Dark Nebula: Using the Cloud to Build a RESTful Web Service

Robert Fisher, John Fisher, and Peter Bui
Department of Computer Science
University of Wisconsin - Eau Claire
Eau Claire, WI 54702
{fisherr, fisherjk, buipj}@uwec.edu

Abstract

Today, many web sites are now taking advantage of cloud computing systems to rapidly build scalable web services. In order to gain experience and to learn about this emerging development platform, we constructed an application that uses Google App Engine to make a URL shortener. This web service can be accessed via a traditional web browser, but its notable feature is its RESTful API which allows for integration with both native and web applications along with automation and testing. Our paper describes the design and implementation of our cloud application, discusses the challenges in using Google App Engine, and evaluates the URL shortening web service.

1 Introduction

With the emerging adoption of cloud computing systems, web sites are no longer just web pages, but also services that provide programmatic interaction. To explore the use of cloud computing in developing a RESTful web service [6], we chose to develop a simple URL shortener. This custom URL shortening service transforms and condenses lengthy URLs into concise URLs that are ready for distribution. While URL shortening services already exist, we created a custom application to do the following: explore cloud computing via Google App Engine [1], manage a RESTful web service, and build libraries for interfacing our service. The main contributions in this paper are a discussion of our experience in utilizing a cloud computing platform such as Google App Engine in building a programmable web service and an evaluation of our final system.

2 Design

An overview of the architecture of our URL shortening service, Dark Nebula, is provided in Figure 1.

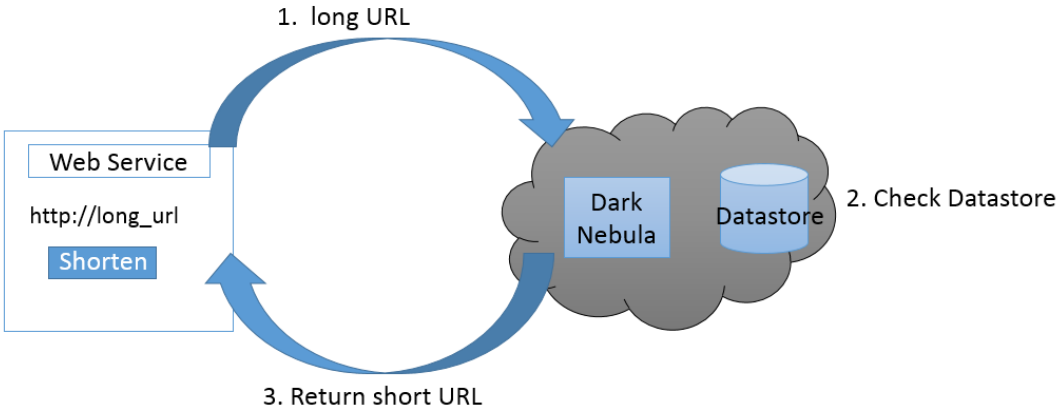

Figure 1: Dark Nebula: POSTing Process

The first step is to register a URL with the web service. This is done by the following:

1. **Send "long" URL:** The first step of the system involves sending a HTTP POST request containing the long or original URL and the response format. The POST is sent to the submission form under the domain. This step can be done with entering data on a web page or executing a POSTing script.

2. **Check Datastore:** Once the POST is made, Dark Nebula performs a query to check the Google Datastore [5] if the request contains a new or existing URL. If it finds a new long URL, the web service inserts the new entity into the Datastore. Otherwise, it retrieves the existing URL.

3. **Return "short" URL:** After the Datastore check, it returns the result based on the specified response format. The web service returns a short URL with a short URL identifier in hexadecimal with a domain prefix.

