5-2014

Smart Utility Poles

Jason Yen-Shen Chan
jchan5@utk.edu

Team 5 – Smart Utility Pole

Final Report

Josh Clark – Lead
Jason Chan
Zach Fisher
Will Thatcher
Alex Wetherington

Executive Summary

The Smart Utility Pole project is an effort sponsored by Dr. John Simmins at the Knoxville location of EPRI, the Electronic Power Research Institute. The goal of the project is to develop an open platform that can support smart grid functionality. The platform is intended to be mounted at the top of a utility pole, monitor changes in loads at transformers, and report events such as fallen utility poles. The system can then communicate its status to a base station (most likely a local substation) and can be designed to take corrective actions or alert the utility company that repairs need to be made. The system will be designed to be very flexible, and will allow the end user to develop applications for both individual platforms and the system as a whole.

Requirements

1. Hardware Platform

A. The hardware platform must be able to be mounted on a utility pole.

B. The hardware platform must be able to fit in a final assembly that is less than one foot wide and long, and less than four inches tall.

C. The hardware platform will rely on power drawn from 120/240 V power lines, or from other sources of energy harvested (E.g., solar, vibrational, etc.). It will not be possible to obtain power from voltage sources higher than 240 V for safety reasons.

a. For the purposes of this project, the system can run off of power from a 120 V wall socket

b. The power requirements must not exceed what could be sourced in the field.

D. The hardware platform must be capable of running a real-time software platform as described in section 2.

E. The hardware platform will be an open, modular design that can support multiple functions related to smart grid technology.

F. The system must be able to run for 3 hours in the event of a power failure.

2. Software platform

A. The software platform must be able to respond to input from a central computer or from other platforms in real time.

B. The software platform must allow the utility company to easily develop, test, and deploy software applications on the system.

C. The software platform must allow for the utility company to access the system by logging in both remotely and locally.

D. The software platform will use open source software to implement the above features, although the final product and any applications developed for the system do not have to be open source.

3. Networking

A. The system must be capable of communicating with a central computer and other systems of the same design deployed in the field.

B. The system must be capable of communicating through multiple interfaces.

a. For the purposes of this project, proof that the system can network successfully over two interfaces is sufficient.

C. The network must be resilient and be able to recover in the event of a failure in the network.

4. Sensors

A. The system must be able to detect if the utility pole it is attached to is in good condition.

B. The system must be able to detect if there are any problems within the system itself.

C. The system will allow for the utility company to install additional external components that the system can interface with.

a. The system will provide a standard set of interfaces over which the external components can communicate to the system. (E.g., USB, Ethernet, I2C, serial ports, etc.).

D. The system will be able to communicate sensor and external component data to the central computer or to other similar systems in the field.

Research, Prototyping and Evaluation

Research – Hardware

One of our biggest concerns at the start of the project was the level of power that the system would run on. Because we were unsure of the availability of power for our system, we came up with three platform concepts – a low-power, medium-power, and high-power design – and let our sponsor determine which of the platforms would be best suited to meet the requirements. The three platforms were as follows:

Low Power Model:

- 1x Arduino Uno
- 1x Raspberry Pi Model A
- 1x XBee Radio
- 1x XBee Arduino Shield
- 1x USB A-B Cable

The low-power model is designed to be able to run entirely off of solar-charged batteries. Each system would be controlled by a low-power Arduino Uno that is connected to a wireless mesh network using the ZigBee wireless mesh protocols. The Arduino would control the Raspberry Pi, waking the Pi up from sleep when it needs to process a large amount of input. When the Pi is not needed, the Arduino is still capable of reading sensors, communicating with simple components, and maintaining the wireless mesh network. This would allow the system to communicate its status and respond quickly to simple stimuli while consuming less than 0.5 W, a power level easily sustainable by solar panels. One disadvantage of this model is that it would not be possible to run the Pi continuously, which would add a significant processing delay into the system.

Medium Power Model:

- 2x Raspberry Pi Model B
- 1x Ethernet Cable
- 1x USB Wi-Fi Dongle

The medium-power model is designed to respond quickly to external events and process input faster than the low-power model without the large power requirements of the high-power model. The

system would consist of two Raspberry Pis, one of which maintains the wireless mesh network using the USB Wi-Fi dongle while the other Pi performs sensing and command-and-control functionality. Both Pis are capable of running more powerful software than the low power systems. If it were determined that more processing power were necessary, a third Pi could be inserted into the system as well as an Ethernet switch to network the three boards. If the system were required to run at lower power, it could be possible to put one of the boards in sleep mode and wake it when necessary. The benefit of this approach is that it is a very flexible and powerful platform, although it requires significantly more power than the low-power model. Another alternative for this model is to use BeagleBone Blacks instead of Raspberry Pis.

High-Power Model:

- 1x UDOO Quad Core

The high-power model is designed to be an easy, all-in-one solution for a processing platform. The UDOO board has all of the components necessary to meet the requirements of the system, including a built-in Wi-Fi card, plenty of I/O ports, and multiple processor cores for fast processing. However, the UDOO has high power requirements and runs at 12 VDC as opposed the 5 VDC of the other two systems. In addition, the four processors are very likely overkill in many situations, and there is no way to reduce power consumption of the board.

Comparison of the three models:

Model	Pros	Cons
Low-Power	• Sustainable by solar power • Well suited for small sensor networks	• Very complex system design • Cannot run a full OS • Solar power is more expensive than rest of system
Medium-Power	• Very flexible • Capable of responding in real-time • Can run a full OS	• Requires a steady source of power • Overhead from networking multiple boards in system
High-Power	• Very simple system design • Capable of responding in real-time • Can run a full OS	• High power consumption • Not as flexible as other options

It was decided that the medium-power model would best suit the requirements of this project because of its flexibility and processing power.

As part of the design described above, we were deciding between two boards to build our system with: the Raspberry Pi Model B and the BeagleBone Black. Both are small, single-board computers capable of running Linux. The Raspberry Pi Model B has a 700 MHz ARM processor, 512 MB of RAM, and 26 GPIO (general purpose input output) pins. The BeagleBone Black has a 1 GHz ARM processor, 512 MB of RAM, and 92 GPIO pins. Both boards have Ethernet, USB, and SD card capabilities. The BeagleBone black consumes 210-460 mA @ 5 V, and the Raspberry Pi consumes 160-360 mA @ 5V. The BeagleBone is slightly more powerful than the Raspberry Pi, at the cost of consuming slightly more power. We determined that a Raspberry Pi should be powerful enough to meet the needs of this project, and given the flexibility of our design, more Raspberry Pis could always be added to a system if more processing power were needed.

The boards will process data from various sensors and transmit this processed data to a base station which can store the data. For our the default set of sensors installed on every platform, we have

decided to use a current sensor, a gyroscope, and a temperature and humidity sensor. The current sensor measures the power draw of the system to ensure that it is working properly. The gyroscope and the temperature and humidity sensor work to monitor the status of the platform. They help ensure that the system is working properly; that it is not overheating and that the platform (and the pole) is upright and stable. Additional components and sensors can be added over a variety of interfaces to add more functionality to the system.

