IBIS and The Google App Engine

A Master's Thesis

Bas Boterman (`bboterm@cs.vu.nl`)
Dept. Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

April 15, 2009

Abstract

This document describes our attempts to make use of the Google App Engine *[8] for scientific purposes. The Google App Engine provides free resources to run a web applications on Google's infrastructure. To realize this, Google makes use of a framework, which runs in a* Python *[5] environment, and more recently, also features a* Java *[?] environment. We would like to use the resources offered by Google for scientific purposes concerning the* Ibis *[1] in particular. It contains a detailed design of both client and server, as well as implementation issues and source code.*

Contents

1 Introduction

In April 2008 [12], Google launched a platform for building and hosting web applications on their infrastructure, called the *Google App Engine* [8]. Based on cloud computing technology, Google App Engine uses multiple servers to run an application and store data. In addition, Google automatically adjusts the number of servers to handle requests simultaneously.

2 The App Engine

...

2.1 The Runtime Environment

As of April 8th 2009 [?] the Google App Engine provides two programming languages to build and run web applications. Besides *Python* [5], also the *Java* [?] programming language is featured to develop web applications and run them on Google's resources. Since the feature of running your applications in Java is still relatively new, we focused on writing our server in Python.

A Python web application interacts with the App Engine web server using the CGI protocol. An application can use a WSGI-compatible web application framework using a CGI adapter. App Engine includes a simple web application framework, called *webapp*, to make it easy to get started. For larger applications, mature third-party frameworks, such as *Django* [?], work well with App Engine.

The App Engine currently only supports Python 2.5.3. The Python interpreter runs in a secured "sandbox" environment to isolate your application for service and security. The interpreter can run any Python code, including Python modules you include with your application, as well as the Python standard library. The interpreter cannot load Python modules with C code; it is a "pure" Python environment.

2.2 Limitations

Although the Google App Engine provides us with resources, it also has some limitations we should consider. All applications hosted on the App Engine servers, are run within a sandbox. An application can only access other computers on the Internet through the provided URL fetch and email services and APIs. Other computers can only connect to the application by making HTTP (or HTTPS) requests on the standard ports. An application cannot write to the file system. An app can read files, but only files uploaded with the application code. The app must use the App Engine *datastore* for all data that persists between requests. Application code only runs in response to a web request, and must return response data within thirty seconds. A request handler cannot spawn a sub-process or execute code after the response has been sent.

2.3 Quotas

The Google App Engine is free of use up to a certain level of used resources, after which fees are charged for additional storage, bandwidth or CPU cycles required by the application. Currently the following quotas, for using the free version of the Google App Engine, are maintained:

- **CPU Time** - 46.30 hours/24h
- **Outgoing Bandwidth** - 10.00 Gbytes/24h

- **Incoming Bandwidth** - 10.00 Gbytes/24h
- **Data Storage** - 1.00 Gbytes
- **Secure Outgoing Bandwidth** - 2.00 Gbytes/24h
- **Secure Incoming Bandwidth** - 2.00 Gbytes/24h
- **Request size** - 10 MB per request
- **Response time** - 30 seconds per request
- **Data Sent to datastore** - 12.00 Gbytes/24h
- **Data Received from datastore** - 116.00 Gbytes /24h
- **Datastore CPU Time** - 62.11 hours/24h

From May 25th, new quotas will take effect [6], which are:

- **CPU Time** - 6.5 hours/24h ($0.10 for every additional hour)
- **Bandwidth** - 1 Gbyte (in and out)/24h ($0.10/$0.12 for every additional GByte in/out)
- **Stored Data & Email Recipients** - these quotas will remain unchanged ($0.15 for every additional GB stored per month)

2.4 Monitoring

For us to monitor the quota already used by the various Ibis services, we can login to the administration panel. All quotas as described above can be monitored from this website, and additionally, we can see when our quotas will be reset (being midnight, Pacific time). Historical note: The 24-hour replenishment cycle was introduced in December 2008. It replaced a more complicated system of "continuous" replenishment, to make it easier to report and control resource usage. Besides the monitoring of quotas, also various logs are available (e.g. error logs, debug logs, and access logs).

3 Server Design

Below we will describe some high-level design decisions we made, prior to implementing our server on the Google App Engine.

3.1 Server Structure

The server will have an iterative structure and servers clients on a one-thread-per-client basis; it will receive a client request, process it and return a reply (all in HTTP as described below). Since the App Engine distributes each request to possibly a different server, concurrency can occur, so a form of synchronization is needed.

The datastore uses *optimistic locking*[1] for concurrency control. An update of an entity occurs in a transaction that is retried a fixed number of times if other processes are trying to update the same entity simultaneously. Our application can execute multiple datastore operations in a single transaction, which either all succeed or all fail, ensuring the integrity of data.

Google provides us with an *SQL* (Structured Query Language) alternative, called the *Google Query Language* (GQL). An example GQL query is shown in figure 1. These queries are rather powerful when using functions like find(), as described below. Also, some security issues are triggered using the GQL. Those are described in Section 3.6.

```
db.GqlQuery("SELECT * FROM MyModel WHERE s1 >= :1 AND s2 < :2", "foo", "bar")
```

Figure 1: GQL Query example.

3.2 Client Functions

The server provides some generic functions as an interface to its clients. These functions adhere to the javadoc of the *JavaGAT*[3] (for now). Below we will describe how these functions would be implemented.

3.2.1 HTTP Requests

The Google App Engine only offers functions in a sandbox, as stated above. It is not possible to make connections to the App Engine, other than using HTTP or HTTPS. Furthermore, once a HTTP request is issued, a response is expected within a few seconds, otherwise the connection times out. There is no specification on the maximum size of an HTTP request/response.

Google offers an URL Fetch function, which supports five HTTP methods: GET, POST, HEAD, PUT and DELETE. For the functions described below, it would be

[1]Optimistic locking (also known as optimistic concurrency control) is based on the assumption that most database transactions don't conflict with other transactions.

useful to have an HTTP method that would allow us to send some (binary) data (i.e. POST or GET), after which we do not only get a success return value; but also data as a return value. This would be especially useful for the `find()` function. The HTTP POST function servers our needs best, since the GET function translates all data into a URL formatted String (which would be inappropriate for binary data). Secondly, according to the HTTP RFC , the default HTTP response is a status line (e.g. "200 OK"), followed by a message body. This suits the needs for the implementation of our server.

Our Java Adaptors could easily use these methods to authenticate themselves (with or without Google Accounts), and call functions, possible sending data as parameter of a function (using *SOAP*[2] or *JSON* (JavaScript Object Notation) [4] for example).

For Python *simplejson* [11] is available, which is a simple, fast, complete, correct and extensible JSON encoder and decoder for Python 2.4+. It is pure Python code with no dependencies, but includes an optional C extension for a serious speed boost. From Python 2.6, JSON is contained in the standard library, but since Google makes use of Python version 2.5.2, it is likely that simplejson is needed.

3.2.2 Functions to Implement

Looking at the javadoc of the JavaGAT [2](the AdvertService in specific), there are some basic functions we can implement using the App Engine. First of all a very important function is the function called `marshall()`, which takes a random object (stream of bytes) and creates a String representation of this object. This object can then be used by the following functions:

- `void add(MetaData, Object)`; Add an object with meta data to the datastore
- `void delete(path)`; Delete an object from the datastore
- `String[] find(MetaData)`; Query the service for entries matching a specified set of meta data
- `Object getInstance(path)`; Gets an object from the datastore
- `MetaData getMetaData(path)`; Gets meta data from the datastore
- `String getPWD()`; Returns the current element of the name space used
- `void setPWD(path)`; Specify the element of the name space to be used as reference for relative paths
- `void exportDataBase(target URI)`; Imports the advert database from persistent memory located at the given URI

[2]SOAP stands for Simple Object Access Protocol, a protocol specification for exchanging structured information in the implementation of Web Services. [13]

- `void importDataBase(source URI)`; Exports the advert database to persistent memory located at the given URI

Note that these functions are specific to the JavaGAT AdvertService, but could also be used for the IPL registry bootstrap service, and possibly could be placed somewhere in ibis.util for other use.