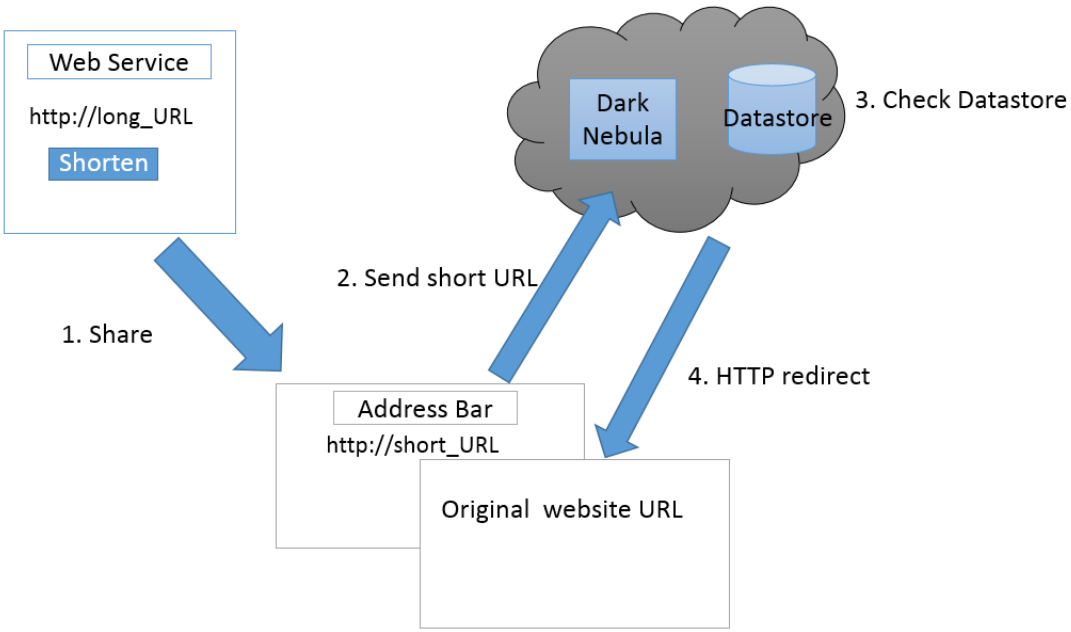

Figure 2: Dark Nebula: Redirection Process

The second step is to use the short URL obtained from the web service. This is done by the following:

1. **Share:** The short URL is distributed with other users that need a condensed URL.

2. **Send "short" URL:** The short URL is sent to the web service as HTTP GET in a web browser's address bar.

3. **Check Datastore:** Dark Nebula translates the short URL by querying the Datastore and retrieving the corresponding long URL.

4. **HTTP redirect:** After the lookup, Dark Nebula uses the retrieved long URL and automatically redirects the user to the long URL page via HTTP 302.

3 Implementation

To implement this project, we registered a domain name *go.yld.me* for the web service. We set up a Linux development environment with the Google App Engine SDK [5] and programmed the web service in Python. With this SDK, we took advantage of Google App Engine's *schema-less* data storage system, also known as the Datastore [4]. The main obstacles and challenges that we encountered primarily dealt with learning how to effectively program with a RESTful API, managing URL data in the Datastore, and debugging on a remote platform that we had no control over. To develop our system, we relied heavily on Google App Engine's *Admin* console for monitoring log files, and we performed extensive testing on our local development environment before deployment to the production cloud server.

