# POW-R High Level Architecture Document

Grace De Geus
`gdegeus@vtc.edu`

Charles Hathaway
`chathaway2@vtc.edu`

Nate Pickett
`npickett@vtc.edu`

Niloc Quimby
`nquimby@vtc.edu`

Forest Immel
`fimmel@vtc.edu`

November 10, 2012

# Contents

# 1 | Introduction

## 1.1 Purpose of this Document

This document describes the high level design of all components of the **POW-R!** (**POW-R!**) system.

## 1.2 Architectural Assumptions

It is assumed that an entire system consists of one Server, and one or more Satellites.

## 1.3 Summary of Requirements

The POW-R project aims to produce a low cost, low powered, device that will allow home owners to track their power consumption on a per-device basis. The hope is that by allowing home owners to see where all their power is going, they can take steps to disable or replace the most inefficient offenders.

To accomplish the stated purpose, this project provides the following components:

- Satellites
- The Server
- The Display

Each of these components have their own requirements. The basic summary of each one would be:

### 1.3.1 Satellite Requirements

- Ability to plug into standard **NEMA!** (**NEMA!**) 5-15 mains electrical outlets
- Two outlet sockets on the opposite side of the plugs (if it takes up two outlets, it should provide two outlets)
- Less than 5% error on current and voltage reading
- Power draw less than 1**W!** (**W!**) per Satellite
- Broadcasts information up to 500**M!** (**M!**)
- Transmits every 1.0 seconds
- Zigbee compatible
- Encrypted communication

### 1.3.2 Server Requirements

- Physical Server inside the monitored building
- Runs on less than 10**W!**
- Hosts web server for Display
- Button to connect Satellites
- Ability to connect to the user's network
- Zigbee compatible
- Encrypted communication

### 1.3.3 Display Requirements

- User Management
  - Add users
  - Delete users
  - Modify user information
  - Modify user permissions
- Group Management[1]
  - Add groups

[1] Optional

    - Delete groups
    - Modify groups
    - Group permissions
- Device Management
    - Add Devices
    - Disable Devices
        - Does not include deleting Devices, since that would invalidate history
    - Modify Devices
        - Rename
        - Change outlet association
    - Device Groups[1]
        - Device groups are logical groupings of Devices that can be viewed as a single Device in the View Data section
        - Create Device groups
        - Delete Device groups
        - Modify Device groups
- Satellite Management
    - Add Satellites
    - Remove Satellite
- View Data
    - View Device Power Consumption
        - Shows the user power consumption on a per-device (or device group) basis
    - View Power Consumption Over Time
        - Shows the user power consumption over a specified range of time
        - Includes total and per-device power consumption
    - View Device Cost Over Time[1]
        - Shows the user the cost to run a device, or group of devices, over a specified time range
        - Shows the user a comparison of device costs over a specified range with a given interval

    - E.g. Show them the device cost every month for the past year

- Power Bill Guesstimator[1]
  - View cost to run a specified device over a period of time in the future
  - View cost to run multiple devices over a period of time
  - View expected power bill if a device is added
  - Allow the user to specify cost of power for
    - Specific time ranges
    - Specific power-usage ranges
    - E.g. Power cost the user $0.08 per kW/hr for the first 500 kW/hrs, and $0.10 after that
- Factory reset[1]
  - Allow the user to reset the system to default settings via...
    - A form
    - A physical button located on the server
- Software upgrade[1]
  - Allow the user to upgrade the system to the newest software version

# 2 | Hardware Design Overview

## 2.1 Description

The **POW-R!** project is comprised of two main hardware components, the Server and the Satellites. The Server refers to the physical hardware from which the Display shall be served. It also acts as the data center for all Satellites associated to it, collecting data and storing it in a hard drive. The Satellites will talk over ZigBee specification to the one Zigbee module connected to the Server. That one Zigbee module can talk to the Server over **USB!** (**USB!**).

Figure 2.1 shows the layout of the hardware system.

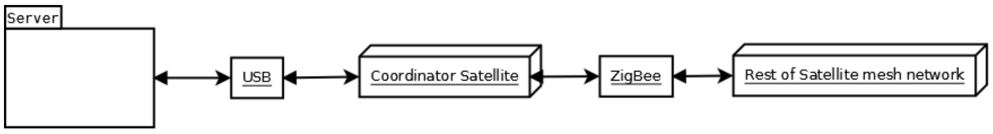

Figure 2.1: System Overview

## 2.2 The Server

### 2.2.1 PC Hardware

#### Mainboard

The Server's main hardware component is the mainboard, a SYS9400-ECX Developer-Ready Reference Platform. It's a small form-factor, low-power machine with the following specs:

- 1.6 GHz Intel Atom E6XX Series Processor
- 1 GB DDR2 RAM

- Roughly 6" by 4"
- Various connection interfaces:
  - 2x **SATA!** (**SATA!**) ports
  - Header for **SATA!** power
  - Ethernet port
  - 5x **USB!** 2.0 ports
  - **GPIO!** (**GPIO!**) pin header

#### Potential Problems

If for whatever reason using this mainboard falls through, it should be noted that the requirements for the Server hardware concern not just specifications, but interfaces as well. In particular, this project requires at least Ethernet, 2 **USB!** ports, a **SATA!** port, and accessible **GPIO!** pins.