The boards will be networked through an Ethernet switch and powered by a USB hub sourced by a wall adapter. Obviously, the USB power adapter is just used for the purpose of prototyping the system and will be replaced by a different power supply in the field. One of the boards in the system will have a Wi-Fi dongle so that the systems can communicate wirelessly if necessary. Otherwise, the system will communicate to other platforms through its Ethernet interface.

Research – Software

When considering options for the design of the product's software component, we needed to provide an environment for the end user that would allow for extra components (such as a camera or panic button) to be added onto the system and seamlessly utilize the network to perform desired tasks. For this reason, we initially considered Android on Raspberry Pi. The Android OS was a desirable solution to the problem, as it naturally allows for easy and safe development of apps by the end user. However, as it turned out, there are significant compatibility issues for this combination, partially stemming from the fact that Raspberry Pis do not provide graphics acceleration, and the Android OS relies heavily on graphics. Even if the decision was made to use BeagleBone Black, our hardware alternative, instead of Raspberry Pis, we would still run into many of the same issues.

After encountering this roadblock, we began to look into possible alternative options that would provide similar ease of development and safety as Android would have. As a result of this search, we are now considering using Raspbian, a form of Debian made by the creators of Raspberry Pis, which, as the name suggests, is optimized to run on a Raspberry Pi. Because of this, it has a larger community than that for running Android on Raspberry Pis, and thus more available resources.

Since we no longer have the development environment provided by Android to pass to the end

user, we needed to consider a method to emulate such an environment for our system. What we are currently looking at is writing a system controller in Java that listens for new components added to the system, and starts them up. A plug-in interface will be provided to the user, and he or she will be able to create a class that implements the interface and provide the system controller with his or her custom class file. The controller will then read the file, execute the user's program through the interface based on certain triggers specified by the user, and broadcast any data collected by the new components through the network. Finally, the system controller will encourage good programming practices by enforcing logging and standardizing access to sensors.

For the software design of the project, we started with a concept where there was a single base station connected to all poles through a self-healing wireless mesh network. Each pole had both a sensing unit and a networking unit. On some given pole, there would be a primary thread managing all plugin threads, which it would fork off of itself based on the available plugins installed at runtime. These plugins would be executed according to their own code, however their access to the sensors would be restricted by a user interface with the sensors, in order to prevent race conditions from appearing within sensor writing and reading. Whenever there was information the sensing unit wanted to send to the base station, whether by the plugin execution code or an explicit request from the base station, the sensed values are transferred to the networking unit, which then sends it back to the base station. From here, the base station obtains the information and updates a database accordingly.

After some prototyping and research, it was determined that this design would indeed be feasible without significant modification. The most signifiant change was a result of our choice of networking method, and was that some given base station would instead be capable of becoming a sub-station that connects to a central station. As it turns out, this change was a more favorable one for more than simply bringing the product to a usable state, but additionally enables the system to be expanded on to utilize existing protocols put in-place by utility companies.

Research – Networking

Our ultimate design needed to feature a system which could communicate over long distances in a manner that was inexpensive, unrestrained, and reliable. Our first inclination was to use a mesh

topology or a daisy chain of devices to wirelessly transmit data across the network. We ultimately decided that a wireless mesh topology could meet all of our needs if we were able to find hardware to support such a design and if the protocols were robust.

The first implementation we researched utilized WiFi as the backbone of the network. There are a number of WiFi devices that are supported by the Linux kernel and many of these are relatively inexpensive. As for the mesh networking protocols, there were a number of articles citing the viability of open source routing protocols built for Linux such as Babel, B.A.T.M.A.N, etc. If we could build on top of these protocols, we would meet all of the requirements for the network. Once research on this route turned up positive we decided to build a prototype using very cheap WiFi dongles and the Babel protocol.

Building the prototypes went smoothly. The WiFi dongles were easily recognized by Linux and the Babel software was available in the Raspbian repositories. This made installation as easy as plugging into a USB port, installing software from the repositories with the Debian Aptitude package manager, and running the software.

Unfortunately, evaluation of the prototype revealed numerous issues with this design. The routing protocols were running but no routes to other nodes could be found. After tedious testing, hacking, and more research it was discovered that much of the Linux support from the manufacturer of our WiFi dongles had disappeared. Open source drivers had replaced it in the current kernel and we could not revert to an older kernel version without risking the integrity of our other components. The specific issue was that most inexpensive chipsets no longer support the Ad-Hoc protocols which are relied upon by all mesh networks. If we wanted to continue working with WiFi, we would have to spend more money on standalone wall powered dongles with high gain antennae.

Our second design choice looked back to the very beginning of the project at one of the early implementations of the entire system that we threw out. We had considered a low power option of our Smart Utility Pole box which used a low-power Arduino and ZigBee device to maintain the network

while the high-power Raspberry Pi was only woken up to do any significant processing work. Our second design would utilize the ZigBee device if we could find some way to let it communicate with the Raspberry Pi.

Research into that design turned up the XBee board along with its USB explorer dongle which allows the RF board to snap into headers on the dongle. The dongle can then be plugged in to any USB port and the whole device is then accessed through device files.

XBee is a module which implements the 802.15.4 or ZigBee protocols designed by Digi International Inc., heretofore referred to as Digi. The 802.15.4 protocol is a point-to-point protocol which requires a network architecture that is commonly found today. There is one coordinator node, several router nodes, and even more end device nodes. The end devices are write-only devices which collect information that is then routed back to the coordinator node through any number of router nodes in the network.

ZigBee is a mesh network protocol where there is again one coordinator node. However, each end device is also a router node. These nodes can send and receive data amongst themselves in a mesh pattern which allows the network to route any node failures within the network. Like our first design with the WiFi dongles, we needed a mesh architecture that would allow us to communicate across long distances, between nodes who were not in range of each other.

XBee denotes the difference in 802.15.4 radios and ZigBee radios by using a series number. Series 1 is the former protocol and Series 2 the latter. Additionally, within each series there is a normal module and a PRO module. The PRO provides much greater range (~1 mile) at the expense of power drawn.

Building the prototype was again very straightforward. The XBee devices had to be flashed with firmware using Digi's proprietary Windows software called X-CTU. We were able to run X-CTU in Linux in the Wine windows emulation program. Once the devices were configured, we were able to

read and write to each board by sending data to and listening for data from the device file that the dongle was running under.

Evaluation of this design turned out to be positive and as more research was done, multiple libraries for high level programming languages were found that enabled us to communicate across the ZigBee network without having to worry about complex RXTX commands.

Design

Design – Hardware

The purpose of the system was to take measurements regarding the condition of the utility pole and relay the information back to a base station. To accomplish this we used various sensors to gather the data, single board computers to process and relay this data, radio modules to handle the communication between boards, and a network router to manage information flow.

The main component of the system is a Raspberry Pi Model B single board computer. It contains a 700 MHz CPU [1]. It powers at 5 V with 700 mA running 3.5 W of power and hosts two USB 2.0 ports, HDMI output, audio and video outputs, and hosts 26 GPIO pins for I/O capability [2]. It also features an SD card slot from which the operating system is read [1]. The size of the board itself is 85.0 mm x 56.0 mm making it suitable for an embedded project such as ours [1]. The project requires a low-power computing platform for monitoring and communicating, and the Raspberry Pi fits the criteria by being a powerful yet cheap computer.

