GIS in the cloud: implementing a Web Map Service on Google App Engine

J.D. Blower
Reading e-Science Centre,
University of Reading,
RG6 6AL, United Kingdom
+44 118 378 5213
j.d.blower@reading.ac.uk

ABSTRACT

Many producers of geographic information are now disseminating their data using open web service protocols, notably those published by the Open Geospatial Consortium. There are many challenges inherent in running robust and reliable services at reasonable cost. Cloud computing provides a new kind of scalable infrastructure that could address many of these challenges. In this study we implement a Web Map Service for raster imagery within the Google App Engine environment. We discuss the challenges of developing GIS applications within this framework and the performance characteristics of the implementation. Results show that the application scales well to multiple simultaneous users and performance will be adequate for many applications, although concerns remain over issues such as latency spikes. We discuss the feasibility of implementing services within the free usage quotas of Google App Engine and the possibility of extending the approaches in this paper to other GIS applications.

Categories and Subject Descriptors

C.4 [**Performance of systems**] – *Performance attributes; Reliability, availability and serviceability*

H.2.8 [**Database management**]: Database applications – *Spatial databases and GIS*

H.3.5 [**Information Storage and Retrieval**]: On-line Information Services – *Web-based services*

General Terms

Experimentation, Performance, Reliability

Keywords

Geographic Information Systems, Open Geospatial Consortium, Service-Oriented Architecture, Web Map Service, raster, cloud computing, scalability

1. INTRODUCTION

The use of service-oriented architectures in Geographic Information Systems (GIS) is becoming increasingly widespread. This approach helps to hide the technical details of the datasets in question by exposing them through standard, implementation-neutral web interfaces, potentially making them available to wider audiences. The interface specifications of the Open Geospatial Consortium (OGC) are the focus of much current activity in this area ([2], http://inspire.jrc.ec.europa.eu/). Of these specifications, the Web Map Service (WMS [4]) is currently the most widely employed (e.g. [1], [5], [6]). WMS servers serve custom-generated georeferenced images that can be precisely overlain on top of other images in a GIS.

Achieving high levels of reliability and performance in such systems is challenging: such services are often processor- and disk-intensive and many of the smaller data providers (including university research groups) do not have the capability to run highly-available server infrastructures of this nature. One common way in which to increase the performance of Web Map Services is to pre-generate imagery, limiting the possible geographic extents of the images to a finite set of tiles [5] which are then served as static images. This reduces the flexibility of a "full WMS" to produce a "tiled WMS", which reduces server load and increases scalability.

The recent emergence of cloud computing brings new possibilities in service deployment. Services can be deployed in environments that can be made to scale up or down as required, with the service provider only being charged for actual usage. The task of running the hardware itself is outsourced, allowing service providers to concentrate on the software. The Google App Engine (GAE) different from many cloud environments (http://tinyurl.com/dlbllz). Instead of providing a virtualized operating system into which any software can be installed, GAE provides a custom hosting environment, for which applications have to be specially developed. Once an application is deployed in GAE, many concerns of scalability are handled automatically by the infrastructure: new service instances are dynamically provisioned as demanded by the current load. Furthermore, there is no charge for using this environment, provided that an application does not exceed certain quotas (discussed below), although service providers can expand these quotas by enabling billing. The low cost and automated scalability make GAE an attractive target for investigation.

In this study, we take some first steps towards evaluating the suitability of GAE for hosting GIS services by implementing a

Web Map Service in the GAE environment. Section 2 discusses the requirements of this system, section 3 discusses the development of the software itself, section 4 evaluates the performance of the resulting system, and finally section 5 draws conclusions about the suitability of GAE for this and other GIS applications.

2. REQUIREMENTS

The Reading e-Science Centre (http://www.resc.rdg.ac.uk) hosts a Web Map Service for environmental data, which provides visual access to many raster datasets, particularly the outputs of numerical simulations of the ocean and climate. Most users access the WMS through a custom-built web client known as Godiva2 [1], which displays the data overlain on top of a selection of background maps that provide context. We have found it very difficult to find a freely-available third-party service of these background maps that is sufficiently reliable for our needs. Therefore this study was primarily designed to meet our immediate needs of implementing a reliable and performant background map service, as well as evaluating the potential of GAE to fulfil more sophisticated requirements.

For this application, we need a WMS that can serve imagery in a variety of coordinate systems, particularly WGS:84 latitude-longitude and north and south polar stereographic. Other coordinate systems will likely be required in future. The Godiva2 client typically accesses background maps through tiled map services, although a full WMS is also required for certain applications. Low latency is an important requirement to support interactive browsing of images on a draggable, zoomable dynamic map component.

