Final Technical Report
Ouroboros

2015-05-21

Team Cobra

Andrew Lyne Gabriel Marcano James Kuglics Jared Smith

Project Sponsor

Nathan Ransom

Faculty Coach

Rick Weil

Table of Contents

1. Project overview

The current state of embedded system solution prototyping at Harris requires software developers to individually engineer their own prototype servers and web services, without much support for automation. This lack of automation support leads to developers creating many different implementations of servers and services for embedded devices, resulting in a lot of time being spent by developers essentially re-writing similar systems. Ouroboros is meant to address this problem by providing a code generation platform developers can use to generate servers quickly which can then be compiled and run on embedded systems. In addition, the generic interfaces exposed by the Ouroboros-generated servers means that it is possible to developers to share plugins that connect into the servers. The primary goal of Ouroboros is to reduce the time developers spend setting up test web servers on embedded devices, leaving more time for actual testing of the embedded system.

The scope of Ouroboros is split into two major components, the code generation and the run-time behavioral aspects of the project. For the scope of the code-generation aspect, Ouroboros covers everything from taking in an input configuration file to outputting portable C++ code integrated with the open source Mongoose web server. For the scope of the run-time behavior aspect, Ouroboros covers the C++ and REST APIs it exposes when the server itself is compiled. Technically speaking, compiling the server is outside of the scope of the project, but for convenience the project has been configured to use Autotools for compiling the customized server.

2. Basic requirements

The requirements of the Ouroboros project are outlined in the following list:

1. Ouroboros will provide a means for describing the operating state of a system, consisting of parameters and external functions, by means of a configuration file.
2. Ouroboros will use the provided configuration file to generate code that can be compiled and run in an embedded system that can run a Mongoose web server.
3. Ouroboros will, in addition to generating custom embedded server code, also generate a Web UI using HTML and CSS that will be served by the custom server compiled from the generated server code. The Web UI will allow for the display and modification of the system state via a web interface.
4. Ouroboros will provide a REST API through the compiled embedded server that can be used by a 3rd party client networked application to modify the state of the device, including changing parameters and running functions specified in the configuration file.
5. Ouroboros will provide a C++ API with the compiled embedded server that can be used by 3rd party software run on the embedded device to modify server parameters and register and execute functions described by the configuration file given to the platform for server creation.
6. Ouroboros will provide a callback mechanism for its C++ and REST APIs in order to allow for 3rd party software to respond to server state changes.

Ouroboros functions as a pipeline for generating custom web servers for platforms that can run the Mongoose web server. The input to the project is a custom XML configuration file which details the start state of the web server. This input XML configuration is given to the Ouroboros code generator, which in turns produces the necessary components to build the customized web server. Once the web server is compiled and deployed on the target platform, the server exposes a REST API that web services can query for information about the server, and a C++ API via a plugin interface, that local applications can hook into at startup and interface with the server that way. The C++ and REST APIs support the notion of callbacks, which can be used to notify external components of state change of specific components of the server.

The general outline of the pipeline is shown in the following figure:

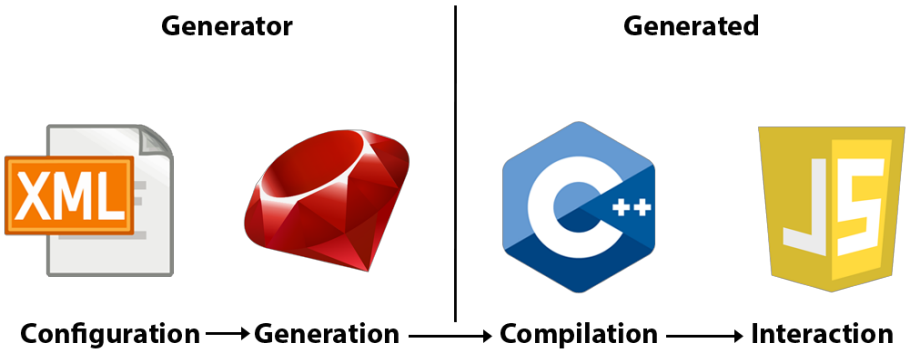

Figure 1. Graphical depiction of Ouroboros pipeline.

Human interaction for setting up Ouroboros is only needed for the steps between the four major steps in the pipeline. For setting up Ouroboros, human or external interaction is only needed for creating the input XML describing the initial server state, running the code generator, compiling the generated C++ code, and running the resulting binary. Third party applications or web services that interface via the provided APIs must also be developed separately from Ouroboros.