A good example are the `getPWD()` and `setPWD()` functions. Both functions are necessary to make the AdvertService adapter for the Google App Engine. However, these functions won't be very helpful for the other uses of our service. That's why they won't be implemented into our library (which communicates with the App Engine), but they will be in the adapter (locally).

3.3 MetaData

With the above in mind we also have to specify the MetaData object for other services. Generally, a MetaData object consists of a number of key value tuples, where both the keys and the values are Strings. A MetaData object should contain some basic functionality:

- `String get(key)`; Gets the value associated to the provided key.
- `String getData(int)`; Gets the value associated to the key retrieved by getKey(int).
- `String getKey(int)`; Gets the i-th key of the MetaData.
- `boolean match(MetaData)`; Match two MetaData objects.
- `void put(key, value)`; Put an entry in the MetaData object.
- `String remove(key)`; Removes an entry specified by the provided key.
- `int size()`; Returns the number of entries in the MetaData.

Furthermore, the MetaData object implements `Serializable`, which means the object can be converted into a sequence of bits so that it can be stored.

3.4 Datastore Layout

The App Engine scalable datastore stores and performs queries over data objects, known as entities. An entity has one or more properties, named values of one of several supported data types. A property can be a reference to another entity, to create one-to-many or many-to-many relationships.

3.4.1 Data Types Needed

As we look at the javadoc from the AdvertService, we notice that we need to store Advertisable objects in the Google Datastore, which is basically a key-value pair. The client requests to store a sequence of bytes, which is marshaled by the server and stored in the datastore.

The keys could be stored as a String, for example (as can be seen in the javadoc of advert.MetaData). It would be nice to implement some sort of hierarchy for the keys, if we consider that the server application could also be used as a bootstrap server for the IPL registry. This could be implemented by the "Key", provided by the Google Datastore.

The values are best stored as a string of bytes, since the client is allowed to store any object in the datastore as wished. Google provides a data type for these kind of data, which is called the BlobProperty . Blob is for binary data, such as images. It takes a String value, but this value is stored as a byte string and is not encoded as text, which is exactly what we need.

3.4.2 Datastore Functions

Besides traditional functions that Google provides to manipulate the datastore (provided by the data modeling API, defined in `google.appengine.ext.db`); Google also provides an SQL like way to communicate with the datastore, called the GQL.

3.4.3 Garbage Collection

Since the capacity of the datastore is limited to 500MB, we need some sort of *garbage collection* to keep our service usable for storing new data. This is only possible by overwriting old data with new data (i.e. removing non-used items to make room for new data). Removing old items is called garbage collection.

TTL Initially we could give every data item stored in the server a Time to live (TTL). This TTL could have a fixed value (for example 10 days), after which the data will be removed from the datastore, making room for new data. Alternatively, we could give data items a dynamic value for their TTL, depending on the usage of the datastore (i.e. if the datastore is almost full, we will give data a shorter TTL).

On every request (or once every x requests), we could query the datastore for expired items and delete them from the datastore accordingly. Another option would be that we remove the items which are expired as soon as we run out of free storage space. This would give less overhead than starting a garbage collector every request.

FIFO and LRU Secondly, if we would not give a TTL value to each data item stored in the datastore, we could remove the oldest item as soon as we run out of free storage space. By giving all items a timestamp as soon as we store then in the datastore, we could find the oldest and remove it from the datastore to make room for new data. Also we could make something like a LRU list (Least Recently Used), after which we update the timestamp as soon as a data item was referenced or edited. This way, only the data items that are least referenced are deleted, as soon as we run out of storage.

Our Solution For now we would like to guarantee that a data item at least exists for a fixed time span in the datastore. This way we prevent data items of being removed from the datastore too quickly. Secondly the TTL should not be updated if a data item is referenced. This could lead to clients referencing them only the keep them alive in the datastore. Our main purpose for the TTL is to prevent clients from storing data in the datastore and forgetting to delete it (which results in an overfull datastore). Finally we won't need garbage collection every request, but only when the datastore is reaching its limit (e.g. when 90% is used). When the datastore is full and there is no data to be evicted, we return an error to the client, which throws an `AppEngineResourcesException`. The client can then try again at a later moment in time, or use another server.

3.5 Authentication and Privacy

Google provides two forms of authentication with its App Engine.

- By means of a Google Accounts (also used for Gmail, iGoogle, etc)
- Google Apps for your Domain

For the first option, Google's unified sign-in system is used. All a user needs is a valid email address (it doesn't need to be a Gmail address!) to sign up for a Google Account. For the second option, users of Google Apps for your Domain can choose to restrict all or part of their web application to only those people who have a valid email address on their domain.

3.5.1 Authentication through Google Accounts

For our purposes it would be most attractive to make use of the authentication through Google Accounts. An application can redirect a user to a Google Accounts page to sign in or register for an account, or sign out; using simple functions like `create_login_url()`. After a successful login session the application can retrieve a User object to authenticate a user.

If one is to log in manually (`create_login_url()` was called), one would see a login page, similar to that of other Google services, like Gmail for example. Once one has pushed the 'Sign in' button, username and password are sent over HTTPS, after which a cookie is stored at the user side, containing a session ID (ACSID). This cookie is used for all sessions until logout is initiated.

Note that this function is not very useful for our client, so we will have to program the login process ourselves. There are solutions to interact with Google data services, using cURL for example .

3.5.2 Own Authentication Scheme

Second possibility would be to drop the concept of Google Account authentication, and write our own authentication scheme (for example using a private-public key pair). This has the advantage that we could apply a much more sophisticated authentication scheme and give more guarantees with respect to the issues stated above. A disadvantage is that it takes a lot more time and knowledge to implement your own authentication scheme, while there is a perfectly fine solution at hand (i.e. Google Accounts).

Another way of not using the authentication through Google accounts would be to use the Django framework, which is also supported by the Google Apps Engine. Various examples of this authentication scheme exist.

3.5.3 Server Authentication

Both authentication schemes give us two options for server setups.

- **Everyone Runs Its Own Service**; This way one could only authenticate himself by using a Google Account (i.e. one Google Account per organization), or restrict the application for only the domain of the organization using this service, or even to oneself. This way one would run out of resources less quickly than if one would use a (global) public service. Also, it is more reliable than using a public service; there is less chance of people reading and/or editing data stored by the service.
- **One Public Service for Everyone**; This way we would run an open service, accessible for everyone, without the need for a Google Account to authenticate oneself. A possible downside of this is that everyone can read/edit everyone's information. Also global public use might make the service run out of resources fast. Thus, for one global public service we can't give many guarantees.

3.5.4 Conclusion

Note that both solutions do not necessarily implement hierarchy in users. There will be an interface for the administrator, but that stands aside of the service we are offering (this interface is offered by the Google App Engine). Otherwise it seems logical to use the Google Accounts for authentication (if we want to run a 'private' service). For now we will implement two different servers; one without any form of authentication (i.e. a public server), and one with authentication (i.e. only the owner is able to use it, and additional users can be added after the server is deployed).

3.6 Security

Finally we should take a look at security besides user authentication. More decisions should be made as described below.

3.6.1 (In)Secure Connections

Google provides the means of having both insecure connections (HTTP, port 80), as having secure connections (HTTPS, port 443). Naturally some data we would rather not share in the open (for example passwords), so that would be a use for HTTPS. The question is, if it would be wise (and necessary) to run all traffic over HTTPS. We have to keep in mind that our bandwidth for HTTPS is limited (five times less than our HTTP bandwidth), but is still a reasonable amount (see Section 2.3). Plus we don't give any guarantees. If we run out of HTTPS, we just have to wait another 24 hours and it will work again. Either way configuration is done beforehand , when it comes to secure connections.