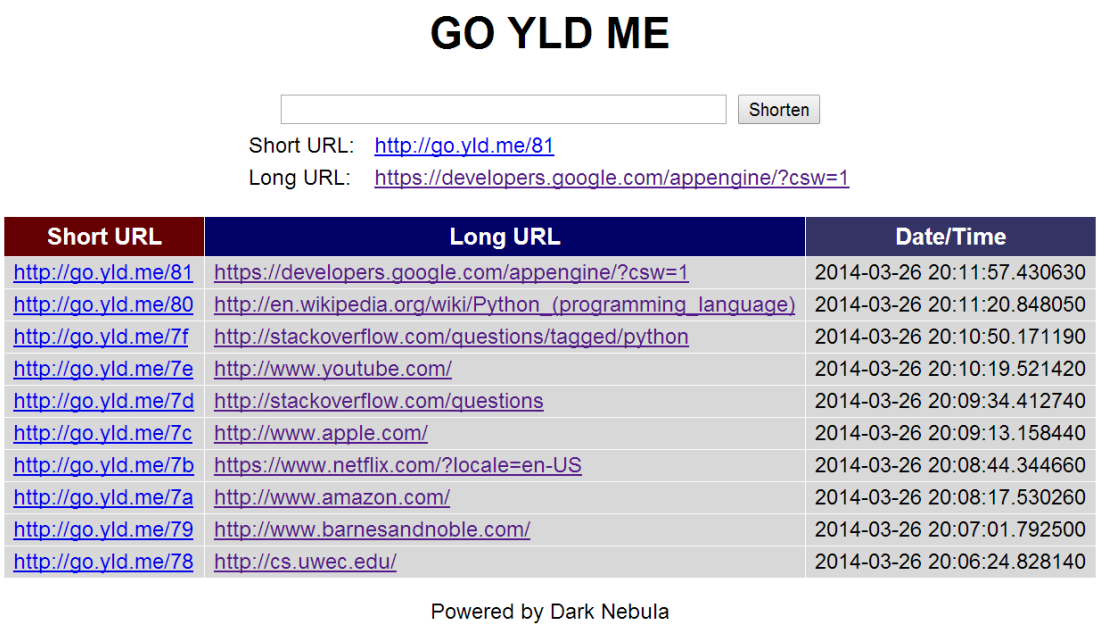

Figure 3: *go.yld.me* interface

3.1 Conversion Functions

The underlying principle behind URL shortening is the back and forth conversion of lengthy URLs to short URLs for redirection. Therefore, the first step in developing URL shortening capabilities required writing two URL path conversion functions. Ultimately, this allows us to yield a *short_identifier* to maintain a reference among short URLs and their original lengthy counterparts.

1. **identifier_to_string16:** Returns a *short_identifier* string that represents the original URL's identifier property in terms of the alphabet passed in. Since we are using a hexadecimal alphabet, we were simply converting the original long URL into its hexadecimal counterpart.

2. **string_to_identifier16:** Returns the original long URL's identifier property as a string by reverse engineering the short_identifier. This function, which takes in the same alphabet as its predecessor, essentially converts the hexadecimal identifier into decimal.

Function Name	Input	Output
identifier_to_string16	100	64
string_to_identifier16	3e8	1000

Table 1: Conversion Functions

3.2 NDB Datastore

Next, we wrote a *URL* class using the NDB Datastore API from Google App Engine [5]. This Datastore employs a schema-less storage system which, unlike a typical relational database, provides a less constrained and scalable storage platform for our application [4]. Using this approach meant simply defining the entity as a class, similar to defining a table, and its respective properties: *identifier, long_url, date, clicks, and clicks_date*.

Property Name	Return Type	Property Description
identifier	integer	A unique row in the Datastore. Arbitrarily set to start at incrementing at 0.
long_url	string	The original URL record submitted in a transaction.
date	date/time	A timestamp automatically recorded for each submitted URL.
clicks	integer	A count of the total number of clicks or redirects using the respective short URL.
clicks_date	date/time	A timestamp holding the last time the respective URL was redirected with the short URL.

Table 2: URL Model Properties

3.3 Handlers

In order to shorten submitted lengthy URLs on our website we needed to write two classes to handle HTTP GET and POST. These classes utilize the conversion functions discussed earlier, URL entity class, and webapp2 functions to display the contents of the Datastore and to ensure proper redirection of shortened URLs [7].

HomeRequestHandler: Performs the two following HTTP functions.

1. **GET:**

(a) **URL path with short_identifier:** The moment this function runs, it deletes any expired URL entities, as specified by the *clicks_date* property, to clean the Datastore. Next the *short_identifier* is isolated and passed into the *string_to_identifier16* conversion function to yield a respective *long_url_identifier*. Next, the Datastore uses the aforementioned *long_url_identifier* and queries its records to select the matching *identifier* property associated with an original URL. Before the final redirection occurs, it updates the *clicks_time* and increments the *clicks* properties. Lastly, Dark Nebula redirects to the original website with the long URL.

(b) **URL path without short_identifier:** The 10 most recently submitted URLs are queried to retrieve a *short_identifier*. With this information it can combine the prefix web address with the *short_identifier* as its path. At this point, the most recent URLs are displayed in a table with their respective properties on the home page.

2. **POST:**

(a) **Web Page:** Dark Nebula processes the input to validate whether or not the submitted URL is a well-formed, functional URL. Also, there is a query executed to check and prevent duplicate entries. If a new URL is POSTed then a new URL entity is created with an identifier and the long URL. After this, it returns the home page after storing the new entity in the Datastore.

(b) **POSTing Script:** When making a POST through a Web browser, the default response is returned as HTML. In other words, the home page is redisplayed with the most recent URLs. If the POST was sent with a *"response_format"* attribute in the form field set as *"text,"* by using a programmatic request, then only the new short URL will be returned by the GET function. This allows for a programmatic interaction for the performance tests that only return short URL strings.

