            io_loop.add_timeout(now + 1, stop_loop)
        else:
            io_loop.stop()
            logging.info("Shutdown.")
    stop_loop()

def signal_handler(sig, frame):
    logging.warning("Caught signal: %s", sig)
    tornado.ioloop.IOLoop.instance().add_callback(shutdown)

def run():
    # setup signal handling
    signal.signal(signal.SIGTERM, signal_handler)
    signal.signal(signal.SIGINT, signal_handler)

    io_loop = tornado.ioloop.IOLoop.instance()

    app = Application(config.settings, io_loop)

    # Hook RabbitMQ consumer into Tornado IOLoop
    logging.info("Hooking RabbitMQ Consumer into IOLoop...")
    io_loop.add_timeout(time.time() + .1, app.rabbitmq.run)
    logging.info("complete.")

    logging.info("Starting Tornado on Port %s", config.settings['tornado']['port'])
    app.listen(config.settings['tornado']['port'])

    io_loop.start()
```

Appendix F - rabbitmq.py

```
import json
import logging
import pika
import socket
import sys
import urllib
import uuid

from shoal_server import utilities

from time import time
from pika import adapters

class Consumer(object):
    """
        Basic RabbitMQ async consumer. Consumes messages from a unique queue that is declared
when Shoal server first starts.
        The consumer takes the json in message body, and tracks it in the dictionary `shoal`
    """
    def __init__(self, settings, shoal):
        """constructor for RabbitMQConsumer, uses values from settings file"""
        self.host = "{0}/{1}".format(settings['rabbitmq']['host'],
                                     urllib.quote_plus(settings['rabbitmq']['virtual_host']))
        #self.shoal = shoal
        self._connection = None
        self._channel = None
        self._closing = False
        self._consumer_tag = None
        self._settings = settings
        self.shoal = shoal

        self.queue = socket.gethostname() + "-" + uuid.uuid1().hex
        self.exchange = settings['rabbitmq']['exchange']
        self.exchange_type = settings['rabbitmq']['exchange_type']
        self.routing_key = '#'
        self.inactive = settings['squid']['inactive_time']

    def connect(self):
        """establishes a connection to the AMQP server with SSL options"""
        # gets SSL options from settings files
        failed_connection_attempts = 0
        ssl_options = {}
        try:
            if self._settings['rabbitmq']['use_ssl']:
                logging.debug("Using SSL")
                ssl_options = {
                    "ca_certs": self._settings['rabbitmq']['ca_cert'],
                    "certfile": self._settings['rabbitmq']['client_cert'],
                    "keyfile":  self._settings['rabbitmq']['client_key'],
                }
```

```
        except Exception as e:
            logging.error("Could not read SSL files")
            logging.error(e)

        logging.debug("Connecting to %s", self._settings["rabbitmq"]["host"])
        connection = adapters.TornadoConnection(pika.ConnectionParameters(
                                            host=self._settings['rabbitmq']['host'],
                                            port=self._settings['rabbitmq']['port'],
                                            ssl=self._settings['rabbitmq']['use_ssl'],
                                            ssl_options = ssl_options,

connection_attempts=self._settings['rabbitmq']['reconnection_attempts'],
                                           ),
                                            self.on_connection_open,
                                            stop_ioloop_on_close=False)
        return connection

    def on_connection_open(self, unused_connection):
        """opens channel and connection"""
        logging.debug("connection opened.")
        self._connection.channel(on_open_callback=self.on_channel_open)

    def on_connection_closed(self, connection, reply_code, reply_text):
        """stops IO connection loop"""
        self._channel = None
        if self._closing:
            self._connection.ioloop.stop()
        else:
            logging.warning('Connection closed, reopening in 5 seconds: (%s) %s',
                            reply_code, reply_text)
            self._connection.add_timeout(5, self.reconnect)

    def close_connection(self):
        """closes connections with AMQP server"""
        logging.debug("closing connection.")
        self._connection.close()

    def reconnect(self):
        """stops current IO loop and then reconnects"""
        # This is the old connection IOLoop instance, stop its ioloop
        self._connection.ioloop.stop()
        if not self._closing:
            # Create a new connection
            self._connection = self.connect()
            # There is now a new connection, needs a new ioloop to run
            self._connection.ioloop.start()

    def on_channel_open(self, channel):
        """opens connection on channel"""
        logging.debug("channel opened.")
        self._channel = channel
        self._channel.add_on_close_callback(self.on_channel_closed)
```