3. Constraints

The only build-time technical requirements imposed upon the project was that Ouroboros must use the Mongoose embedded web server as the basis for the generated server, and that the C++ version that the generated code should target should be C++03. From a runtime perspective, the compiled Ouroboros server must expose both a C++03 and a REST API. For this project in particular, the target platform is a Raspberry Pi model B+, which runs on a BCM2835 System on Chip (SoC), which in turn runs with an ARM ARM1176JZF-S CPU. Ouroboros should be portable to any target platform that is supported by the Mongoose web server, including any hardware capable of running Windows and POSIX compliant systems, but this assertion was not tested and was not a part of the scope of this project.

4. Development process

The team selected to use an iterative process for developing the system. The general idea behind the process was to have two major iterations for the project, one encompassing the first semester, and another encompassing the second, with all the development activities (requirements gathering, design, etc.) being stepped through in each iteration. The process further developed in the second semester, where the team began doing smaller iterations weekly, beginning with a planning meeting to determine what needed to be accomplished that week, working on development, determining if there were any blockers and resolve them, and then reporting on the progress for the week during the weekend.

The team selected this process with the hopes of being able to show working versions of the system to the sponsor at the end of the two major planned iterations. The sponsor approved of the process, specially because of the ability of the process to provide working systems at the end of each major iteration.

In terms of communication, the process requires a lot of communication at the beginning of iterations in order to ensure that requirement elicitation is done correctly, and also at the end in order to show progress. With the advent of the smaller weekly iterations, the team would be able to give reports on the work accomplished that week, as well as provide current prototypes of the system including work merged in that week.

As for team roles, Andrew acted as team coordinator. This entailed setting up meetings, acting as the single point of communication between team, coach, and sponsor, and keeping the team on task with deliverables and deadlines. The remaining team members each became domain experts on a specific subsystem for the system: Andrew with the XML configuration, James with the Ruby generator, Gabriel with the C++/Mongoose backend, and Jared with the front-end Javascript UI. The domain experts became the primary developers for their respective sections and essentially had a final say, besides decisions made by the sponsor, for decisions in their domains.

5. Project schedule: Planned and actual

After finishing with requirement elicitation and some initial design the first iteration, the team took a look at the requirements of the system, and discussed the difficulty of implementing the different requirements. Since the goal of each iteration was to have working systems at the end, the team could not ignore sections of the project, but instead could opt to only offer some functionality in the different steps of the pipeline. As a result, the team split up the features as follows:

Release 1:

1. Creating code for a simple custom Mongoose server for an embedded device
2. Definition of most simple types for XML configuration file.

3. Generate code to provide REST API so that users can have limited interact with the device
4. Generate code to provide HTML pages that the mongoose server will serve

Release 2:

1. Code generation tool will be used to:
    a. Interact with and connect with the Mongoose server on an embedded device
    b. Generate code to expose a C++ API for developers to easily hook components of the device with the Mongoose server.
    c. Generate code to allow for custom functions to be handle through requests
    d. Allow for callback to be accessed via both the C++ and REST APIs.
2. Create and provide access to CSS files to style the generated HTML pages.
3. The code generation tool will provide input validation for users so that users will not be able to send bad data as parameters to the device
4. The code generation tool will auto-generate documentation for the code that it also generated (Stretch goal)

The schedules for both major releases were met, with one minor slip for the first release. Although the system at the time of the first release could meet all of the requirements the team sought to implement by that time, it still required some coaxing in order to get the system to function as desired (some values had to be hard-coded in, work-arounds had to be implemented, etc.). The team fixed the problems that were left from the first iterations in the first weeks of the second semester. This slip in the first release did not impact the team's ability to finish the project in time. The stretch goal for the second release was not met.

6. System design

As indicated in section 2 in figure 1, Ouroboros has four major components, configuration, generation, compilation, and interaction. Each of those components is outlined and explained below.

6.1. Configuration

The configuration of the server is made by an XML file. The XML file uses a DSL defined by the schema. The schema allows for two main tags, <group> and <field>. Groups can contain other groups (subgroups) and fields. The fields have types which are defined as function, `stringField`, `enumField`, `booleanField`, `unsignedIntField`, `signedIntField`, `unsignedByteField`, `signedByteField`, `unsignedShortField`, `signedShortField`, `floatField`, and `doubleField`. Figure #2 below shows the hierarchy of how these classes interact with each other.