3.4 Dark Nebula Shortener Module

Listing 1 shows the Python functions which describe how one can shorten a URL through a programmatic execution. By tracing the algorithm for both Dark Nebula and TinyURL, they follow the same steps: define the domain, encode a URL, make a POST request, and retrieve the shortened URL. But there is two minor differences which make these two functions unique. In step 2 and 3 of *darknebula_shorten*, it encodes and POSTs both a *long_url* and *"response_format"* set as *"text."* Then in the step 5, the *short_url* is returned using the response data. However, in step 2 and 3 of *tinyurl_shorten*, it only encodes and POSTs a *long_url.* In addition, the response from this POST retrieves the entire web page in HTML. In order to obtain the short URL from TinyURL [8], the function uses a regular expression at step 5 and 6. By providing the extra *"response_format"* data, the web service determines that it needs to only respond with a short URL. Otherwise, the default "html" would respond to the POST request with the entire web page in HTML.

```
def darknebula_shorten(long_url):
    #1. Specify the domain
    domain='http://go.yld.me'
    #2. Define and encode the long_url and response_format
    values = {'long_url': long_url, 'response_format': 'text'}
    data = urllib.urlencode(values)
    #3. POST data
    req = urllib2.Request(domain,data)
    #4. Open Dark Nebula (go.yld.me)
    response = urllib2.urlopen(req)
    #5. Retrieve the short_url
    short_url = response.read()
    return short_url

def tinyurl_shorten(long_url):
    #1. Specify the domain and regular expression
    domain_regex = '(?<=text=\")(http:\/\/tinyurl.com\/.*)(?=\">)'
    domain='http://tinyurl.com/create.php'
    #2. Define and encod the long_url
    values = {'url': long_url}
    data = urllib.urlencode(values)
    #3. POST data
    req = urllib2.Request(domain,data)
    #4. Open TinyURL
    response = urllib2.urlopen(req)
    the_page = response.read()
    #5. Use regular expression to match the short_url on page
    match = re.search(r''+domain_regex,str(the_page))
    short_url = ''
    #6. Retrieve the short_url if match is found
    if match:
        short_url = match.group(0)
    return short_url
```

Listing 1: Dark Nebula Python Module.

4 Evaluation

We evaluated our system's translation and redirection mechanism by utilizing real URLs that we inserted into the Datastore [5]. We created test cases that checked for duplicates and shortened various sized URLs. After this step, we shared the shortened URLs with others to use. Once we shortened a URL, we could send the hyperlink to students for them to utilize. During our testing, redirection was immediate and reliable within different Web browsers. At the moment the system's Datastore has not been tested for competing transactions and performance with lots of data.

4.1 Performance Testing

To test the performance of Dark Nebula, we ran automated speed tests to compare with TinyURL.com, another URL shortening service [8]. These scripts imported the Dark Nebula module we developed and called the shortening functions from section 3.4 for each service. In doing so, we could examine whether our cloud-based URL shortener could even compete with a commercial website. Because the Dark Nebula web service provides a *"response format"* as a hidden input in the POST form, a short URL can be directly returned to the user through the GET request. To retrieve the short URL from TinyURL we developed a regular expression to parse the page after the GET request. We also ran timing tests for Dark Nebula to observe the differences when a new URL is stored into the Datastore and when it already exists.

4.1.1 Transaction Statistics

URLs	**Validation**	**Dark Nebula Avg. Time (s)**	**TinyURL Avg. Time (s)**	**Avg. Ratio (Dark Nebula/TinyURL)**
100	No	0.2371	0.4552	0.5209
100	Yes	0.5814	0.4552	1.2774

Table 3: Average Execution Times for URL Transactions

URLs	**Validation**	**Dark Nebula Std. Dev. (s)**	**TinyURL Std. Dev. (s)**
100	No	0.1575	0.5531
100	Yes	0.3625	0.5531

Table 4: Standard Deviation Times for URL Transactions

4.1.2 URL Validation for Web Services

To have validation in our system means that short URLs will only be generated based on a successful HTTP or HTTPS request. This means that long URLs must have a http:// or https:// prefix. Any attempt to input a long URL without such prefixes, will be rejected by Dark Nebula. Furthermore, these long URLs must link to actual functioning web page that return a HTTP 200 status code. Any other status code will reject the long URL. As far as we know TinyURL appends a http:// prefix to the long URL, but it accepts fake or non-functioning long URLs and a short URL would still be generated. This is what it means to have no validation.