#### Storage

The Server's mainboard is connected via **SATA!** to a 40GB **SSD!** (**SSD!**).

#### Power Supply

An adapter rated for 12**VDC!** (**VDC!**) @ 3**A!** (**A!**) is used to connect the Server's board to mains electricity.

### 2.2.2 Add-Ons

The Server provides for the computational needs of the **POW-R!** project; more hardware shall be added before the utility needs of the project are met.

### 2.2.3 Server Hardware IO

#### IP! (IP!) Display

The **IP!** of the Server shall be displayed somewhere on it's casing. The user will enter this **IP!** into their browser to access the Display.

This will be achieved by connecting an Arduino via **USB!** to the Server, and housing it inside the Server's casing. The Arduino will be connected to an

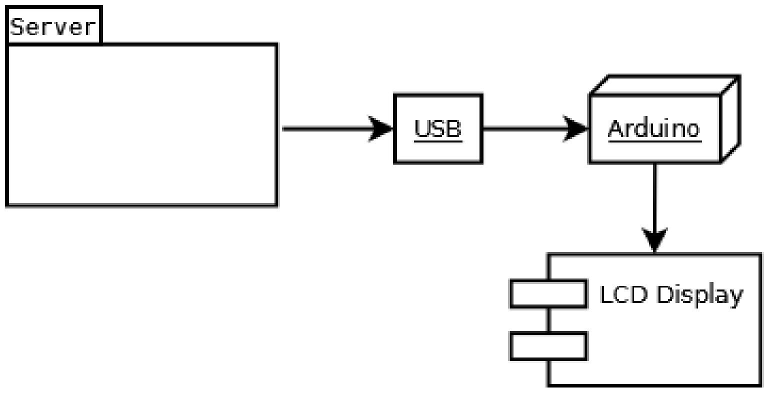

Figure 2.2: **IP!** Display Diagram

**LCD!** (**LCD!**) display which will output the network **IP!** of the Server. Figure 2.2 shows the interaction between Server and Arduino.

The **IP!** address of the system can be obtained from the operating system, and sent to the Arduino one byte at a time. This works well, since the Arduino connects over serial and thusly takes a byte at a time.

#### Power Switch

A standard rocker switch will be added to the case to provide the user with a way to turn the Server on and off. The style of the switch will clearly signify "On" or "Off".

#### Factory Reset[1]

This button should intentionally be placed somewhere inconvenient: Sunken into the case far enough that you need a skinny rod (such as a paperclip) to push it, and in a spot that chaotic forces (children, mean people, God's divine will) will not notice it.

### 2.2.4 Server Hull

The Server needs a protective casing for several reasons. On the physical level, a durable casing will protect the hardware. The casing also serves to render unnecessary ports inaccessible to users. Lastly, the casing shall meet client expectations of what a server looks like.

As far as prototyping goes, the casing can be as simple as a folded piece of aluminum with holes cut in it to fit the **IP!** display, the buttons, and any ports that must be exposed. As an end-game product, the casing would probably be more professional and streamlined.

### 2.2.5 Talking to the ZigBee Mesh

The mesh network that the ZigBee modules form (hereafter referred to as "the Mesh") doesn't directly communicate with the Server at all. This means that there's a clear separation between the computational/utility hardware, and the measurement hardware.

The only point of the Mesh that the Server can communicate with is the "ZigBee coordinator." Every ZigBee mesh needs exactly one coordinator. Holding the title of "Master" in the Mesh, it performs many tasks that will be discussed in the Satellite section. In regards to the Server, the coordinator stands out in the following ways:

- Receives all messages from all Satellites in the Mesh
- Sends measurement data to the Server
- Receives commands from the Server

The coordinator communicates to the Server using serial protocol over USB cable.

### 2.2.6 Potential Problems

As of right now, the Server has not been explored, and as with the rest of this project, is unfamiliar territory.

## 2.3 The Satellites

The Satellites are made up of two sets of hardware: Instruments for measuring current and voltage, and radios for communicating that data to the Server. The end product shall have a hard casing around it, **NEMA!** 5-15 sockets on one side, and prongs for a **NEMA!** 5-15 male end on the other. However, it should be noted that prototyping will most likely involve all of the hardware being spread out on breadboards.

### 2.3.1 Measurement Hardware

Current and voltage running through the mains outlet to the Satellite's associated device must be measured and sent to the Server.

### 2.3.2 Communications Hardware

For the communication between the Server and the Satellites, we will be using the Xbee Series 2 Radio Modules.

#### XBee Series 2 Radio Modules

The XBee Series 2 Radio Module is configured to be running the ZigBee mesh firmware. We chose this module because of it's low current draw. When configured as an end device, it draws at most 40 m**A!**. In addition to it's low power draw, it also has a considerable indoor transmit range of 133 ft.

#### Interface Board

The Digi Interface board is the interface board that we will be using to connect to the Xbee Series 2 Radio Modules. We chose this due to it's simplicity in communicating with the Modules.

