Visualization in Power and Energy in Real-Time

Timothy Pace
University of California, Santa Cruz
Santa Cruz, CA, USA
tmpace@ucsc.edu

Bryan Smith
University of California, Santa Cruz
Santa Cruz, CA, USA
brabsmit@ucsc.edu

Abstract— **New technology finding its way into the home has opened up a gateway to collect power statistics and other information in a fast and efficient way. To keep up with demand from consumers, some electric companies provide primitive visualizations of their power usage and quality of power. A problem arises from hosting all this data on a centralized network, introducing problems such as low sample rates and extremely high latency. Under the guidance of Professor Patrick Mantey, Bryan Smith and Timothy Pace worked with graduate student Paul Naud to develop a system that provides real time voltage monitoring using existing sensors found on APC's UPS (Uninterruptible Power Supply). The system was developed with two goals in mind: scalability and efficiency. A Raspberry Pi was employed to store the data and host the website, and the WebSocket protocol was used by the client to facilitate the live content.**

I. Introduction

The project began with a simple goal in mind: use the existing sensors on board the UPS as a means for providing power statistics. The UPS itself provided minimal power statistics through software called PowerChute. In order for PowerChute to collect data from the UPS and provide statistics on thing such as blackouts, under voltages or over voltages, it must be connected to a PC running the PowerChute software via USB. Keeping a PC running to collect power statistics is not very practical. In order to record blackouts the PC must remain on always. Our idea was to use a device that would be cheap, but would still be able to collect data and provide useful statistics. The solution we came up with was to use a Raspberry Pi, a cheap development board capable of running a full Linux OS. We used the Raspberry Pi to run daemon software written in Python to collect data by polling for statistics over the USB connection. The Pi then stored this data into a database and provided a simple easy-to-use web interface to visualize the data and provide more useful statistics on the data.

A. Setting up the Pi

In order for the Raspberry Pi to be prepared upon boot for interfacing with the UPS, we first wrote a configuration file for the network interface.

This allows the Pi to connect to a predefined wireless network. In addition, we wrote an init.d bash script to start our Python script as a daemon. Init.d scripts are a specially formatted bash script that allows the system to boot up and start daemons in a particular order so that runtime dependencies are met in the correct order. This ensures that if the Pi doesn't last on the UPS's battery, it will still boot up correctly and be ready to go when plugged back in.

II. Data collection Daemon

Our data collection daemon was written in Python. The daemon collected data from the UPS by polling for data over the USB connection. Any changes in data are reported to any clients connected to the daemon via a WebSocket, and data changes are stored into a MySQL database. Users can then use the data collected here in conjunction with the front-end website to visualize the data.

A. pyUSB

Python provided a simple and easy to use USB library called pyUSB. This allowed us to make Control Transfer requests over the USB connection to the UPS to request different types of data. Control transfers are typically used for command and status operations. They are essential to set up a USB device with all enumeration functions being performed using control transfers. [1] In order to figure out the format for the control transfer requests to the UPS, we used a USB packet sniffer called USBlyzer. We ran USBlyzer in parallel with PowerChute so that we could see what types of requests PowerChute was making. Using this method we were able to determine several different control transfer opcodes used in requesting data form the UPS. Figure 1 shows the different opcodes we discovered and used in our daemon.

TABLE I
Control transfer opcodes

x06	Request a Boolean indicating whether the battery is charging.
x22	Request the percentage that the battery is charged.
x23	Request how much run time is left on the battery in seconds.
x31	Request the current in
x50	Request the total load on the UPS.

B. Data Collection

The data collection portion of the daemon is run in a separate thread from the rest of the program. This thread is responsible for the polling mentioned above. While polling, this thread keeps track of the current and past levels of data and makes note of changes by updating clients that are connected to the WebSocket, and storing the changes into a MySQL database. This allows us to prevent recording redundant data by only recording changes in data. In order to record special events such as blackouts, under voltages, and over voltages, this thread also takes note of how long the voltage stays at a certain level and records the appropriate information. Unfortunately, the sensors on the UPS only allowed for a maximum sample rate of 6Hz, or once every 150ms. Because of this bottleneck, the data collection had a maximum resolution of 6 power cycles.

