Princeton University

Senior Thesis

A Wired Sensing Suite for Data Center Climate Monitoring

Author:
Ryan C. O'Shea

Advisor:
Prof. David Wentzlaff

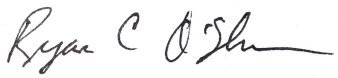
Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering

Department of Electrical Engineering
Princeton University

May 2, 2016

HONOR DECLARATION

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.



Ryan C. O'Shea

A Wired Sensing Suite for Data Center Climate Monitoring

Ryan C. O'Shea

ABSTRACT

This senior thesis builds on the foundations of two previous independent work projects to design, build, test, and deploy a scalable network of temperature sensors for data center environments. There exists a current and accelerating trend of computing workloads migrating from personal devices to cloud-based services. That growth has inspired an expansion of data center facilities across the world, which need to ensure that server hardware is kept cool enough for safe, continuous operation using incredibly sophisticated and expensive cooling systems. Taking a small-scale working prototype, this thesis miniaturized and minimized the cost of its sensing hardware, rewrote its software to ensure long-term stability, and established an architecture enabling the monitoring of an unlimited number of server cabinets across any number of data center facilities. The measurements collected by this system are organized and displayed by a new, powerful web application that allows data center operators to effectively monitor their facilities, providing critical insights about the efficacy of their cooling systems. In total, 500 sensors and peripherals were produced, while a smaller subset were deployed to the Princeton HPCRC data center, where they are actively collecting data for the facility's managers and our own research purposes. Within the next year, we hope to have our system deployed at full scale across the HPCRC, both to equip those at the facility with a powerful tool for climate monitoring and to provide a large body of high-resolution temperature data for further study.

ACKNOWLEDGEMENTS

My foremost thanks are offered to my incredibly supportive faculty advisor **Prof. David Wentzlaff**, whose guidance and past involvement with this project were absolutely essential to the progress that was made during this thesis. I also extend my gratitude to **Prof. Naveen Verma** for graciously agreeing to be my second reader. **Mike McKeown**, **Jon Balkind**, and **Mohammad Shahrad** from Prof. Wentzlaff's Princeton Parallel Group also provided me with vital support as I explored unfamiliar tools and technologies during this past year. At the Princeton HPCRC, **Matt Petty** and **Michael Monaghan** were extremely accommodating and supportive of our efforts to equip their server racks with our system. **Phil Fugate** was also instrumental in determining and configuring our device networking strategy.

This project received an incredibly generous amount of funding from the **Department of Electrical Engineering** and from the **School of Engineering and Applied Science** via the MacCracken Independent Work/Senior Thesis fund. Due to the unusually expensive nature of this thesis, especially regarding the fabrication and assembly of PCBs, Prof. Wentzlaff also contributed a moderate amount of funding to the project. I owe a (literal) debt to the Department, the School, and Prof. Wentzlaff for making this project financially possible. Thanks also to **Linda Dreher**, Purchasing Coordinator for the Department, who assisted me a great deal in placing all the necessary orders.

I would like to thank especially all of my friends at Princeton who provided me with encouragement and support throughout this process. First, I am forever grateful for the extraordinary amount of kindness shown to me by **Tess Marchant**, who has selflessly supported me through every setback and success that has passed by during this project. I would not have been able to finish this thesis without her help. To all of my friends and fellow officers at the **Princeton Tower Club**, who have given me the closest thing to a second home I have ever experienced, I would like to thank you all from the bottom of my heart. Life after graduation will not be the same without your wonderful presences. In particular, I'd like to thank **Alex Cuadrado**, who helped immensely in reviewing the final draft.

Lastly, to my family — Mom, Dad, & Dan — your incessant support and love have not only enabled this thesis, but provided me with the opportunity to produce it in the first place. This thesis is for you and for all you have done for me in the last twenty-two years.

To anyone else whom I neglected to mention, thank you.

Sincerely,
Ryan

Contents

Chapter 1

Introduction

1.1 Motivation

In the past decade, the paradigm of personal computing has undergone a major shift. Mobile devices wielding the same computational power of desktop PCs have become increasingly prevalent in developed parts of the world. As the capabilities of both mobile and other personal devices have grown, more complex and compute-intensive applications have been developed to serve customers using faster devices. These two areas of growth have not progressed at the same speed, however, and, as such, applications powered by the Internet have moved their compute workloads from consumer devices to "The Cloud" — to servers and computer clusters hosted in large data centers. In order to provide useful experiences to customers, applications need to leverage computational power beyond the limited capabilities of a low-power mobile device or even a laptop PC. No longer limited by the speed of personal devices, application developers can continue to provide increasingly complex services by deploying their computational backends on servers in a data center.

An entire business sector has emerged from this change. Internet companies frequently pay

for computational power and storage space with which to power their applications, either obtaining these from a third party data center provider like Amazon or via the establishment of their own data centers, an option typically reserved for the largest technology companies due to the prohibitive cost of building and maintaining a data center.

Meanwhile, the speed of CPUs has grown — and transistor size has decreased — generally in accordance with Moore's Law, but power consumption has not followed the same downward trend expected by the law. Instead, supply voltage has remained nearly constant and as such, building faster processors in recent years has necessitated increases in energy consumption and heat output per chip [1]. Following this, the most substantial limiting factor in the growth of data centers is power consumption — both directly powering computational equipment and powering the massive systems required to keep them cool. Data center power consumption doubled worldwide from 2000 to 2005 [2] and quadrupled in the decade prior to 2012 [3]. Even though this growth did not live up to decade-old predictions [4], data centers still consumed $4.5B in electricity each year in the US by 2006 [5] and have only expanded their footprint since.

More important even than the overall rate of growth of data center energy usage is the proportion of the energy used that goes towards cooling rather than directly to work. By some estimates, cooling can consume 40-45% of a data center's total electricity usage [6], which has turned cooling efforts and systems from necessary, though secondary, features of these data centers to an aspect of their design as important as the networking and servers themselves. Any system as large, complex, and expensive as a data center cooling system

is critically concerned with its efficacy. If any areas of the data center's "data hall" (the room which actually houses the servers and is the target of nearly all cooling efforts) receive uneven treatment from the cooling apparati, prolonged operation could result in permanent, costly damage to computing hardware or the facility itself. This could come in the form of "hot spots," in which hardware in a small part of the facility is allowed to heat past the low temperatures required for continuous operation under load. It could also take the form of moisture collecting via condensation in more complicated cooling schemes, the result of which could be electrical damage to hardware or even a fire hazard. On an unrelated, but interesting note, server racks with moving components — like spinning hard disk drives — might suffer from unwanted vibration that could damage components in similar ways.

In response to these concerns, and with demonstrated interest from the Princeton University High Performance Computing Research Center (HPCRC),[1] this project aims to create an inexpensive means of monitoring the climate in a data center to allow site operators to identify possible problems as they arise and to ensure that their computing resources can operate safely at full capacity.

1.2 Background

This project began in 2014 as a student independent work project to design a sensor board (both circuit and PCB layout) to allow some sort of controller device to communicate with

[1] A nearby data center serving Princeton University faculty and students

small, inexpensive temperature sensors over the I^2C bus protocol. The circuit also incorporated a series of switches to allow for easy configuration of the sensor device's address on the I^2C bus, which is required to be unique for each device on the bus.

The initial work was performed by Rahul Subramanian '14 [7] under the guidance of Prof. David Wentzlaff, under whose control it remains. It was continued by Peter de Groot '15 as another independent work project [8], where the designs were miniaturized and manufactured as boards and a working subsystem of sensors connected to a controller was deployed on a small rack in the EE Parallel Lab. It successfully displayed its gathered data via a simple web application running on the bus master device. Further efforts were contributed by Felix Madutsa '19 to digitally reconstruct some of de Groot's work and expand on the web application used to display sensor data.

Further details on the system design and the contributions of the author's predecessors are provided in Chapter 2.

1.2.1 Milestones of Current Project Iteration

This iteration of the project aimed to move the components of the system to a state that can be more easily and affordably mass-produced than previous versions. It fully developed a new capability to operate beyond a small-scale system involving a single controller device attached to an array of sensor boards, instead orchestrating the operation of many instances of these sensor-controller subsystems as one functional system. The final design of the system was deployed in a live data center environment, the Princeton HPCRC. Lastly, it involved

the development of a highly functional web application for managing a deployed instance of the climate monitoring system so that site operators can both utilize the system for climate monitoring and manage their systems effectively.

1.3 Related Work & Industry Context

Climate monitoring of data centers is an active field of research and development, so this section aims to recognize related efforts in the field and explain some of the design choices made in this attempt at a solution. The latter half of this section makes limited use of prior research already done in de Groot's iteration of this project, found in [8].

Most importantly, the use of a wired network to link the sensors together was a key design choice. The primary factor in this decision was cost. Keeping the cost of the sensor suite as low as possible is a major concern for this project, as each deployed instance of the system could consist of dozens or hundreds of sensor boards collecting data at various locations across a data center's data hall. Each increase in the price of an individual sensor board is amplified across all boards, and wireless communication equipment would at the least on the order of double the cost of each sensor board. Wired communications, however, allow us to make use of much cheaper cables to carry data between each sensor and the controller. The signal integrity across a wired connection is also likely higher, especially on a relatively low-frequency bus protocol like I^2C. Adding wireless communications to the board would also necessarily increase the size of the board, especially since it would then need to find room for the wireless transmitter itself and some sort of microcontroller as well, which would

be required to interface between the sensor's I^2C protocol and the controls of the wireless transmitter. This increase in size and price (especially since microcontrollers are much more expensive than even the most expensive components on the current version of the board) would make the sensor board increasing impractical as a small, inexpensive, unobtrusive addition to a server rack, which this system aims to be.

Regarding other efforts aimed at data center climate monitoring, IBM proposed a much different solution involving a roving platform capable of collecting data as it wandered around a facility [9]. Rather than a low-cost network of devices scattered across a room, this solution relied on very low quantities of relatively expensive equipment. This system has the advantage of allowing data center operators to diagnose a problem area hyper-specifically by moving the sensing equipment directly to that area, though as noted in [8], it also has the significant disadvantages of having very low time resolution due to the physical movement needed to record data in different locations as well as operational interference caused by automated vehicles traveling in the same space as humans.

Another example closer in design to the system laid out in this paper is Raritan's data center climate monitoring solution [10]. Raritan sells a series of sensors and controllers that can mount on to a server rack. In their system, each temperature sensor hangs from one wire, rather than having multiple sensors placed along a single wire, as they are in our system. While this allows for greater plug-and-play capabilities, cable clutter would quickly become an issue if a data center would like to position many of these sensors, say, along a vertical line down the side of a server rack. Also, it seems that Raritan's sensors need to be connected

Figure 1.1: Geist's 'Monitor' rack-based solution [11]

to a rack outfitted with their "intelligent rack" technology, which obviously necessitates a more sizable investment on the part of the data center operator.