### 2.3.3 ZigBee Standard

ZigBee is a standard that uses **IEEE!** (**IEEE!**) 802.15.4 wireless protocol as it's foundation. It's particularly useful for setting up ad-hoc mesh networks. Much like how any Bluetooth devices can talk to each other, any ZigBee devices nearby can talk to each other (so long as they have the proper permissions).

#### Why ZigBee?

The advantage of using ZigBee is that it lines up with the requirements of the **POW-R!** project. It's a specification targeted for low-power, low-data applications that need to be secure.

Another advantage of using ZigBee is that XBee radios, which also fit the requirements of the **POW-R!** project, are ZigBee-compliant, so the forming of a mesh network can happen quickly and easily.

#### Device Types

The nodes of a ZigBee network are obviously ZigBee devices, but from the network's perspective, there are three distinct types of ZigBee devices:

- Coordinator
- Router

- End Device

The coordinator runs the network, keeps it healthy, makes sure everybody is talking when they should be, and overall, just maintains the Mesh. It is also the device that is typically plugged into a computer via **USB!** or some other serial interface, meaning that this is the node you want to send data to. There cannot be a ZigBee network without exactly one coordinator.

Routers are fully functional ZigBee modules, capable of taking care of themselves and routing messages across the Mesh. Routers are capable of being "parent" devices.

End devices are low-power ZigBee devices that don't do any routing. They can only talk to one other device, which is referred to as it's "parent" device. Only a router or a coordinator may be a parent device.

In the **POW-R!** project, the coordinator will be attached to the Server so that it may also act as a data bridge between the Server and the Satellites. The rest of the Satellites are routers; There will be no end devices in the **POW-R!**'s mesh network.

#### Network Topology

Zigbee supports four network topologies, shown in Figure 2.4. A ZigBee network is defined by two rules:

- A ZigBee network must have two or more ZigBee devices
- One of those devices *must* be a coordinator

As mentioned before, the **POW-R!** project will use the Mesh configuration, but without end devices. This means that the Mesh will consist entirely of routers and one coordinator.

### 2.3.4 Potential Problems

XBee Series 2 radios and their interface boards have as of yet been unexplored. This means that as with the rest of this project, there will be about as much learning as there will be doing. To counteract this somewhat, the engineers in charge of weaving the Mesh have learning materials that are perfect for this project.

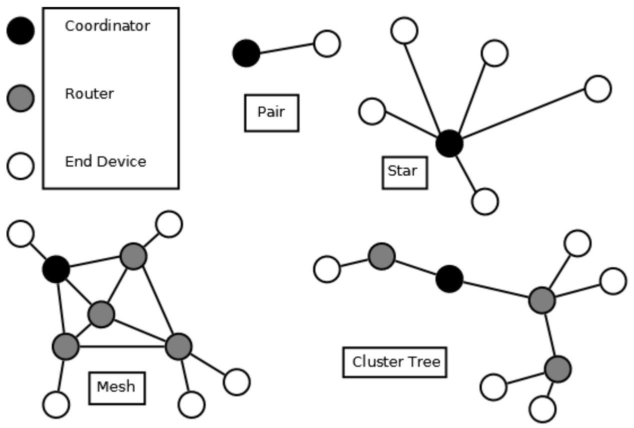

Figure 2.3: ZigBee Network Topologies

# 3 | Software Design Overview

## 3.1 Description

The **POW-R!** project uses various pieces of software to control the Satellites, provide an **API!** (**API!**) for displays, and generate pages for the primary display. The **POW-R!** project will consist of the following pieces of software:

- The microprocessor code
- The server code to communicate with the Satellites
- The server code to present the **API!**
- The client code to render the display for the user

The microprocessor code is documented in the hardware section.

### 3.1.1 The Display Backend

The general plan for the backend is to write modules for a Django web application. These modules will either be running in the background and reporting the data given to the server from the Satellites to the database through the **REST!** (**REST!**) **API!** functions, or responding to user requests sent through the **REST! API!**.

The logic for using Django as the base, and write modules around Django, is one of necessity. We do not have enough time or manpower to make everything we need from scratch. Therefore, we will be using several open source utilities to assist us.

Django will give us an easy-to-use database interface. It will deal with sanitizing all **SQL!** (**SQL!**) queries, and make sure all output is escaped (i.e. no **XSS!** (**XSS!**)). Using Django gives us almost all of the security benefits listed in our non-functional requirements.

To make our **REST! API!**, we will utilize an open-source Django library called tastypie. Tastypie gives us both model resources (that is, **API!** elements that are basically the database tables) and the ability to create custom resources that may, or may not, map to the database.

Another advantage of Django is that it comes with a very extensive, expandable, and adaptable authentication system. This will help accomplish several optional and required features almost instantly.

### 3.1.2 The Display Frontend

The general plan for the frontend is to serve up some fairly static pages, and use extensive amounts of Javascript to fill in the content and animate things. The frontend will communicate with the backend through the **REST! API!**. This will be fairly simple since the backend **API!** will support multiple formats, including **JSON!** (**JSON!**).