3.6.2 GQL Queries

Although the GQL provided by Google is really useful for lookup functions, they also have a downside. SQL queries are generally vulnerable to SQL injection , which is a technique that exploits a security vulnerability occurring in the database layer of an application. Basically it would make it possible to insert an SQL statement within a statement, and execute custom code. For example, if we take the query from Figure 1, and "foo" or "bar" were to be variables, then they could be replaced by code, such as shown in Figure 2.

```
'foo';DROP TABLE users;SELECT * FROM MyModel WHERE s2 LIKE '
```

Figure 2: GQL Injection example.

This would effectively execute three statements, of which the middle one could do serious damage to the database. Since there is nothing in Google's documentation about SQL (or GQL) injection, and we haven't tested it ourselves, we don't know if it will work with GQL. Although we don't provide any guarantees towards the user, it is something we should consider whilst implementing lookup functions, needed for our services. The most obvious solution will be to check for ';' in our find queries, before executing them. Checking will be done at both client and server side (in case, someone writes a malicious client).

4 Client Implementation

Below we describe what the client side (Advert class) will look like.

4.1 HTTP(S) Connections

Since our client is implemented in pure Java, it is useful to make use of the `java.net.URL` class, which allows us to make HTTP connections to the App Engine. By creating an URL object like shown in Figure 3.

```
URL url = new URL("http://ibis-advert.appspot.com/");
HttpURLConnection httpc = (HttpURLConnection) url.openConnection();
```

Figure 3: Opening an HTTP Connection.

Now it's possible with the use of the various functions in both classes to get all information needed; like status-codes, HTTP Headers, and the message body. This we will discuss below when we talk about the HTTP response.

In addition to HTTP connections, it is also possible in Java to make HTTPS connections, however, this is not as straightforward as making HTTP connections. To make an HTTPS connection we will need to import the `java.security.Security` class (amongst others). In addition, if we look at Figure 4, we see that the code adds us to the `security.provider` list in `java.security`. After these lines of code we will be able to make an HTTPS connection just like we did with making HTTP connection above (i.e. `url.openConnection()`). All sample code is fully operational at our branch in the JavaGAT SVN Repository .

```
Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());

Properties properties = System.getProperties();

String handlers = System.getProperty("java.protocol.handler.pkgs");
if (handlers == null) { //nothing specified yet (expected case)
    properties.put("java.protocol.handler.pkgs",
        "com.sun.net.ssl.internal.www.protocol");
}
else { //something already there, put ourselves out front
    properties.put("java.protocol.handler.pkgs",
        "com.sun.net.ssl.internal.www.protocol|".concat(handlers));
}
System.setProperties(properties);
```

Figure 4: Setting up an HTTPS Connection.

4.2 HTTP POST method

Secondly it is of great importance to create an HTTP POST request for both authentication and sending data to the App Engine.

4.2.1 Sending URL Encoded Strings

The simplest option is to send a various number of URL encoded strings. An example of making such an HTTP POST request can be seen in Figure 5. We enable sending output to the connection just created, calling `urlc.setDoOutput(true);`. After making this call we are able to open an OutputStreamWriter to which we can write our POST data. Multiple fields are separated by an ampersand, and variable names and values are separated by the equal sign. In this example we write the author's name and content to an HTTP POST request.

```
/* Setting up POST environment. */
httpc.setRequestMethod("POST");
httpc.setDoOutput(true);
OutputStreamWriter out = new OutputStreamWriter(httpc.getOutputStream());

/* Writing POST data. */
out.write("author=bbn230&content=test");
out.close();
```

Figure 5: Making an HTTP POST request.

4.2.2 Sending Binary Data

In addition to just making HTTP post requests, it is also possible to send binary data (commonly used for uploading files to web servers via HTTP). Again, this is a step more difficult. We have to make sure the server expects a binary string of data. This is shown in Figure 6. After the request has been made, the content can be extracted from the message body by the server and can be stored accordingly.

```
/* Setting up POST environment. */
urlc.setDoOutput(true);
urlc.setRequestProperty("Content-Type", "application/octet-stream");
```

Figure 6: Making a binary HTTP POST request.

4.2.3 Sending a Combination of Both

Multipart/Form-Data HTTP natively supports a standard for sending a combination of both URL encoded Strings and binary data. This is specified in *RFC1867* as *multipart/form-data*. Although this would seem perfect for our purposes, we decided not to use this standard for sending a combination of URL encoded Strings and binary data. RFC1867 specifies that every part of the message should be seperated by a boundary, chosen by the client. The downside is that this boundary cannot appear in the content of the payload. Since our payload can be virtually anything, we decided to go a different direction, because otherwise we would have to perform a linear search on all our data to see if a boundary token would accidentally appear in our payload.

Custom Scheme Instead of using boundaries to define the start/end of our payload, we could make use of special bytes which define the length of our payload. The layout of this scheme is shown in Figure 7. To begin with we start by sending the length of the path, which is our first four bytes. Now the server knows how many bytes the path will be (which is variable n), after which it knows the meta data will follow. The meta data also starts with four bytes, indicating the length (variable m). Finally, the object is sent, again, by sending the object length as the first four bytes and the object (variable size p) right after that.

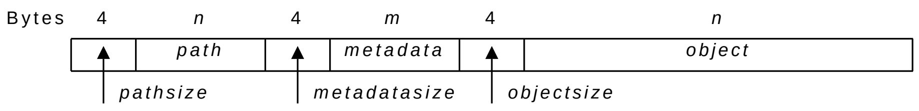

Figure 7: Custom Payload for Sending Binary Data.

Our first attempt of sending the meta data was to structure it like this: key1=value1: key2=value2: key3=value3: etc. At the client side, we would have to escape actual content containing the equals sign or a colon, by placing a backslash in front of them. We had to decode this again at the server side, to make successful searches. Note that the earlier mentioned JSON is designed, hence very effective, in sending such key-value pairs. If we take this even further, JSON can do a lot more than just encoding key-value pairs.

JSON Besides serializing a MetaData object, JSON can actually serialize our whole payload for us. This means that we don't have to work with neither boundaries, as used in multipart/from-data, nor with payload sizes. Since both Java and Python can use libraries which implements all JSON functionality, we won't have to program a new mechanism of sending multiple Strings and binary data in one request.

To give the reader a small idea of how JSON works, we will try to encode te payload above into a serial JSON String. The message consists out of three parts: a path, meta data, and a binary object. So we will make a JSON array, with each entry containing a part. The first part is just a String, which is a value for the first entry. The second part is something more complicated, which we will encode as a JSON obect, storing key-value pairs. The third entry again is a bit more difficult, since we can only store Strings, numbers, booleans, and null. That's why we formatted the binary object into a *Base64* encoded String[3]. The only downside is that Base64 encoding has a overhead of about 1.35 of the original data length. For more information about the JSON format, we refer to [4].

Once encoded, we can serialize the JSON array to a String (see Figure 8), after which it will be sent to the server. We can decode this String using simplejson at the server side, retrieving our JSON array back (see Section 5.3.2).

```
["/home/bboterm/app-engine/",{"key1":"value1","key2":"value2","key3":"value3"},
  "R0lGODlh(...)KSAAOw=="]
```

Figure 8: An example of a serial JSON array.

Note that all three parts need to be sent in one request, because suppose we send all three parts in separate requests, requests could get lost, which would then lead to inconsistencies.

4.3 HTTP(S) Login

Finally we want to 'simulate' the login process through Google accounts. To achieve this, we need the technique used above: making an HTTPS POST request.

4.3.1 Google App Engine login page

We have looked at the login page of a typical App Engine application, and we have stripped the unnecessary HTML from the login form. As we can see from the result shown in Figure 9, the login form is quite comprehensive. The login form has a lot of extra (hidden and redundant) fields, which we did not expect to find.