Geist offers a number of rack-based climate monitors (systems that slide into a slot in a server cabinet like server would), which have onboard temperature sensors and the capacity to attach up to four external sensors for expanded sensing capability [11]. These systems are larger and significantly more expensive, due to their more substantial footprint. They also require data center operators to possibly sacrifice space in their server cabinets to fit the monitor, as opposed to our door- and rack-top-mounted solution, which simply mounts inside the frame of of a cabinet. The Geist units also host a web application locally and provide access over an ethernet connection to the outside Internet, which is a feature of the de Groot implementation of this project, but which we are replacing in this iteration with a global web application handling multiple sets of sensors.

The takeaway from this brief survey of the current market landscape is that even professionally designed systems aimed at climate monitoring for data centers come with drawbacks,

expenses, and trade-offs. Though the industry is relatively standardized, each data center is different, and each operator may have differing thoughts on what role monitors should play in the space they control. As such, there is great room for a variety of approaches, so our minimalist design might serve the market well.

Chapter 2

Prior-generation System Overview

2.1 Overview

This section describes in detail the state of the climate monitoring system before this thesis's work was conducted, as many of the design details remain relevant or perhaps unchanged. The following is based on the work in [7] and [8]. Some additional work was provided by Felix Madutsa, though no written source exists to document that work. For brevity, further citations to this material may be omitted, and the description references the state of the system as it was when this iteration of the project began in September 2015, so differentiation may not be made between the contributions of each author to the project, and descriptions will focus on the combined design of the final de Groot/Madutsa version.

2.2 Sensor Board

2.2.1 Sensor Board Instrumentation Circuit

TMP175 Sensor

The most fundamental component of our system — the one towards which the majority of the fall term's efforts were directed — is the circuit and board design for the actual sensor boards. The main component is a simple, inexpensive digital temperature sensor. The one chosen for this project is a Texas Instruments TMP175 chip in a surface mount LM75 package [12]. The sensor provides its readings via the two-pin I^2C bus, which of course implies that a secondary, master device be present in the system to instruct the sensor to take readings and then to gather that data. The sensor provides measurements in a digital format accurate to 1°C.

Addressing

Multiple TMP175 sensors can be attached to the same I^2C bus by varying the address of each sensor on a given bus to avoid collisions. The sensor has three addressing pins, each of which use trinary logic (ground, reference voltage, or floating voltage), providing a total of 3^3 or 27 possible addresses. Therefore, we can place up to 27 TMP175 sensors on a single bus.

To accomplish this, the circuit incorporates six DIP switches. Each address pin of the sensor is controlled by two switches. One, when switched on, pulls the pin high via a pull-up resistor. The other, likewise, grounds the pin. Leaving both switches off results in a floating/random voltage, and the third logical option for the pin. The presence of the pull resistors have the added benefit of preventing a short circuit if both switches are turned on.

Connectors

The circuit uses a 10-pin IDC connector (in 2 rows of 5, 0.1" pitch), which in turn connects to a 10-conductor ribbon cable, to provide power and I^2C data from the controller. The 10-pin design is used for two reasons. First, the type of cable used is inexpensive. Second, the arrangement of pins assures that if the ribbon cable connector were to be attached to a board in the wrong orientation, no permanent damage would be done to the board. Each board is attached to the cable via an identical IDC connector crimped to the proper location on the cable.

Other components

The circuit has an LED which illuminates to confirm that the device is receiving power. It also has two decoupling capacitors between power and ground, one to prevent against short-term power fluctuations and another with greater capacitance to prevent against longer-term fluctuations.

Circuit Schematic

The diagram for the circuit as it stands currently is shown in Figure 2.1. This circuit only has a few very minor differences from the version in the de Groot iteration of the project. During this thesis, the schematic was rebuilt using different parts for nearly every component, but the only electrical changes were the values of the decoupling capacitors used.

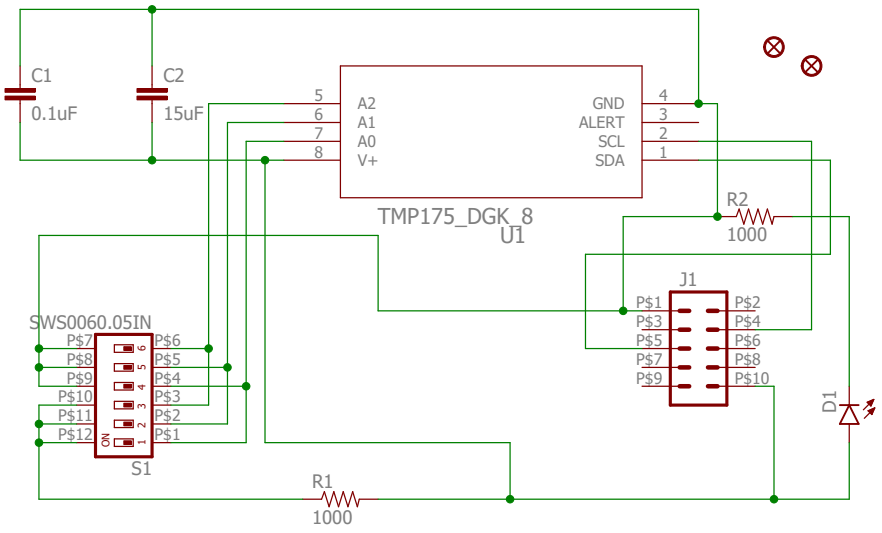


Figure 2.1: Current instrumentation circuit diagram for sensor board. To clarify, `J1` is the cable header.

2.2.2 PCB Design

The circuit described was produced on a small 2-layer printed circuit board. The board only has components on one side for reduced costs. It has primarily surface-mount components for easy assembly, but the header for the ribbon cable is a thru-hole component. The design is shown in Figure 2.2.

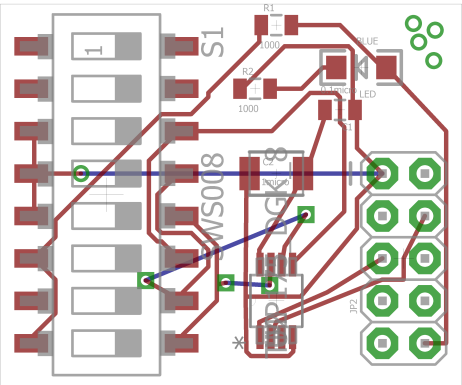


Figure 2.2: Original design for sensor board PCB
Dimensions: 1.1" × 0.9"

We identify some issues in this design:

- Non-45° traces are used
- DIP switch component is unnecessarily large and has 2 unused switches
- Mixed-type vias
- Two layers are used for routing, but large wasted areas present which could be consolidated via more efficient routing
- Thru-hole component is an obstacle for cheap automated assembly
- Design calls for large electrolytic capacitor, when smaller/cheaper ceramic replacement could be used

These concerns are addressed by the new design, which can be found in §3.2.

2.3 BeagleBone Black Controller

To manage gathering data from the sensors, a BeagleBone Black running custom software is utilized. The BeagleBone is connected to a cape to aid in providing the proper connections.[1] The software periodically queries the full address space of the TMP175 sensors and detects the addresses of each sensor connected to the bus. It then periodically gathers temperature readings from those sensors, keeping track of which sensor provided each reading. The BeagleBone supports two I^2C buses, so in theory each BeagleBone can manage up to $2 \times 27 = 54$ sensors.

In the inherited version of the project, this data was saved in an SQL database and presented to users via a web application that is hosted via an instance of the Apache webserver running on the BeagleBone. The BeagleBone needed only to be connected to the Internet via its Ethernet port, and the application could be accessed by visiting the BeagleBone's IP address from any web browser.

This method of displaying climate data is limited to one sensor-controller subsystem, as the web application can only display the data provided by the sensors the BeagleBone is physically connected to. A solution for expanding past this limitation is outlined in Chapter 4.

[1] The details of this setup, which have been omitted for brevity, can be found in [8]. However, an updated design for the cape is described in full in §3.3.3

Chapter 3

A New Generation of Sensing Hardware

3.1 Overview

This chapter outlines the work done to redesign aspects of the climate monitoring system's electrical hardware. The related files can be found in the project's GitHub repository [13] in the `/dcsense-board` directory.

3.2 Miniaturizing the Sensor Board

The majority of the work carried out in the fall term focused on preparing the sensor board for inexpensive and reliable mass production. A few priorities were designated for the redesign beforehand, including:

1. Use only surface-mount parts (significantly decreases the cost of professional turn-key assembly)
2. Miniaturize the design as much as possible (decreases cost of fabrication and minimizes the system's footprint on server cabinets)

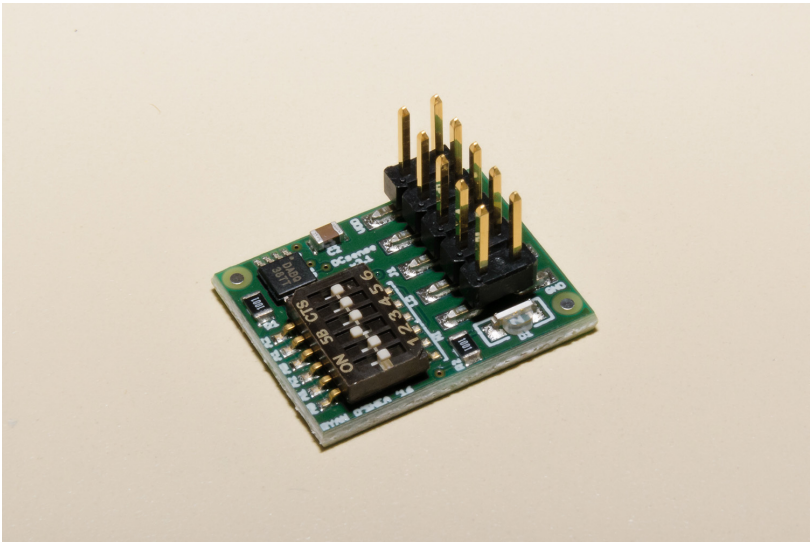


Figure 3.1: The new (v3.1) DCSense sensor board, professionally manufactured by Advanced Circuits and Green Circuits

3. Improve the board's electrical characteristics by using PCB layout best practices
4. Prioritize price over size when size is not vital

These priorities informed the decisions described below.

First, it was recognized that the cable header, rather than the DIP switches, should constrain the size of the board. The first change was thus a move to much smaller DIP switches (from 0.1" pitch to 0.05"). We also changed from an eight DIP switch component to one offering only the six needed for addressing. Next, the sole thru-hole component (the cable header) was removed in favor of a surface-mount replacement. For a brief period, we considered reducing the pitch of the cable header from 0.1" to 2mm, but that idea was abandoned upon the realization that the IDC connectors and cable needed in that case were substantially more

expensive than the 0.1" versions.[1] One other small change was made to the circuit, namely that we replaced the larger electrolytic decoupling capacitor with a cheaper ceramic one with much higher capacitance for greater protection. The final circuit diagram is shown in Figure 2.1 and a specific list of all parts can be found in Appendix A. Lastly, we included copper fill pours for the ground and power signals on the bottom and top layers to improve the board's electrical characteristics, and we ensured that all traces were aligned to a standard grid and oriented at 45° angle increments to comply with PCB fabrication standards.