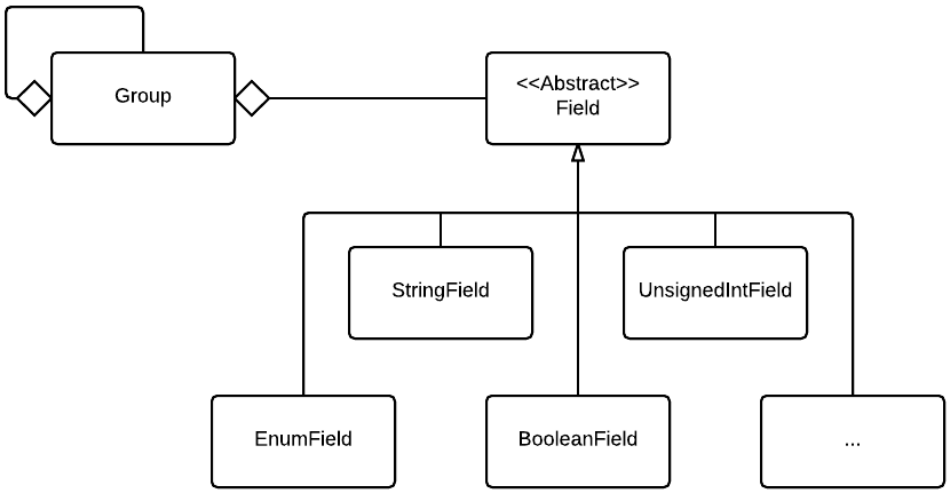

Figure 2. XML data representation.

The choice for XML started with a strong suggestion from the sponsor. There was no prior experience with any team members, so other alternatives were explored. The two other markup languages discussed included YAML and JSON as they were newer and simpler options for the team's purposes. In the end the team decided to go with XML because of the sponsor's strong preference and because XML offers strong validation through the use of user-defined schemas. This allowed for the team to build their own small DSL within XML for describing the system and easily enforcing it during generation.

These field types are used to define the attributes of the target device. With the exception of functions, all are data types you would typically find in a low level programming language such as C or C++. Functions are actions to be triggered which executes code written by the user. The function field type defines the name of the function to be called, and an empty stub is generated in the server code for the user to implement themselves.

6.2. Generation

The generation phase is the second component of the generator system. This part of the code is a ruby program that transforms the XML into compilable C++ code through the use of a Ruby gem called RGen (a model driven development framework) while using custom-implemented logic checking. The execution is done on the command line with the name of the XML configuration file as input.

The sponsor's initial suggestions for parsing of the XML was to use either Ruby or Python. The decision was based primarily on the team's prior expertise in Ruby programming. With the discovery of the RGen gem, Ruby became the primary choice for our implementation of this subsystem.

Before the XML is parsed, the Nokogiri gem is used to validate the XML against a predefined schema. This is done to ensure that the user of the system has used correct XML syntax and has used only the tags and attributes that have been defined in the schema. Once the schema has been used to validate the XML, it is then parsed using RGen.

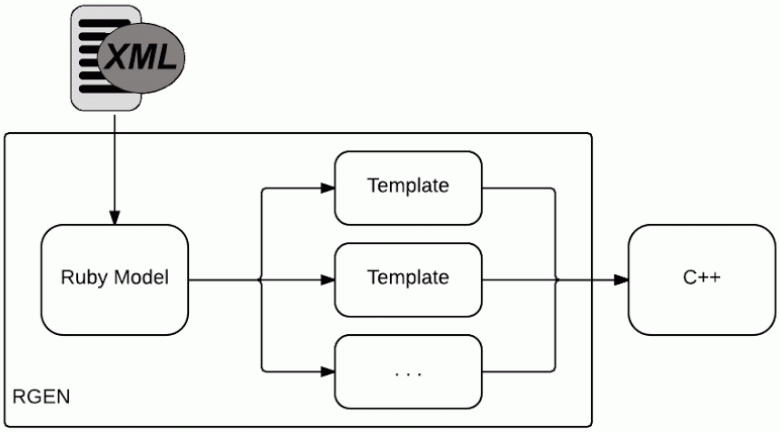

Figure 3. Data and processing flow for code generation.

After parsing the XML, RGen builds a model (based on a predefined metamodel that was built using RGen's own internal DSL) that represents the data inside the XML. The model is then checked for logical correctness based on the constraints placed on the data by the user of the system. For example, an integer field with a value of 20 and a min value of 25 would not make logical sense. In these cases, Ouroboros will tell the user where errors were found and terminate.

Upon successful logic checking, the model is then passed to a few generation templates. These templates utilize the ERB syntax with some added functionality from RGen. This added functionality makes recursing through our tree structure very simple and efficient. The output of these generation templates is compilable C++ code which represents the data defined in the XML configuration.

Originally, the generation aspect of the code was quite significant to the overall system, generating many different files. After concerns of how to test such dynamically generated code would be carried out, the team managed to refactor it down to just one C++ file describing the attributes in XML in the form of C++ code.