To assist us on the frontend, we will be using several open source libraries. They will all serve a very clear function.

- Backbone.js - A library to provide basic functions dealing with fetching and syncing data with the server
  - Backbone.js will deal with tracking the "model" portion of our Javascript code
  - Backbone.js will be made compatible with our **REST! API!** by using a small extension called backbone-tastypie
- iCanHaz - A library that provides advanced templating functionality This is client-side templating, and relies on several "fake" **HTML!** (**HTML!**) tags
- RequireJS - A library that deals with tracking and loading assets asynchronously This includes loading libraries that we will be using
- jQuery - jQuery will be used for both parts of the front end and for the AJAX support
  - We will use jqplot for dealing with graphs
  - We will use the jquery UI to deal with most of our widgets and such

## 3.2 Program Flow

This program will have three main "threads". One thread would be responsible for listening to messages from the Satellites, and another would be responsible for listening and responding to requests on the **API!**. The last thread would be responsible for serving up static files for the main display.

The Satellite thread will be responsible for taking the logging information provided by the Satellites and storing it in the database. Each Satellite will report it's information to the server once every second. Therefore, all this thread needs to do is react to incoming connections.

The **API!** thread would provide the Representation State Transfer **API!** (**REST!**) that will be used by all future displays to get data in a standardized, well defined way. This thread will NOT be serving up images or page templates. This is done to make best use of the caching functionality included in the HTTP server software that will be used.

The last thread will be running a "dumb" web page that servers up static HTML and Javascript files. This thread will also retrieve images and **CSS!** (**CSS!**) files from the server.

## 3.3 Data Flow

This application will use a **SQL!** database to store all data. Each module will interact with the **REST! API!**, which interacts with the database as needed.

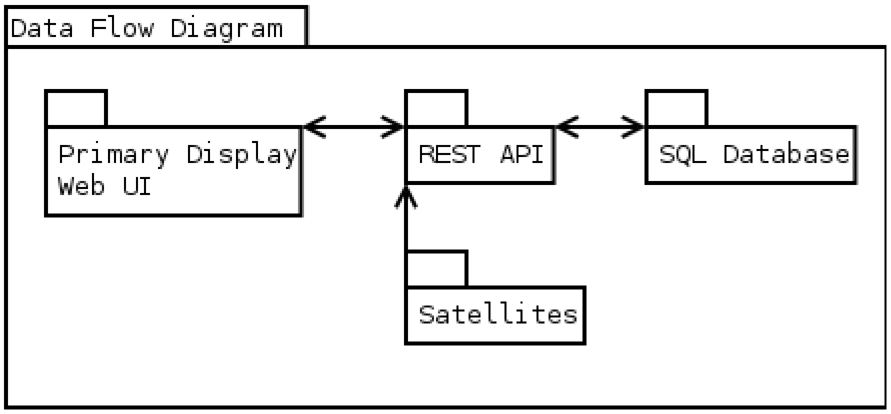

Figure 3.1: Data Flow Diagram

Figure 3.1 shows how various components will be interacting. Notice that the database is not directly accessed by any components, but goes through the **REST! API!** instead. This allows us to provide a standard interface that should be more-or-less accessible from any programming language or system. This will also allow us to use some well-known security features, such as **TLS!** (**TLS!**).

## 3.4 Description of Software Modules

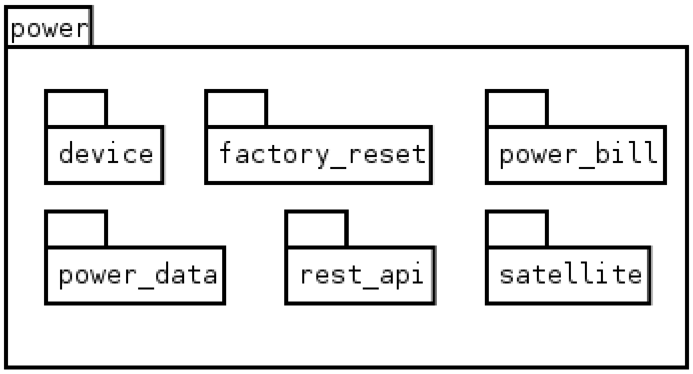

Figure 3.2: Listing of software modules

Figure 3.2 contains all the Django modules that will be created inside the actual Django "website" itself. This is basically the folder hierarchy of our web application source code. The "power" folder consists of folders for each module (not including the various open-source modules we are using, since those are being controlled via **PIP!** (**PIP!**)). Each of these contains a `models.py` file, which specifies what the database for that module should look like, an `__init__.py` file which will register the module with the **REST! API!**, and whatever other files that module needs.

## 3.5 Potential Problems

Using open-source libraries requires us to learn them. This is a big problem because of the time constraints on this project, and will require us to spend almost as much time learning as we will be in development (possibly more). This can lead to frustration, especially in light of the fact that we will need to learn the languages in addition to the libraries.

However, using well developed and designed libraries allows us to skip over a good part of the development and planning. We will be able to use extensive **API!**'s without taking responsibility for maintaining the software. This will allow us to focus on our product instead of the all the small things that make it up.