The new design is **47% smaller** in terms of area than the original design. It uses only surface-mount components so that the board can be easily assembled by an automatic pick and place machine. Ignoring the fixed cost of non-design-related portions of PCB fabrication and assembly, the new design **costs 20% less to fabricate** due to its smaller footprint and **34% less to assemble** due to the removal of thru-hole components.[2]. Tabulating the prices for each component,[3] the previous design comes in at $3.57 per board, while the miniaturized version goes up a bit in price to $4.20. Sadly, the smaller DIP switches — though they make the smaller form factor possible — are the main driver of the price increase. Adding it all up, the new board still comes in at about **10% less expensive** than the original board at a much smaller and much less obtrusive form factor. The final layout of the board can be found in Figure 3.3.

After a small initial order of boards for testing purposes, we had 500 copies of the board

[1] Note that because we use a 2×5 header at a pitch of 0.1", the pitch of the cable itself needs to be half that, or 0.05". The same applies to any other header pitch values.

[2] Based on quotes for PCB fab and assembly from Advanced Circuits, given a quantity of 500 and lead time of 2 weeks

[3] Assuming a quantity of 50

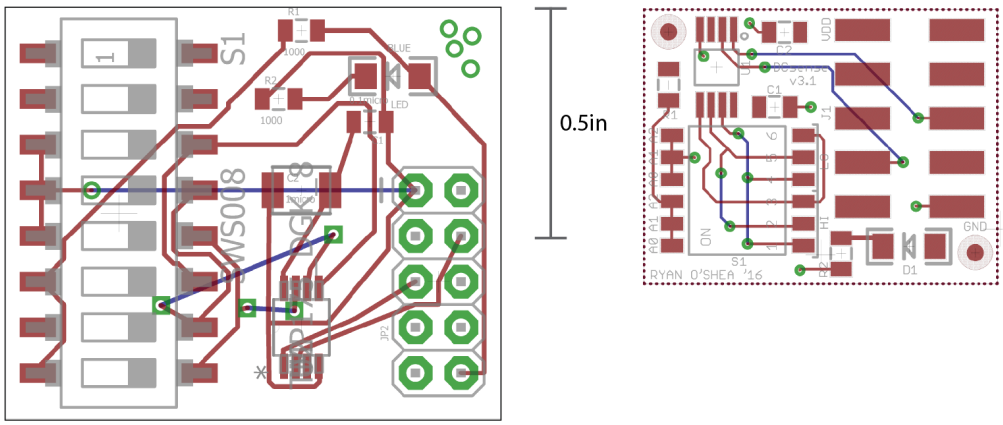


Figure 3.2: De Groot PCB v2.0 (left) vs. miniaturized PCB v3.1 (right)
To relative scale: 1.1"×0.9" vs. 0.8"×0.65"
Copper pours not shown for component visibility

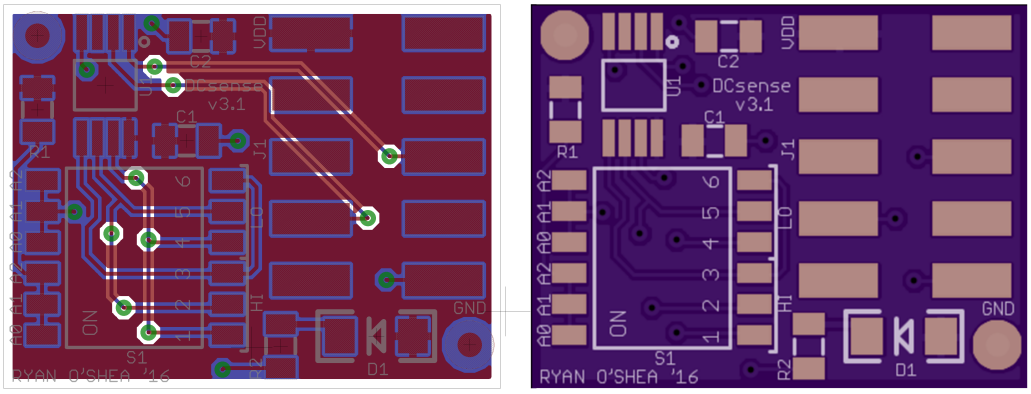


Figure 3.3: v3.1 PCB design with visible copper fill pours
for power and ground (left)
Preview of fabricated board without components (right)

fabricated by Advanced Circuits, and they were professionally assembled by Green Circuits for a total cost of $3,970.00. Photographs of the finished sensor board can be found in Figure 3.1.

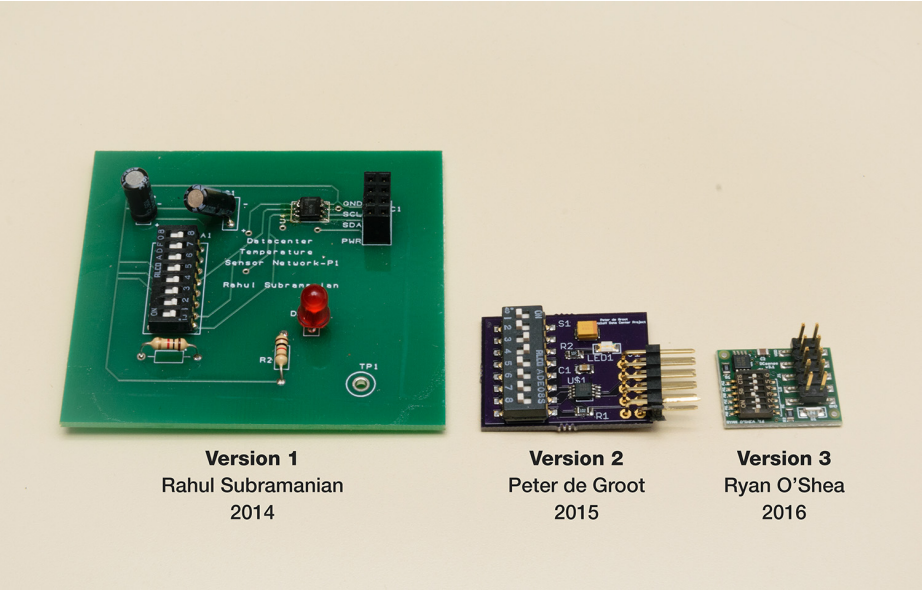


Figure 3.4: Side-by-side comparison of 3 generations of the sensor board

3.3 The Sensor-controller Subsystem

3.3.1 Overview

Aside from the sensor board, not much has changed in this thesis in the remainder of the electrical hardware comprising the sensor-controller subsystem. However, in some cases, the exact hardware has not been documented previously, so this section intends to fill those gaps by documenting some of the minor changes made.

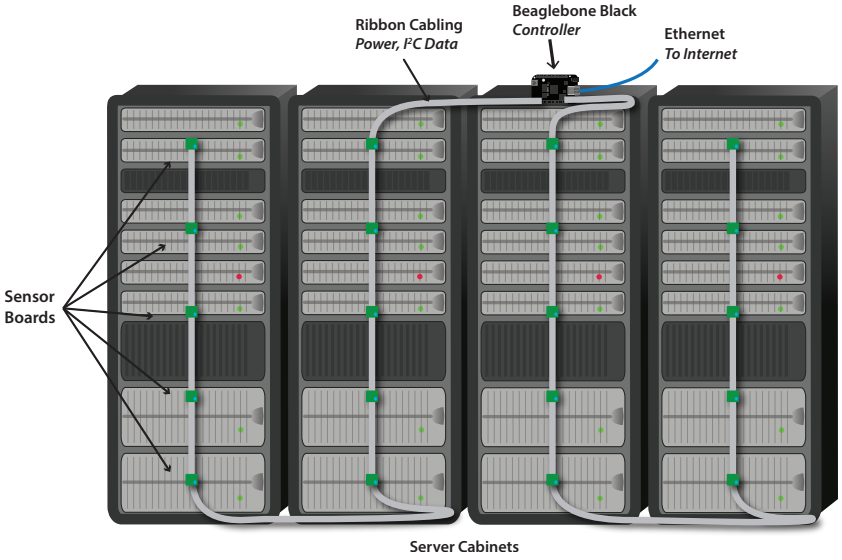


Figure 3.5: Mockup layout of sensor-controller subsystem on a server cabinet group. Note that sensors are mounted on the doors of the cabinet, which have hinges on the right. Cable is fed through the door frame and then down the inner surface of the door for sensor placement.

3.3.2 BeagleBone Black

As mentioned before, the subsystem relies on a TI BeagleBone Black to be present to interface with the sensor boards. Our particular implementation used revision A5B.

System Configuration

Our BeagleBones run Debian rather than the stock Angstrom distribution of Linux. Not much configuration is needed, though a few features are essential for proper operation. First, the devices need a working Internet connection via the onboard Ethernet port. A few packages must be installed, including `ntp` for automatic setting of the system time and `python/python-smbus` to run the controller application.

The Python application should ideally be configured to run at startup after a delay of a few minutes (which allows the BeagleBone to initialize its Ethernet interface and establish an Internet connection). This can be accomplished via a script placed and registered in `/etc/init.d` which sleeps for the necessary time before executing the application.

Additionally, the BeagleBones must be configured to enable the third onboard I^2C bus, which is disabled by default. On the BeagleBone, this bus is `I2C1`, while the other on-by-default bus is `I2C2`.[4] `I2C0` exists but is used internally and not accessible to users. Enabling this bus proved difficult and the process varies by BeagleBone revision and operating system. For this reason, among others, we prepared a fully-configured system image that can be cloned to a MicroSD card. Inserting one of these MicroSD cards into a BeagleBone will cause it to boot from the card into an environment

Figure 3.6: View of one of the server cabinets at HPCRC bearing our sensor system

[4]Note that while BeagleBone labels the buses `I2C1` and `I2C2`, Linux tends to map these in reverse as `/dev/i2c-2` and `/dev/i2c-1`, respectively.

perfectly configured for our application. Exact details on this process can be found in the project GitHub repository [13] in the README within the `/beaglebone-controller` directory.

For our particular installation at the Princeton HPCRC, the BeagleBones are placed on a subnet with no outside Internet access. However, the main DCSense server (see §4.2) has access to that subnet, so the server is configured as a proxy service which the BeagleBones use to access the Internet for a few necessary services. The MicroSD card includes this configuration, although it won't be necessary for any BeagleBones deployed outside of Princeton's campus.

3.3.3 BeagleBone Cape

In order to connect to the cables carrying the sensor boards, the BeagleBone utilizes a configured version of SparkFun's BeagleBone Proto Cape. The cape with our custom circuitry serves two primary functions: providing external pull-up resistors for the I^2C buses and providing headers to connect the two bus cables to the correct power, ground, and GPIO (bus) pins on the BeagleBones. Each cape was hand-soldered to implement the circuit shown in Figure 3.8.

