ION API for Electronic Signage Final Report

Scott Hazlett

University of British Columbia

EECE 490L

April 08, 2014

LETTER OF TRANSMITTAL

Scott Hazlett
UBC Computer Engineering
2356 Main Mall
Vancouver, BC. V6T 1Z4

April 9th, 2014

Vincent Wong, Technical Supervisor
Department of Electrical and Computer Engineering
2332 Main Mall
Vancouver, BC. V6T 1Z4

Dear Professor Dr. Vincent Wong,

Enclosed is the project report titled ION API for Electronic Signage that was commissioned in January 2014 to integrate the UBC ION metering data with UBC electronic signage.

This report discusses the implementation of ION API as a web-based interface using HTTP queries. This report is restricted to considering meter readings of only electricity consumption. As well, this report assumes usage patterns that are not exposed to the public.

The main findings on the report are:

1. The ION API serves as a protective layer to the ION database by preventing possible malicious access and by lowering the load on the database by hosting cached results
2. The ION API lays the groundwork for developing upstream integration with other platforms for consuming data as well as for developing downstream integration with other producers of data

I would like to thank Wilson Lo, Senior Programmer Analyst for Communication and Collaboration Technologies, for being instrumental in forming key design decisions and providing valuable technical assistance. I would also like to thank Jamil Rhajiak, UBC Communications Services Coordinator, who has met regularly with myself to help assist with the integration of this project's data with UBC signage.

Finally, I would like to sincerely thank you for having the opportunity to work on this project.

Scott Hazlett
Computer Engineering student

Encl.

University of British Columbia

Electrical and Computer Engineering 490L

ION API for Electronic Signage
Final Report

Scott Hazlett

With special mention
Wilson Lo
Jamil Rhajiak

8th April 2014

ABSTRACT

This report investigates a practical approach to integrating a database with UBC's Enterprise Cool Sign infrastructure that powers all campus electronic signs. UBC's ION database collects real-time electricity usage data for all buildings on campus but the data is relatively inaccessible. This project designed and deployed an ION database API that serves to channel this real-time data to electronic signs across both university campuses. The ION API is underpinned by an Apache webserver running Perl CGI scripts that service HTTP GET requests with URL parameters. The ION API was designed to reduce load on the ION database by caching previous search results. Furthermore, the ION API attempts to present the raw data in a graphical way that is appealing to passerby's so as to affect societal change. Due to ongoing restructuring in the ION database, this project worked with static file of exported data from the ION database. Care was taken to read the database row by row to closely mimic the live database. The ION API was a success deployment allowing for two forms of operations: comparing electricity consumption for one building for the 24 hour period preceding, and another providing inter-building simple comparisons.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

GLOSSARY

API: *Application Programming Interface* is the published abilities of a used program to its users.

CGI: *Common Gateway Interface* is a way for users to connect to a URL and do more than HTTP is designed to do by launching an environment, say PHP or Perl, which can then run a script as if the script was launched locally on the machine.

CSV: *Comma-Separated Values* file stores data structured along rows by separating them with a carriage-return character and along columns by separating them with the comma character. CSV files popular because they are easy for programs to process.

kWh: *kiloWatt* hour is a measurement of electrical work and represents the amount of electricity needed to power a 100 Watt bulb for 10 hours.

SIS name: SIS names can be looked up under the details tab of a building in Wayfinding UBC and uniquely identify a given UBC buildings. Most UBC buildings have an SIS name although the University Services Building is a notable exception.

XML: *eXtensible hypertext Markup Language* is a popular text-based markup language that allows for simple transmission of structured data. One popular example of XML usage is an RSS feed.

1. INTRODUCTION

This project investigates an optimal approach to delivering meter data to UBC Signage. The ION API for Electronic Signage project proposes to adapt data collected about campus electricity usage and present it on campus digital signs in a form that is consumable within the 6-second attention span of passing people.

Visitors to UBC as well as the local community invariably pass by many of these signs in any given day. I feel that this is a good opportunity to contribute towards lasting campus behavioural change through awareness of resource consumption.

This project works with existing UBC signage infrastructure to allow streaming of consumption data to any sign. As well, this work lays the groundwork for downstream integration with personal mobile devices as well as upstream integration with additional sources of data.