Another potential problem with using the suggested architecture will be clients that don't like using Javascript in their browser. Using heavy javascript allows us to offset a good bit of the load from our low-powered server to the clients computers, which will have a good deal more power. If we wanted to support clients that don't like using javascript (E.g. security nuts), we would be looking at adding a significant load to our server (text-processing can consume a good bit of resources, especially string manipulation). Although we may be able to do it, for the first version we will simply add "Javascript support" as a requirement for the clients.

## 3.6 The Backend

### 3.6.1 Authentication

#### Description

This module describes the architecture of how the backend will manage features related to users and groups (authentication features).

The backend authentication management provided by Django is very robust. It includes user authentication, access control (including groups and individual permissions), administrative panels, and logging of who does what.

#### Program Flow

The authentication module will be called whenever a user tries to access something. It will verify that the user is logged in, has permissions, and even write a short log message if it is an administrative function.

#### Data Flow

Because this module is not being written by us, it will not be following the standards we are using. Therefore, it will be using the standard Django classes to access database objects instead of going through the **REST! API!**.

While that is not ideal, it should be OK anyway. This part of the Django framework has been tested extensively, in many production environments. There is a low risk that something will not work, and if there is a problem there is a very high chance that upon being reported to the Django project, it will be fixed quickly.

Something worth noting is the way passwords will be handled. Django hashes all passwords in a variety of algorithms, as specified in the password field. It also salts the passwords. This is much better than the lazy way of hashing it (maybe) and storing it in a database column. For more details, please reference `https://docs.djangoproject.com/en/dev/topics/auth/`.

#### Potential Problems

Like I stated above, the largest potentional problems will be bugs in this piece of the code. Because we did not create, and we will not be supporting it, any bugs that we find will need to be reported to the Django framework community and we will have to wait for them to roll out a fix.

Another problem that might be common is the difficulty in expanding on the user profiles. This will not be a problem for us because Django did such a good job in keeping their framework extensible. All we need to do is declare a OneToOneField in our model, and add a few panels to the administrative interface and we will have an instant user profile. Not only that, but we will have successfully separated user information from the authentication information, which is considered good practice as it is faster and more secure.

### 3.6.2 The REST! API!

#### Description

This module describes the architecture of how the backend will manage features related to providing access to the **REST! API!**.

The **REST! API!** will provide a `urls.py` file, which will provide a handler for all **REST!** methods. These could be achieved by simply using the tastypie **URL!** (**URL!**)'s, which can be generated by calling `include(api_v1.urls)` in the urlpatterns object.

#### Program Flow

The **REST! API!** will be activated by the Django chain when a request comes in for the access to the **API!**, or when another module directly calls functions in the **API!** during their response to a request from the user.

Having each module register itself when the system launches should work, but there may be a problem with things getting registered twice.

#### Data Flow

All data will be returned to the calling client either as a Python dictionary (if called directly from another module), or as **JSON!**, **XML!** (**XML!**), or **YAML!** (**YAML!**) through a web request (as specified in the query filter using ?format=json).

The **REST! API!** will directly access the database. Initially, tastypie will be dealing with converting the **HTTP!** (**HTTP!**) request to a method on the

Django objects. Django will be generating the **SQL!** that actually gets run, and encapsulating objects in Python classes.

#### Potential Problems

At first, we will be using a prepackaged Django module to help provide the **REST! API!**. This could be problematic because we don't know a lot about the implementation of the **API!**, and if there is a problem, tracking it down could be difficult.

The good news is that django-tastypie is well supported by the community, and source code is liberally commented. It follows Python commenting standards, and should be fairly easy to navigate. In addition, members of our team have already used tastypie for projects in both work and recreation.

The most problematic bit about using tastypie for the initial run-through is the compatibility problems it has with Backbone.js. However, the community has developed an addition to Backbone to help deal with this. It is called Backbone-tastypie. It is a simple modification to the Backbone sync operations that allows it to communicate with a standard tastypie **REST! API!**.

When the web server is launched, it initialized all modules by calling their `__init__.py` file. In this file, each module can register themselves with the **API!**. This will allow all modules to be self-contained, which is ideal in that it will allow us to upgrade one module without breaking everything else.

Early experiments indicate that the `__init__.py` file is called twice during the loading of the system. This could be solved a number of ways, or it might not even be a problem. It's very possible the first time the module is loaded it is just being checked for syntax and compiled, then the program is restarting with the compiled files.

### 3.6.3 Satellite

#### Description

This module describes the architecture of how the back end will manage features related to managing Satellites.

The Satellite Management back end is in control of maintaining the information in the database pertaining to Satellites. The back end will also generate the framework for the web pages. It will pass the basic form of the Satellite Management page to the front end to be filled in with additional formatting.

### Program Flow

The back end Satellite Management module will be activated by a request from the front end Satellite Management. This module will complete the request by adding information to the **SQL!** database or querying said database for specific data.