```
    # setup the exchange
    self._channel.exchange_declare(self.on_exchange_declareok,
                                   self.exchange,
                                   self.exchange_type)

def on_channel_closed(self, channel, reply_code, reply_text):
    """closes connection on channel"""
    logging.warning('Channel was closed: (%s) %s', reply_code, reply_text)
    self._connection.close()

def on_exchange_declareok(self, unused_frame):
    """callback for exchange"""
    logging.debug("exchange declared")
    # setup queue
    self._channel.queue_declare(self.on_queue_declareok, self.queue, auto_delete=True)

def on_queue_declareok(self, method_frame):
    """callback for queue"""
    logging.debug("binding %s to %s with %s",
                  self.exchange, self.queue, self.routing_key)
    self._channel.queue_bind(self.start_consuming, self.queue,
                             self.exchange, self.routing_key)

def on_consumer_cancelled(self, method_frame):
    """closes channel on consumer"""
    if self._channel:
        self._channel.close()

def acknowledge_message(self, delivery_tag):
    """acknowledges that consumer has received the message"""
    self._channel.basic_ack(delivery_tag)

def on_cancelok(self, unused_frame):
    """calls close channel"""
    self.close_channel()

def stop_consuming(self):
    """stops consuming, exits out of basic consume"""
    if self._channel:
        self._channel.basic_cancel(self.on_cancelok, self._consumer_tag)

def start_consuming(self, unused_frame):
    """starts consuming registered callbacks"""
    # add cancel callback
    self._channel.add_on_cancel_callback(self.on_consumer_cancelled)
    self._consumer_tag = self._channel.basic_consume(self.on_message,
                                                     self.queue)
def close_channel(self):
    """closes channel"""
    self._channel.close()

def run(self):
    """sets up connection and starts IO loop"""
```

```
        try:
            self._connection = self.connect()
        except Exception as e:
            logging.error("Unable to connect to RabbitMQ Server. {0}".format(e))
            sys.exit(1)
        self._connection.ioloop.start()

    def stop(self):
        """stops consuming and closes IO loop"""
        self._closing = True
        self.stop_consuming()
        self._connection.ioloop.start()

    def on_message(self, unused_channel, basic_deliver, properties, body):
        """Retreives information from data, and then updates each squid load in
           shoal if the public/private ip matches. Shoal's key will update with
           the load if there's a key in Shoal. geo_data will update or create a
           new SquidNode if the time since the last timestamp is less than the
           inactive time and a public/private ip exists"""
        logging.debug("Received message: %s", body)
        external_ip = public_ip = private_ip = None
        curr = time()

        # extracts information from body of AMQP message
        try:
            data = json.loads(body)
        except ValueError as e:
            logging.error("Message body could not be decoded. Message: {1}".format(body))
            self.acknowledge_message(basic_deliver.delivery_tag)
            return

        try:
            key = data['uuid']
            hostname = data['hostname']
            time_sent = data['timestamp']
            load = data['load']
            squid_port = data['squid_port']
        except KeyError as e:
            logging.error("Message received was not the proper format (missing:{0}),
discarding...".format(e))
            self.acknowledge_message(basic_deliver.delivery_tag)
            return
        try:
            public_ip = data['public_ip']
        except KeyError:
            pass
        try:
            external_ip = data['external_ip']
        except KeyError:
            pass
        try:
            private_ip = data['private_ip']
        except KeyError:
```

```
            pass

        data["last_active"] = time()
        # if there's a key in shoal, shoal's key will update with the load
        if key in self.shoal:
            self.shoal[key].update({"load": load,
                                    "last_active": time()})
        # if the difference in time since the last timestamp is less than the inactive time
        # and there exists a public or private ip, then add the geo_data of its location
        elif (curr - time_sent < self.inactive) and (public_ip or private_ip):
            geo_data =
utilities.get_geolocation(self._settings['general']['geolitecity_path'], public_ip)
            if not geo_data:
                geo_data =
utilities.get_geolocation(self._settings['general']['geolitecity_path'], external_ip)
            if not geo_data:
                logging.error("Unable to generate geo location data, discarding message")
                self.acknowledge_message(basic_deliver.delivery_tag)
                return
            else:
                data['geo_data'] = geo_data
                self.shoal[data["uuid"]] = data
```