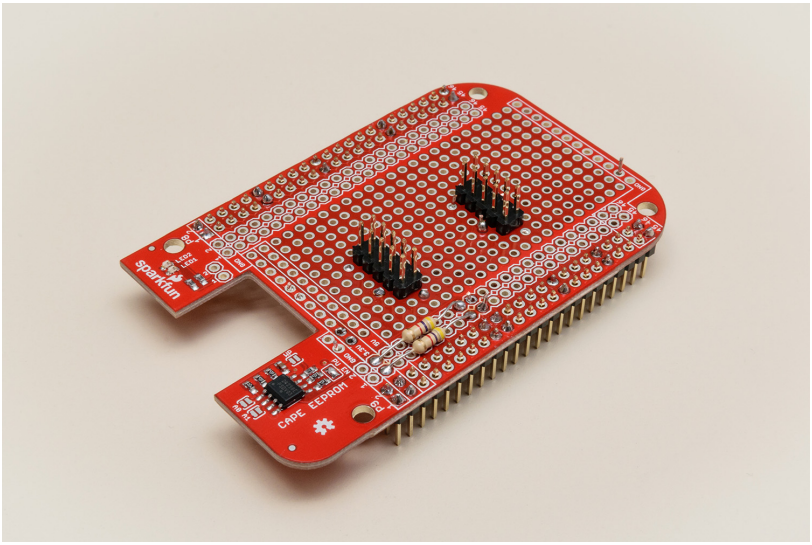


Figure 3.7: Top view of a completed BeagleBone cape with the hardware needed to support the sensor subsystem (two resistors are missing, as they were added to the design after the photo was taken)

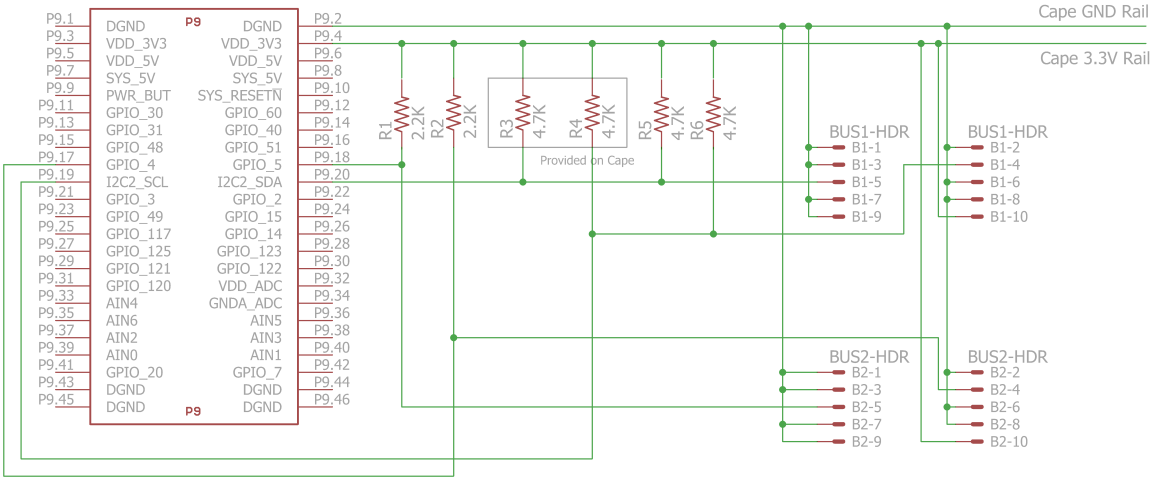


Figure 3.8: Custom cape circuit schematic. Note that 2 additional 4.7kΩ resistors are used in parallel with the cape's built-in 4.7kΩ pull-ups to reduce the pull-up value for both buses to 2.2–2.35kΩ

3.3.4 Cabling, Connectors, & Mounting

To connect the sensor boards to the BeagleBone cape's cable headers, we use 10-conductor 0.05"-pitch ribbon cable. 2×5 position 0.1"-pitch female IDC connectors crimped onto the ribbon cable are used to connect both the cables to the cape headers and the sensor boards to the cable. The cable itself is run along the structure of a server cabinet and down the front of the cabinet door. Zip ties are typically used to secure the cable in place.

Chapter 4

Top-level Network Architecture

4.1 Overview

Moving the system to a state that is able to handle sensing for an entire data center required the development of an architecture that can scalably handle many sensor-controller subsystems operating simultaneously and separately. To accomplish this goal, we have designed a top-level system that uses no additional infrastructure in the data center facility and allows for nearly instant access to sensor data and powerful and flexible system management.

Each BeagleBone, as before, connects to the Internet via an Ethernet cable, which are readily available in a datacenter, as network cables run above and to every server cabinet to provide their servers with Internet access. However, instead of hosting a web application to display sensor data via Apache running on the BeagleBone, the device communicates sensor data back to a central server. The devices exist on a private subnet with no Internet access, preventing unauthorized access to sensor data or remote manipulation of the BeagleBone controllers by outside parties.

The server established to accept the sensor data is the same one which serves the DCSense web application. The servers and their architectures are discussed here, while the application is discussed in Chapter 6.

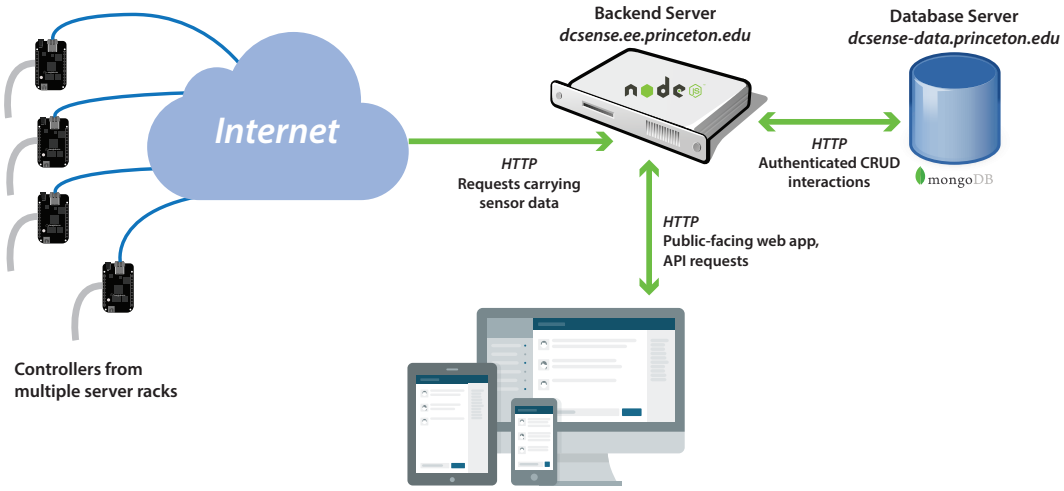


Figure 4.1: Overview of top level system

4.2 Primary DCSense Server

The primary server is a VM provided by the Department of Electrical Engineering. It sits at `http://dcsense.ee.princeton.edu` and runs Ubuntu Server Edition, and a custom Node.js server running on port 80 handles all of its functionality. It performs the following functions:

- Static serving of the frontend AngularJS web application
- Securely implementing access control for the web application
- Providing an API for the frontend application and handling all such AJAX requests
- Handling incoming sensor data from many BeagleBone sensor controllers and safely transmitting them to the storage database in an easily queriable format

- Acting as management console for deployed BeagleBone devices (only this server has network access to the BeagleBones' subnet, so any management must go through the server)

4.3 Discrete Database Server

The purpose of this thesis is twofold. The first is to design and implement the system being described to be functionally useful to the data center operators for which it is deployed. The second is to collect data for the research purposes of Prof. Wentzlaff. This second goal necessitates that temperature data be gathered approximately every five seconds by each sensor. For a six-month data gathering period over the full 500-sensor suite, that amounts to approximately 1.5 billion readings and 172 GB of storage space. Querying a database with more than a billion records is intractably slow, even using a database designed for large collections like MongoDB. To solve this problem, we decided to store the readings twice: for each BeagleBone's group of sensors, every single measurement would be stored in one collection, while only every ten minutes, the incoming group of readings is also stored in a different collection which is accessible to users via the frontend application. This much smaller database collection ensures that the system can continue to operate quickly even over long periods of time, while higher-time-resolution data can continue to be collected for research purposes.

The amount of storage required necessitated the use of a dedicated database server. We procured a VM with 400GB of storage space to fill this role. It also runs Ubuntu Server Edition.

MongoDB runs as well, and the database has authentication enabled, so all traffic between the main DCSense server and the database server are authenticated using MongoDB's built-in authentication. We were careful to avoid including the MongoDB authentication details, which are needed by the Node.js server, in the publicly available code in our project repository.

Chapter 5

A Robust Controller Codebase

5.1 Overview

The BeagleBone Black controllers utilize a custom application written in Python to handle communication with the connected sensors (and now, to transmit sensor data to our central server). A version of this application was inherited at the outset of this thesis, but substantial changes needed to be made over the course of the project. The source code for the application can be found in Appendix C and in the project's GitHub repository [13] in the `/beaglebone-controller` directory.

5.2 Major Changes

Readying the system for a production environment necessitated a nearly complete overhaul of the Python code running on each BeagleBone, interacting with the sensors, and gathering climate data. The code written for [8] accomplished the goal of detecting the sensors connected to the BeagleBone and gathering temperature measurements from them at a regular

rate. However, there were a number of changes that needed to be made both to improve the application's robustness and to add the necessary features for the project's new architecture.

5.2.1 Long-term Stability

The controller code base, as inherited, suffered from frequent crashing. Because I^2C does not support any kind of error correction, any errors that appear in the messages traveling across the bus – which might occur when cables are moved or sensors are connected or disconnected – cause an exception to be thrown in the `python-smbus` library that handles the bus communications. In the code, those exceptions caused a complete crash of the script. One of the first changes made during this thesis was to establish procedures for error handling. In the current code base, errors aren't just caught to allow for continued execution, but a new board status module keeps track of any errors that occur, the exact details of those errors, and other relevant data (e.g. how many tries has the controller made since it was able to obtain good readings from the sensors). The module also communicates those errors back to the server alongside every submission of sensor data, so the server can keep track of and quickly identify problems at a distance. This allows for remote detection and diagnosis of problems in the system without the need either to check each BeagleBone manually or make an onsite visit to an affected facility.

These changes have been very successful in practice. After their implementation, we were able to achieve continuous operation for at least weeks at a time, and as far as we can tell,

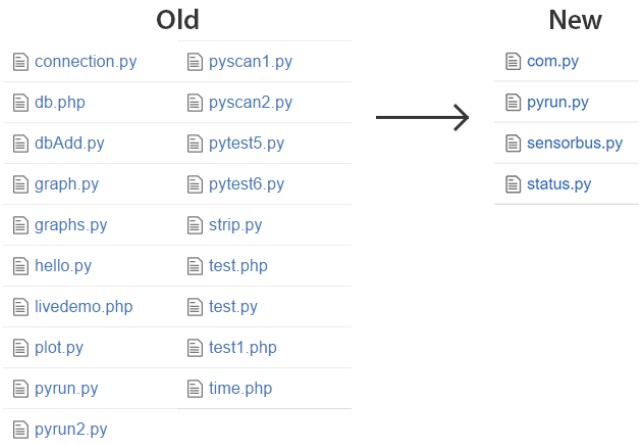


Figure 5.1: Top-level comparison of old and current BeagleBone code

under normal conditions, the BeagleBones should be able to continue running and collecting measurements indefinitely.