### Data Flow

The Data Flow will behave in much the same manner as the Program Flow. The back end Satellite Management module will receive data from the Satellite Management front end via the **REST! API!**. This includes data that needs to be changed in the database, and requests for data. This module will then modify or query the database as needed, returning the requested data or a confirmation of modification to the front end through the **REST! API!**.

### Sub-modules

Communication with the Satellites:
This modules serves as an interface and a form listener. It will receive **POST!** (**POST!**)ed commands for dispatch to the ZigBee network via a serial-port connected ZigBee. It will also receive the incoming data from the ZigBee network and store it in the Database. The data will consist of the current and voltage readings from each Satellite on the network.

## 3.6.4 Devices

### Description

This module describes the architecture of how the back end will manage features related to Device Management.

The Device Management back end will be in charge of maintaining the database with information pertaining to the Devices. The back end will also generate the framework for the web pages. It will send the basic format of the Device Management page to the front end to be filled in with additional formatting.

### Program Flow

The back end Device Management module will be activated by a request from the front end Device Management. This module will complete the request by adding information to the **SQL!** database or querying said database for specific data.

#### Data Flow

The Data Flow will behave in much the same manner as the Program Flow. The back end Device Management module will receive data from the Device Management front end via the **REST! API!**. This includes data that needs to be changed in the database, and requests for data. This module will then modify or query the database as needed, returning the requested data or a confirmation of modification to the front end through the **REST! API!**. The data for the basic format of the web pages will also be sent from the back end to the front end through the **REST! API!**.

#### Sub-modules

The Device-Satellite Association:
Devices will be associated to a Satellite at all times, unless the Device is disabled. That way we can associate the power consumption data from the Satellite with the correct Device. This association will be made in the database. These associations not be static, Devices will be able to be moved from one location to another. By associating the Device with each new Satellite it is physically connected to we can track that Device regardless of its location.

This module will implement the associations between the Devices and the Satellites.

### 3.6.5 Power Bill Guestimator

#### Description

The Power Bill Guestimator backend will be in charge of maintaing a record of utility power costs for different times and usage rates. These rates are set by the user on the front end and stored in the database. It will be able to calculate the cost of the power used between certain time ranges and give that data to the front end to be formatted.

#### Program Flow

Data flows through this module when the front end requests the cost of power used for Satelites. This triggers a query of data from the database of the power usage history for that Satelite (or combination of Satelites) and a query of the rates that applied to the time period. With that data the module will calculate how much power was used at each rate and give that data back to the front end.

#### Potential Problems

Since rates can vary based on time of day, total power used, or other unknown factors, keeping track of rates may be tricky. Research must be done to find out the most common ways utilities charge for power and be able to store rates accordingly.

### 3.6.6 Factory Reset

#### Description

This module describes the architecture of how the backend will manage features related to the factory reset. The factory reset will restore everything to the exact same way it was when the system came out of the "factory". This means that all data will be wiped, and custom modifications will be wiped, and any software upgrade will be wiped.

#### Program Flow

This function will be invoked either through a call passed down from the web interface (it will be handled by the **REST! API!**, and the call will be intercepted and result in an action) or by a program picking up on the push of the physical factory-reset button.

#### Data Flow

Doing the factory restore will involve a good bit of trickery. First, we will need to put a partition on the disk that stores all the files that make up the "live" running system. This might be best done by simply putting a tarball and a script on a small partition at the end of the disk. The script will be responsible for wiping the primary partition, re-creating it, and extracting the tarball.

Getting the system to boot into the "restore" mode will involve modifying grub settings on the fly. We can do this, and it shouldn't be too hard, but it feels like a really bad idea.

After the machine is in recovery mode, something will need to launch the script to do the restore. This shouldn't be too difficult if just use an old-school `rc.conf` script or something of the sort. The script will also restore grub, so we don't have to worry about returning that to it's old good state.

#### Potential Problems

One of the biggest risks here is failing to boot into the restore partition and ruining our grub configuration. This would cause the customer to need to send us back the unit, we would have to flash it, and send it back. If we can get this feature working however, we will not have to worry so much about breaking the web UI by accident.

## 3.7 The Frontend

### 3.7.1 Authentication

#### Description

This module describes the architecture of how the frontend will create pages that allow the user to manage user and groups.

The authentication front end will consist of both the "login" page, and the user management pages. These pages either post directly to the server using standard form-submission methods (ie, not **AJAX!** (**AJAX!**)) or by using **AJAX!** (if we have time to recreate the entire management system).

#### Program Flow

The user will first be presented with a login screen, which will take two pieces of information:

- Username - The user's username
- Password - The user's password

After these two pieces of information have been verified (ie, that password belongs to that user), the user will be logged in.

From here, the user has all kinds of options. The only ones that are interesting to this module is the settings panel, which includes a link to the user administration panel. The user administration panel will, initially, be a themed Django-admin view. Time permitting, this may be recreated to be a **AJAX!** type interface.

#### Data Flow

All data will be sent to the server through **POST!** requests, not through the **REST! API!**. This will allow us to skip over this module almost entirely. This is not consistent with everything else, and will need to be remade later on if there is time.