The sensors that were used in the system include an ACS712 30 Amp Current Sensor on a breakout board, an Arduino GY-521 MPU-6050 Module 3-Axial Gyroscope/Accelerometer module, a DS18B20 Digital Temperature Sensor. Additional hardware used includes an ADS1015 12-Bit 4-Channel ADC, and an Xbee Pro Series 2B ZigBee Radio Module.

The ACS712 current sensor is a fully integrated, hall-effect based linear current sensor powered by 5 V which can sense currents ranging -30 A to +30 A with a sensitivity of 66 mV/A [3]. It outputs a voltage which corresponds with the amount of current flowing through its sampling, establishing a baseline of half of the supply voltage powering the IC when no current is being read (i.e., for a power supply of 5 V, when reading no current, it will output 2.5 V) [3]. For the project, the current sensor was intended to be used to read how much current was being driven into the overall system to measure its power consumption.

The Arduino GY-521 is a breakout board which contains an MPU-6050 sensor [4]. The sensor houses a 3-Axial Gyroscope and a 3-Axial Accelerometer, and the breakout board itself houses a voltage regulator allowing it to be powered from 3.3 V to 5 V [4]. The values can be read via I^2C, and with the built in 16-bit A/D converter, it allows for a wide range of values to be read [5]. Through testing, it was discovered that it also contained a temperature sensor. Although not intended to be used for temperature sensing, this components was tested against the DS18B20 Digital Temperature to determine which was a more capable device. The results of this testing can be found later in this report. This intended use of this device is to monitor the system's orientation to determine whether the utility pole is upright, leaning, or has fallen.

The DS18B20 Digital Temperature Sensor is a 1-wire sensor which can read from -55 to 125°C with up to 12 bits of resolution and an accuracy of approximately +- 0.5°C [6]. The device can be powered from 3 V to 5 V and it has a maximum conversion time of 750 ms [7]. This device is used to determine the temperature of the system. Overheating of the system could cause poor performance so it is imperative that this device works properly. The results of testing device can be found later in this report.

The ADS1015 ADC is a 12-bit precision analog-to-digital converter hosting 4 input channels and allows for sampling at up to 3300 samples/second through I^2C [8]. The entire module can be powered from 2 V to 5.5 V and can be set to sample continuously at 150 uA or in a single-shot mode, which allows for it to be powered down after sampling [9]. The ADC translates analog signals, such as the ones from the current sensor, into a digital signal which can be used by the Raspberry Pi.

During initial designs of the board, it was discovered that the Linux kernel running on the Raspberry Pis does not support mesh networking via Wi-Fi. To create a mesh network, the communication was done through Zigbee due to its range and low power consumption. The XBee Pro Series 2B radio module (XBP24-BZ7PIT-004) manufactured by Digi International allows for Zigbee communication which was essential to the project [10]. The module has a range of up to 2 miles in an outdoor

environment, allows for an RF data rate of 250 Kbits per second, and operates at 220 mA, or 63 mW [11]. The module for the project utilized a PCB antenna which allows for reliable communication without a cumbersome receiver [10]. A Sparkfun XBee Explorer dongle was also utilized to allow the module to be powered through a USB port rather than through the Raspberry Pi's GPIO pins [12]. This was to ensure that the device received adequate power. It required 3.3V input and we didn't yet know what the power consumption of our other devices would be.

Other miscellaneous hardware used for the project included USB A-A micro cables to power the boards, SD cards, a wireless networking router, four Edimax wireless USB dongles, two 4-port USB hubs, and two USB AC/DC wall adapters. These components help with powering the entire system as well as communication between the Pis and a base station.

Design – Software

The main purpose of the software platform is to provide an easy to use framework that incorporates all of the common features of applications that would run on this platform. We debated for a long time over what features should be included in this framework, as including too much would put too much burden on our schedule and including too little would lead to a hard to use system. We eventually decided that the framework should consist of three major interfaces: an interface through which custom user code is executed and monitored, an interface for communicating with the various peripherals connected to the system, and an interface for networking with the other Pis inside and outside the box. These three systems would provide the end-user an easy way to develop plug-ins that are simple and easy to use without putting too much burden on our team.

The first system mentioned loads custom user code in what we call "plug-ins." A plug-in is simply a Java source file that has been compiled against our Smart Pole software library and packaged into a jar (Java archive) file. This jar file can then be reused on any platform (assuming the framework

has not been modified) without having to rebuild the plug-in. All the user has to do is create a class that implements a simple interface in Java and include it in the build path. This allows our Plug-in Manager to find, load, and report data from the plug-in. This is done using a few methods in the plug-in class. First, the plug-in is loaded and validated. This process is done automatically by Java when we load in the jar file. The first action that the plug-in will take is the "execute" method. This method defines all the actions taken by the plug-in during its normal life. Most plug-ins are design with an infinite loop inside the execute method that repeatedly perform a simple action, and only stops when a status variable has been unset. This status variable is typically set by the Plug-in Manager when it receives a notification that the plug-in needs to stop (e.g., the jar file has been deleted), but it can be set in the execute method if the plug-in has entered an unrecoverable state. The plug-in interface also includes methods that return status and logging information. The plug-in can choose to implement these however they want, but our framework currently expects both of these methods to return Json objects that contain only one level key-value pairs. These methods are called periodically throughout the lifetime of the plug-in by the main thread to report the status of each plug-in back to the base station.

Another major component of the software is the IO framework that allows thread-safe access to peripherals such as I2C or 1-wire devices. The IO framework is designed with a simple model: only one thread can be communicating with a peripheral at a time because the communication has to go over a physical bus. In order to implement this thread-safe access pattern, each of the device types has its own singleton class that actually performs the reads and writes. Then there is a "generic device" class that accesses the device through the singleton class. Finally, the framework includes several "physical device" classes that are subclasses of the "generic device" class. These physical classes represent actual devices that access the peripherals such as the gyroscope or temperature sensor. With this pattern, it is ensured that there is only one class (the singleton class) actually communicating with a bus, and the singleton pattern enforces the rule that there can be only one instantiation of the singleton class. So it is possible to achieve thread-safe access with many threads running. One improvement that could be