We are not sure what all hidden fields do, and if they are really necessary, but simulating this login page by posting all this would make our HTTP POST request quite large.

4.3.2 Google Account Authentication API

An alternative for simulating the login page is the *Account Authentication API* [7]. The Account Authentication API comes in two forms. One form for installed *Google Apps* [9], which is called *ClientLogin*, the other is for Web Apps, and is called *OAuth* or *AuthSub*.

[3]Base64 refers to a specific MIME content transfer encoding. It encodes binary data by treating it numerically and translating it into a base 64 representation.

```
<form id="gaia_loginform"
action="https://www.google.com/accounts/ServiceLoginAuth?service=ah
    &amp;sig=d71ef8b8d6150b23958ad03b3bf546b7"
  method="post"
  onsubmit="return(gaia_onLoginSubmit());">

<input type="hidden" name="ltmpl" value="gm">
<input type="hidden" name="continue" id="continue"
  value="http:// bbn230.appspot.com/_ah/login?
  continue=http://bbn230.appspot.com/" />
<input type="hidden" name="service" id="service" value="ah" />
<input type="hidden" name="ltmpl" id="ltmpl" value="gm" />
<input type="hidden" name="ltmpl" id="ltmpl" value="gm" />
<input type="hidden" name="ahname" id="ahname" value="Personal" />
<input type="text" name="Email"  id="Email" size="18" value="" />
<input type="password" name="Passwd" id="Passwd" size="18" />
<input type="checkbox" name="PersistentCookie" id="PersistentCookie"
  value="yes" />
<input type="hidden" name='rmShown' value="1" />
<input type="submit" class="gaia le button" name="signIn" value="Sign in" />
<input type="hidden" name="asts" id="asts" value="">
</form>
```

Figure 9: Stripped down version of the App Engine login page.

ClientLogin Typically, ClientLogin is used for Installed applications that need to access Google services protected by a user's Google or Google Apps. To make use of this API, we will make a HTTP POST request to `https://www.google.com/accounts/ClientLogin`, to which we send our credentials as shown in Figure 10.

We created a Google Account for testing purposes and made a successful connection using that account. The response of the ClientLogin is shown in Figure 11.

Currently, 'SID' and 'LSID' are not used by the *Google API*, so we just need to extract the 'Auth' value. Usually this value can be used directly on a service its API (when providing a developer key), but unfortunately, the Google App Engine is not listed in the Google data API library [10]. There is a workaround for still being able to connect to the App Engine, by connecting to a App Engine's login page like Figure 12. In this example, `Auth` is a variable that contains the token acquired by the example of Figure 11.

Connecting to this page will return a cookie with a session ID, just like the one described in Section 4.4.2, only with a token called 'ACSID', with which we can identify ourselves to the App Engine for every subsequent request.

AuthSub Web applications that need to access services protected by a user's Google or Google Apps (hosted) account can do so using the Authentication Proxy service. To

```
StringBuilder content = new StringBuilder();
content.append("Email=").append(URLEncoder.encode("johndoe@gmail.com",
  "UTF-8"));
content.append("&Passwd=").append(URLEncoder.encode("north23AZ", "UTF-8"));
content.append("&service=").append(URLEncoder.encode("ah", "UTF-8"));
content.append("&source=").append(URLEncoder.encode("Google App Engine",
  "UTF-8"));
```

Figure 10: Preparing our ClientLogin credentials.

```
HTTP/1.1 200 OK
Content-Type: text/plain
Cache-control: no-cache, no-store
Pragma: no-cache
Expires: Mon, 01-Jan-1990 00:00:00 GMT
Date: Mon, 16 Mar 2009 14:07:25 GMT
X-Content-Type-Options: nosniff
Content-Length: 497
Server: GFE/1.3
----
SID=DQA(...)bjG
LSID=DQA(...)nSV
Auth=DQA(...)ub4
```

Figure 11: A response from ClientLogin.

maintain a high level of security, the proxy interface, called AuthSub, enables the web application to get access without ever handling their users' account login information. Before using, verify that the Google service to be accessed supports the Authentication service. Some Google services may allow access only to web applications that are registered and use secure tokens. We have not tried AuthSub to authenticate ourselves at the App Engine. If the ClientLogin remains to fail, we could consider AuthSub as our backup. We will not use it as our primary authentication mechanism because if our users would want to install an advert server, it would require extra effort, because they need to register themselves first before this service can be used.

4.3.3 Cookies

Also, to maintain session with the server, we want to pass a session ID through cookies. Also, this can be done through Java by setting the appropriate headers. For example, if we want to send a cookie with the name "SID" and the content "abcde", we could code this in Java as shown in Figure 13. This code will add the cookie to the HTTP request, after which the response headers and body can be fetched.

```
url = new URL("http://bbn230.appspot.com/_ah/login?auth="+ Auth);
```

Figure 12: Logging In for a Session Cookie.

```
urlc.addRequestProperty("Cookie", "SID=abcde");
```

Figure 13: Sending cookies in Java.

4.4 HTTP Response

After the HTTP request is sent, we are able to receive the HTTP response. To begin with, we will download the response headers sent by the web server, after which we can retrieve the body.

4.4.1 Headers

How to recieve the headers of an HTTP response is shown in Figure 14. Already, by receiving the response code, we can distinguish errors from successful requests. Response codes starting with a 1 are informational and those starting with a 2xx mean success. If the response code is starting with a 3xx are used for redirection, and 4xx and 5xx are used for errors (client and server respectively). For example: 200 stands for success, 403 for authentication required, and 404 for not found.

```
/* Retrieving headers. */
System.out.println(httpc.getResponseCode());
System.out.println(httpc.getRequestMethod());
for (int i = 0; httpc.getHeaderField(i) != null; i++) {
    System.out.println(httpc.getHeaderField(i));
}
```

Figure 14: Retrieving HTTP response headers.

After checking the response code, we can retrieve additional headers. In our case (using the App Engine as our web server), we have listed an example of receiving a succesful response (Figure 15). Again, a status header is sent, followed by the server type, the date, and some other info. Note that “Content-Length” can be really useful to determine our buffer size in the next step.

```
HTTP/1.0 200 OK
Server: Development/1.0
Date: Wed, 11 Mar 2009 11:25:05 GMT
Cache-Control: no-cache
Content-Type: image/gif
Content-Length: 18006
```

Figure 15: Example of HTTP response headers.

4.4.2 Cookies

A 'special' type of HTTP header that can be retrieved from sending an HTTP request are cookies. Just like all the other headers, it is retrieved with the code stated above. When printed to screen, it will look similar to the request shown in Figure 16. This code above is taken from retrieving the URL `http://www.google.nl/`. As you can see, this cookie consists out of multiple parts, but would normally be stored as one cookie in your browser. In this case, the cookies name is "PREF", and its content is "ID=8f3c26(...)zrigB6". Finally it comes with an expiration time and date, a path, and a domain. This information will be needed when we want to authenticate ourselves to the App Engine.

```
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Date: Wed, 11 Mar 2009 11:57:38 GMT
Expires: -1
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie:
  PREF=ID=8f3c262a9b38330e:TM=1236772658:LM=1236772658:S=L2ORt13rUlzrigB6;
  expires=Fri, 11-Mar-2011 11:57:38 GMT; path=/; domain=.google.nl
Server: gws
```

Figure 16: An HTTP response including cookies.

4.4.3 Message Body

Once we've received all headers and parsed them accordingly, we are able to get the actual body. As we can see from Figure 17, the most important thing is getting the stream, provided by the `getInputStream()` function. After we have done this, we can basically do anything with our InputStream. In this case we print it to the console, but we could also save it in a buffer and manipulate it, or write it do disk, etc. After we are done receiving the InputStream, we close the InputStream, effectively closing the connection.

```
BufferedReader in =
  new BufferedReader(new InputStreamReader(httpc.getInputStream()));
String inputLine;

while ((inputLine = in.readLine()) != null) {
    System.out.println(inputLine);
}
in.close();
```

Figure 17: Retrieving the HTTP response's body.