6.3. Compilation

The majority of the generated server is actually static C++ code that is not generated by the generator step. Originally, the team began to implement the code interfacing with the Mongoose webserver in C, but then the team decided that since it had to support a C++ API eventually, all of the project might as well be in C++. This turned out to be a good decision as

it allowed leveraging object oriented programming and C++'s powerful generic template system in order to abstract out the different components of the system. The general architecture of the server is shown below:

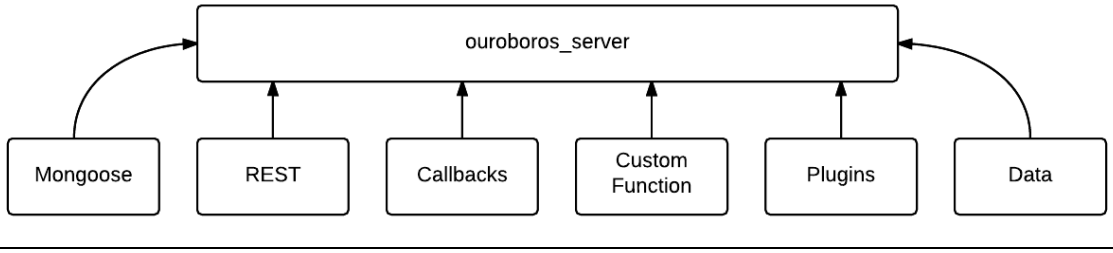

Figure 4. General server architecture.

As shown in Figure 4, the final, compilable server is composed of the Mongoose web server, code for handling REST requests, code for handling callbacks, code for implementing the custom function hooks, code managing the C++ plugin mechanism, and the generated data store code. All components communicate with each other through the ouroboros_server, which effectively acts as a façade for the other subsystems.

6.3.1. Mongoose

The Mongoose web server code is not modified by Ouroboros— it is pulled directly from the Cesanta GitHub repository, and interfaced with the rest of the application. The Mongoose web server exposes the ability to customize its behavior, such as the port to listen on, and the ability to listen for specific server events and act accordingly, by registering a special callback function. It is via this special callback mechanism that the REST API, including REST callbacks, is set up with Mongoose.

6.3.2. REST

The REST subsystem consisted of a generic wrapper for translating the incoming REST URL requests, as well as functions within ouroboros_server that handled the different types of REST calls available, specifically requests for groups, fields, callbacks, or functions. These handlers request resources from ouroboros_server as they need it to fulfill their requests.

6.3.2.1. REST syntax

Generally speaking, the period character is used to designate nesting of groups, so that '`group1.group2`' means that '`group2`' is nested inside '`group1`'.

For all the following syntax examples and explanations, the assumption is made that all requests are in the remote host `foobar.net`, and that there is a group called '`group1`' that has a field called '`field1`' and a function called '`function1`'. Another field, '`field2`', is present at the root group level. In other words, the data structure looks something like:

```
root
  |---> field2
  |---> group1
          |---> field1
          |---> function1
```

Figure 5. Data layout for examples listed in this section.

See section 6.3.2.6. for an explanation about how the names of groups, fields, and functions are processed by the system.

6.3.2.2. Fields

Syntax for accessing fields follows one of the following two formats:

`/groups/{groups}/fields/{field}`
for cases where the field is in a non-root group

`/fields/{field}`
for cases where the field is in a root group

Fields will respond to PUT and GET REST requests, and will return an HTTP status code 400 (Bad Request) for all other verbs.

The following code examples show the syntax expected for GET and PUT requests.

```
GET /groups/group1/fields/field1 HTTP/1.1
Host: foobar.net
```

Figure 6. Sample HTTP GET request for a field.

```
PUT /fields/field2 HTTP/1.1
Host: foobar.net
Content-Length: 16

{ "value" : 30 }
```

Figure 7. Sample HTTP PUT request for a field.

6.3.2.3. Groups

Syntax for accessing groups follows one of the following two formats:

`/groups/{groups}`
for cases where the group is in a non-root group

`/groups/`
for cases where the group is the root group

Groups will respond to GET REST requests, and will return an HTTP status code 400 (Bad Request) for all other verbs.

The following code examples show the syntax expected for GET requests.

```
GET /groups/group1 HTTP/1.1
Host: foobar.net
```

Figure 8. Sample HTTP PUT request for a group.

6.3.2.4. Functions

Syntax for accessing functions follows one of the following two formats:

`/groups/{groups}/fields/{function}`
for cases where the function is in a non-root group

`/fields/{function}`
for cases where the function is in a root group

Functions will respond to PUT and GET REST requests, and will return an HTTP status code 400 (Bad Request) for all other verbs.