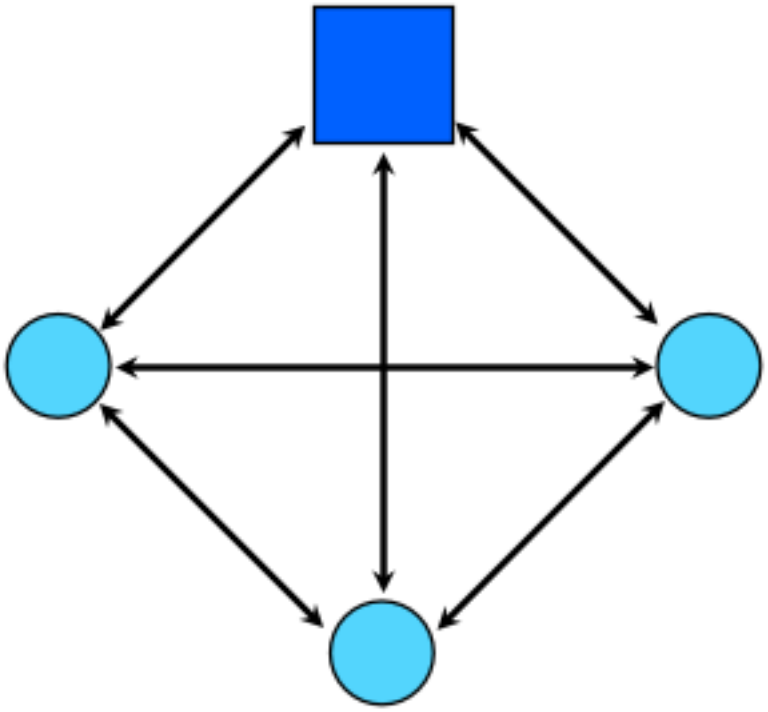
made to this system include implementing special read/write locks that can determine if it is safe for two devices to interleave their commands (if they are communicating with different devices, or are communicating with the same device but not changing its state). In the final project, we only implemented the I2C and 1-wire serial interfaces, but it would be simple to implement a new interface for a different protocol using our existing libraries as a starting point.

The final part of the system involves networking the Pis together. This consists of two parts: communication with the Router Pi from within a system, and communicating between a Router Pi and the Control Pi. The second type of communication is covered in the networking section, so we will focus on the first type. All communication between Pis within a system is done using XML-RPC. XML-RPC is a protocol for specifying, querying, and executing remote procedures on a server. In our project, each Pi is running an XML-RPC server that other Pis can communicate with. However, the communication is only designed to work between the Router Pi and a Sensor Pi. It is certainly possible for two Sensor Pis to communicate over XML-RPC, but the end user would have to add code to the framework to accomplish this. At the moment, each XML-RPC server loads one of two sets of functions, those for a Router Pi or those for a Sensor Pi. This is determined when the server starts up by a configuration file on each system. Basically, each Pi determines if its IP address matches the Router Pi's IP address (both are hard-coded in the configuration file) and loads the proper set of functions. This configuration allows the Routers to request information (status, errors, logs, etc) from each of the Sensor Pis and allows the Sensor Pis to alert the Router Pi in the event of a critical failure. One benefit of using XML-RPC over other, possibly more lightweight methods is that using XML-RPC allows us to keep all the interactions in the same address space as the main process, which simplifies development. The XML-RPC functionality could easily be replaced with a low-level socket-based interface, although the XML-RPC interface is very easy to use and extend.

One important feature of the software platform is that is designed to be easy to reconfigure or extend. Our design highlights these features by allowing easy development of end-user applications

(plug-ins) and easy configuration for each individual Pi in the system (each Pi has its own configuration file that determines how certain plug-ins behave). It is important to point out that every Pi in our tests was running the same software platform. The configuration files determined the behavior of Pi in the system, and including different plug-ins allowed us to create specific behavior when necessary. If at any time we had swapped the configuration files and plug-ins on any two Pis, the system would work just as well as before.

Design – Network



The above diagram shows the basic mesh topology of the network design. The square is our base station which contains the coordinator node of the network while the circles denote each of the sensor nodes which route information back to the base station. To accomplish this design we chose to have each Smart Utility Pole box contain a Raspberry Pi responsible for maintaining the network and another Raspberry Pi responsible for listening to its sensors.

Within each box the two Raspberry Pis are connected via Ethernet cat5e cables across a basic Ethernet switch. Communications are then handled with XML-RPC which allows us to keep all of our software in Java.

Continuing with that trend, we wanted to be able to keep our ZigBee communication in Java as well. One of the aforementioned software libraries we discovered during research was xbee-api. Andrew Rapp developed this library and it handles a significant amount of the low level communication with the XBee module.

The xbee-api package provides for things like packet acknowledgement, logging, etc. but there is maximum packet size of 256. Because of that a new Java class was developed that can parse large payloads into individual data segments that the XBees are capable of handling. The structure of the new data segments constructed in our software is modeled by the following:

1 byte - packet size (max 245)

1 byte - packet type (TODO)

1 byte - data compression type (TODO)

4 bytes - packet number (max 16777215)

4 bytes - total number of packets (max 16777215)

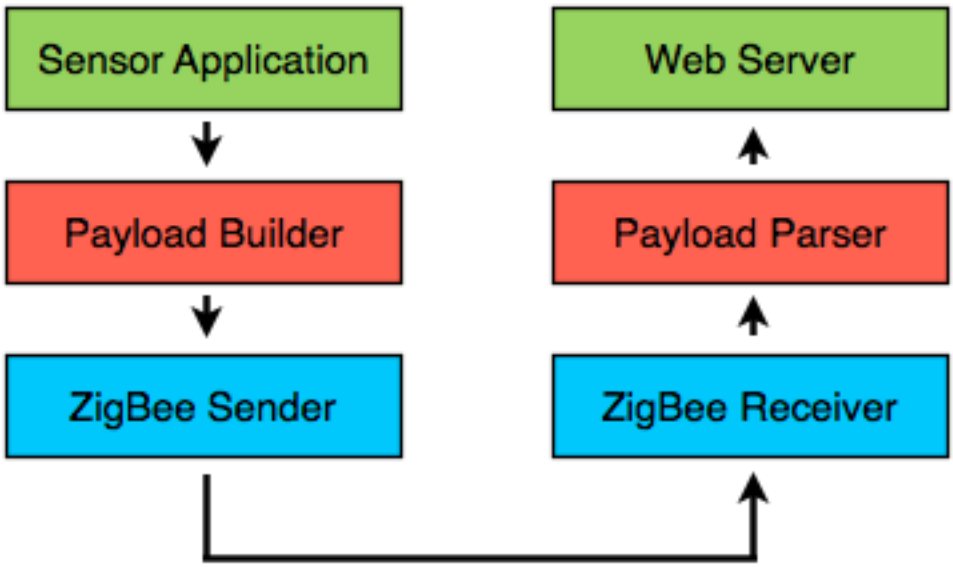
245 bytes - data segment

The construction method takes a string as data and parses it into these segments and then sends them via the xbee-api built in sending classes.

On the other side of the network, we designed a threaded class that takes packets from the xbee-api receiving code. This receiver method is always listening on the base station system. Our custom class spawns a new thread when a packet is received and that thread either assigns the packet to its respective packet queue or, if it is the last packet meant to go in that queue, rebuilds the data.

The internal data structure used is a Java HashMap keyed on the ZigBee address of the sender. The value associated with each key is a list of the data received. When a packet is received, we already

know the expected number of packets. Once the size of the list matches that number, each data segment is turned from its binary representation into a string. Once each packet has been turned into a string and all of those strings put together, the data is sent to the appropriate application which is decided by the packet type. There is a network diagram detailing the flow of data on the following diagram.



As you can see, data is generated by the sensor applications. The data is then passed to the payload building/parsing layer where we add all of our metadata to the packets. After that the data is passed to the ZigBee layer where xbee-api and the ZigBee protocol take care of sending the data back and forth.