5 Server Implementation

Below we describe our (advert) server in more detail.

5.1 User Authentication

As described in the Server Design we will implement two servers, one without user authentication (i.e. a public server without any guarantees), and a server which requires authentication through Google Accounts.

5.1.1 Google Accounts

The most basic and obvious way to achieve authentication is single user authentication (through Google Accounts). Basically only the owner is allowed to use the advert service. According to the documentation provided by Google, there is no variable that indicates if a user is owner of the application. Nonetheless there is another function called `is_current_user_admin()`, which is available in the `google.appengine.api.users` package. By default the owner of the application is administrator, and additional administrators can be added through the administration panel. This fits our needs perfectly.

Once a client logged in through the Google login procedure (e.g. a Google login page), the request handler at the server returns a user object, as shown in Figure 18. This object can then be used to identify a user.

Other functions included in the `google.appengine.api.users` package are:

- `create_login_url(dest_url)`; which returns a URL that, when visited, will prompt the user to sign in using a Google account, then redirect the user back to the URL given as `dest_url`.
- `create_logout_url(dest_url)`; which returns a URL that, when visited, will sign the user out, then redirect the user back to the URL given as `dest_url`.

```
user   = users.get_current_user()

if not user:
    self.error(403)
    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write('Not Authenticated')
    return

if not users.is_current_user_admin():
    self.error(403)
    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write('No Administrator')
    return

...
```

Figure 18: Authenticating a User.

5.2 The Datastore

5.2.1 Datastore Layout

The Google App Engine datastore is not like a traditional relational database. Data objects, or "entities", have a kind and a set of properties. For our server we need a set of different data objects to store in our datastore.

(Advert) Objects Obviously, the main purpose of our server is to store binary (advert) objects. For this purpose we designed the following entity to represent our advert data (Figure 19). First of all we will store a path, which will be used for reference of a unique object (i.e. a key), which will be structured as a directory structure. Secondly, we will store the author's name, for legal issues. Next is the actual Advert object, which is stored as binary data. Finally we add a TTL as will be discussed below.

MetaData At our server, we would like to store meta data not as a binary object, but as readable strings of data. This way, we can search meta data when it is queried. According to the AdvertService API, a MetaData object should look like Figure 19. In this object, we just need to store key-value pairs, which are String properties. Every object will be linked to a parent (Advert object), as soon as it's created. Note that there is redundant information inside this object, being the `path` value. We did this to simplify our GQL queries as described in Section 5.3.4.

```
class Advert(db.Model):
  path   = db.StringProperty()
  author = db.UserProperty()
  ttl    = db.DateTimeProperty(auto_now_add=True)
  object = db.BlobProperty()

class MetaData(db.Model):
  path   = db.StringProperty()
  keystr = db.StringProperty()
  value  = db.StringProperty()
```

Figure 19: An Advert Object.

5.2.2 Transactions

Since the (Advert) object and the MetaData object are stored sequentially, it is essential that they are stored either both, or neither of them, to ensure consistency. Fortunately, the Google App Engine datastore provides a mechanism for ensuring both are stored or, if the datastore fails, nothing is stored. Using transactions, we can call a function and maintain atomicity. All functionality of the App Engine can be used inside a function that is called in transaction, except for GQL functions.

```
try:
  db.run_in_transaction(store, json)
except db.TransactionFailedError, message:
  logging.error(message)
  ...
```

Figure 20: Transactions.

A small example of how transactions work inside the App Engine is shown in Figure 20. In this example we encapsulated the transaction function `store`, with arguments `json`, into a `try`, `except` statement. If the transaction fails, the operation's effects are not applied, and the datastore API raises an exception, which in turn is caught by the `except` statement and an error message is attached. The transction could fail due to a high rate of contention, with too many users trying to modify an entity at the same time. Or an operation may fail due to the application reaching a quota limit. Or there may be an internal error with the datastore.

Note that to ensure consistency of the datastore, we will also use transactions to remove objects from the datstore. Again this is done sequentially. If one delete function would fail, we would end op with an inconsistent copy of the datastore. Hence we will again use transactions for atomicity and thus consistency.

5.2.3 Path Encoding

To have the AdvertService work properly, we need some path encoding to save Advert objects in the datastore. The Google App Engine itself provides one solution, because every entity in the datastore has a key. When the application creates an entity, it can assign another entity as the parent of the new entity. Assigning a parent to a new entity puts the new entity in the same entity group as the parent entity. Every entity belongs to an entity group, a set of one or more entities that can be manipulated in a single transaction. An entity without a parent is a root entity. An entity that is a parent for another entity can also have a parent. A chain of parent entities from an entity up to the root is the path for the entity, and members of the path are the entity's ancestors. The parent of an entity is defined when the entity is created, and cannot be changed later. The downside of this solution is that we have to create empty MetaData objects for path nodes that are empty.

Another option is to save our own paths, using a namespace that identifies a path, something like the UNIX system does (e.g. `/home/bboterm/.`). The advantage of this model is that it works exactly analogue to the JavaGAT AdvertService. The disadvantage of this is that we have to parse our own pathnames, and keep track of them by means of a non-existing mechanism (i.e. something that we need to build ourselves). Since this is not a major obstacle, we will stick with the second approach.

5.3 Public Server Functions

Below we will list some of the major functions the advert server will offer to the client. These functions will be used by, for example, the JavaGAT AdvertService adaptor for the App Engine.

5.3.1 Logging In

Since we are using ClientLogin service (as described above) to authenticate ourselves to the App Engine, we won't need a separate login page. Sending our credentials (session IDs) to the Appe Engine should be enough to authenticate ourselves. For every request, our server will use a small section of code, which looks like the code segment stated in Figure 21, to authenticate the user.

5.3.2 Receiving and Storing Binary Data

An important function of the Advert Service is to accept and store binary data.

Binary Data Only For accepting just binary data, we can receive the entire message body and write it to a variable, which is stored in the datastore accordingly. A sample code of this function is shown in the code segment of Figure 22. The first line identifies the class, which is called by the RequestHandler (depending on the given URL). The second line says that we are expecting a POST request. Then, a new `Bin()` data model is created (which in our case only contains a field `data = db.BlobProperty()`).

```
class AnyClass(webapp.RequestHandler):
  def get(self):
    user = users.get_current_user()

    if user and users.is_current_user_admin():
      self.response.headers['Content-Type'] = 'text/plain'
      self.response.out.write('Hello, ' + user.nickname())
    else:
      self.response.http_status_message(401)
      self.response.out.write('Some error.')
```

Figure 21: A MetaData Object.

Then we fetch the body and put in a temporary variable, before we store it as a `db.BlobProperty()`in the database by calling `bin.put()`.

```
class Download(webapp.RequestHandler):
  def post(self):
    bin = Bin()
    uploaded_file = self.request.body
    bin.data = db.Blob(uploaded_file)
    bin.put()
    self.redirect('/')
```

Figure 22: Accepting Binary Data.

Combination of Binary Data and Strings Although meta data will be stored in a different class, we won't apply a different function to store MetaData objects, for the simple reason that if one of these transfers would fail, it could lead to inconsistencies (see Section 4.2.3).

When receiving our binary object as a combination of both binary data and unicode Strings, it will be received as one stream of bytes (assuming we're not using the multi-part/formdata method of Section 4.2.3. This means that we will have to dismember the message ourselves, which is not as straightforward as it seems. The Google App Engine currently only supports Python 2.5, which does not support bytes or byte arrays for data manipulation.