This report first provides background on the existing UBC infrastructure for electronic signage and the Cool Sign Enterprise software that powers the signs. Then this report transitions into the deployment of ION API and the primary design decisions. Further along, this report reviews some implementation highlights including providing for multiple behaviour based on templates and soft. Finally, this report concludes with the possibilities of upstreaming and downstreaming as well as some achievements and concerns.

2. EQUIPMENT AND METHODOLOGY

The following subsections will cover existing infrastructure as well as instrumental personnel.

Personnel

Working with *Wilson Lo*, Senior Programmer Analyst for Communication and Collaboration Technologies, and *Jamil Rhajiak*, UBC Communications Services Coordinator, we have deployed dynamic content to display nodes using the Cool Sign infrastructure. Because a primary goal of this project was to integrate well with existing infrastructure, we did not consider alternate signage software.

Cool Sign Version 4 to 5

However, UBC is currently undergoing an upgrade from Cool Sign version 4 to version 5. Most notably, Cool Sign version 5 will now offer operators the option to display a website natively on signs as an alternative to manually creating content as was necessary with version 4. This is a big incentive for operators to shift towards dynamic content on a website. A unified front can be easily displayed on multiple platforms, for example, mobile devices and web browsers.

UBC Signage enjoys a wide audience and so its security is accordingly well-guarded. Thus, it was a natural choice to deploy the ION API on a Virtual Server Service provided by UBC IT Services. This keeps the entire system protected behind the same virtual network. The system is specifications are 1 CPU, 2 GB RAM, 32 GB Tier 1 System Disk, RHEL 6 OS. This accrues an annual operating cost of about C$125 per year from the ECE department.

Cool Sign and Data Watcher

Cool Sign in an enterprise software that UBC licenses to display contents on electronic signs across both campuses. Figure 1 shows how the Cool Sign Data Watcher periodically queries sources of information, in this case the ION API, used for broadcasting dynamic content to UBC electronic signs. The Data Watcher is configured by a Data Watcher configuration file which is very flexible and suits this project:

1. We can configure the Data Watcher to connect to any URL
2. We can configure the Data Watcher to handle an XML-formatted result
3. We can configure the periodicity of updates