We chose to display the data on a web dashboard just to easily demonstrate that we can detect problems and then use that data at the base station. However, this end application can really be anything. Because the data is in ASCII string format when it comes out of the network, it can be written to files, inserted into a database as we have done, or passed to other applications that the end user designs.

Our network design really embraces the “open platform” mentality we have strived for with this project. We simply are able to take data from any sensing node and communicate it back to the base

station in a format that any application can use. We aren't concerned with what the top layer applications might be. We are simply concerned with getting those applications' data where it needs to go.

Testing

The Smart Pole system consists of three major areas – the hardware that makes up the platform, the software that runs on each Raspberry Pi, and the network formed by connecting multiple platforms. In order to fully test our system, we had to test and validate all of the hardware components individually before integrating them into the system in software. Then we had to test the software and network individual platforms to create the Smart Pole system as a whole. This report covers the testing that we have done and the testing that will need to be done in the future to ensure that the system can function as intended.

Hardware Testing

The hardware platform consists of several components that the rest of the system relies on to function properly. These components must be validated before it is possible to test the rest of the system. This report will focus on testing the various sensors used in our project: the DS18B20 temperature sensor, the GY-521 MPU-6050 Gyroscope/Accelerometer board, the ADS1015 analog to digital converter, and the circuit designed to measure the power line voltage. The other hardware components in the system (an Ethernet switch, a ZigBee radio, and an SD card) are proven technologies and will be tested as part of the system as a whole in the software and networking sections.

From the start, it should be noted that the USB hub that was powering the Raspberry Pis was causing the voltage to drop from 5 V to 4.73 V. While testing the current sensor, there were some notable problems. While the power was being supplied to the board, the resistance across the sensing terminals across the Raspberry Pi was causing the voltage to drop even further. Because of this, the Raspberry Pi that was used for sensing received inadequate power for long term operation. Thus, it was decided that the current sensor was to be removed from the system. If the current sensor were to be integrated into the design, it is highly recommended that a high quality power supply be utilized to power the sensor and the rest of the boards.

Temperature Sensor

The Smart Pole platform has a temperature sensor to monitor the temperature inside the project enclosure and report an alert if it is outside operating conditions. Although more accurate temperature tests of the complete, encased system will need to be conducted by EPRI over the summer and in future semesters, we conducted high-temperature experiments with a Delta Design 9023 Environmental Test Chamber. This was done to test the upper range and accuracy of the DS18B20 Digital Temperature Sensor and the temperature sensor on the GY-521 MPU-6050 Gyroscope/Accelerometer chip, as opposed to stress-testing the system.

To test the temperature sensor, we placed a Raspberry Pi and the two connected, aforementioned sensors into the heat chamber. The temperature in the heat chamber was set to 45°C and was increased in increments of 5°C up to 65°C, each time waiting for the sensors' values to stabilize before recording. A graph detailing the results of the experiment is shown in Figure 1, displaying the graphical data for the entire range of the test. Figure 2 provides a closer look on the higher temperatures highlighting the disparity between the two sensors' values.

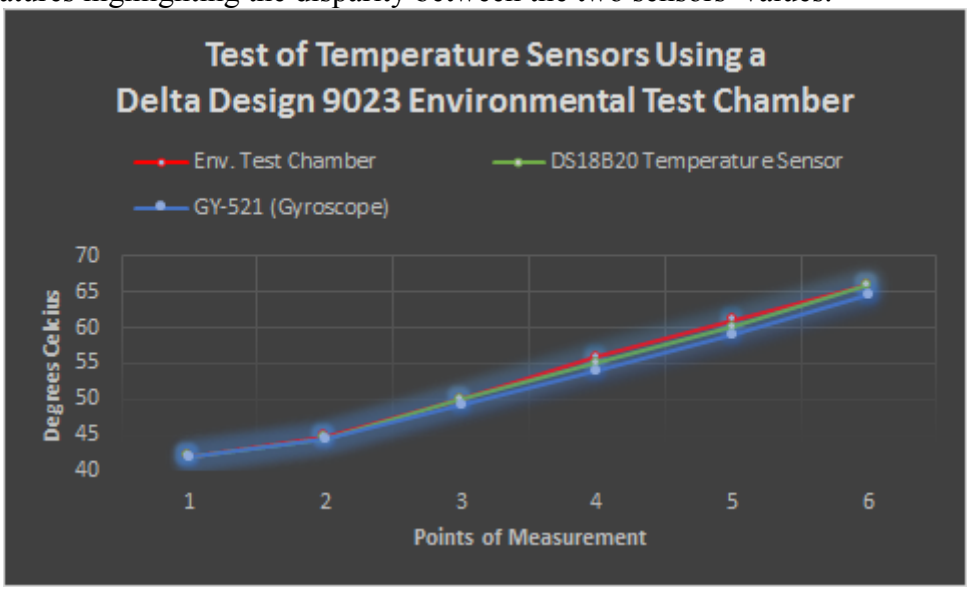


Figure 1: Complete Temperature Test Results

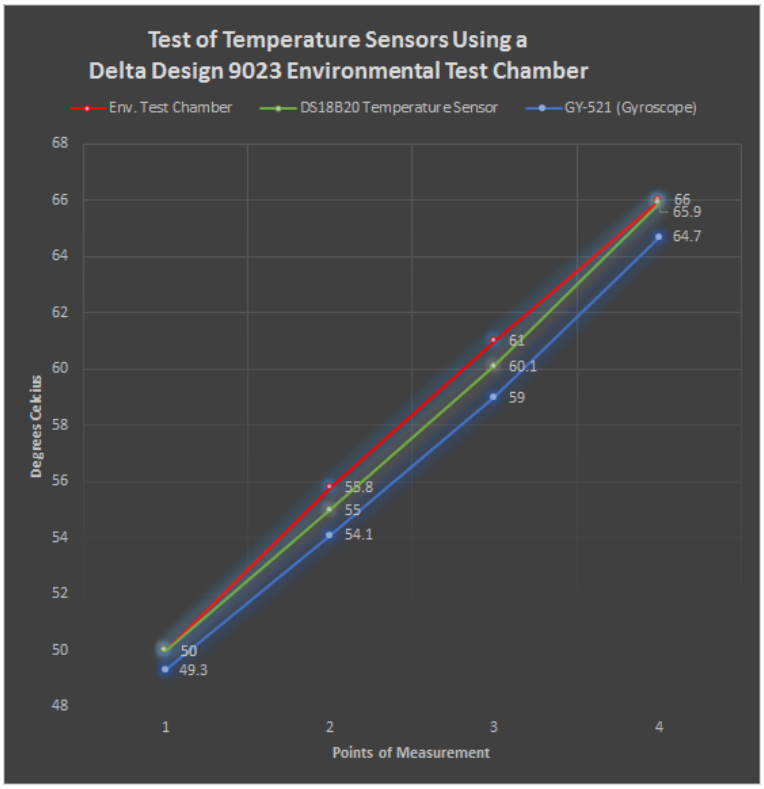


Figure 2: Top Temperature Results

The DS18B20 Digital Temperature Sensor proved to be the more accurate and more quickly adapting of the two. The temperature sensor in the GY-521 MPU-6050 Gyroscope/Accelerometer proved to be less accurate and slower to adjust. Based on these results, it is recommended that the DS18B20 Digital Temperature Sensor be used as the primary temperature sensor of the unit. In this endeavor, a sensor which can change more rapidly and read more accurately is the better option of the two.