The most common background map requested by clients is the NASA Blue Marble Second Generation composite satellite image, which is available at multiple resolutions. For the Godiva2 site, the 5400x2700 JPEG image is adequate for our needs and this is the source image we use in this study.

We targeted the Java environment within GAE (a Python environment is also available), with the intention of re-using many existing investments in Java GIS technology, including our own ncWMS software (http://ncwms.sf.net). We aimed to use only open-source software and to strive as far as possible to remain within the free usage quotas of Google App Engine, in order to maximize the sustainability and reusability of our work.

3. DEVELOPMENT

3.1 Limitations of the GAE environment

The GAE software is somewhat restricted in order to allow for the automatic expansion of capacity by provisioning new service instances, which may be distributed across many physical servers. For the purposes of this study, the most important restrictions are:

- No files may be written to the server's local filesystem. All data must be written to Google's persistent store, or to the memory cache ("memcache"), both of which are distributed over many servers to provide high capacity.
- Many important classes in the standard Java Development Kit are not available, including those in the javax.imageio package and most of the contents of java.awt. These packages contain much of the code that would usually be used for image manipulation.
- Applications may not spawn background threads. This means that the present versions of many Java GIS libraries, including Geotoolkit (http://www.geotoolkit.org), cannot be run.
- The amount of random-access memory available to each service instance is low. Precise figures are not available, but tests suggest that the current limit is around 100MB. Applications must therefore make very sparing use of RAM.

These restrictions aside, GAE provides an environment close to that of a standard servlet container. Capacity is restricted by per-day and per-minute quotas, whose impact upon this application is discussed in Section 4 below.

3.2 The implementation

The above restrictions mean that implementing a full WMS within Google App Engine is far from trivial. A full discussion of the implementation is beyond the scope of this paper, but the source code and more discussion of the details of the implementation are available from http://code.google.com/p/gae-wms/.

When a WMS request for an image is received, the server must work out the geographic coordinates of each pixel in the requested image. These coordinates are then mapped onto corresponding nearest-neighbour pixels in the source image. These pixels are then extracted and assembled to produce the final image. The key challenge in this implementation was to implement an efficient means for extracting pixels from the source image. Given that the javax.imageio package is not available in GAE, and that we were unable to find a suitable replacement open-source Java JPEG-reading library that runs in GAE, the source image was uncompressed into raw ARGB (4-byte integer) pixels before being uploaded to GAE.

The resulting uncompressed raw pixel array is 58.3 MB in size, which exceeds GAE's limit of 10 MB per file. By splitting the pixel array up into chunks of 500x500 pixels (1 MB in size), the image data could be uploaded to GAE. We experimented with storing these chunks in the distributed persistent store and the memcache system. However, we found that during even moderate usage in the test cases (section 4 below) the application would exceed its free per-minute quotas for reading from these data stores. Therefore we found that the best approach was to store the chunks in the application's local filesystem, which is not associated with a quota for data reading. A data abstraction layer was then implemented that treats the mosaic of image chunks as one large virtual image. For image output, we located an open-source PNG-writing library that we adapted for GAE (http://catcode.com/pngencoder/). The GAE image transform service was used to convert PNG images to JPEGs.

Because new service instances can be provisioned automatically by the GAE infrastructure, it is beneficial if the instances have a short startup time. Therefore we found it advisable to avoid the use of large web frameworks such as Spring.

4. TESTING

We wished to evaluate the performance of the above implementation under different conditions of server load. Since we have no control over the behaviour of the GAE environment, the results of such tests may not be precisely reproducible; however, the relative performances of different server

configurations are instructive, as are some general features of the system.

4.1 Methodology

We used Apache JMeter (http://jakarta.apache.org/jmeter/) to set up and run a number of test scripts, each of which involved a single client machine making repeated requests to the server for images. Each test started with a single thread, ramping up linearly over at least 30 seconds to a maximum number of threads, which were then sustained for at least a further 30 seconds. Each thread looped repeatedly through a set of 42 image requests in random order. Each image was of size 256x256 pixels, representing three "zoom levels" in WGS:84 latitude-longitude coordinates (the first zoom level contains two images, one covering each hemisphere; the second level splits each of these images into four, and so forth).

We tested three different server configurations. In the "fully-dynamic" configuration, all images were generated dynamically from the source data, with no caching taking place within the application. In the "self-caching" configuration, the server was set up to hold all generated images in the GAE memcache system as JPEGs or PNGs. Subsequent identical requests receive the image directly from the memcache (tiling clients will naturally produce a number of identical requests). Finally, in the "static files" configuration, the client was set up to request a finite set of static image files from the server, bypassing the WMS interface. (This last configuration is similar to that of a TileCache server, http://tilecache.org/.)