4.1.3 Transactions Without Validation

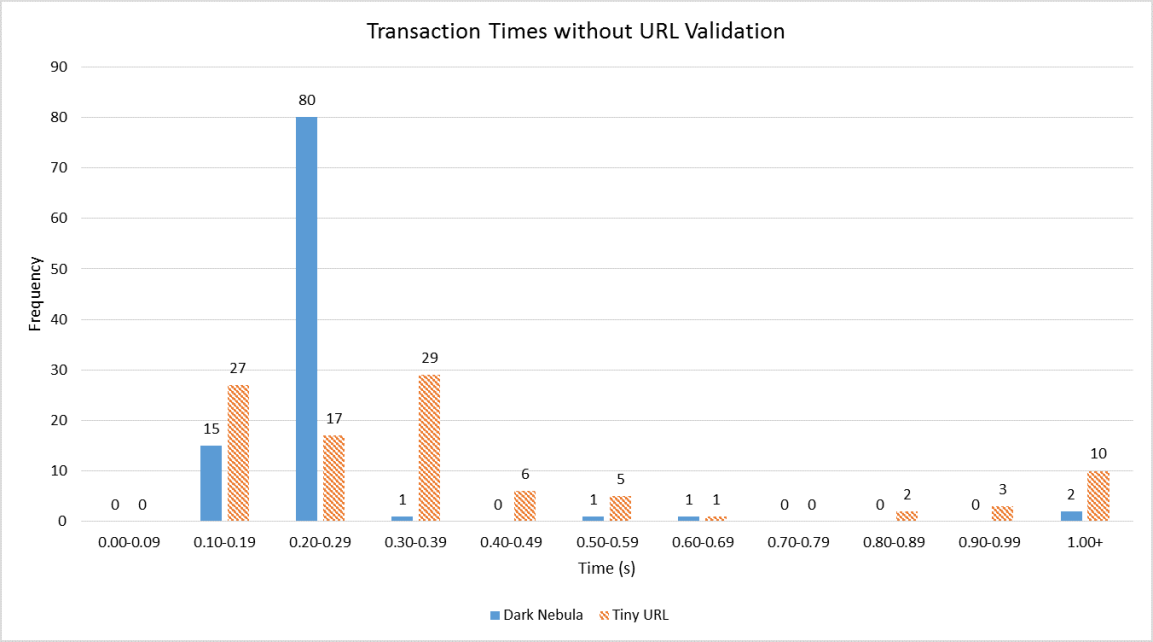

Figure 4: Performance comparison with 100 URL transactions without validation

Analysis: In this test case we wanted to determine the consistency of making URL shortening requests for Dark Nebula and TinyURL. Figure 4 displays the times for 100 individual URL Transactions without validation. This graph shows the consistency of Dark Nebula compared to TinyURL. For example, there was 80 transactions that completed in the 0.20-0.29 seconds interval. Table 3 shows that the total time ratio of Dark Nebula to TinyURL was 0.5209, or approximately a 50% reduction in time. In Table 4, the standard deviation for Dark Nebula was 0.1575, which was approximately a 30% of TinyURL's standard deviation. The graph's distribution is skewed right (meaning the right tail end of the graph is longer in comparison) for both web services, since there were some outliers for a few transactions. This behavior was more apparent when making requests for TinyURL and were less dramatic in Dark Nebula.

Explanation: The most notable observation from Figure 4 is the trade-off that occurs with validation processing. When no validation was done to check for well-formed,functional URLs, Dark Nebula processing time was frequently quicker than TinyURL. Under this test, Dark Nebula is more like TinyURL in the sense of no validation, and our web service still performed better. There is much we do not know about the technology TinyURL implements, but our test data just displays that this web service was slower in comparison. for each URL.