Given the current environment and equipment, it was not possible to conduct a low temperature test during the course of this project. Because the recommended operating temperature of a Raspberry Pi is between 0°C and 70°C, it is highly recommended that EPRI follow up on our high temperature test with a lower temperature test. The project sponsor, Dr. John Simmins, informed the team that EPRI owns a type of cold environment chamber which is used for testing of this nature. To conduct a more realistic test, the unit should be encased in its container, which EPRI plans to make, in order to trap the heat generated from the components themselves. This will help evaluate a more complete system rather than just exposing a bare system and its components to frigid temperatures. This test will simulate a complete unit mounted on a utility pole undergoing the harsh conditions of winter.

Gyroscope/Accelerometer

The Smart Pole platform contains a Gyroscope/Accelerometer to determine if the utility pole is standing upright. To date, the only testing done on the GY-521 MPU-6050 Gyroscope/Accelerometer has been to determine its reading accurately at a static angle. This is because the sensor reacts strongly when the system is moving, but when left still tends to converge towards the expected orientation. The sensor was tested with a program that calculated the orientation 20 times per second and printed the results once per second. The output of this program was compared against a gyroscope application on a smart phone. We found that the sensor could vary by about +/- 2°, but would always be centered around the correct value.

EPRI plans to do further testing on the Gyroscope/Accelerometer which includes measuring the system at dynamic angles. Once a complete unit can be properly encased and mounted on a pole, EPRI plans to take a unit to one of their testing facilities and use a controlled test fall of a utility pole to determine, both the durability of the equipment and the quick response and accurate reading of the Gyroscope/Accelerometer sensor.

Analog to Digital Converter

The analog to digital converter in our system is used to translate analog values sampled from the outside world into digital values that our software can interpret meaningfully. Therefore it is very important that this sensor be correctly calibrated. We tested this sensor in a variety of ways: comparing the values in software against 1) a multimeter measuring the same signal, 2) known values, such as 3.3 V, 1.65 V, or ground, and 3) comparing the four channels on the ADC to see if there was any difference between them. After this testing, we determined that the ADC was most accurate when sampled as a differential pair with the input signal as the positive end and ground as the negative end. This helps to reduce the amount of noise in the resulting values and corrects for any DC offset in the ADC itself.

Power Line Voltage Measuring Circuit

The system should be able to verify that the power line to which it's connected runs at a

constant AC voltage of 120 VRMS. Since we are unable to connect the system directly to a power line, we used standard outlet voltage, at the recommendation of our project supervisor, Dr. John Simmins. Because the signal would eventually be fed into the ADC, a circuit was built to ensure this 120 VRMS was stepped down to a usable voltage. The signal would need to be rectified signal (i.e. only positive) of less than 3.3V to be able feed into the ADC. A small PC-34-125 transformer stepped down the voltage from 120VRMS to a safer voltage. The signal was then fed into a bridge rectifier before being stepped down once more by a voltage divider. Figure 3 shows the circuit used. The rectified waveform was then passed along through the ADC and fed to the Raspberry Pi, where our software would sample the values.

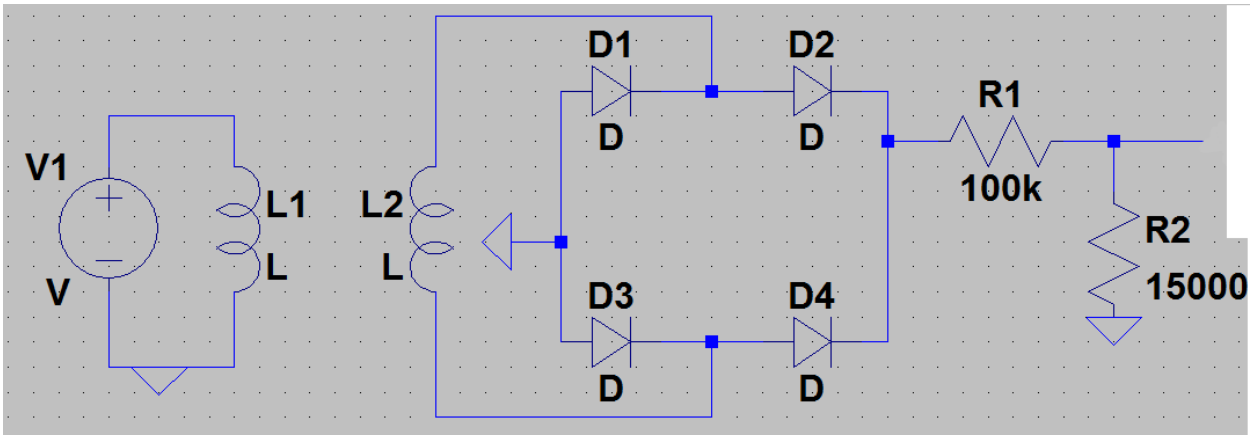


Figure 3. Voltage divider circuit

The rectified waveform is sampled as quickly as the software will allow it (around 1000-1500 Hz) to get a better estimation of the original signal. The software attempts to recreate the sinusoidal form of the signal by tracking its expected phase and correcting for the distortion caused by rectifying the voltage. The reconstructed signal is interpolated slightly to smooth out the results. Then the RMS value of the waveform is estimated by integrating the square of the reconstructed signal. Then the calculated value is scaled (from ~3 V to ~120 V) based on the values of the components in the voltage dividing circuit. This method typically produces results within +/- 0.5 V RMS of the expected voltage when compared against a multimeter. To get more accurate readings of the primary voltage, it would require an ADC that can handle a larger range of voltages (less precision lost from voltage division),

negative voltages (less error from phase distortion), or a software system that can sample at a much higher rate (less error from interpolating the sampled points).

Software Testing

Once all of the hardware components were individually tested, we could begin testing the software components. The majority of our software testing focuses on resource sharing issues because our design consists of many independent threads running on a single-core platform. To test these issues, we created several test cases that create a "worst-case" environment and compared their performance against a "best-case" environment. Listed below are the results of the scenarios that we have tested.

System response time under heavy load

One measure of how well the system will perform in the field is how well it responds under a heavy load. For this test case we will define the "round trip packet time" as the time from when the Control Pi requests the status of a Router Pi until the Router Pi responds with the status of its system. This time period includes the time that it takes the Router Pi to query each of its Sensor Pis over the XML-RPC interface. As such, it will be very dependent on the load on each Pi in the system. To simulate a heavy load, the Smart Pole software is running a plugin that performs calculations on random numbers in an infinite loop. The results of this experiment are shown below.

Test #	CPU %, Router	CPU %, Sensor	Avg RTPT (s)	% RTPT	Samples
1	Low	Low	2.206	1.000	100
2	Low	High	2.291	1.039	100
3	High	Low	2.633	1.194	100
4	High	High	2.915	1.321	100

Table 1: Comparison of average Round-trip packet time for different CPU loads.

