Not exactly the Internet of Things for Outdoor Lighting: Technology Review

Oregon State University CS Senior Capstone Group 22
Malcolm Diller, Sean Rettig, Evan Steele
Client: Victor Hsu

June 10, 2016

1 Client

Victor Hsu
Oregon State University
Phone: 541-737-4398
Email: hsuv@onid.orst.edu

2 Problem Statement

Outdoor lighting seems like a simple problem to solve, but the solutions on the market today are less than ideal. The standard transformer/timer combos available in the big box stores are rudimentary and clunky at best (constantly needing to be adjusted for the changing sunset and sunrise times), but are reasonably priced. The new-generation smart apps for home automation are flexible and fancier, but are quite spendy and lock you into a specific protocol. So why not use an open platform running on commodity hardware? Easy to use, reasonably priced, and highly customizable–that is our goal.

3 Project Description

Our system will consist of a wireless network of tiny "client" computers that each control up to 4 sets of lights and are controlled by a central "server" computer, which will automatically send out commands to the clients when it's time to turn on or off. The central node will run a control program that can be easily accessed via a touch screen, a web browser, or a mobile device, where the user can locally or remotely control each light individually. Want your lights to turn on at sunset and then dim gradually as the sun rises? Simple. Want your lights to flash when you're throwing a party? Just press a button. The control interface will allow users to easily set "rules" for what their lights do and when, depending on the time of day, the sun/moon position, and potentially even triggers such as weather conditions or calendar dates. This system will be easily extensible to potentially control other devices as well, such as garage doors, sound systems, and more.

4 Pieces

1. OS for the PI

   (a) Raspbian
   Custom built for the PI, this operating system would provide many important features, and be a solid foundation to run on. Installation is very straightforward, and it has support all across the Internet. This means that the Pi would have to run the entire Raspbian operating system, so there will be many utilities we don't need that will be included anyway. The image is easily acquired and it can be burned to a SD card and booted straight away. It will likely require heavy configuration to get all the necessary services running and the extra ones to never run.

   (b) Yocto meta-recipe
   We will not need all of the fancy features that Raspbian offers, and one way we can strip our OS down to the features we need is by building a Linux system using Yocto. The Yocto build system is complicated, but our group has experience building layers for various images. It will allow us to build a custom distribution with every package we need, init.d scripts, and service scripts for only the services we need. It will require build time, and time to burn to installation media. Kevin noted that we could get a build server for the project by the end of the term, which would give us good hardware for Yocto builds.

   (c) OpenWRT
   An open source router firmware package that allows embedded devices to perform complicated network tasks. Only recently was it recompiled to work with the Raspberry Pi. Our collective experience with the firmware is limited, so it would have an exceptional learning curve. It looks like it would be able to handle all the services we would need, but there isn't a whole lot of flexibility for other additions. It's also very difficult to debug or run tests with.

   We have decided to use a Yocto build of Linux for the final build. This decision was made since the Yocto build will only contain exactly what we need, will allow us to easily incorporate new packages with the recipe system, and can be easily shared on our Github repository as a meta-layer for others to build. Proof of concept builds will just use Raspbian with the packages installed.

2. Server Program Implementation

   (a) Python / MicroPython
   The ESP8266 can run MicroPython for GPIO access, making Python the obvious choice for the server program as well. Using Python's socket API, we can easily connect the devices and execute code to work with the GPIOs. The overhead of Python is minimal when considering the ease of use, and MicroPython was practically designed for the ESP8266, so there's lots of support available. Installing it is as simple as flashing a binary to the ESP8266, and the Python shell is supported. However, we would likely load in a Python script to handle incoming commands from the server. The ESP8266 has sufficient storage for one large Python script.

   (b) C / libmraa
   Running the system through C is possible, thanks to libmraa which provides useful abstractions from the /sys/class/gpio and puts it into easy-to-use C libraries. While we could use these libraries, the Python variant provides an easier interface with only slightly more overhead. Additionally, the library contains SWIG-generated wrappers for a variety of languages, including Python, C++, and Perl. The repository is maintained by Intel's IoT Development team, and our team has contribution experience with libmraa.