4.1.4 Transactions With Validation

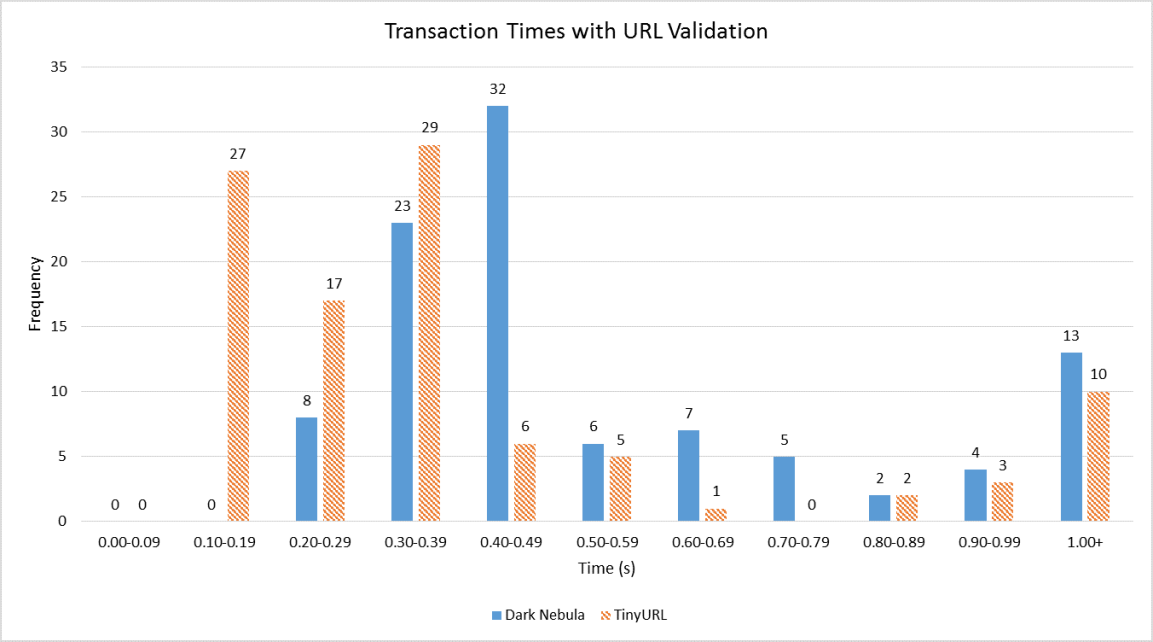

Figure 5: Performance comparison with 100 URL transactions with validation

Analysis: This test compares the times for 100 URL Transactions with validation components in Dark Nebula's web service. In Figure 5, it shows that the times for Dark Nebula's shortening requests took longer than TinyURL. Table 3 displays that the total time ratio was 1.277, or about a 25% increase in time. In Table 4, the standard deviation for Dark Nebula was 0.3625, which is about two-thirds of the standard deviation for TinyURL. To put it differently, Dark Nebula experienced more deviation when validation occurred so its performance resulted in a slower and less consistent processing.

Explanation: In contrast to Figure 4's result, when validation was implemented as displayed in Figure 5, Dark Nebula's transactions yielded more fluctuations with longer run times. However, TinyURL accepts invalid URL strings to shorten (i.e. http://dn_random.edu), while Dark Nebula does not. So with the additional overhead of validation in Dark Nebula, it demonstrates how performance can be significantly affected by choosing to handle erroneous input. Both web services are skewed to the right, but Dark Nebula was simply shifted while maintaining a slightly smaller standard deviation with a higher average time. Overall, it was expected that validation would add noticeable performance changes in Dark Nebula in some samples.

4.1.5 Influence of Regular Expressions

Dark Nebula Time (s)	Dark Nebula Validation Time (s)	TinyURL Time (s)	TinyURL Regex Time (s)
0.2545	0.3941	0.2983	0.0004

Table 5: Single Transaction Times

Analysis: Even though we used a regular expression to check for the matching short URL only for TinyURL, the amount of time to match is negligible and therefore can be discounted as a potential factor in the time increase in TinyURL. The quicker Dark Nebula times without validation could be attributed to our service's simplicity in a variety of ways. Perhaps Google App Engine's schema-less Datastore performs faster storage and queries [4]. Though, we don't know how TinyURL stores, queries, and processes short URLs. But regardless of this unknown, there appears to be less work when opening up Dark Nebula (without validation) versus TinyURL after making the POST with the long URL.

Explanation: Regular expressions in themselves are not intensive operations for our web service to perform. It doesn't have the overhead of validating a string of text but rather just the matching of it. Likewise, the time needed for our Datastore to insert or query data is more expensive with form fields to pass through. Therefore, the time delay for TinyURL gained from regular expressions is negligible.