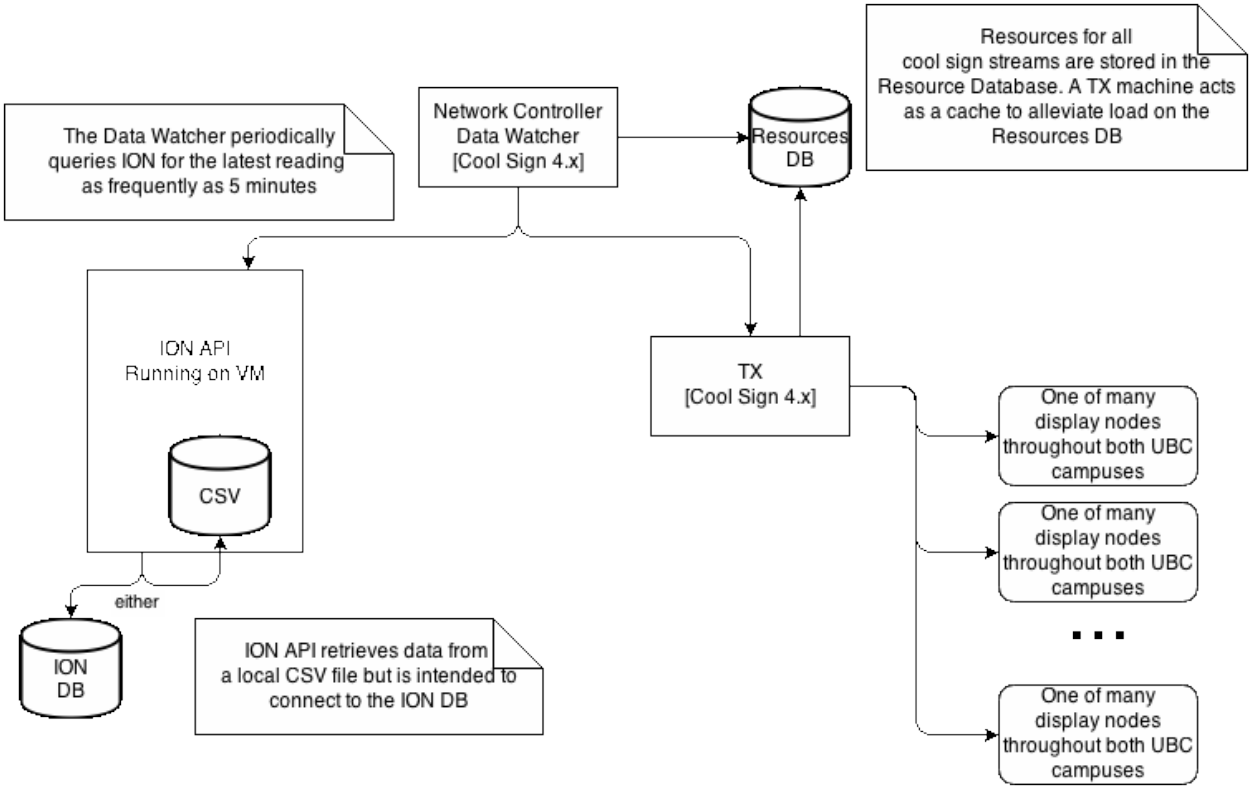

Figure 1: System Diagram of Integration of ION API with UBC Signage Infrastructure

We have created a dedicated Data Watcher that queries the ION API via HTTP and receives XML-formatted data on the electricity consumption of a building. All results can then be automatically streamed to any UBC display node through the Cool Sign infrastructure.

URL Parameters and XML Results

The Cool Sign Data Watcher can query in different ways. A common way is to query the data source using an HTTP GET request to a URL with parameters. Similarly, Cool Sign can receive data in various formats. We chose XML-formatted results because there are many other nodes already configured in this fashion, and because XML-formatted data is easily processed by most applications.

The following shows the XML structure that we expect after querying the ION API. The URL for the ION database is not currently registered with the DNS server and so it is directly accessed by IP address.

```
Content-type: text/html
<item>
    <Block>
        <BldName>USB</BldName>
        <Consump24>2026</Consump24>
        <Date>07/04/2013</Date>
    </Block>
</item>
```

Here, the XML entries have been indented for clarity but should actually be on one long line to not break compatibility with the Data Wacher. This XML reply states that the University Services Building “USB” consumed 2026 kWh in the past calendar day.

Handling the Present ION Database

The ION database is currently undergoing a migration in naming and reordering to better reflect the current deployment of meters with physical buildings. Each building may have smaller buildings for which the same department is responsible for. Thus, many buildings have multiple meters. For example, MacLeod is responsible for Rusty Hut.

This project has focused only on kWh readings of energy consumption as it is more widely deployed. As well, UBC is in the process of migrating completely from steam to hot water and so there is flux with those meters.

The ION database can export data for a given period, for example one year and multiple buildings, into CSV format. This project has been using such a file for development. It contains the readings for the calendar year of 2013 across 14 meters which aggregate to 9 buildings. Each reading is taken at an interval of 15 minutes. We extracted the kWh columns for each meter to shorten the file size.

3. ION API DESIGN DECISIONS

The following sub headings will discuss the design and implementation of the ION API.

Model-Controller Design Pattern

This project used a Model-Controller design pattern to separate decision making from the database access. `IonController.pl` is the URL-accepting interface and only performs basic sanity checks and sanitization on the URL parameters. It then dispatches the work to `IonModel.pm`.

`IonController` is a Perl script that can be called directly and has a `.pl` extension, meanwhile `IonModel` is a Perl module that contains a constructor and member methods and has a `.pm` extension. The former is analogous to a main method in C or Java, while the latter is analogous to a software library.

Reactive API with Apache and CGI

The ION API is a reactive host in that it does not store any state other than previously cached results. This is easily implemented with the Common Gateway Interface "CGI" which allows for URL queries to run local scripts in various languages, for example, PHP, Perl, and Python. This project choose Perl as it performs well with per-line file processing. In conjunction with this, we chose Apache as the webserver because it is widely deployed and easy to setup.

Perl XML Library

This project uses the Perl Module *XML::Generator* to generate XML-formatted replies. We hope to reduce the number of programmer errors by re-using proven software libraries.