The following code examples show the syntax expected for GET and PUT requests.

```
GET /groups/group1/fields/functions1 HTTP/1.1
Host: foobar.net
```

Figure 9. Sample HTTP GET request for a function.

```
PUT /groups/group1/fields/functions1 HTTP/1.1
Host: foobar.net
Content-Length: 45

{ "parameters" : { "a" : "30", "b" : "20" } }
```

Figure 10. Sample HTTP PUT request for a function.

6.3.2.5. Callbacks

Syntax for accessing functions follows one of the following two formats:

`/groups/{groups}/fields/{fields}/callback`
: for cases where the field is in a non-root group

`/groups/fields/{field}/callback`
: for cases where the field is in a root group

Callbacks will respond to POST requests, and will return an HTTP status code 400 (Bad Request) for all other verbs. Unlike all other REST calls, upon success it returns HTTP status 201 (Created).

The following code examples show the syntax expected for POST requests.

```
POST /groups/group1/fields/field1/callback HTTP/1.1
Host: foobar.net
Content-Length: 36

{ "callback" : "barfoo.net:8082/" }
```

Figure 11. Sample HTTP POST request for a function.

6.3.3. Callbacks

Ouroboros supports registering callbacks/notifications through both the C++ and REST APIs. Internally, this registration and notification system is implemented via the observer pattern. An internal `subject` class, associated with a callback function, observes specific fields, which

then call a `notify` method on the `subject` whenever the associated field changes. `Subject`s then call the associated callback whenever they are notified. This approach is used for both APIs. The main difference between the two APIs is that the C++ API allows for plugins to register actual callback functions with a `subject`, while the REST API only allows for a web service to register a website to which the JSON representation of the field that changes will be sent on a notification.

6.3.4. Custom functions

Custom function can be called via the REST or C++ APIs. Upon server initialization, all functions declared in the XML are effectively null or empty functions. In order for a function to do useful work, a plugin must register a function with the function system that will be called whenever the function is invoked. For example, the XML file could have a 'shutdown' function, and then a plugin could register a function with the function system so that whenever the 'shutdown' function is invoked its own registered function is called as well.

6.3.5. Plugins

The C++ API is exposed via a plugin mechanism that allow for 3rd party applications to access the ouroboros_server directly and query its state. All plugins must use C++ and must export a C function (in C++ this is done by using `extern "C"`), with the following signature:

```
extern "C" bool plugin_entry(ouroboros_server& aServer);
```

Figure 12. Function signature for function called by plugin mechanism at initialization.

Even though all plugins must implement the above function in C++, this function can be used as a shim to interface with other languages, such as Python.

The plugin mechanism for the current version of Ouroboros is implemented using POSIX dynamic library loading. This limits the current version of Ouroboros to POSIX systems, but it should be able to port it to other systems by extending the plugin system to use a compile-time strategy pattern, which would allow for the dynamic loading mechanism to be changed at compile time. Currently, moving to a non-POSIX system would only require for the `plugin_manager` class to be rewritten, specifically the manner in which code is loaded dynamically.

Ouroboros assumes the existence of a 'plugin' folder in the same directory as the executable is run, which it will scan for files that expose the function shown in figure 12.

6.3.6. Data

Ouroboros uses the name under the `<title/>` tag for accessing elements within the generated data structure. Originally, the team meant that users could use the exact same title provided in the XML tags for accessing data, but due to bugs in the internal regex libraries the

team used for parsing URLs, Ouroboros now lowercases and replaces spaces in the titles with an underscore (as an example, “A Number” is interpreted as “a_number” by Ouroboros). Both the C++ and REST API use this normalized version of titles for lookups.

The generated data is kept within a tree structure within Ouroboros. The code generator output provides two functions, one to create a new copy of the tree structure, and another to release the resources used by the structure, which hook into the mechanism for creating the tree within Ouroboros. This data tree structure uses a similar syntax as the REST API for accessing its data, the only difference being that it takes the nested group and the target field or function as separate arguments. In theory, the underlying data structure generated by the code generator does not need to be a tree, so long as the data structure exposes the data as a tree. Ouroboros represents exposes the internal data in the same order as it is outlined in the XML configuration file.

6.3.7. Autotools

Compiling the custom Ouroboros server was originally beyond the scope of the project, but the team began to use Makefiles to help compile the servers so that the team could test these more efficiently. Eventually, the team’s Makefile system grew to be complex and new additions and bugfixes were beginning to take longer to accomplish, so the team decided to refactor the build system and use Autotools, which offered a lot more support for customizing builds, with the added bonus of allowing users to specify cross-compilation details. The team did run into problems with different versions of Autotools using different macros for syntax, so the team suggests developers updating their autotools installations to the newest possible. Ouroboros was developed and tested using the following versions of Autoconf and Automake:

Automake version 1.15
Autoconf version 2.69

It is possible to compile Ouroboros using older versions of Autotools, but it will require modifications to the different build-related configuration files in the system.