5.2.2 Code Semantics

The inherited code base was extremely disorganized. There was no coherent sense of which files performed which task. Functions were also named meaninglessly (e.g. "`pytest5()`"). Much work was conducted during this thesis to reorganize the code into a good example of object-oriented programming principles. The main functions were consolidated into organized classes, the effect of which can be seen in Figure 5.1. Note that some of these changed reflected pruning of unneeded features, though additional features were also added, as discussed in §5.2.3. The new code has also been thoroughly commented to aid anyone who may develop it in the future.

5.2.3 New Features

The new design of our climate monitoring system necessitated the addition of new features — namely, formatting the collected sensor data in an organized data structure with necessary metadata and then sending that formatted data to the DCSense central server. These features were implemented and thoroughly tested during this thesis.

5.3 Codebase Overview

Below is a brief overview of the functionality provided by each file in the new code base. The full source can be found in Appendix C.

- **`pyrun.py`:** The project's main module. Initiates the gathering of sensor data, reporting of errors, and transmission of readings to the DCSense server.
- **`sensorbus.py`:** A class which represents one I^2C bus with connected sensors attached to the BeagleBone. Two instances of this class are instantiated by `pyrun.py` to represent the two buses used for sensors. This class handles all communication with the sensors.
- **`com.py`:** This file handles transmission of gathered sensor data to the DCSense server via an HTTP request. Before the request is sent, the process creates a fork of itself so that the HTTP request can be sent in the background without blocking the main execution flow of the remainder of the program.
- **`status.py`:** This class represents the error status of the BeagleBone as described in §5.2.1.

Chapter 6

Monitor Web Application

6.1 Overview

This chapter describes the functionality provided by and technologies powering the DCSense web application, which allows data center operators to view data gathered by the sensor-controller subsystems deployed in their facilities. This section of the thesis refers entirely to software, but due to the large quantity of code, it is not reproduced in the appendices. Instead, for full implementation details, please browse the project's GitHub repository [13] in the `/server-app` directory.

6.2 Demonstration

The application is live in production at `http://dcsense.ee.princeton.edu`. Please feel free to log in using the username `testuser` and password `password`.[1]

[1] As this is the account of a simulated user, it has the capability to create and destroy data that is useful for demonstration purposes. Please feel free to browse around the site and explore its functionality, but please refrain from making any changes to the server configuration so that other users can have the same demonstration experience. However, any changes made here will *not* affect the experience of real user accounts on the site.

6.3 Stack

The application was built using the MEAN[2] stack, a fully-JavaScript stack designed for fast single-page web applications, like the one being discussed. Because of the sheer quantity of data we needed to store and manage, MongoDB was selected as the database technology due to its high performance on large database sizes. As a NoSQL database which uses BSON (a slightly modified form of JSON[3]) to store data, it was sensible to use frontend and backend frameworks that work nicely with JSON. These factors combined with prior experience led to the selection of Node.js as our backend server and API handler (powered by the Express.js framework) and AngularJS as our frontend framework for single-page application development. jQuery, Twitter's Bootstrap, and the d3 visualization library were also used quite a bit to build the user interface.

6.4 User Interface

The application has primarily one page, a dashboard designed to easily access climate data for an entire data center facility. Figure 6.2 shows an example of the dashboard in use.

Each user can see data only from the BeagleBones present in the facility to which they are assigned. This allows the system to be scalable to multiple physical facilities and multiple users across those facilities. In fact, the application will support any future installations of the sensor system, even ones outside the facilities originally considered during this thesis.

[2]MongoDB + Express.js + AngularJS + Node.js
[3]JavaScript Object Notation

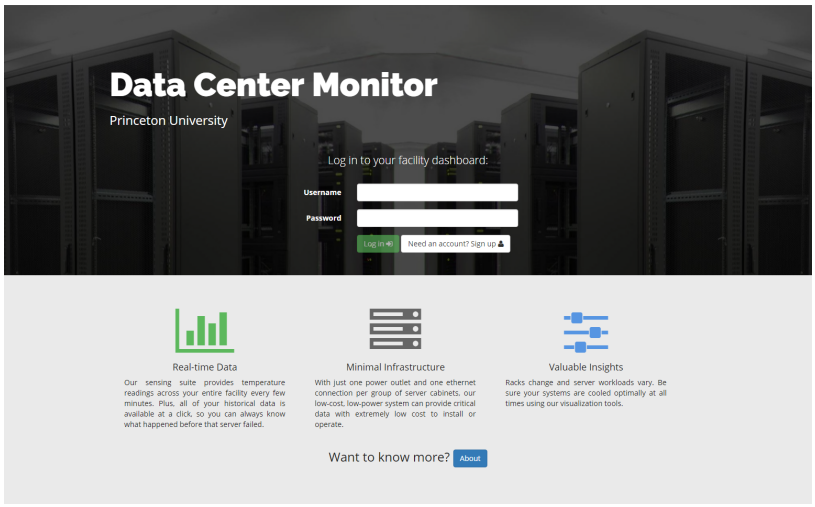


Figure 6.1: Welcome/Login Page

6.4.1 Server View

For each BeagleBone that transmits sensor data back to the application, the site creates a representation of the group of server cabinets (up to 10 cabinets wide) that the BeagleBone is mounted on. Either an administrator or a user can map each sensor for which readings are available to its location on that server cabinet representation. In the server view, a user selects a date and time from which to view climate data, and the appropriate temperature measurements are displayed directly on the server cabinet graphic, with readings shown exactly where their corresponding sensor is positioned. Dragging a large slider allows users to scrub back and forth through time to see the state of the server cabinet group at any time in the past two weeks. In Figure 6.2, we show an example 2-wide server cabinet group displaying its data. The application assumes that each cabinet has five sensors mounted vertically on its exterior, which is how they arranged in our deployed instances.

The interface allows users to select which which server cabinet group (i.e. which BeagleBone) to view climate data for, and the server view discussed here is the default display mode. Clicking on any sensor causes a graph to be displayed, showing the sensor's measurements over the last two weeks. This interactive graph gives a visual indication of how a server cabinet's climate may have changed over time, especially if any changes were out of the ordinary.

Sensor Mapping Tool

Clicking the lock icon in the bottom right corner of the server view allows users to use an intuitive tool to map the address of the sensor present at a given location on a server cabinet to that location on the web application's server representation. Note from Figure 6.3 that users do not need any special knowledge of the system to perform this mapping. Instead of selecting the address of the sensor in question, which is not immediately apparent to an unfamiliar user, it instead asks them to select the sensor based on the appearance of its address switches, which can be seen directly on the board. This process is likely done during installation by someone familiar with the system, but could for this reason be done by a data center operator with simple instructions.

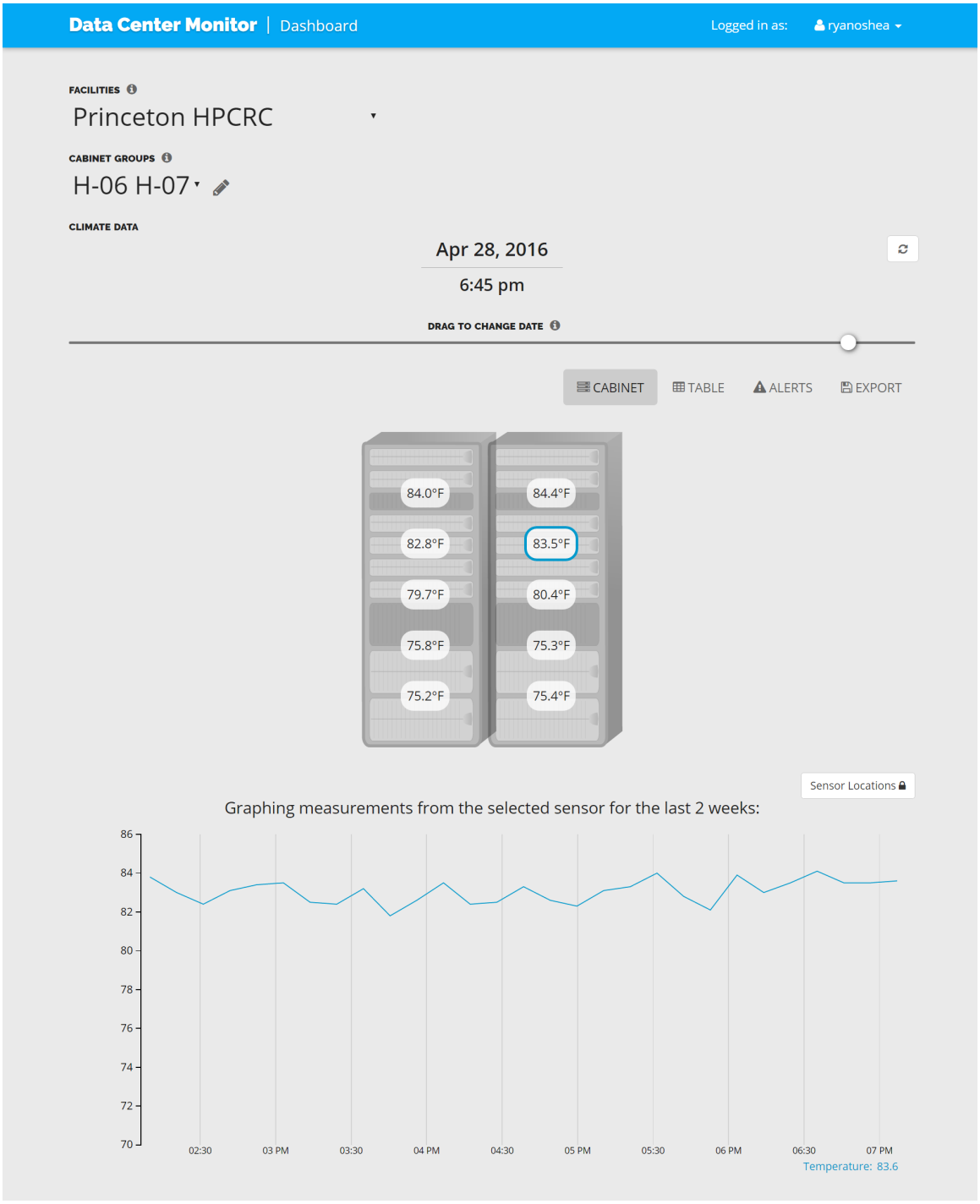


Figure 6.2: The facility dashboard, showing data for a group of server cabinets at the Princeton HPCRC, where the user is viewing the temperature history for a particular sensor

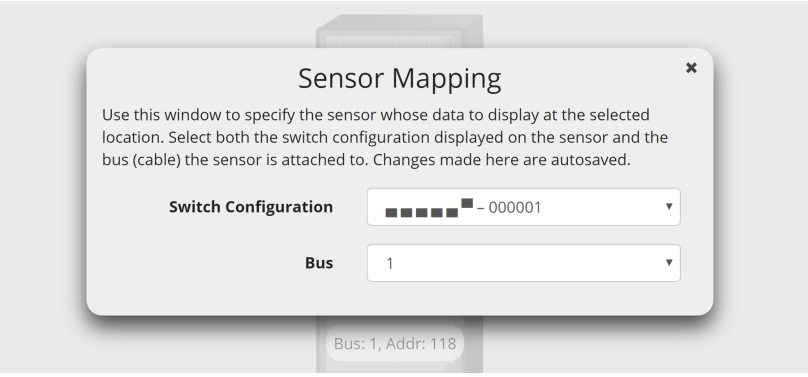


Figure 6.3: The sensor location mapping tool, showing how a user can indicate that the sensor at this location had its address switches configured as shown

6.4.2 Table View

Users can instead choose to view the data for a particular time (as determined by the timeline slider discussed above) as a table. This may be useful for users who want more organized access to the data, for example to copy-paste it for some other purpose. Clicking any sensor listed in the table switches to server view, selects the sensor at its location on the cabinet group, and displays its history graph.