We still managed to extract data using string manipulation. Essentially when the App Engine receives the entire body as shown in Figure 22, it is stored as a raw string (no encoding) before it is stored as BLOB in the datastore. We made use of that raw string as shown in Figure 23. In this example, we receive the message body and store it into the raw string called 'bytes'. After this we read the first byte (`bytes[0:4]`), we use ord() to make it an integer, knowing the length of the data sent. After that we send our Content-Type headers and write the rest of the body to the standard output.

```
bytes = self.request.body
length = ord(bytes[0:4])
self.response.headers['Content-Type'] = "image/gif"
self.response.out.write(bytes[5:length])
```

Figure 23: Manipulating a Raw String.

Simplejson Instead of using our own representation of unicode Strings and binary data, we could also use a JSON representation. Once a serial JSON array or object has been sent (see Section 4.2.3), we are able to load it into a server-side JSON object and decode it according to Figure 24. The `simplejson.loads()` function decodes a serial JSON String representation into a JSON array, after which we can access its array entries like a regular array.

```
body = self.request.body
json = simplejson.loads(body)
...
advert.path   = json[0] #extract path from message
advert.object = json[2] #extract (base64) object from message
...
for k in json[1].keys():
  ...
  metadata.keystr = k
  metadata.value  = json[1][k]
...
```

Figure 24: Decoding a JSON object.

The same goes for a JSON object. We can construct a for loop like shown in Figure 24, by iterating through all the keys and retrieving their key-value pairs.

For reasons mentioned in Section 4.2.3, the object is not send in binary, but in unicode String format. For this reason, we can't store it as a `db.BlobProperty()`, since this property expects unencoded binary data only. The only option is to use the `db.TextProperty()`, since it is not limited to 500 characters, like the `db.StringProperty()` is.

Note that storing a JSON object is done in an atomic transaction, as described in section 5.2.2.

5.3.3 Storing Meta Data

Since MetaData needs to be searchable at the server-side, we will have to extract the key-value pairs from the POST request and store them in the datastore accordingly. This will be done in the opposite way of sending them at the Client side (see Section 4.2.3).

As of yet, the Google App Engine does not support Tuples, or key-value pairs as a data type. Therefore we created our own MetaData class as described above. Also, this MetaData cannot be stored into a list, because lists only supports primary data types like integer and string. For that matter, we add an ID to the MetaData, which is a child of the binary object stored in the datastore.

Another option would be to maintain two lists of strings, where the indexes of the lists link the key and the value of the MetaData. However, we are not sure of those lists maintain order (which could mess up the MetaData).

Finally, we could also append all data in two strings and have them separated by special delimiter. Downside of this method is that we could run out of the maximum length of strings (which is 500 bytes).

Once data has been stored successfully, we send an HTTP 201 (Created) status code, and return the TTL in the message body. If an entry already exists at the specified path, that entry gets overwritten, and a warning is issued. This is done by sending an HTTP 205 (Reset Content) Status Code and a warning in the message body.

5.3.4 Finding MetaData

When the MetaData has been stored in the datastore, we should be able to process find() requests by clients. First a MetaData object is received, which is then tried to be matched with MetaData already present in the datastore. For this purpose we will use the GQL as described above.

To process the find() request, we have to check all the key-value pairs given by the client and compare them to the key-value pairs of all MetaData objects in the datastore. Since all key-value pairs are stored in one 'table', we have to group them by path, after which we can check all key-value pairs per MetaData object.

Comparing the MetaData object sent by the user with the MetaData present in the datastore can be achieved in two ways. One approach is to make a selection of all paths available and check the key-value pairs on a per path basis. This is done by the GQL query shown in Figure 25.

To obtain an array of all paths (without duplicates), we would usually use the 'DISTINCT' keyword from SQL. Since the GQL does not support the 'DISTINCT' keyword we have to make our own unique array of keys. We do this by using the `Set()` data type, which maintains a unique set of paths (duplicates are automatically overwritten).

After we have a distinct set of all paths, we iterate through the list and match every key-value pair with the key-value pair given by the user. If one pair does not match, we remove the path from the list, and start over inspecting the next path in the list. Eventually we will have a list of paths which contains all paths that meet all

```
query = db.GqlQuery("SELECT * FROM MetaData")

paths = Set()

for bin in query:
  paths.add(bin.path)

paths  = list(paths)
self.response.out.write(paths)

for path in paths[:]:
  for k in json.keys():
    query = db.GqlQuery("SELECT * FROM MetaData WHERE path = :1 AND keystr = :2
                        AND val = :3", path, k, json[k])
    if query.count() < 1:
      paths.remove(path)
      break

if len(paths) < 1:
  self.error(404)
  self.response.headers['Content-Type'] = 'text/plain'
  self.response.out.write('Not Found')
  return

self.response.out.write(simplejson.dumps(paths))
```

Figure 25: Functionality of `find()` Function.

requirements given by the user.

The return value of all metadata found is a String array, which again needs to be formatted in such a way that it can be sent as one String. Obviously we will use simplejson to do so. If no entry is found, we will send a 404 response code, accompanied by the text 'Not Found' in the message body.

A second option to implement the `find()` function would be to first find all paths that meet the requirements given by the first key-value pair. Then do the same for the second key-value pair and accordingly take the intersection between the two search results. This process should be repeated until all key-value pairs have been checked. The downside of this approach is that there is no obvious method to perform the intersection between two search results at the Google App Engine. This approach requires a *for-loop* to iterate through all items in the first search result and check them with the second.

5.3.5 Returning an Object From the Datastore

This function will send binary data back to client. Basically, this function only sends an Object to the standard output (the HTTP connection), which looks like Figure 26.

Of Course, we first need to fetch the requested object from the datastore, which will be done using GQL Queries.

```
self.response.headers['Content-Type'] = "application/octet-stream"
self.response.out.write(bin.data)
```

Figure 26: HTTP Response with Binary Data.

5.3.6 Removing an Object From the Datastore

To remove an (Advert) object from the datastore, a user calls the `delete()` function, with a pathname as argument. Obviously, the most straightforward way to remove an item from the datastore is to query all objects and meta data that match the pathname given. Naturally, we want to perform the removal of the object and associated meta data in transaction, because we do not want inconsistencies in our datastore (i.e. querying meta data of which the associated object does not exist anymore).

The only problem we have now is that we cannot perform queries inside a transaction, since the Google App Engine does not allow us to do so. The solution that Google provides is either to work with keys (which is not feasible in our case), or prepare your queries before performing the transaction. We adapted this solution by sending our query results to a `remove()` function, which iterates through the query results and deletes all objects and meta data given to the function.

5.4 Private Server Functions

Below we will describe some of the private server functions provided for the advert service.

5.4.1 Garbage Collector

As described above, we will use some form of TTL to determine whether a data item stored is still needed in the datastorage. When a data item is expired, it will be removed automatically using a mechanism called the garbage collector. The source code of our garbage collector can be found in Figure 27.

By calling `gc()`, we activate our garbage collector, which, in turn, executes a GQL query, selecting all data that has been added to the datastore earlier than ten days ago. To achieve this we use the `timedelta()` object, provided by the `datetime` class. A timedelta object represents a duration, the difference between two dates or times, and is ideal for our garbage collector.

The query results in an iterable list of advert objects that can be removed from the datastore. We should not forget that also all the MetaData associated with the Advert object should be removed alongside the object itself. Therefore we wrote a function

```
def gc(): #garbage collector
  query = db.GqlQuery("SELECT * FROM Advert WHERE ttl < :1",
                      datetime.datetime.today() + datetime.timedelta(days=-10))

  for advert in query: #all entities that can be deleted
    advert.delmd()     #delete all associated metadata
    advert.delete()    #delete the object itself
```

Figure 27: The Garbage Collector.

inside the advert class, called `delmd()`, that automatically deletes all its metadata. Once this is done, the object itself is deleted and the garbage collection process is finished.