6.3.8. Testing

Since most of the code for Ouroboros is static, the team was able to implement some unit testing to check it. The unit testing uses Google’s C++ testing suite, GoogleTest, as the framework that powers testing, which in turn hooks into Autotools. One of the issues the team encountered while working with GoogleTest is that while it claims to be C++98 compliant, it is not really. In fact, according to Google itself, they do not intend to support the pedantic compiler flag[1], meaning that the suite does not really conform to C++98. Since Ouroboros targets C++03, which is an extension of C++98, this discovery was not welcome by the team since it was identified late in the development process. The team decided that it would remain using GoogleTest for unit testing and document the incompatibility, since this does not impact the server code directly.

[1]Issue 152: g++ -pedantic produces compile errors:
https://code.google.com/p/googletest/issues/detail?id=152

6.4. Interaction

After compiling the server, the resulting binary exposes several forms of interaction, including a local C++ API (explained in section 6.3.5), a RESTful JSON API (explained in section 6.3.2), and a web UI that allows for developers to view and modify server state directly via a web browser. These three forms of interaction share common functionality for modifying the internal state of the embedded web server running on the target device.

An original consideration for the implementation of the UI was whether to implement the UI code in CoffeeScript or JavaScript. CoffeeScript was considered for its readable, ruby-like syntax, but was ultimately decided against due to the need to convert CoffeeScript to JavaScript and that users were more likely to already know JavaScript because of it's prevalence in the open-source and web development communities.

The C++ API exposed by the plugin mechanism allows for plugins to change server values, and can be used to register callbacks so that the device can be controlled based on changing values in the server. For example, a light on the server could have several states for emitting different colors of light. In this server the light state could be represented as an enum with a finite number of options corresponding to the colors allowed. The plugin could then register a callback to listen on the light state enum, and as a result adjust the actual light color based on the current state of the variable.

The REST API allows access to the server via an HTTP interface which other web services not located on the target device can use to alter the server state. The API communicates through RESTful routing and uses JSON as the format of the data payload. The server allows for GET, PUT, and POST HTTP requests, depending on the type of request (refer to section 6.3.2 for more details). In general, GET requests query the state of the server, PUT requests are used for updating values on the server, while POST requests are used to register callbacks on the server for change events.

The web UI uses the simplest form of interaction with the web server. Visiting the website will load an index HTML file, a JavaScript file, and CSS styling files. The HTML file is very barebones and includes the structure of the page as well as hidden templates for use with the JavaScript code. The JavaScript code is not generated in any way and stands alone from the server. It can parse the JSON objects returned from the API and uses the complete JSON representation of the server to build the page. The JavaScript code uses JQuery to generate the navigation links, the groups, and their fields in the form of the appropriate inputs corresponding to the field's data type. The JavaScript file then builds event listeners on page load as well for submitting AJAX requests to the server. One current assumption of the UI code is that there are no fields under the root group, as in all fields are placed under subgroups to the root group. The server will continue to function if this assumption is violated, but the UI may not display the fields in the root group correctly.

The site is styled using bootstrap for the purposes of familiarity and customizability. Heavy use of classes are used for the majority of styling. This was done to minimize file size when being served from the server. Additional customization can gained by changing the colors defined in bootstrap, thus allowing for site-wide style changes. An additional CSS file is

loaded so that any developer or user familiar with CSS can modify it to alter some aspects of the UI.

7. Process and product metrics

There were two metrics that the team kept track of, time and commit frequency. Time was measured by manual time entries on our work, and commit frequency was measured by GitHub automatically. Time tracking was effectively measured the first semester by the team.

After the first semester, it was determined that time tracking was not helping the progression of the project. Although it was a good personal metric for determining the accuracy of our time estimates, the concurrency of which we worked on other classwork as well didn't make the estimates accurate. When the second semester started, diligence in its tracking fell of because of the manual work involved and because of the lack of usefulness.

Commit frequency turned out useful. After the first semester it was determined that most of our work was done during the week, usually just before the team meetings. There was little work being done on the weekends when the team was supposed to be doing work. Using this insight, the team modified its process. The team started to have Sunday check ins where they reported their work so that their tasks would be top of mind on the weekends for reporting at the end. This change in process resulted in more work completed on weekends, but still, relative to the work being committed on Tuesdays and Thursdays before meetings, the work done over the weekend was not as significant, shown in figures 13 and 14. Still, these figures only show when commits were made to GitHub, and not when the actual work was done.