In each of the three test suites, a number of test scripts were run, with different maximum numbers of threads. The throughput (the number of images received by the client from the server per second) was calculated by calculating the mean and standard deviations of a large number of "instantaneous" throughputs, each of which was estimated by calculating the time required to receive a certain number of images (typically 10) in a sliding window over the dataset. Care was taken to verify that the figures for throughput were representative and not affected by transients. The average latency in each case (the time a client observes between a request being made and the first response) was calculated by averaging over the recorded latencies in the same time window as was used to calculate the throughput.

4.2 Results

Figure 1 shows the maximum sustained throughput for successful requests versus client thread count in each of the three test suites. The error bars (one standard deviation) illustrate the variability in the measured throughput. In the "static files" configuration, the server is capable of serving over 500 requests per second; this is the limit of GAE's default capability under free usage. With 50 or more client threads, many requests therefore failed with an HTTP response code of 503 ("Service Not Available"), which we interpret as GAE blocking the test client's requests, perhaps defending against a potential denial-of-service attack.

The "self-caching" configuration was capable of serving around 200 requests per second, although we again saw many requests fail with "503" error codes with 100 client threads. In this case, we may be exceeding a different quota; by default GAE places a limit of 30 simultaneous dynamic requests within the limits of free usage.

The fully-dynamic configuration achieved around 25 requests per second with 20 client threads, but requests again failed with "503" error codes with over 10 client threads. The origin of these errors is less clear, since the observed latencies in this case were not sufficient to bring the server to a limit of 30 simultaneous dynamic requests.

A similar, but slightly lower, level of performance was recorded when requesting images in polar stereographic projection. The slight decrease in performance is due to the need to reproject the data from the original WGS:84 latitude-longitude system.

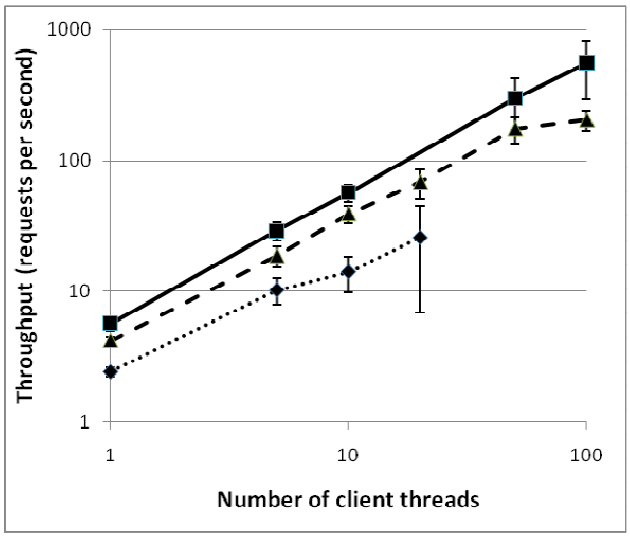

Figure 1: Maximum sustained throughput (images served per second) of the server in static files configuration (solid line), self-caching configuration (dashed line) and fully-dynamic configuration (dotted line). Error bars represent one standard deviation from the mean. In each case the trials were stopped when a large error rate was observed.

Figure 2 shows the average latency for successful requests during the same time period used to calculate the maximum sustained throughput in each case. In each of the three configurations, the increase in latency with increasing numbers of client threads is small compared with its variability. These simple averages do not capture the whole story. The first request in a test script typically causes a new virtual machine to be provisioned; such requests (known as "loading requests") are associated with high latencies (a second or more). Other spikes in latency (of up to four seconds) were observed, not all of which were associated with loading requests.

The latency of the static-tiles configuration (around 180 ms) represents an approximate minimum latency. Some proportion of this will be network latency (the client test machine was in the United Kingdom; the geographical locations of the GAE servers are unknown). The typical time to retrieve an image from the memcache in the "self-caching" configuration was 15-20 ms, which accounts for some, but not all, of the difference between the "self-caching" and "static files" configuration; the remaining ~50 ms can be considered as the overhead of a dynamic request. The higher latency of the fully-dynamic configuration is mainly associated with the extraction of data from the source files.

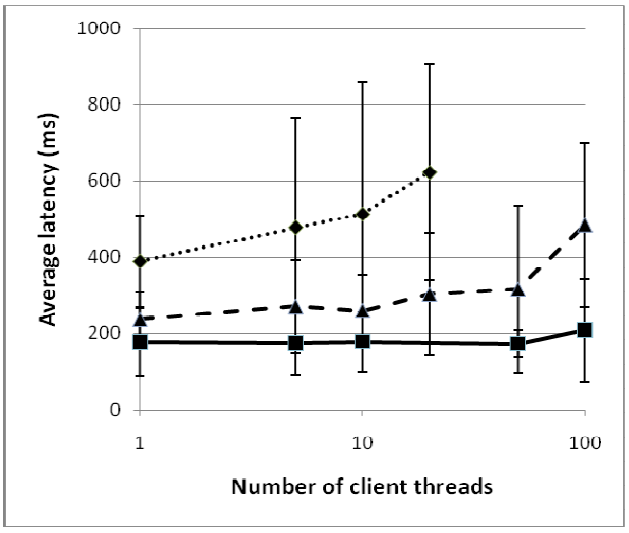

Figure 2: Average latency of the server in the three configurations. Key as in Figure 1.