   (c) sysfs
   The most direct, but perhaps least reliable method is to just use the sysfs interface. This involves doing things like echoing values into files and reading the raw files for values. For instance, accessing the SPI interface for a device registered with major number 0 would look like: `cat /sys/class/spi_master/spi0/spi0.0/iio:device0` which is not ideal or clean. Additionally, it could break if we try to use it on systems with different sysfs layouts. It would be useful to use sysfs for testing, but we should not use bash scripts like this for our final product.

   We have decided to use Python and MicroPython because of compatibility issues and ease-of-use for the devices. Python will run on both the ESP8266 and the Raspberry Pi with all the libraries we need to run the software. Our concerns about overhead are minuscule since the Python will be executing very simple code; it is just interacting with a relay switch. It will mesh well with our web interface, so we'll be using it as a CGI script in the web server.

3. Web Site Implementation

   (a) JavaScript
   An easy language to work with, Javascript will allow us to use many different, responsive templates such as react.js and node.js for control. Node even has wrappers for sysfs functionality which may even allow us to deploy it on the ESP8266 devices for remote control. Using JS for the web interface would offload the majority of work from the Pi to the web browser, which could be useful depending on the complexity of our interface. If we end up using a GPIO abstraction library such as libmraa, it should be noted that most of them have wrappers to Javascript code.

   (b) Django
   A web framework known best for rapid deployment, we could examine this technology as a way to power our entire web interface and backend system. It would have a longer ramp-up time since none of our group members are fully comfortable with Django. However, if we decide that the ramp-up time is worth it, we could see this platform as an all encompassing management tool. Django is well-supported, but does have large learning curve and would require exhaustive group research to understand the entire system.

   (c) PHP
   The hypertext preprocessor is, at first glance, perhaps not the best system to run our backend off of, but we're considering it for a few reasons. It can execute our CGI scripts with the mod_php module for the webserver, our group has significant working experience with the language, and it just works out of the box. It may require additional configuration, but from a 10,000 foot view it accomplishes all the tasks we need. It can mesh well with a wide variety of other techologies we may end up using, and is highly modular.

   We have decided to use the HTML/JS/PHP stack for our project, which means we'll be building a system from the ground up. We can point the Javascript to the CGI files to execute the Python that will change the relays at the remote end, use PHP for the web backend and database access, and use HTML for the pages themselves (although this one will likely just be generated through the PHP code too) to create our full web server stack. This does not rule out us using other technologies such as node or Django as a subsystem.

4. Server/Client Communication

   (a) AD-HOC
   In this communication method, the devices are configured for AD-HOC mode, or a packet-radio system. The wireless adapters need to be able to support AD-HOC mode and be set to a specific channel and IP range. Given the difficulty we've had with AD-HOC networks in the past, we'll maybe try to avoid this mode, but it would be useful as a proof-of-concept for socket communication. However, this is likely not a long-term solution.

   (b) Central WAP
   We would configure one Raspberry Pi as a central Wireless Access Point and have the ESP8266 units connect to it as wireless clients. The advantage to this mode is that we could use the Raspberry Pi to connect to the Internet to perform tasks such as talking to the Wunderground API, which is not possible in AD-HOC mode. This would require that the Pi run a DHCP server, along with other routing services.

   (c) CAS
   A Cental Authority Service model leans more toward the well-defined Internet of Things model. All of our devices rely on an external server for almost all their actions. It would coordinate between the devices, actually store all the data and perform all the heavy lifting, relying on the Pi and ESP8266 devices only for GPIO access. A drawback to this model is the additional server we'd need to configure and rely on, especially since the system would stop working if the server ever becomes inaccessible.

   We have decided to use the WAP model because it provides the benefits of the AD-HOC and CAS models without all the additional configuration burdens. A Yocto meta-recipe for the Pi's WAP functionality would be simple, and connections between the ESP8266 devices would become a trivial task. This does mean the Pi has to run extra services (like DHCP), but it should be more than capable of running the limited services we will require of it.