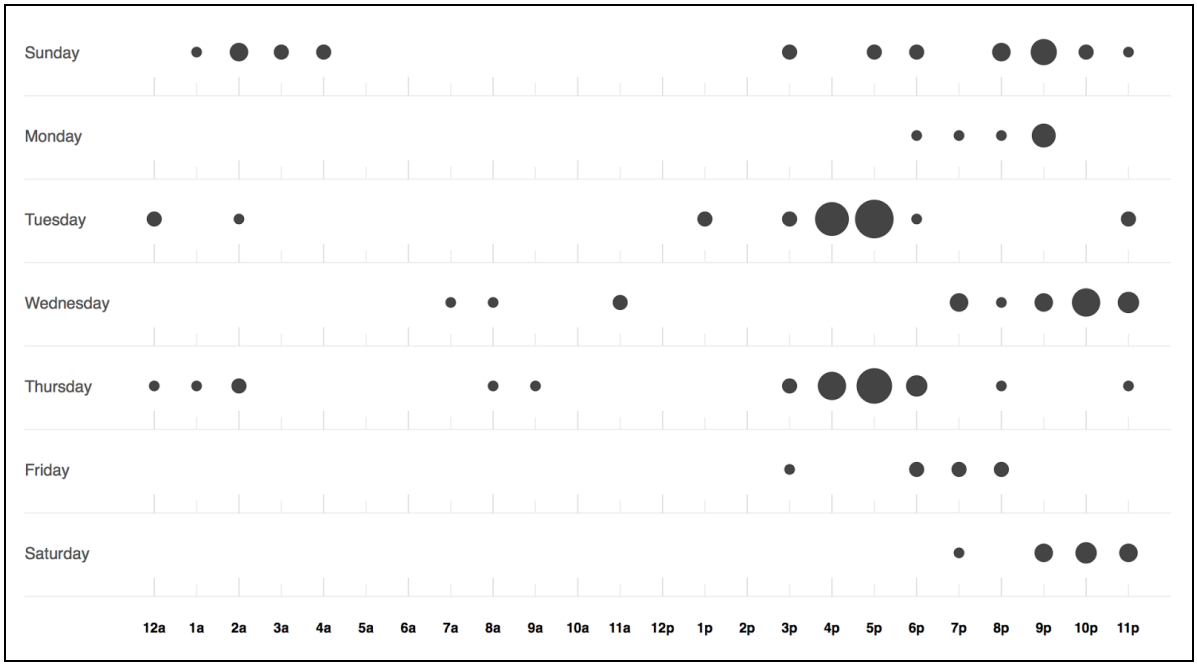

Figure 13. Commit punch-card for first semester.

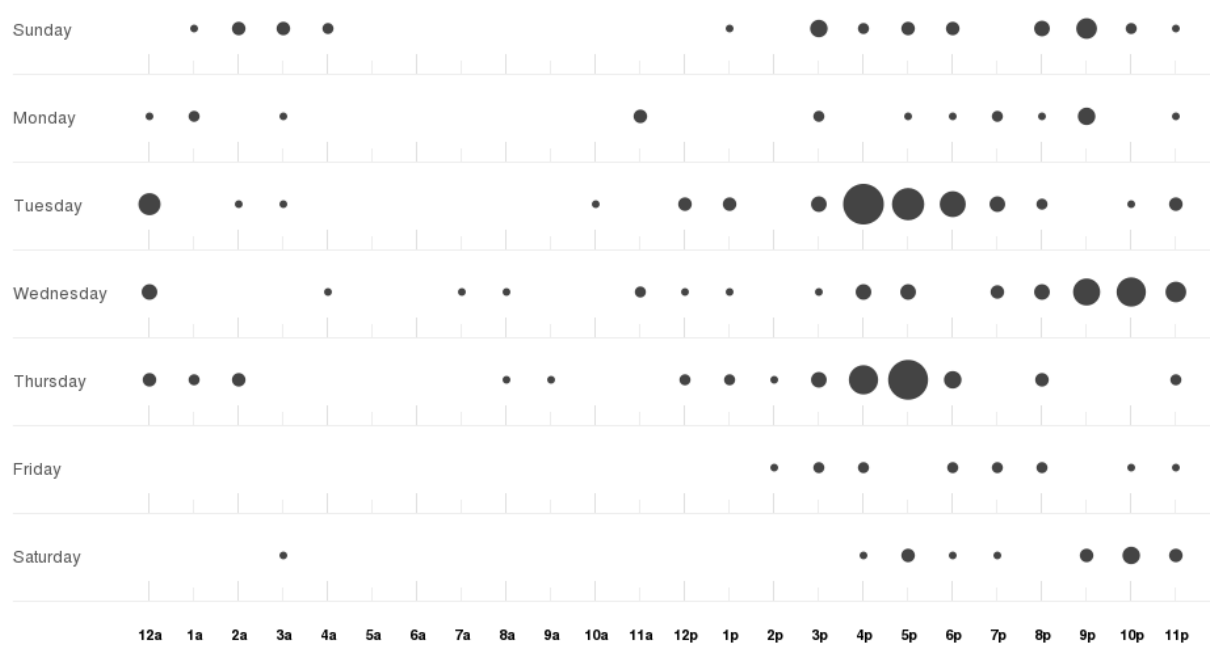

Figure 14. Commit punch-card for second semester.