#### Potential Problems

Themeing the admin interface to match our site will be tricky. At best, it will involve telling Django to include some extra stylesheets or using a different template. At worst, it will involve modifying the Django stylesheets directly. Either option works, but due to version control it would greatly preferred to be able to tell Django what styles to use.

### 3.7.2 Satellite

#### Description

This module describes the architecture of how the frontend will create pages that allow the user to add and delete Satellites associated with the system.

It will receive a basic framework for the web pages in **HTML!** from the back end module and use Javascript to render the rest of the page. The pages will be created with a mix of **HTML!** and **CSS!** from the back end and heavy Javascript from this module. These pages include the Satellite Management page and the Add Satellite wizard.

#### Program Flow

The user will be presented with the option to add or delete a Satellite. Adding a Satellite allows the user to then associate it with a Device. Deleting a Satellite will disassociate the Device from the Satellite, leaving it unassociated to anything. The Satellite will then no longer be available to be associated with a Device.

#### Data Flow

The Satellite Management front end sends a request to the back end through the **REST! API!**. The back end returns either a confirmation of action or the requested data to the front end via the **API!**. This module receives the basic format for the web pages from the back end Satellite module. It then fills in the additional and page-specific content.

### 3.7.3 Devices

#### Description

This module describes the architecture of how the frontend will create pages that allow the user to manage devices in the system.

It will receive a basic framework for the web pages in **HTML!** from the back end module and use Javascript to render the rest of the page. The pages will be created with a mix of **HTML!** and **CSS!** from the back end and heavy Javascript from this module. The Device Management front end includes the Device Management page, Add Devices page, and the Confirm Disable Device dialog.

#### Program Flow

The user will be presented with the option to add, edit, or disable a Device. Adding a Device allows the user to name the new Device and associate it with a specific Satellite. Editing a Device allows the user to change the Satellite associated with the Device. The user will not have the option to edit the name of the Device once it has been created in order to maintain consistant data. Disabling a Device will remove it from all graphs and tables and data will no longer be collected on the Device. Devices cannot be deleted in order to maintain past data.

#### Data Flow

The data will be sent to the Device Management back end module through the **REST! API!**. The back end returns either a confirmation of action or the requested data to the front end via the **API!**. This module receives the basic format for the web pages from the back end Device module. It then fills in the additional and page-specific content.

### 3.7.4 Frontend - View Data

#### Description

This module describes the architecture of how the frontend will create pages that allow the user to view various pieces of data in various formats.

The View Data front end is responsible for providing the user with data regarding each Device and Device Group. It will provide three separate graphs and tables with data specified by the user; Device Power Consumption, Monthly Power Consumption, and Power Consumption Over Time.

This module will be using jqPlot to render the graphs on the web page. This will allow the back end to provide the data for the content and the front end to plot the graphs as needed.

### Program Flow

From the Home page the user will be able to select which graph or table they would like to view. These graphs and tables provide the user with information pertaining to the Devices specified and within the time-frame selected.

### Data Flow

This module requests data for the graphs and tables from the back end through the **REST! API!**. It receives a response and renders the graphs and tables with the data provided.

### Potential Problems

The main sources of concern with this module have been stated previously in Section 3.5. We will be facing a large learning curve as we must familiarize ourselves with the software we will be using, such as jqPlot. There is also the problem where some users will not have Javascript supported in their browsers. This will keep them from being able to view any of the information presented by this module using Javascript or jqPlot. These problems have been addressed previously.

### Sub-modules

Device Power Consumption:
This is a table that will contain the name, current power consumption, and average power consumption rate of each selected Device or Device Group. The content will be sortable by each field.

Monthly Power Consumption:
This is a graph with power in watts on the vertical axis and time in months on the horizontal axis. The user will specify which Devices or Device Groups and the range of months they would like to view data for. The graph will then display a line graph with a separate lines for each separate Device, showing the power usage over time.

Power Consumption Over Time:
This is a graph that allows the user to specify not only the Devices and Device Groups to view, but the time interval as well. The user will specify whether time, on the horizontal axis, will be measured in days, weeks, months, quarters or years. The units of the vertical axis will be power in watts. The graph will then display a line graph with separate lines for each separate Device, showing the power usage over time.

### 3.7.5 Power Bill Guestimator

#### Description

This module describes the architecture of how the frontend will create pages that allow the user to view guestimates of their monthly power bill, in various scenarios.

It will receive a basic framework for the web pages in **HTML!** from the back end module and use Javascript to render the rest of the page. The pages will be created with a mix of **HTML!** and **CSS!** from the back end and heavy Javascript from this module.

#### Program Flow

When the user would like to estimate a power bill they first have to select what devices they want to view and for what time period. With that information selected, this module asks the backend for the power cost data for said devices.

When the user is modifiying the utility company rates they will be presented with a wizard that ill guide them through inputting them. This wizard will ask them several questions about how the utility company charges for power and will give data to the backend accordingly.

#### Potential Problems

One potential problem is having a utility company have rates change in a way we did not anticipate. One other issue is not having rates defined before trying to calculate the bill.