Examination of the total resource usage following the tests revealed that the resource that is likely to be exhausted first (in terms of the GAE free-usage quota) is the outgoing bandwidth from the server. Currently this quota is 1GB per day. The average size of each JPEG image in this study was around 10kB, meaning that the server can serve around 100,000 JPEG images per day at no cost. Equivalent PNG images had an average size of nearly 100kB, giving a limit of around 10,000 PNG images per day.

5. DISCUSSION

This study has demonstrated that it is possible to implement a full Web Map Service for raster data within the Google App Engine environments. The system developed is sufficiently scalable and performant to meet the needs of many applications, including the particular application discussed in section 2. A fully-dynamic WMS can serve 10 simultaneous clients with an average latency of ~500 ms for 256x256 images. The same WMS, with self-caching enabled, could serve tiles to 50 simultaneous clients at latencies of ~300 ms. Implementing a static tile cache (in which all images are pre-generated) is very easy and can serve images at a rate of several hundred per second. A number of outstanding concerns remain, which may limit the potential of this system to be used for certain applications:

The cause of the "503 Service Not Available" errors. These errors appear to occur when the application exceeds certain free-usage quotas under heavy load. It is possible that the problem occurs due to the use of a single test client: perhaps GAE automatically blocks a client that is making large numbers of requests. Testing the server with many distributed clients would test this hypothesis and would also be a more realistic simulation of typical server load.

Loading requests and other latency spikes. The high latencies associated with loading requests (discussed in section 4.2) have strong implications for lightly-used services. In such situations, most requests may occur after periods of quiescence and therefore become loading requests with high latencies. Other latency spikes (which may be due to temporary server contention) also act to degrade a user's experience, even though average latencies are acceptably low for many applications.

Serving many source images. The chosen method for storing the source image (as binary chunks in the file system; see section 3.2 above) means that a single application cannot serve large numbers of high-resolution images within the free quotas (there is currently a per-application limit of 150MB of static files). This could be mitigated by storing the source images as compressed JPEGs (or other formats), at a cost of increased CPU usage and latency (and perhaps RAM usage) because of the need for decompression.

The question of access control to the service is beyond the scope of this study but some level of access control may be necessary to prevent clients from accidentally or maliciously consuming excess resource. There is currently no widely-agreed mechanism for controlling access to Web Map Services, and implementing such controls would break compatibility with most existing client tools.

Extending this WMS solution to serve vector data efficiently would be challenging. Many existing mature software systems (e.g. GeoServer, MapServer, PostGIS) have been developed over many years to address this problem and they would not generally be easy to port to the GAE infrastructure because of the limitations discussed in section 3.1 above. Perhaps the greatest challenge would be to implement an efficient spatial search algorithm within the GAE persistent data store.

6. ACKNOWLEDGMENTS

This work was supported by the UK NERC Reading e-Science Centre contract and by the EU MyOcean project. The work of the NASA Earth Observatory in providing the Blue Marble Imagery is gratefully acknowledged. The author is grateful to Hugo Mills for useful discussions on cloud computing.

7. REFERENCES

[1] Blower, J.D., Haines, K., Santokhee, A., and Liu, C. 2009. Godiva2: Interactive visualization of environmental data on the web. Phil. Trans. Roy. Soc. A, 367, 1035-9

[2] Granell, C., Dıaz, L., Gould, M. 2010. Service-oriented applications for environmental models: Reusable geospatial services. Environmental Modelling and Software, 25, 182-198

[3] OpenGIS Web Map Service Interface Specification version 1.3.0, Open Geospatial Consortium document OGC 06-042

[4] OpenGIS Web Map Tiling Service Candidate Implementation standard, Open Geospatial Consortium document OGC 07-057r6

[5] Panagos, P., Van Liedekerke, M., Montanarella, L., Jones, R.J.A. 2008. Soil organic carbon content indicators and web mapping applications. Environmental Modelling & Software, 23, 1207-9

[6] Zhang, C., Li, W. 2005. The Roles of Web Feature and Web Map Services in Real-time Geospatial Data Sharing for Time-critical applications. Cartography and Geographic Information Science 32, 4 (Oct. 2005), 269-83