It is clear from the results that the system does not scale well under heavy load. This is expected because the Raspberry Pi only has a 700 MHz single-core processor. To get better performance in the future, it is recommended to either 1) choose a more powerful processing platform,

or 2) redesign the software system in a language that is more performance-oriented than Java.

I/O response time with thread contention

Another problem arises when too many threads attempt to use the low level I/O on the Pi to communicate with the hardware. These resources in software are directly connected to the hardware through the kernel with no synchronization for multiple threads, so our software platform has to include its own synchronization. What this means is that only one thread can access a specific hardware component (e.g., the I2C bus) at any given time, and if a second thread tries to access the same resource, it will wait until the first thread is finished. If too many threads attempt to access one resource at the same, it is possible for starvation or deadlocks to occur.

For this test, we will create several threads that repeatedly access the I2C bus and measured the average, minimum, and maximum access time of each thread. In this case, "access time" will be measured from the time that the thread spends within the synchronized block of code (including the time it spends waiting on other threads). We have not yet had time to run these tests, but we expect to see results similar to the ones in the previous section.

System-wide testing

In addition to these worst-case scenarios, we will also be testing the various paths through which data can flow in the system. This includes the following paths:

1) Status Request: Control Pi → Router Pi → Sensor Pi
2) Status Response: Sensor Pi → Router Pi → Control Pi → Web Server Pi
3) Alert: Sensor Pi → Router Pi → Control Pi → Web Server Pi
4) Check-in (Heartbeat): Router Pi → Control Pi → Web Server Pi
5) Check-in Request: Router Pi → Sensor Pi
6) Check-in Response: Sensor Pi → Router Pi

Network Testing

We decided upon two scenarios which we could test for in the network as it currently stands. The first scenario was simulating the failure of any node in the network. The second scenario involved the sending and reconstructing large data via the code that we designed to handle such things.

In the first scenario, the simulation was rather difficult. Our prototype contained only two nodes: a coordinator for the network and a router. Because of time constraints and our limited prototype we were never able to test what happens when a node fails in the network. However, the test plan is rather simple. A timeout variable should be added to the coordinator and it should start listening to the network. Two sensing nodes should then send at least one ping back to the coordinator to let it know that they were both working at one point. At that point, one sensing node should be turned off to simulate a node failure. Once the elapsed time since the last ping exceeds the timeout value the coordinator would recognize the downed node but the network should continue working. One side note on this part is that during testing of our second scenario, we had multiple SD card failures which corrupted the file system for those nodes. However, during this, the dashboard reflected that the nodes continued to report back. That could be a sign that the network was at least able to withstand failures but the limited testing environment gave us no proof of that.

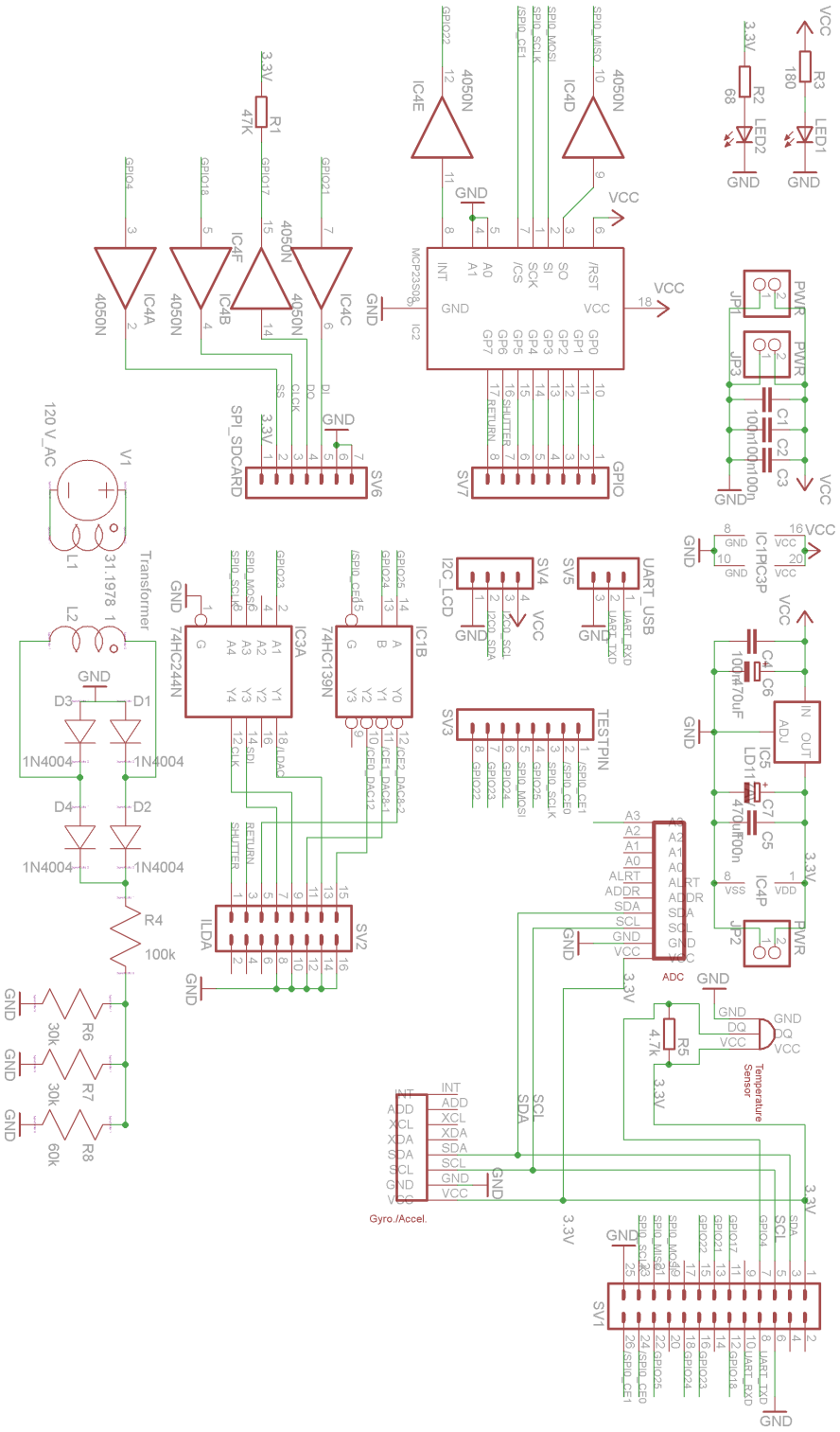
The second scenario we tested was the sending of large data over the network. We wanted to test this because we ourselves had developed the code to handle this issue. We tested by constructing a file that was larger than the ZigBee's maximum payload size and sending it to the coordinator where it should have been rebuilt completely. Evaluation of our test results revealed that our tests were successful except in the very rare case that our data size was within 11 bytes of the maximum payload size. After examining the code that handled the rebuilding of the code we realized that 11 bytes was the size of our header. It turned out that we had a math error in our code that was causing the second packet of size 11 bytes to be ignored and it would get dropped. Correction of that bug fixed the error and all of our subsequent tests for this scenario were successful.