A Source/API

A.1 Advert.java

```
public class Advert {

        private Communications comm = null;

        public Advert(String server, String user, String passwd)
          throws AuthenticationException, IOException {
                comm = new Communications(server, user, passwd);
        }

        /**
         * Add a {@link byte}[] to the App Engine, at an absolute path, with
         * {@link MetaData} included, to the datastore. If an entry exists at the
         * specified path, that entry gets overwritten, and a warning is issued.
         *
         * @param bytes
         *            {@link byte}[] to be stored.
         * @param metaData
         *            {@link MetaData} to be associated with the passed bytes.
         * @param path
         *            Absolute path of the new entry.
         * @throws AppEngineResourcesException
         *             This exception is thrown when the App Engine runs out of
         *             resources.
         */
        public void add(byte[] object, MetaData metaData, String path)
          throws MalformedURLException, IOException, AuthenticationException,
          AppEngineResourcesException, NoSuchElementException,
          RequestTooLargeException {
                JSONArray  jsonarr = new JSONArray();
                JSONObject jsonobj = new JSONObject();

                Iterator<String> itr  = metaData.getAllKeys().iterator();

                while (itr.hasNext()) {
                        String key   = itr.next();
                        String value = metaData.get(key);

                        if (key == null) {
                                continue; //key can't be null (value can)
                        }

                        jsonobj.put(key, value);
                }

                String base64 = new sun.misc.BASE64Encoder().encode(object);

                jsonarr.add(path);
                jsonarr.add(jsonobj);
                jsonarr.add(base64);

                comm.httpSend("/add", jsonarr.toString());

        /*
         * TODO: return TTL of data stored at server (optional)?
         */
        }


        /**
         * Remove an instance and related {@link MetaData} from the datastore at an
         * absolute path.
         *
         * @param path
         *            Path is an absolute entry to be deleted.
         * @throws NoSuchElementException
         *             The path is incorrect.
         */
        public void delete(String path)
          throws MalformedURLException, IOException, AuthenticationException,
          AppEngineResourcesException, NoSuchElementException,
          RequestTooLargeException {
                comm.httpSend("/del", path);
        }

        /**
         * Gets an instance from the datastore at a given (absolute) path.
         *
         * @param path
         *            Absolute path of the entry.
         * @return The instance at the given path.
         * @throws NoSuchElementException
         *             The path is incorrect.
         */
        public byte[] get(String path)
          throws MalformedURLException, IOException, AuthenticationException,
          AppEngineResourcesException, NoSuchElementException,
          RequestTooLargeException {
                String base64 = comm.httpSend("/get", path);

                return new sun.misc.BASE64Decoder().decodeBuffer(base64);
        }

        /**
         * Gets the {@link MetaData} of an instance from the given (absolute) path.
         *
         * @param path
         *            Absolute path of the entry.
         * @return A {@link MetaData} object containing the meta data.
         * @throws NoSuchElementException
         *             The path is incorrect.
         */
        public MetaData getMetaData(String path)
          throws MalformedURLException, IOException, AuthenticationException,
          AppEngineResourcesException, NoSuchElementException,
          RequestTooLargeException {
                MetaData   metadata = new MetaData();

                String result = comm.httpSend("/getmd", path);

                JSONObject jsonobj = JSONObject.fromObject(result);
                Iterator<?> itr    = jsonobj.keys();

                while (itr.hasNext()) {
                        String key   = (String)itr.next();
                        String value = (String)jsonobj.get(key);

                        if (key == null) {
                                continue; //key can't be null (value can)
                        }

                        metadata.put(key,value);
                }

                return metadata;
        }

        /**
         * Query the App Engine for entries matching the specified set of
         * {@link MetaData}.
         *
         * @param metaData
         *            {@link MetaData} describing the entries to be searched for.
         *            No wildcards allowed.
         * @param pwd
         *            Current working path.
         * @return a {@link String}[] of absolute paths, each pointing to a
         *         matching entry. If no matches are found, null is returned.
         */
        public String[] find(MetaData metaData, String pwd)
          throws MalformedURLException, IOException, AuthenticationException,
          AppEngineResourcesException, NoSuchElementException,
          RequestTooLargeException {
                JSONObject metadata = new JSONObject();
                JSONArray  jsonarr  = new JSONArray();

                Iterator<String> itr  = metaData.getAllKeys().iterator();

                while (itr.hasNext()) {
                        String key   = itr.next();
                        String value = metaData.get(key);

                        if (key == null) {
                                continue; //key can't be null (value can)
                        }

                        metadata.put(key, value);
                }

                String result = comm.httpSend("/find", metadata.toString());

                jsonarr = JSONArray.fromObject(result);

                return (String[]) jsonarr.toArray();
        }
}
```

A.2 Communications.java

```
class Communications {

        private static final int MAX_REQ_SIZE = 10000000;
        private static final int MAX_DB_SIZE  = 1000000;

        private String cookie;
        private String server;

        Communications(String server, String user, String passwd)
          throws MalformedURLException, ProtocolException, IOException,
          AuthenticationException {
                this.server = server;
                authenticate(user, passwd);
        }

        private static final String CLIENTLOGIN =
                "https://www.google.com/accounts/ClientLogin";

        /**
         * Function to set up SSL in the system properties.
         */
        private static void setupSsl() {

        }

        /**
         * Function to authenticate to the Google App Engine
         * @param server
         *                                 Server to connect to.
         * @param user
         *                                 User's email address for identification.
         * @param passwd
         *                                 User's password for identification.
         * @return a {@link String} which contains a cookie with a session ID.
         * @throws Exception
         *                                 Failed to authenticate to the App Engine.
         */
        void authenticate(String user, String passwd)
          throws MalformedURLException, ProtocolException, IOException,
          AuthenticationException {

        }

        /**
         * Function to send an object over HTTP.
         * @param server
         *                                 Server to send the object to.
         * @param cookie
         *                                 Cookie for identification.
         * @param object
         *                                 Object to be send to the server.
         * @return
         *                                 Returns the response body in {@link String} format.
         * @throws Exception
         *                                 Failed to send object to server.
         */
        String httpSend(String ext, String payload)
          throws MalformedURLException, IOException, AuthenticationException,
          AppEngineResourcesException, NoSuchElementException,
          RequestTooLargeException {

        }
}
```