4.1.6 New and Old Transactions

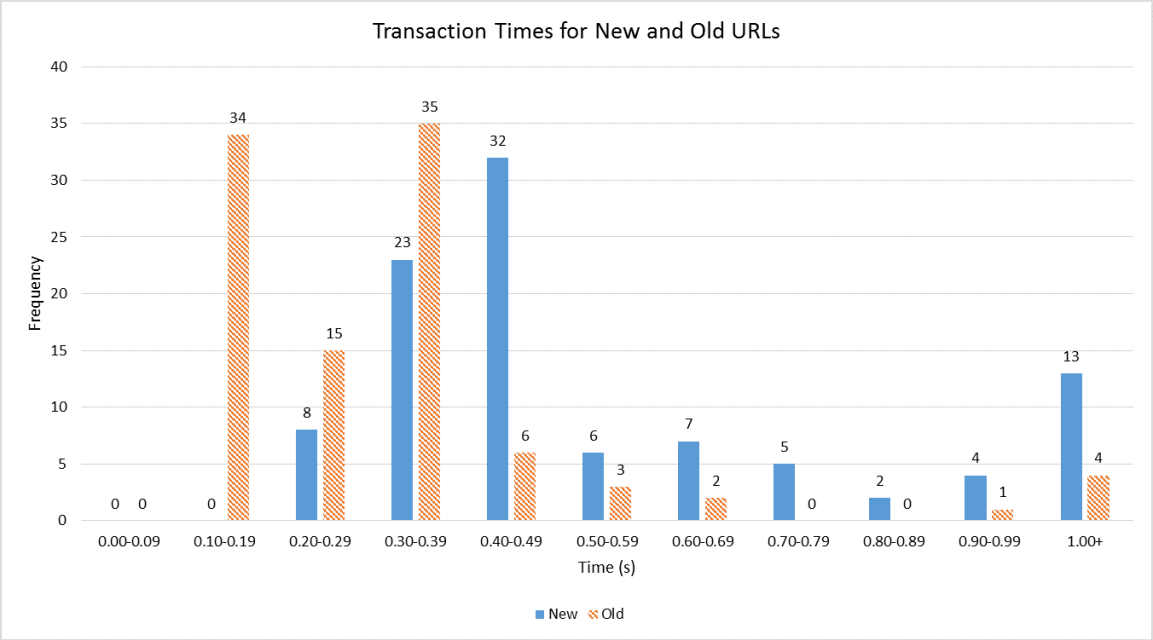

Figure 6: Performance comparison with 100 new URLs and old URLs

Analysis: In Figure 6, we compare the amount of time it takes to process a new URL and a preexisting one in Google App Engine's Datastore [5]. When a new URL is POSTed, it is stored in the Datastore, and then the web service returns the short URL. If the URL already exists, then the short URL is returned. From the trial, it shows that there is more variation and inconsistency with new URLs while older URLs are returned quicker most of time.

Explanation: The relative disparity between the new and old times makes sense because the amount of processing needed. If the URL already exists in the Datastore then less work should is needed for processing since it has already happened once before. Conversely, when a new URL is inserted the service has to jump through all the steps of shortening the URL. Simply, Dark Nebula does not need to do more work than necessary.

4.2 Input

In order for our system to correctly process incoming URLs it expects them to be well-formed and error-free. We had mentioned that Dark Nebula validates long URLs in section 3.3, but we will need to explain how and why we implemented this functionality. Below are the following scenarios which may occur.

1. **Character Length:** URL strings must not be greater than or equal to 500 characters in length. Google App Engine's Datastore [5] imposes this limit, and thus these strings will not be processed and result in an internal server error. So our validation function prevents long URLs that exceed this length. However, TinyURL [8] is able to process URL's with string lengths greater than or equal to this limit. This demonstrates an example of where the system falls short.

2. **URL Format:** The system expects well-formed URL strings. So under the hood of validation, Dark Nebula tries to fetch the long URL, and if a HTTP status code is 200, then the long URL was valid. Otherwise, an exception occurs and thus the long URL is not stored since it does not yield a functional URL.

3. **Predicting Short URLs:** Due to the nature of how we generate *short_identifier* properties, one can notice some potential risks. Unfortunately, it could be possible for input to abuse this design mechanic when the submitted URL contains the *https://go.yld.me/* prefix. These URLs would be valid under normal circumstances, but if a long URL contained a short URL that was previously entered or had not been generated yet, then an indirect reference would occur. This would either return a redirect loop or a maximum recursion depth error. To address this issue, we do not allow long URLs to contain this prefix with an *short_identifier* by using a regular expression validator.