6.4.3 Alert View

This view provides a tool that users can use to see a list of any temperatures above a provided threshold that have been measured at the server cabinet group either in the last two weeks or during a provided timeframe. Figure 6.4 shows the tool in action.

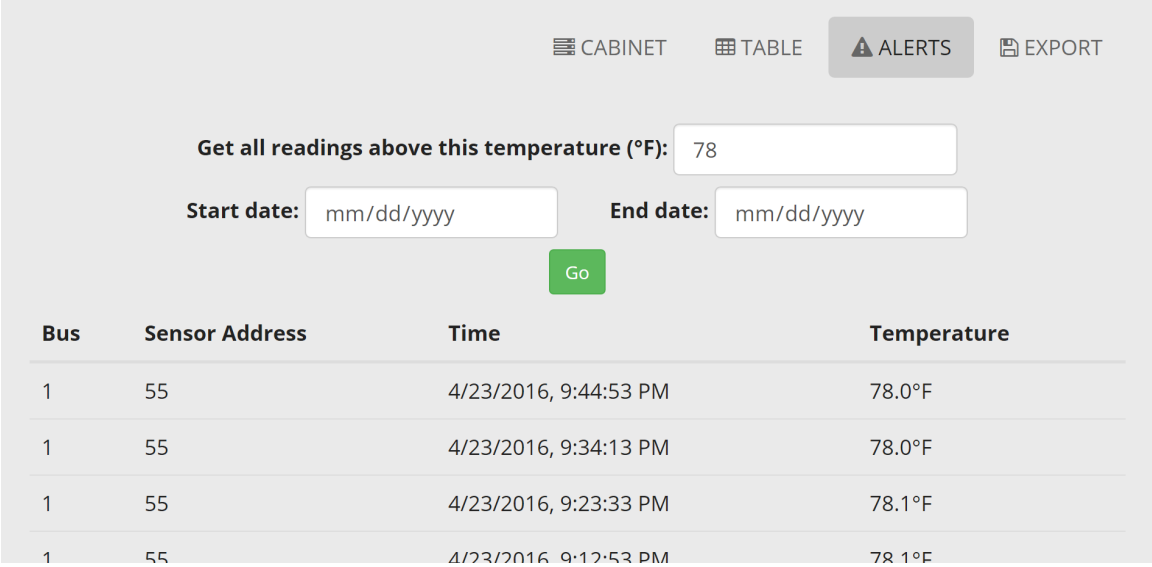


Figure 6.4: The tool provided for searching for high temperature readings in a given time frame

6.4.4 Export Tool

We have also provided an export tool that allows users to export all data from across their entire facility within a given time frame. The exported data is downloaded to their computer as a CSV with all available metadata. This provides utility both for the data center operators who might want to use a more sophisticated tool to analyze their facility's climate and for the research purposes of Prof. Wentzlaff and the Princeton Parallel Group.

6.5 Backend

The Node.js application comprising the application's backend serves many vital functions. This section explores them in depth.

6.5.1 User Authentication

The application uses its own authentication system. Developing this in a secure way was the target of a substantial amount of development work, as we want to be sure that no user can access another user's data, since data centers value security quite highly. User accounts are stored in a fairly cryptographically secure way in the database. The application stores a user record in the following way:

- **Username:** The user's username in plaintext
- **Token:** The password is *never* stored in plaintext. Instead, we store a unique cryptographic hash (technically, an HMAC) of the user's username, password, and a unique 20-character salt created when the user signed up.

$$\text{Token} = \text{HMAC}_{\text{SHA256}}\left(\text{username} \| \text{password} \| \text{salt}\right)_{16}$$

- **Salt:** The random 20-character string described above
- **Access Level:** A number indicating whether the user is a normal user or an administration

When logging a user in, the server generates a random 25-character string to serve as a session token. That token is stored in a database collection along with the user's username and an expiration date one week in the future. That record is also sent to the client to be saved as an authentication cookie.

Page Access Control

The frontend application relies on an AngularJS service to control whether or not a a page is loaded in the browser for the currently logged-in user. This service sends the authentication

cookie present in the client's browser to the server's session status endpoint, which indicates whether or not the page should be loaded. Using a service ensures that no data is sent to the client until the deferred promise object is resolved and the server has confirmed that the user has access.

Data Access Control

All AJAX requests for data to the backend (including sensor data, details about server cabinets/facilities, etc.) are authenticated using the authentication cookie mentioned above. In addition, the server checks the access level of the user to determine whether or not the user is authorized to make the type of request received. This goes further to specific server-side functions that check whether or not a user has permission to access data regarding a specific facility or cabinet group. Most API handlers are arranged as callbacks from these permission-checking methods, executing only when access is granted.

Known Vulnerabilities

While we are confident of the security of this user authentication scheme for most use cases, it does suffer from a few unlikely vulnerabilities. If an attacker were to copy the authentication cookie from a client's machine and install it in their own browser, they would be able to authenticate as that user, as the server does not currently check that the client has not changed. Also, although the database server uses MongoDB's user-based authentication, the data is not encrypted at rest, so an attacker with physical access to the server could

compromise the security of the data we would like to keep protected (though they would not be able to read user passwords due to the way the backend stores them). However, both of these scenarios are outside the threat model we considered and are improbable enough to not have been taken into account in building the system.

6.5.2 Sensor Data Storage

The server accepts and stores readings POSTed to

`http://dcsense.ee.princeton.edu/api/sensors/submitreadings`

in the format of an array of objects following the following fields:

- **'controller':** The MAC address of the device (BeagleBone) that submitted the readings formatted as a string ("xx:...:xx"). Used to link reading objects together by the server cabinet group they share.
- **'bus':** An integer (1 or 2) used to identify the cable upon which the sensor that provided this reading resides
- **'sensor_addr':** The decimal I^2C address of the sensor that provided this reading
- **'time':** An ISODate object (standard in JavaScript or Python) representing the exact timestamp of this reading according to the device. This value is optional, as the readings are timestamped for storage server-side.
- **'temp':** Floating point number representing the measured temperature in degrees Celsius

Timing

The server expects to receive an array of readings from every controller device once every 5 seconds. Each of these readings is stored in the database `ResearchReadings` collection,

which is not used by the frontend web app and not accessible via the server's REST API. This collection is meant to be used privately by Prof. Wentzlaff and his group for research purposes.

Every 120 received sets of readings (approximately every 10 minutes), the readings are also saved to the `Readings` collection, which is used by the application as the data to display data to users.

By only storing a subset of collected data in the production collection, we can ensure that that users have access to a fast web application. The number of readings in the smaller collection should scale very well, whereas the higher-time-resolution data would be difficult to query at an acceptable speed due to its massive size.

Extensibility

The server is designed to collect readings from sensors in a device-agnostic way. There is no feature specific to the BeagleBone controllers that we use for this version of the system that is relied upon by the web application. Instead, the application was specifically designed to be as decoupled as possible from the hardware providing it with sensor data, so that the application can be used in future revisions without the need for changes.

6.5.3 Users, Facilities & Server Cabinets

We have discussed already how the application organizes sensor data according to the controller device that submitted the data, representing that device as a group of server cabinets with its own custom mapping of sensors to locations on the cabinet group covered by the controller device. Moving up from that, the application conceptualizes controllers in groups called facilities, which represent a specific location in which the sensor system has been deployed. Administrators use a special control panel to assign controllers for which data exists to the appropriate facility. In a normal use case, each controller would only be assigned to one facility. However, the system supports scenarios where the administrators would like to grant a user access to a subset of a full facility (for example, if the Princeton Computer Science Department would like to be able to access data for their servers inside the Princeton HPCRC). In these cases, administrators could create two facilities, where the first facility includes all controllers in the real-world facility and the second facility contains only a subset. Any customizations (most importantly, controller renaming) made by one user will not affect the controller seen by another user. Only the data will remain the same.

The same tool allows administrators to assign users to specific facilities, which grants the users access to data from all controllers in that facility. The system can easily grant access to a facility's data to multiple users, which we imagine would be a common use case.

Finally, administrators have access to all data, organized by the facilities present in the database. This access allows for extremely easy troubleshooting in the event that a user has

difficulty accessing their data or managing sensor mappings. For full details regarding how these mappings and configurations are represented in the database, please see Appendix D.

Chapter 7

Deployment Efforts

7.1 Strategic Partnership with Princeton HPCRC

From the outset of the project, the Princeton HPCRC was considered as a potential location to install a first group of sensors. When approached, the HPCRC staff was extremely willing to grant us access to a limited number of server cabinet groups across their data hall for a temporary installation of our system.

Starting in the Spring semester of this thesis, we began talks with the HPCRC to plan our installation. We ordered 500 copies of the sensor board and planned originally to attempt to install 10 BeagleBones, each with up to 50 sensors (covering up to 10 contiguous server cabinets each). However, it took a significant portion of the semester to gather all the required parts needed to begin installation, so that process began in April with somewhat reduced scope.

7.2 Obstacles

Our deployment efforts ran into a number of obstacles, each of which somewhat limited the quantity of sensors we were able to get installed in the facility by the conclusion of the project.

Figure 7.1: View of one of the server cabinets at HPCRC bearing our sensor system

7.2.1 Networking

While the solution for connecting the BeagleBone controller's to a Department of Electrical Engineering subnet was functional, the work at the HPCRC required to provide each of our deployed BeagleBones with network access was completed just days before the due date of this thesis. For security reasons, all networked devices installed at HPCRC must have ports manually opened before they can send or receive traffic outside the data center. The Office of Information Technology is responsible for making these changes, but during our deployment efforts, their ticket queue had grown exceptionally large, and so they were unable to complete our relatively low-priority request until the last possible moment.