Local Caching and Calculations

An important design decision was to have all the calculations be performed on the ION API host so that no unnecessary load is placed on the Cool Sign DataWatcher. Similarly, The ION API will cache past results to serve requests faster and to alleviate the burden on the ION database.

4. ION API IMPLEMENTATION

The following subsections cover interesting implementation details.

Generating the Cached Result

The Perl code illustrating the core functionality for a query is shown below. This subroutine will generate the cache file for a query. If it does not already exist, it will query the ION Database to generate the cache file. Otherwise, it does not contact the ION Database.

```
sub execute() {
  my $class = shift;
  # This will generate the regex used for the starting
  # row (smallest reading) to the end row (largest reading)
  # from which we will take a delta
  $class->generate_regex_for_time_bounds();

  # This will perform the mapping from SIS name
  $class->lookup_ion_name_with_sis_name();

  # This will assign $class->{'_cache_filename'}
  $class->generate_cache_filename();

  # This will create the cached result if it does not
  # already exist.  If it exists, no need to create it.
  # Otherwise, it reads this from the
  # ION database line-by-line and saves the result
  # to the designated cache file
  $class->generate_cache_file();
```

```
  # Now the client can call get_reply_string();
}
```

First a regular expression is formed for the starting and ending timestamps that we are interested in. Next, we look up the relevant meters based upon the SIS name for the UBC building.

With this information, we can now generate the deterministic name of the cache file that has this information. If the file is not found, it will query the database for the relevant result and save it in XML format as a cache file.

After a return from this subroutine, a called to the following code will return the XML-formatted result. It merely reads in the entire file and pipes it to the client.

```
sub get_reply_string($) {
  my $class = shift;
  open CACHE, "<$class->{'_cache_filename'}" or die "Could not open cache
file $class->{'_cache_filename'} for reading";
  while (<CACHE>) {
    print $_;
  }
  close CACHE;
}
```

While this algorithm performs well with a single access, in its present state it can have concurrency issues which are discussed in section

PROJECT CONCERNS

This subsection will consider some concerns.

Concurrent Access.

Parsing the ION Database Row by Row

Each row corresponds to one 15 minute window and one sample. The first six rows and five columns for this file are shown below. There has been minor editing for clarity.

```
                     , USB_641, Comp_sci_2,Macleod_312, Rusty_Hut_307,…
Timestamp            ,     kwh,        kwh,        kwh,           kwh,…
01/01/2013 12:15:00 AM, 9591521, 436197.75  ,         0,    4157702.75,…
01/01/2013 12:30:00 AM, 9591565, 436257.0938,         0,    4157731.5 ,…
01/01/2013 12:45:00 AM, 9591608, 436316.4375,         0,    4157760.5 ,…
01/01/2013  1:00:00 AM, 9591652, 436375.875 ,         0,    4157790   ,…
```

To calculate the consumption for a one hour period, we took the difference between the reading at the starting time, say 1:00:00AM, and the reading at one hour later, say 2:00:00AM. Each sample interval is 15 minutes, and so a total of 5 rows are evaluated.

Dealing with ION Database Corner Cases

It became apparent that some entries were not usable, for example they were blank or 0. A design decision in this project was to fail gracefully. If there was a valid reading at both the desired start and end times, we simply took the difference. For case of summing an entire day, it was possible that the end point might be unavailable but a suitable row nearby would be workable. Thus, the algorithm evolved into first looking for the starting row by looking at time, then continuing iteration through rows until a suitable kWh on that row was obtained, that is, a non-zero kWh reading. This will be our lower bound kWh reading. Then, we continue iterating through rows until the ending time is encountered and we retrieve that row's kWh reading as

our upper bound. A subtle point is that at each row we need to keep track of the latest *valid* kWh reading because it is entirely possible that the reading on the final line is not usable.

This way, we only have to traverse forwards.