Figure 15 shows the commits to the project over the timespan of the project. Noticeable features in the graph identify special times of the year, notably the end of the first term, signified by as spike in commits, as well as Winter break, highlighted by a very low amount of commits. Interestingly enough, there was a spike of commits during what appears to be Spring break at the end of March and the beginning of April, and then one final spike at the end of the project.

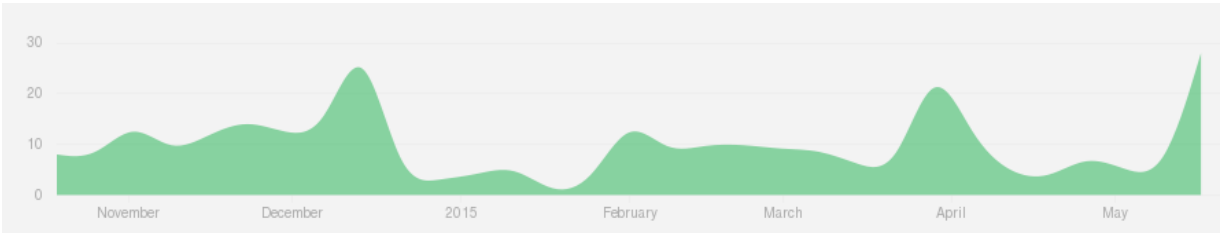

Figure 15. Overall commits for project over time.

8. Product state at time of delivery

The state of the project is feature complete at the time of delivery. All requirements outlined in section 2 of this document have been met. The only feature implemented that was not a part of the requirements was the use of Autotools for compiling the system. Originally, compiling the generated server was outside the scope of the project, and all the team was required to do was to explain what would be required to compile the system. As the size of the C++ server grew, the team began using Makefiles to compile the system for faster testing, but then refactored the build system to Autotools due to the added flexibility and functionality it provided. This allowed the team to compile custom servers quickly for testing purposes. Instead of removing this feature, the team opted to leave it since developers using the project may find it useful. In addition, one of the goals of Ouroboros is to be able to target embedded platforms, and Autotools provides support for cross-compiling binaries.

Currently enums and functions, while supported on the back end of the system, are not visible through the web ui. This realization came too late in the development process for the team to remedy,but should not be too involved to fix.

9. Project reflection

The team exhibited a great deal of synergy throughout the two semesters of this project. All members of the team contributed to discussions, collaborated on the project throughout all phases of development, communicated well with each other, and voiced concerns within the team when necessary without fear or retaliation. As a result, team morale was kept high throughout the project. This particular project benefited from stable requirements, which remained so for the duration of the project. This allowed for the team to focus more on planning up front, which paid off the second semester with almost all time outside of meetings spent on development.

One are the team could have done better is with keeping the sponsor up to date with the status of the project. At the end of the first semester the sponsor indicated that the team’s communication could be improved, which is something the team targeted to be improved second semester. At the beginning of the second term, the team did better, providing weekly updates with the sponsor and meeting in person with the sponsor every other week. Communication broke down once again after the sponsor had to leave for business travels for a few weeks, and also as the term progressed due to an increased workload on the team members. In general the team tried to email the sponsor weekly giving status updates, but there was at least one time this last term where the team spanned two weeks without giving a status update. On the plus side, the stability of the requirements did allow for the team to progress even in the less-than-optimal amounts of communication that was taking place.

The team would have liked to have done more unit testing. Currently, the unit testing only covers some of the lower level details of the server, but even so it is not very comprehensive. It would also have been more enlightening to employ other forms of testing more concretely,

such as formalizing functional testing and doing more static testing, such as running static code checkers against Ouroboros.

Overall, the team enjoyed working on Ouroboros. Half of the team had experience working with web development, the other half with embedded systems, so over the course of the project a lot of information exchange took place, enriching the experience the team gained from the project.

10. References

1. Ouroboros repository, https://github.com/TeamCobra/ouroboros
2. Issue 152: g++ -pedantic produces compile errors: https://code.google.com/p/googletest/issues/detail?id=152