### 3.7.6 Factory Reset

#### Description

This module describes the architecture of how the frontend will create pages that allow the user to reset the system back to factory default settings. It will receive a basic framework for the web pages in **HTML!** from the back end module and use Javascript to render the rest of the page. This front end module will be a page that will allow the user to initiate a Factory Reset.

**Program Flow**

When the user loads this page they will be presented with an option to perform a factory reset. If they choose that option, this module will ask the user if they are sure and list the consiquences of doing so. If the user still agrees, it will request a factory reset to be performed from the back end module.

**Potential Problems**

Making sure only administrators can reset the device to reduce the risk of accidentally breaking it. Also may want to implement a method of backing up and restoring all of the data.

# 4 | The Server

### 4.0.7 Description

The server will be responsible for collecting and recording data given to it by the Satellites, storing the display, and responding to user requests. This section reiterates the hardware of the server, which is also documented in the hardware section because so many parts of it belong specifically to the Zigbee things.

### 4.0.8 The Hardware

#### Mainboard

The Server's main hardware component is the mainboard, a SYS9400-ECX Developer-Ready Reference Platform. It's a small form-factor, low-power machine with the following specs:

- 1.6 GHz Intel Atom E6XX Series Processor
- 1 GB DDR2 RAM
- Roughly 6" by 4"
- Various connection interfaces:
  - 2x **SATA!** ports
  - Header for **SATA!** power
  - Ethernet port
  - 5x **USB!** 2.0 ports
  - **GPIO!** pin header

#### Potential Problems

If for whatever reason using this mainboard falls through: It should be noted that the requirements for the Server hardware concern not just specifications, but interfaces as well. In particular, this project requires at least Ethernet, 2 **USB!** ports, a **SATA!** port, and accessible **GPIO!** pins.

#### Storage

The Server's mainboard is connected via **SATA!** to a 40GB **SSD!**.

#### Power Supply

An adapter rated for 12**VDC!** @ 3**A!** is used to connect the Server's board to mains electricity.

### 4.0.9 Software

In addition to storing our custom written software, which is documented in the software section, the server will utilize a good number of other programs. This section will document what software will be used, and for what purpose.

#### Ubuntu Server 12.04

The base system running on the server will a Ubuntu server base install. We will be installing additional packages on top of this, using the aptitude package manager.

The filesystem will contain 3 partitions.

- /boot - This partition will store the files needed to boot the system, including the Linux image and grub configuration files
- / - This partition will store all the files used during run time, including configuration files, software, and static data
- <Restore> - This partition will contain a minimal boot system, and a tarball will all the factory default software. It will be used to restore the /boot and / systems when requested, but will not normally be mounted

#### Apache2

We will use an Apache2 server with `mod_wsgi` to deal with responding to requests on the **REST! API!** or executing parts of the web application. This server will not be server static files, as that would be a waste of it's features (which includes caching of recently accessed pages to avoid re-executing the same code several times over).

### lighttpd

The lighttpd is also a web server. This web server is much less robust than Apache, but much faster at serving up static files. It will be responsible for giving images, javascript, css, html, and other static content to the user.

### MySQL

Initially, we will use a MySQL database to store data from the Satellites. This may be changed to posgresql if performance becomes an issue, but MySQL is much easier to install and configure.

### Python

To run our application, we will need Python 2.7 installed. We will have a whole bunch of modules, which will be discovered as we are doing the lower level design and implementation, installed along with Python. To track packages we need for Python, we will use pip. In addition to pip, we will use virtualenv to simply the maintance and portability issues of using Python + pip in place of the aptitude package manager. We are not using the aptitude package manager because it does not have the most recent PyPy packages.

### IP! Tables

**IP!** tables will be used to prevent traffic from bridging the server, and to prevent the server from sending out requests anywhere. This will prevent a wanna-be-hacker from using the machine to attack the client's network. Note that this will not help if a hacker got root-access to the Server, but it is better than nothing.

# List of Figures

## 4.1 Acronyms

| | |
|---|---|
| **A** | Amps |
| **AJAX** | Asynchronous JavaScript and XML |
| **API** | Application programming interface |
| **CSS** | Cascading Style Sheets |
| **GPIO** | General Purpose Input Output |
| **HTML** | HyperText Markup Language |
| **HTTP** | Hypertext Transfer Protocol |
| **IEEE** | Institute of Electrical and Electronics Engineers |
| **IP** | Internat Protocol |
| **JSON** | JavaScript Object Notation |
| **LCD** | Liquid Crystal Display |
| **M** | Meters |
| **NEMA** | National Electrical Manufacturers Association |
| **PIP** | Python package installer |
| **POST** | Request Method |
| **POW-R** | Power Outlet Wireless Reporter |
| **REST** | REpresentational State Transfer |
| **SATA** | Serial ATA |
| **SQL** | Structured Query Language |
| **SSD** | Solid State Srive |
| **TLS** | Transport Layer Security |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **USB** | Universal Serial Bus |
| **VDC** | Volts DC (Direct Current) |
| **W** | Watts |
| **XML** | Extensible Markup Language |
| **XSS** | Cross-site scripting |
| **YAML** | YAML Ain't Markup Language |