Once the deployed BeagleBones were brought online, we were able to make some of them fully operational in the extremely short time frame provided.

7.2.2 Signal Integrity

A brief test of the deployed systems was conducted onsite using the first deployed group of sensors, before establishing network access. During that test, we encountered some issues obtaining readings from the sensors in most cases. Further investigation clarified that the length of cable needed to traverse the server cabinet and position the sensors where we needed them (around 15ft in length) was causing problems with the I^2C signals traveling down the cable. This issue was not entirely unexpected, as we previously had decided to reduce the pull-up resistor value to 4.7kΩ for the buses in order to compensate for the long cable lengths needed (reducing the resistance helps compensate for a large RC time constant accumulated over a long wire). We had expected that cable length might become a problem when running cable across three to four server cabinets in a row. However, we had not adequately anticipated the same issue arising when covering only a single server cabinet with sensors.

Figure 7.2: A BeagleBone controller secured in the top section of a server cabinet at HPCRC

System Design Revisions

After further testing offsite, we examined the waveforms of the I^2C messages traveling down a similar-length bus, comparing the signal at the near and far ends of the bus. We noticed relatively slow rise and fall times in the signal transitions, indicating a higher-than-optimal RC time constant. We also saw some significant crosstalk among the signals, even seeing small jumps in our ground wire during transitions on the SDA/SCL lines.

To address these issues, we made some changes to the design of the BeagleBone cape discussed in §3.3.3. Originally, we had neglected to ground the unused pins in the cable headers, resulting in a number of floating cables susceptible to crosstalk. The updated cape design ensured that all of those unused conductors were grounded. We also decided to lower the pull-up resistor value once again, this time from 4.7kΩ to around 2.2kΩ.

After making these relatively simple changes, our signal integrity problems were rectified. Using the new design, we deployed a suite of sensors on the server cabinet in the EE Parallel Lab, which we are now using for our system demo in the test account on the web application.

Should signal integrity become a problem again in a future instance of the system employing longer cables, the next step would be to configure the BeagleBone controllers to user a lower frequency for sending messages over the I^2C bus. Beyond that, the use of shielded (but more expensive) cable would be worth considering if the problems persist.

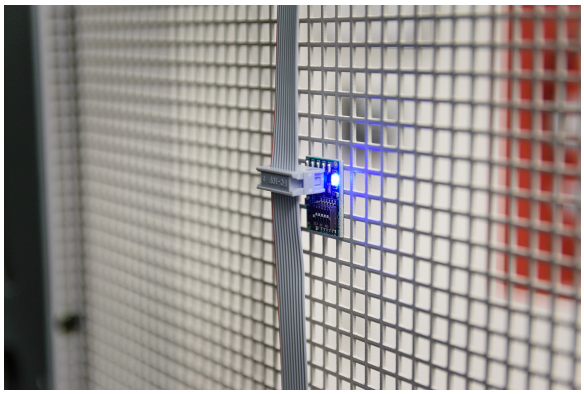


Figure 7.3: View of one of the sensor boards on a server cabinet door at HPCRC

7.3 Deployed Instances

Due to the problems witnessed near the start of our installation, the decision was made to drape each bus cable across only one server cabinet for the remainder of the deployed systems. In an unrelated development, HPCRC staff revised their commitment to the project due to time constraints approaching the end of the semester, and recommended that we install only up to five BeagleBones with attached sensors. In light of the relatively significant portion of materials whose deployment we were forced to sacrifice, the body of sensors deployed at the time of writing is limited to four BeagleBone controllers, each covering 2 server cabinets for a total of 40 sensors.

Despite the obstacles we encountered during our deployment efforts, the hardware currently deployed (which features the older, less stable BeagleBone cape circuit) has been successful to varying degrees at the close of this project. Two of the BeagleBone-sensor systems are

working as expected[1] and are currently sending live temperature measurements to our DC-Sense web application, where staff from the HPCRC can monitor those four server cabinets. Figure 6.2 shows data from one of these cabinet groups taken shortly before the time of writing.

Two of the deployed BeagleBones are experiencing electrical issues related to the length of the bus. Updating the cape should solve these issues, but we did not have enough time to make this change before the project's conclusion.

[1] One of the BeagleBones is unable to obtain readings from just one of its attached sensors, though every other sensor works as intended. Updating the cape will likely solve this problem.

Chapter 8

Next Steps

This chapter discusses immediate next steps that could be accomplished relatively easily by the next person to take on this project, taking into consideration the materials already created and available to them on day one.

8.1 Expansion of Climate Sensing Capabilities

The current system only gathers data on temperature. However, data center operators may be interested in other metrics to maintain the health of their servers. Particularly, we are interested in humidity and vibration. Combined humidity and temperature sensors are quite common, but even the cheapest ones cost more than twice what the lone TMP175 temperature sensor costs, and since that cost increase is non-negligible and multiplied by each board, of which there may be hundreds in a given deployment of the system, the decision was made to postpone the addition of more metrics until another iteration of the project is undertaken.

Introducing new sensors presents additional design challenges. In order to keep costs low, the board needs to incorporate the sensors without the need for a microcontroller to handle communication or data readout from the various new devices. As such, the additional sensors would need to operate over the same I^2C bus as the temperature sensors. The address space of the two sensors could, depending on sensor selection, collide, meaning that potentially fewer boards overall could be attached to one BeagleBone at a time. We could resolve this issue by using an I^2C switch, which would allow the BeagleBone to communicate to devices on something like 8 different I^2C buses using only one bus from the BeagleBone's perspective. That said, there may still be limits to the amount of power the BeagleBone could provide, so hardware might be needed to provide a more robust voltage source if significantly more devices begin to draw power from the controller. Again, the problem is solvable, but will require additional design consideration. The metrics provided might provide valuable insights to the potential user base and could be useful for research purposes, so they may well be worth including.

8.2 Hardware Reliability Improvements

First, though our deployed instances of the system at HPCRC were relatively successful, the next instance of this project should replace the old BeagleBone capes deployed on those systems and replace them with the updated design described in the last chapter.

Second, while our signal integrity issues encountered in the previous chapter were resolved by some hardware changes made late in the project, it would be worthwhile to consider

making some further changes to ensure that the signals are reliable as possible. Specifically, lowering the I^2C bus frequency would likely help ensure more reliable signal transmission over even longer cables. Also, due to the fairly noisy environment found in a real data center (and the lack of error-correction in the I^2C protocol), it may be helpful to look into alternatives to the cheap unshielded ribbon cable currently used by the system to carry the bus and power. Budgetary and time considerations removed that possibility from the scope of this thesis, but it should be considered in any future version for the sake of providing the best possible signal integrity.

8.3 Full Deployment

Full deployment of a large-scale instance of the system was, until recently, a goal of this iteration of this project. Sadly, late-term obstacles prevented us from achieving that goal this time around. Instead, only a small number of sensors were deployed relative to the number that were produced. Any further work on this system should immediately attempt to remedy that situation. Solving the hardware challenges mentioned in the previous section would likely mean that large-scale deployment could begin almost immediately. The system is extremely close to being ready for large-scale use by any type of user. The only obstacles are those hardware issues and the time required to deploy the remaining 460 sensors.

8.4 Optimizing for Research Utility

Finally, the project could benefit from more attention being paid to the research value of its data gathering. Most of the work conducted during this thesis focused on providing a useful tool for end-users who would be using the Data Center Monitor web application. However, a twin goal of the system itself is to provide data that can be useful for future research conducted by Prof. Wentzlaff and the Princeton Parallel Group.

In the project's current state, that data is available.[1] However, the means of accessing it are limited to accessing the database server via SSH and exporting the `ResearchReadings` collection into an acceptable format such as CSV using the very fast `mongodump` tool. Despite the fact that the persons who might need to use this method to access the data have the technical competency to do so, a much better alternative would be to build some easier-to-use tool for exporting data, much like the tool provided for the lower-resolution production data on the web application. Developing such a tool was briefly attempted, but doing so using traditional methods turned out to be unsuitable for such a large quantity of data. Without some other method of handling the massive datasets, the Node.js server would simply freeze for long periods when attempting to export so much data. A more clever solution making use of `mongodump` could easily be implemented, either as a small scripting tool destined for use on the computers of the project's managers or as a feature of the administrator section of the web application.

[1] Or, at least it will be once a full-scale deployment across the HPCRC is completed — a small amount of data-gathering capability exists now, and the infrastructure needed to collect and archive that data is firmly in place thanks to this thesis.

Chapter 9

Conclusion

This senior thesis produced a functional, scalable, flexible, and accessible network of temperature sensors aimed at deployment in data center environments for monitoring one of the most important factors involved in everyday operation of those facilities. Being able to identify hot spots in a data hall — places where the massively expensive and heavily tuned cooling systems present in data centers have failed to do their job adequately — provides an important tool for data center operators who strive to optimize their cooling strategies at all times. Even with a professionally balanced cooling system, servers can move in and out and server workloads can fluctuate by the hour. The heat distribution of data centers thus fluctuates unpredictably, and the operators of these facilities require tools like the one we have developed to ensure their systems run as effectively as possible.

During our real-world deployment of the system, we encountered some scheduling problems that prevented the full-scale system deployment we had hoped for. However, we are confident in our system design, and the currently-functional sensors at the Princeton HPCRC and the EE Parallel Lab are a testament to that work. More sensors can soon be installed to complete our deployment at HPCRC.

Over the course of this year, the system's sensor board was totally redesigned, resulting in greatly improved electrical characteristics, which should aid in reliability during long term operation. Its size was reduced by 47% to fit into a new footprint smaller than the size of a quarter — virtually guaranteed to be unobtrusive enough for even the most appearance-minded data centers. Finally, despite the use of smaller (and often more expensive) components, the price of the board was reduced by 10%, with the vast majority of the cost coming purely from components.

An entirely new architecture was designed and implemented, allowing the system to scale past a single BeagleBone for the first time in the project's history. In fact, the new architecture is virtually infinitely scalable, wherein the only limitations on the number of systems that can be simultaneously deployed are those regarding database capacity — an almost entirely unrelated problem with its own professional-grade set of solutions. The new architecture allows for deployment in any number of facilities across any number of users in any kind of combination.

A powerful web application was built in its entirety to provide an intuitive, highly functional management console for the installed climate monitoring systems. It caters to the needs of our prospective users, providing tools for viewing climate data across a whole facility at a moment in time, examining climate history for a particular area or sensor, instantly finding times and places where temperatures exceeded acceptable levels for even a moment, and exporting data for further analysis in a proper data analysis tool. The application is a modern, fast product that fulfills the expectations of the user base and makes use of

advanced, reliable web frameworks that will ensure its continued operation well into the future. In fact, it is designed to continue to serve the purposes of the project even if the hardware used to gather climate data were to change. Because of the technological stack used, anyone comfortable with JavaScript could easily build upon the existing platform to incorporate any new features that might be needed.