C. Database

To store our data, we used a MySQL database with a couple different tables. Each table kept track of a different statistic collected from the UPS. This allowed us to store data quickly and efficiently and not have to worry about data management. To keep storage needs low, we used the method discussed above in which we only stored changes in data. Additionally, since our collection rate was around 6 Hz, we could not use simple UNIX timestamps to record the data. Instead we used a UNIX timestamp with millisecond resolution.

III. WEB SERVER AND CLIENT

In addition to collecting data, our Raspberry Pi doubled as a web server "backend" for data visualization. A combination of JavaScript and CSS interfaced with the server for the client to make up the "frontend." This pair provided the structure for real-time visualization of data from any array of sensors provided. In order to facilitate live content, the WebSocket protocol was used in our system. Given the multitude of data that is recorded, the server and client must work together to create a fast and easy way to reference the data in a readable way. For data visualization, amCharts was used for its high responsiveness and wide user base.

A. WebSocket

"The WebSocket Protocol enables two-way communication between a client running untrusted code in a controlled environment to a remote host that has opted-in to communications from that code."[2] The WebSocket protocol was chosen instead of POST requests because of how frequent a user can request a new sample point from the system. WebSocket is a layer of abstraction on top of the TCP protocol, allowing for a single connection for traffic in both directions. A WebSocket connection is opened with a handshake that secures the connection between client and server. After the handshake, data can be transmitted using an arrangement of frames. Figure [1] illustrates the outline of a frame. The frames are set up to be sent in succession, with a capping frame at the end. An opcode field denotes the type of message the frame contains, a payload length is specified, and the payload itself is enclosed. Communication from client to server required bit masking, specified by bit eight of the frame. When masked, a frame required the masking key needed for unmasking.

FIGURE 1
WEBSOCKET DATA FRAME

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

B. Server Backend

In constructing the website for data visualization, we experimented with different types of web hosting. Initially, Drupal was considered as the backbone for the webpage, but ultimately was thrown out due to performance issues on the Pi. In the end, just Apache was installed and the website was put together entirely by hand. When a client connects, the initial payload of data contains battery percentage, charging/discharging, percent load, and current voltage. After this initial payload, the server is set up to send only when a) an event changes (change in voltage, battery percentage, etc…) or b) information is requested from the client. In this way, the client receives just as much information as necessary.

C. Client Frontend

Using JavaScript, the client can take full control as to what is being displayed from the database. Since the backend is set up to take flexible requests, the client can specify refresh rate for real time, or select a custom date range to view statically. Because of the premise of the data, it was necessary to set up an easy way to view past "events" on the graph. Blackouts, undervoltages, and overvoltages are all parsed into separate tables and each event is a link to a specific time range when the event took place.

D. amCharts

The graphing library for real time visualization had to be responsive enough to support updating at a rate equivalent to 150ms. Highcharst was considered, however this library had difficulty keeping up with the high refresh rate required by our system. amCharts was then selected based on it meeting our criteria and having a wide user base. amCharts accepts data in the form of a tuple, and x and y coordinate. For real time, and x and y coordinate is fed at the rate specified by the user and the chart is redrawn. When loading a range of data, all points are loaded simultaneously and the chart is drawn once.

ACKNOWLEDGMENT

- Paul Naud, for his assistance in related questions

REFERENCES

[1] Peacock, Craig. "Endpoint Types." USB in a NutShell. Beyondlogic, n.d. Web. 14 Sept. 2

[2] I. Fette and A. Barth. The WebSocket protocol. Internet-Draft draft-abarth-thewebsocketprotocol-01, Internet Engineering Task Force, Jan. 2011. Work in progress.