There is also the case of overflow as these meters all overflow at 10^{10}. First we have to make sure that Perl variables can handle numbers this large. Running the following code at a Linux prompt will show what the current system uses for maximum float values in Perl:

```
$ perl -MPOSIX -le 'print POSIX::FLT_MAX; print POSIX::FLT_MIN'
3.40282346638529e+38
1.17549435082229e-38
```

Next, we need to detect overflow. While iterating to the end kWh reading, for every *valid* kWh reading that we read, we ensure that it is a larger number than the previous valid reading. If it turns out to be a smaller number, and non-zero, then we validate that it is at least smaller by more than $10^{(10-1)} = 10^9$. More specifically, in code:

```
if ($latest != 0
    and ($latest - $previous + $ROLL) < $ROLL_ONE_TENTH) {
  # if the reading is smaller than the previous,
  # we can still take it as long as
  # it was a wrap-around.  We also reject zero values.
  $previous = $lastest;
}
```

This case is treated as overflow and the rollover amount is added to the final result. Overflow happens about once a year for these meters.

Templates

We have created two templates. Accessing template *one* returns the energy consumption for a specified building compared with itself at the same time of day one day ago. The HTTP GET

request accepts the building name and template ordinal, and in return replies with the kWh energy consumption during a one-hour period as well as the date asked for. It also returns a simple result which scores the lowest at 1 and the highest at 5. Currently, we expose the actual kWh usage along with the vote but there is concern that when this database is eventually exposed to the public that we would want to keep that information internal.

Accessing template *two* returns the energy consumption as a comparison between two buildings at a specified time of day one day ago as well as one week ago. Buildings vary widely in the energy consumption and thus only similar buildings would provide meaningful comparison. For example, the comparison of student residence buildings.

To reduce load on the ION API, previous results that have been cached will be saved to disk with the query terms as part of the filename. As the cached values are of ION readings in the past, the cached values will never become stale. This is a simple and deterministic caching mechanism. A scheduled task on the ION API host must periodically trim cached files by sorting them according to their time stamp to prevent exhausting disk space.

5. PROJECT CONCERNS

This subsection will consider some concerns.

Concurrent Access

There is a concern of concurrent access to the database causing an inconsistent state if two nodes are writing to the same file. This could only happen if both connections were running the exact same query at the exact same time. Both threads could see the cached result not existing, and then both start to generate the cached result and write it to file. Because the code writes to cache file one line at a time, there is a chance of corruption if both write at the same time. A possible solution to this would be to save state to the local machine in the form of a semaphore of each file.

It is important to filter all URL parameters from the public as there can be deliberatelu as well as unintentionally malicious parameters. Currently, because the API sits behind the UBC firewall, it is not exposed to this. As well,

Effective Communication and Visualization

Effective visualization has been a hurdle for us as we have found it to be easier to recognize an ineffective visualization than to come up with a novel representation. The result of a comparison is a number between 1 and 5. This can be configured in Cool Sign to map to a table of pre-generated graphics. One visualization is to display an energy-conserving idea when consumption is high, for example, reminding people to switch off lights when not in use. Another visualization would use humor, for example a drenched squirrel chastising the audience's excessive use of laundry driers while he is hand-wringing his own fur.

There really are a lot of possibilities with communicating the idea to conserve energy. Perhaps more important than the specific visualization itself is that the messages should be random or at least appear to passerby's that there is frequently new content each time they look. Otherwise, it is likely for people to gradually tune out the perpetually drenched squirrel.

6. RESULTS

We achieved the project objectives of automating the process of transferring ION Database information to display nodes. As well, we found that the existing infrastructure in place works well with the ION API implementation. Meanwhile, we did not have the opportunity to interact with the public to solicit feedback on visualization methods due to time constraints. As well, an outstanding work item is full sanitization of received URL parameters. I would recommend that persons involved in future work ensure that they fully understand the subtleties of Perl data structure as I had a significant ramp up period due to this. This project was a success and a good interim step for integration with personal mobile devices and web browsers.

7. CONCLUSION

This report investigated how to integrate ION Database meter readings with existing UBC Electronic Signage infrastructure.

We have successfully implemented two templates to accommodate different consumption patterns. In addition, we have laid the groundwork for upstreaming to other data producers such as RSS feeds as well as downstreaming to other consumers such as personal mobile devices.

Although the current implementation reads from a static file, it does so strictly row by row so that it can be easily ported to a live database without much modification. On a similar vein, the ION API was designed with a Model-Controller design patter, completely abstracting one half of the code from the internal representation of the database. The ION API project was a success.