A.3 ibis-advert.py

```
import cgi

from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext.webapp.util import run_wsgi_app
from google.appengine.ext import db
from django.utils import simplejson
```

```

class Advert(db.Model):
  path   = db.StringProperty()
  author = db.UserProperty()
  ttl    = db.DateTimeProperty(auto_now_add=True)
  object = db.TextProperty() #base64

  def delmd(self): #delete all metadata of some object
    query = db.GqlQuery("SELECT * FROM MetaData WHERE path = :1", self.path)

    for md in query:
      md.delete()

class MetaData(db.Model):
  path   = db.StringProperty()
  keystr = db.StringProperty()
  value  = db.StringProperty()

def auth(self): #authentication
  if not users.get_current_user():
    self.error(403)
    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write('Not Authenticated')
    return -1

  if not users.is_current_user_admin():
    self.error(403)
    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write('No Administrator')
    return -1

  return 0

def gc(): #garbage collector
  query = db.GqlQuery("SELECT * FROM Advert WHERE ttl < :1", datetime.datetime.today() + datetim

  for advert in query: #all entities that can be deleted
    advert.delmd()     #delete all associated metadata
    advert.delete()    #delete the object itself

class MainPage(weba0pp.RequestHandler):
  def get(self):
    self.redirect(users.create_login_url(self.request.uri))

class AddObject(webapp.RequestHandler):
  def post(self):
    advert = Advert()
    user   = users.get_current_user()

    if auth(self) < 0: return

    body = self.request.body
    json = simplejson.loads(body)

    query = db.GqlQuery("SELECT * FROM Advert WHERE path = :1", json[0])
    if query.count() > 0: #this entry already exists; overwrite
      query.delmd()  #delete all associated metadata
      query.delete() #delete the object itself
      self.response.http_status_message(205) #reset content

    advert.path   = json[0] #extract path from message
    advert.author = user    #store author
    advert.object = json[2] #extract (base64) object from message

    advert.put() #store object in database

    for k in json[1].keys():
      metadata        = MetaData(parent=advert)
      metadata.path   = json[0]
      metadata.keystr = k
      metadata.value  = json[1][k]
      metadata.put()

    self.response.http_status_message(201) #Created
    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write('Expires: %s', datetime.datetime.today() + datetime.timedelta(days=1
    return

class DelObject(webapp.RequestHandler):
  def post(self):
    if auth(self) < 0: return

    body  = self.request.body
    query = db.GqlQuery("SELECT * FROM Advert WHERE path = :1", body)

    if query.count() < 1: #no matching object found
      self.error(404)
      self.response.headers['Content-Type'] = 'text/plain'
      self.response.out.write('No Such Element')
      return

    for advert in query:
      advert.delmd()  #delete all associated metadata
      advert.delete() #deleting the first entry we find
      break #and stop

    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write('OK')
```

```

class GetObject(webapp.RequestHandler):
  def post(self):
    if auth(self) < 0: return

    body  = self.request.body
    query = db.GqlQuery("SELECT * FROM Advert WHERE path = :1", body)

    if query.count() < 1: #no matching object found
      self.error(404)
      self.response.headers['Content-Type'] = 'text/plain'
      self.response.out.write('No Such Element')
      return

    for advert in query:
      self.response.headers['Content-Type'] = 'text/plain'
      self.response.out.write(advert.object) #returning the first entry we find
      break #and stop

    return;

class GetMetaData(webapp.RequestHandler):
  def post(self):
    if auth(self) < 0: return

    body  = self.request.body
    query = db.GqlQuery("SELECT * FROM MetaData WHERE path = :1", body)

    if query.count() < 1: #no matching object found
      self.error(404)
      self.response.headers['Content-Type'] = 'text/plain'
      self.response.out.write('No Such Element')
      return

    jsonObject = {}

    for metadata in query:
      jsonObject[metadata.keystr] = metadata.value

    self.response.headers['Content-Type'] = 'text/plain'
    self.response.out.write(simplejson.dumps(jsonObject))

class FindMetaData(webapp.RequestHandler):
  def post(self):
    if auth(self) < 0: return

    body = self.request.body
    json = simplejson.loads(body)

    query = db.GqlQuery("SELECT * FROM MetaData")

    paths = Set()

    for bin in query:
      paths.add(bin.path)

    paths  = list(paths)
    self.response.out.write(paths)

    for path in paths[:]:
      for k in json.keys():
        query = db.GqlQuery("SELECT * FROM MetaData WHERE path = :1 AND keystr = :2 AND val = :3
        if query.count() < 1:
          paths.remove(path)
          break

    if len(paths) < 1:
      self.error(404)
      self.response.headers['Content-Type'] = 'text/plain'
      self.response.out.write('Not Found')
      return

    self.response.out.write(simplejson.dumps(paths))

application = webapp.WSGIApplication(
                                     [('/',      MainPage),
                                      ('/add',   AddObject),
                                      ('/del',   DelObject),
                                      ('/get',   GetObject),
                                      ('/getmd', GetMetaData),
                                      ('/find',  FindMetaData)],
                                     debug=True)

def main():
  run_wsgi_app(application)

if __name__ == "__main__":
  main()
```

B Feature/ToDo list

- Splitting up requests larger than 10Meg

C Version History

Note that added/revised text to the previous version is dashed underlined.

v0.8 – 15 April 2009

- Added appendix ‘Feature/ToDo list’
- Added subsection ‘Runtime Environment’
- Added subsection ‘Transactions’

v0.7 – 6 April 2009

- Added subsection ‘Removing an Object From the Datastore’
- Extended subsection ‘Finding MetaData’
- Extended subsection ‘Garbage Collector’
- Extended subsection ‘Storing Meta Data’
- Removed subsection ‘Data Matching’
- Updated appendix ‘Source/API’

v0.6 – 30 March 2009

- Added/Edited code in ‘API’
- Edited subsection ‘User Authentication’
- Extended subsection ‘ClientLogin’
- Extended subsection ‘Receiving (Binary) Objects’
- Extended subsection ‘Sending a Combination of Both’
- Migrated document to LaTeX.
- Moved subsection ‘Garbage Collection’ to ‘Server Design’

v0.5 – 19 March 2009

- Added subsection ‘Sending a Combination of Both Using Our Custom Protocol’
- Extended class ‘Advert.java’
- Extended subsection ‘ClientLogin’
- Extended subsection ‘Finding MetaData’

- Extended subsection 'Receiving (Binary) Objects'
- Removed subsection 'Receiving Meta Data'
- Rewrote subsection 'Garbage Collection'
- Rewrote subsection 'Quotas'

v0.4 – 12 March 2009

- Added chapter 'Server Implementation'
- Added subsection 'HTTP Cookies'
- Added subsection 'HTTP(S) Login'
- Added subsection 'HTTP Response'
- Added subsection 'Sending a Combination of Both'
- Extended subsection 'Authentication and Privacy'
- Extended subsection 'Quotas'

v0.3 – 13 February 2009

- Added chapter 'Client Implementation'
- Changed and extended 'Appendix A: API'
- Extended subsection 'GQL Queries'
- Extended subsection 'HTTP Requests'
- Extended subsection 'Quotas'

v0.2 – 29 January 2009

- Added 'Appendix A: Source Code'
- Added 'Appendix B: Version History'
- Added section 'Security'
- Added subsection 'Authentication through Google Accounts'
- Added subsection 'Datastore Functions'
- Added subsection 'MetaData'
- Added subsection 'Monitoring Quotas'

- Changed subsection ‘Quotas’
- Extended section ‘Authentication’
- Extended subsection ‘Functions to Implement’
- Extended subsection ‘HTTP Requests’
- Removed some ‘?’
- Rewrote section ‘Server Design’

v0.1 – 23 January 2009

- Added chapter Server Design
  - Authentication
  - Client Functions
  - Datastore Layout
  - Limitations
  - Server Structure

References

[1] Vrije Universiteit Amsterdam. Ibis webpage, 2009.
`http://www.cs.vu.nl/ibis/`.

[2] Vrije Universiteit Amsterdam. JavaGAT Javadoc, 2009.
`http://www.cs.vu.nl/ibis/javadoc/javagat/index.html`.

[3] Vrije Universiteit Amsterdam. JavaGAT webpage, 2009.
`http://www.cs.vu.nl/ibis/javagat.html`.

[4] Douglas Crockford. JSON, 2009.
`http://www.json.org/`.

[5] Python Software Foundation. Python Programming Language - Official website, 2009.
`http://www.python.org/`.

[6] Google. App Engine Quotas, March 2009.
`http://code.google.com/appengine/docs/quotas.html`.

[7] Google. Authentication for Installed Applications, 2009.
`http://code.google.com/apis/accounts/docs/AuthForInstalledApps.html`.

[8] Google. Google App Engine, 2009.
`http://appengine.google.com/`.

[9] Google. Google Apps, 2009.
`http://www.google.com/apps/`.

[10] Google. Product APIs, 2009.
`http://code.google.com/more/#products-apis-ajaxsearch`.

[11] Bob Ippolito. simplejson, 2009.
`http://code.google.com/p/simplejson/`.

[12] Paul McDonald. Introducing Google App Engine, April 2008.
`http://googleappengine.blogspot.com/2008/04/introducing-google-app-engine-our-new.html`.

[13] W3C. SOAP Specification, 2009.
`http://www.w3.org/TR/soap/`.