Had there been more time, our team is confident that our design could have successfully passed both of these test cases. Unfortunately we are only able to confirm that one of these two scenarios can be handled appropriately by our system.

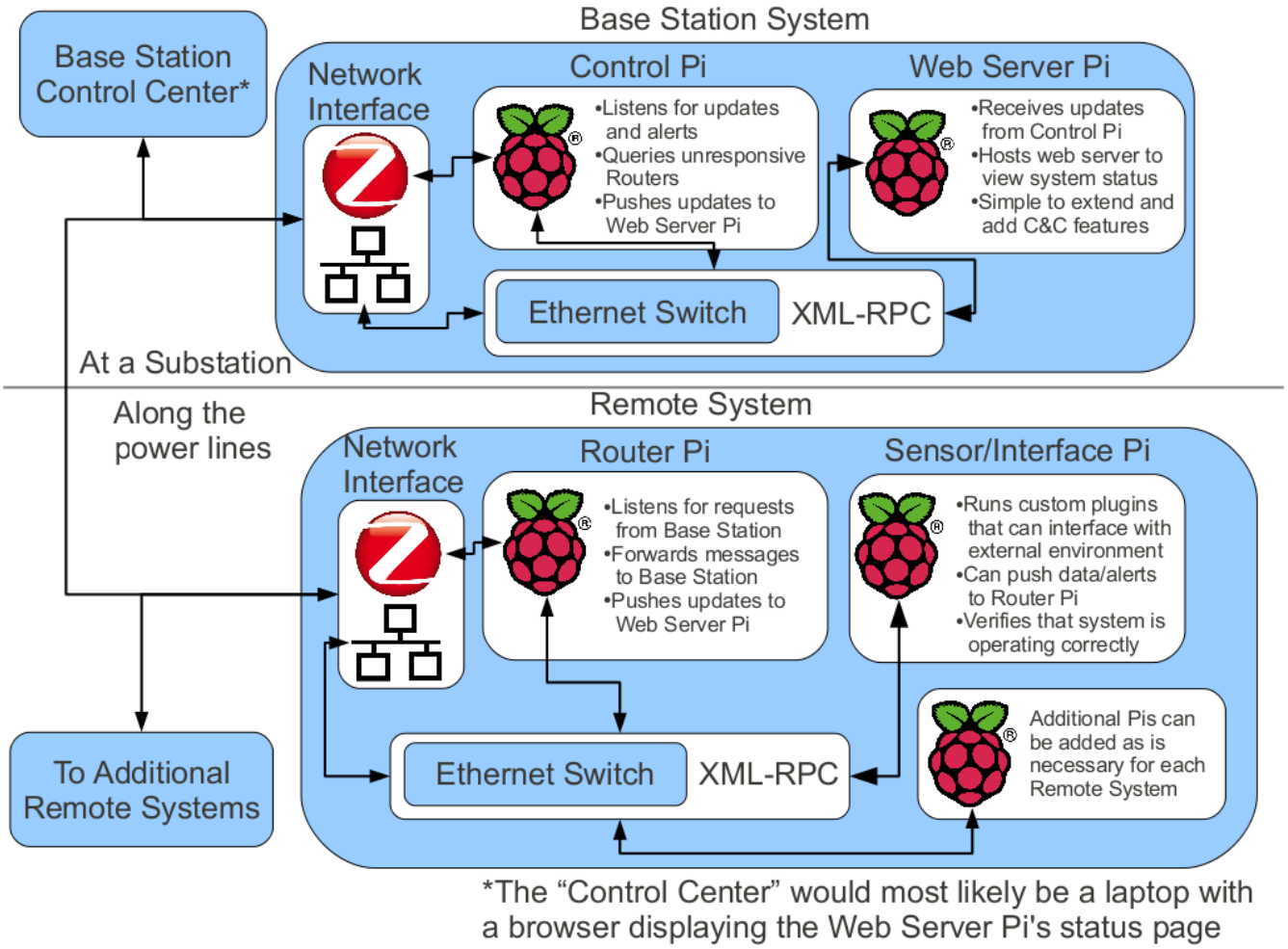
Conclusion

All in all we feel that we have met and even exceeded our requirements for this project. We have designed an open platform that will enable others to develop on top of this system but we have also introduced scalability into the design by allowing large data to flow across our network, accepting any additional number of Raspberry Pi boards that need to be added to the sensor nodes via our XML-RPC architecture, and having support for any sort of end applications that need to utilize the data collected from the Smart Utility Pole units. That's not to say that we couldn't have done things better or managed our time more efficiency. Also there were difficulties along the way which became minor setbacks that we could have avoided had better research been done. In the collective opinion of the Smart Utility Pole team, however, this project has been a success.

Diagrams



Final hardware design



Final system layout

References

[1] "*RPi Hardware*." Elinux.org. 06 Mar. 2014. 28 Apr. 2014. <http://elinux.org/RPi_Hardware>
[2] "RPi Low-level peripherals." Elinux.org. 10 Feb. 2014. 28 Apr. 2014. <http://elinux.org/RPi_Low-level_peripherals>
[3] Allegro Microsystems, "Fully Integrated, Hall Effect-Based Linear Current Sensor IC with 2.1 kVRMS Isolation and a Low-Resistance Current Conductor," ACS712 datasheet, 2006 [Revised Nov. 2012].
[4] "MPU-6050 Accelerometer + Gyro." Arduino, n.d. 28 Apr. 2014. <http://playground.arduino.cc/Main/MPU-6050>
[5] "Kootek Arduino GY-521 MPU-6050 Module 3 axial gyroscope accelerometer stance tilt." Amazon, 15 June 2012. 28 Apr. 2014. <http://www.amazon.com/Kootek-Arduino-MPU-6050-gyroscope-accelerometer/dp/B008BOPN40>

[6] "DS18B20 Digital temperature sensor + extras." Adafruit, n.d. 28 Apr. 2014. <http://www.adafruit.com/products/374>
[7] Maxim, "DS18B20 Programmable Resolution 1-Wire Digital Thermometer," DS18B20 datasheet, Mar. 2007 [Revised Apr. 2008].

[8] "ADS1015 12-Bit ADC - 4 Channel with Programmable Gain Amplifier." Adafruit, n.d. 28 Apr. 2014. <http://www.adafruit.com/products/1083>
[9] Texas Insruments, "Ultra-Small, Low-Power, 12-Bit Analog-to-Digital Converter with Internal Reference, " ADS1015 datasheet, May 2009 [Revised Oct. 2009].
[10] "XBee Pro 63mW PCB Antenna - Series 2B (ZigBee Mesh), " Sparkfun, n.d. 28 Apr. 2014. <https://www.sparkfun.com/products/10418>
[11] Digi International, "XBee®/XBee-PRO® ZB RF Modules," Xbee Pro S2B RF module datasheet, 2010.
[12] "XBee Explorer Dongle, " Sparkfun, n.d. 28 Apr. 2014. <https://www.sparkfun.com/products/9819>