4. **Memcache and Quota:** The Google Datastore permits 50,000 small, write, and read operations. Small operations are neither Datastore reads nor writes, but instead are key allocations, returning keys from a query, or counts. To reduce the cost of small operations, like finding the count of the total URLs present, we attempted to

use Google App Engine's *memcache* [5] to keep track of a counter . This *memcache* stores a key-value pair of the total number of URLs in the Datastore. When a URL was inserted, it incremented the counter. This made assigning the long URL identifier less costly than querying the count. However, using *memcache* to store permanent data leads to duplicate URL "identifiers" and thus incorrect short URL redirections. Apparently, the memcache's eviction policy deletes its contents in unpredictable ways [3]. For instance, the *memcache.incr("key")* could change the key, but then the key but could be different for another machine due to Google App Engine distributive nature [3]. This unpredictable behavior was unacceptable for the URL's *identifier*, so we decided to use a more costly *count* function call.

5. **Maximum Transactions and Quota:** The maximum number of transactions we found was not consistent for every daily instance. Since the quota resets every 24 hours and we used free version of Google App Engine. This is a drawback with cloud based services since the Dark Nebula web service is restricted by this quota. This was a setback to testing since we could only run expensive tests once a day. Despite this, a free Google App Engine application can use up to 1 GB of non-billable data.

5 Related Work

Web services are increasingly employing a variety of technologies and applications trending in the field of Computer Science. Below are the core concepts that our web service, Dark Nebula, utilizes.

1. **The Cloud:** A cloud is a data center which offers distributed computational power and resources [9]. Specifically, cloud computing can be partitioned into service types based on the architectural organization [2]. Specifically, our system utilizes the Platform as a Service (PaaS), Google App Engine, which offers a robust and reliable programming and execution environment [1].

2. **RESTful Web Services:** Originally introduced as an architectural style for distributed systems, Representational State Transfer (REST) is a method of creating applications to work with HTTP protocols and operations [6]. Four RESTful architectural principles include resource identification through URI (Uniform Resource Identifier), uniform interfaces (as in GET and POST operations), self-descriptive messages (as in HTML or text), and stateful interactions through hyperlinks (as in form fields and response messages) [6]. This widespread leveraging of common web standards makes RESTful web services a lightweight development tool for programmers.

3. **URL Shorteners:** URL shorteners are services that transform URLs and condense them into more manageable counterparts. TinyURL is an example of this service with a much more sophisticated interface and *short_identifier* algorithm [8] than Dark Nebula. As such, this service was used a baseline competitor.

6 Conclusion

The motivation of this project is to understand how systems like Google App Engine can be combined with a programmable interface to interact with web services. Writing a library to interact with a RESTful web service reveals the potential of programmatic interfaces. However, based on our performance tests, our system demonstrates a balancing act of speed and correctness. We can design a system to be faster, but then we must tolerate errors. Or, we can design a system to be correct while accepting slower performance. Using the cloud to make a URL shortener presented these benefits and costs. Google App Engine makes it easy to take advantage of evolving cloud technology, but there are still barriers to entry. Data processing is still limited by quotas and billing, which is a reality that most developers must accommodate for in their systems. However, keeping the cloud low to the ground still allows small scale systems, such as Dark Nebula, to allocate resources efficiently. Resource management is often an afterthought for cloud services, but it must be taken into more consideration for smaller applications. Despite the potential of cloud technology, Dark Nebula proves that the sky is not always the limit.

References

[1] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, Thomas Sandholm. What's Inside the Cloud? An Architectural Map of the Cloud Landscape, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[3] Ben Kamens. Dangers of Using Memcache Counters for A/B Tests. http://bjk5.com/post/36567537399/dangers-of-using-memcache-counters-for-a-b-tests,
2012.

[4] Fay Chang, Jeffery Dean, Sanjay Ghemawat, Wilson C. Hsieh,Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4–26, 2008.

[5] Google, Inc. Google App Engine: Platform as a Service. https://developers.google.com/appengine/, 2014.

[6] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big" web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[7] Rodridgo Moraes. WebApp2. http://webapp-improved.appspot.com/, 2011.

[8] TinyURL, LLC. TinyURL. http://tinyurl.com/, 2014.

[9] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, Dec. 2007.