These major improvements to the Princeton Parallel Group Data Center Monitoring project are the product of countless hours of design and troubleshooting, hundreds of commits to a well-maintained code base, and careful consideration on the part of myself, my predecessors, and the project's advisor, Prof. Wentzlaff. I have grown a great deal in my abilities to plan and execute a complex project, design a reliable PCB aimed at mass-production, and to implement robust embedded and web software solutions. Finally, all the work performed to enable these upgrades was carefully documented, either as comments in the code base, instructive READMEs in the project repository, photographs and schematics included side-by-side with source files, or the descriptive chapters of this thesis report. I am confident that anyone with the appropriate skill set attempting to continue this project will find a wealth of supporting documentation to ensure that progress can begin rapidly.

I would like to reiterate my thanks to everyone mentioned in the Acknowledgements at the beginning of this report, especially to Prof. Wentzlaff for his guidance throughout this senior thesis. Thank you for reading.

Appendix A

Sensor Board Bill of Materials

Description	QTY	Ref Des	Mfg	Mfg P/N #	Distributor	Distributor P/N #
FCI CONN HDR DUAL 0.100" SMT	1	JP1	FCI	54202-G0805LF	Digi-Key	609-4723-ND
CTS SWITCH DIP HALF PITCH 6POS 50V (0.05")	1	S1	CTS Electrocomponents	218-6LPST	Digi-Key	CT2186LPST-ND
TDK CAP CER 15UF 10V JB 0805	1	C2	TDK Corporation	C2012JB1A156M085AC	Digi-Key	445-11407-2-ND
DIALIGHT LED BLUE CLEAR 1208 R/A SMD	1	D1	Dialight	5988391117F	Digi-Key	350-3028-1-ND
VISHAY DALE RES SMD 1K OHM 1% 1/8W 0805	2	R1/R2	Vishay Dale	CRCW08051K00FKEA	Digi-Key	541-1.00KCTR-ND
SAMSUNG CAP CER 0.1UF 50V Y5V 0805	1	C1	Samsung	CL21F104ZBCNNNC	Digi-Key	1276-1007-2-ND
TI SENSOR TEMPERATURE SMBUS 8MSOP TMP175	1	U1	TI	TMP175AIDGKT	Digi-Key	296-19882-6-ND

	URL
FCI CONN HDR DUAL 0.100" SMT	https://www.digikey.com/product-detail/en/54202-G0805LF/609-4723-ND/4240461
CTS SWITCH DIP HALF PITCH 6POS 50V (0.05")	https://www.digikey.com/product-detail/en/218-6LPST/CT2186LPST-ND/267321
TDK CAP CER 15UF 10V JB 0805	https://www.digikey.com/product-detail/en/C2012JB1A156M085AC/445-11407-2-ND/3948643
DIALIGHT LED BLUE CLEAR 1208 R/A SMD	http://www.digikey.com/product-detail/en/5988391117F/350-3028-1-ND/3906503
VISHAY DALE RES SMD 1K OHM 1% 1/8W 0805	http://www.digikey.com/product-detail/en/CRCW08051K00FKEA/541-1.00KCTR-ND/1175637
SAMSUNG CAP CER 0.1UF 50V Y5V 0805	http://www.digikey.com/product-detail/en/CL21F104ZBCNNNC/1276-1007-2-ND/3886665
TI SENSOR TEMPERATURE SMBUS 8MSOP TMP175	http://www.digikey.com/product-detail/en/TMP175AIDGKT/296-19882-6-ND/1120838

Appendix B

Sample Quotes for Full Sensor Board Fabrication & Assembly

For a sense of the full price, including tooling, NRE, stencils, etc., of getting the new sensor board fabricated and assembled professionally, a selection of totals from quotes obtained for this project are shown below. The option selected was a mixture of the quotes shown below, as we opted for fabrication with Advanced Circuits and assembly at Green Circuits, paying for shipping between the two.

Vendor\Quantity	**50**	**100**	**500**	**Selected Lead time**
Advanced Circuits	$1,140.00	$1,470.00	$4,055.00	5 weeks
Advanced Assembly	$1,897.03	$3,138.52	$7,727.04	25 days
Hughes Circuits	$3,501.50	$4,052.00	$8,780.00	15 days
Green Circuits	$1,682.95	$2,045.00	$3,968.00	4 weeks

Appendix C

BeagleBone Controller Code

pyrun.py:

```
1  import sensorbus
2  import schedule
3  import time
4  from status import Status
5  import com
6
7  def pyrun():
8
9      board_status = Status().status
10
11     # Detect devices and get readings from all devices on both buses
12     timestamp = time.time()
13     bus1 = sensorbus.SensorBus(1, board_status)
14     readings1 = bus1.get_readings(bus1.scan(), timestamp)
15     bus2 = sensorbus.SensorBus(2, board_status)
16     readings2 = bus2.get_readings(bus2.scan(), timestamp)
17     readings = readings1 + readings2
18
19     # Traces
20     if not board_status['err']:
21         print len(readings)
22     else:
23         print board_status['type']
24         print board_status['msg']
25         print board_status['ioerr_addr']
26
27     # Send readings to server (non-blocking)
28     com.send_to_server(readings, board_status)
```

```

# Re-run every five seconds
schedule.every(5).seconds.do(pyrun)
while 1:
    schedule.run_pending()
    time.sleep(1)
```

sensorbus.py:

```
import string
import smbus
import subprocess
import time
import calendar

class SensorBus:

    def __init__(self, bus, status):
        self.status = status
        if bus == 1 or bus == 2:
            self.busnum = bus
            self.bus = smbus.SMBus(bus)

    def scan(self):

        # Grab i2cdetect output for given bus
        scanoutput = open("scanoutput.txt", "w")
        subprocess.check_call(['/usr/sbin/i2cdetect',
          '-y', '-r', str(self.busnum)], stdout=scanoutput)

        # Start reading i2cdetect output
        f = open("scanoutput.txt", "r")

        # Skip the top line line
        f.readline()
        line = f.readline()

        # Parse text for device addresses, place them in array
        i = 0
        addresses = []

        #check for empty string
```

```
34          while len(line) != 0:
35              a,temp,b = line.partition(' ')
36
37              while len(b) != 0:
38                  a,temp,b = b.partition(' ')
39
40                  if all (c in string.hexdigits for c in a) and not not a:
41                      val = int(a, 16)
42                      addresses.append(val)
43                      i += 1
44
45              line = f.readline()
46
47          subprocess.check_call(['rm', '-rf', 'scanoutput.txt'])
48          return addresses
49
50      def get_readings(self, addrs, timestamp):
51
52          readings = []
53
54          if len(addrs) > 0:
55
56              i = 0
57
58              # Read temp data (in degrees C) for each sensor in addrs
59
60              try:
61                  while i < len(addrs):
62                      # Issue command to take a reading
63                      self.bus.write_byte_data(addrs[i], 01, 96)
64                      # Read back the returned reading
65                      tmp = self.bus.read_word_data(addrs[i], 00)
66                      # Convert to degrees C
67                      temperature = (tmp & 255) + ((tmp >> 12) * (1/16.0))
68                      reading = {
69                          'controller' : self.status['mac'],
70                          'bus' : self.busnum,
71                          'sensor_addr' : addrs[i],
72                          'time' : timestamp,
73                          'temp' : temperature
74                          }
75                      readings.append(reading)
```

```
76                      i+=1
77              except IOError, err:
78                  self.status['err'] = True
79                  self.status['type'] = "IOError"
80                  self.status['msg'] = err
81                  self.status['ioerr_addr'] = addrs[i]
82                  self.status['ioerr_bus'] = self.bus
83                  self.status['num_tries'] += 1
84
85          return readings
```

com.py:

```
1  import requests
2  from requests import ConnectionError
3  import json
4  import sys
5  import os
6
7  server = 'http://dcsense.ee.princeton.edu/api/sensors/submitreadings'
8
9  def send_to_server(data, board_status):
10
11     # Execute post request to server asynchronously
12     try:
13         newpid = os.fork()
14         if newpid == 0:
15             try:
16                 payload = {
17                     'data': data,
18                     'status': board_status
19                 }
20                 headers = {'content-type': 'application/json'}
21                 requests.post(server, headers=headers, data=json.dumps(payload))
22             except ConnectionError, err:
23                 sys.stderr.write('Could not contact server:\n' + str(err))
24             os._exit(0) # Kill child process
25             print "Wasn't supposed to get here."
26         else:
27             os.waitpid(0, os.WNOHANG) # Clear zombie child process
28     except OSError, err:
29         sys.stderr.write('OSError on call to fork(): \n' + err)
```

status.py:

```
import fcntl, socket, struct

def getHwAddr(ifname):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    info = fcntl.ioctl(s.fileno(), 0x8927,  struct.pack('256s', ifname[:15]))
    return ':'.join(['%02x' % ord(char) for char in info[18:24]])

class Status:
    def __init__(self):
        self.reset()

    def reset(self):
        self.status = {
            'mac' : getHwAddr('eth0'),
            'err' : False,
            'type' : '',
            'msg' : '',
            'ioerr_addr' : -1,
            'ioerr_bus': -1,
            'num_tries' : 0
        }
```

Appendix D

Database Schemas

Readings & ResearchReadings:

```
{
  controller: String,
  bus: Number,
  sensor_addr: Number,
  time: Date,
  temp: Number
}
```

See §6.5.2 for full description of the above fields.

Users:

```
{
  username: String,
  token: String,
  salt: String,
  accessLevel: Number
}
```

See §6.5.1 for full description of the above fields.

Ticket:

```
{
  username: String,
  ticket: String,
  expires: Date
}
```

These represent the session tickets described in §6.5.1 used to keep track of which users are currently logged in.

Facilities:

```
{
  name: String,
  controllers: [{ // List of controllers assigned to this facility
    id: String, // Controller MAC
    name: String, // User-defined name (defaults to 'id')
    width: Number, // Number of cabinets in this group
    layout: [ // List of cabinets (should contain 'width' elements)
      { // Object representing a single cabinet's sensor mappings
        // 5-element arrays for the 5 sensors on each cabinet door
        bus: [Number],
        addr: [Number]
      }
    ]
  }],
  owners: [String] // List of users with access to this facility
}
```

These are representations of a full facility equipped with our sensor system. It contains the facility's name, a list of users who have access to data from the facility, and representations of each server cabinet group equipped with sensors (including the controller mounted to that cabinet group and sensor mappings for the cabinet group).

ReadingCounters:

```
{
  controller: String, // Controller MAC address
  counter: Number // Number of readings received since the last set was saved
                  // to the production collection
}
```

These are helper objects used to keep track of when to save a set of readings from each controller to the production `Readings` collection. The counter is incremented until it hits 120, when the next reading is saved to the production database and the counter resets to 0.

Appendix E

Sample Sensor Data

The next page shows plots of sensor data gathered by our system in two different environments — one temperature-controlled server room to mimic performance in a data center environment and one room without climate control to show the sensors response to cyclical changes in temperature across multiple days.

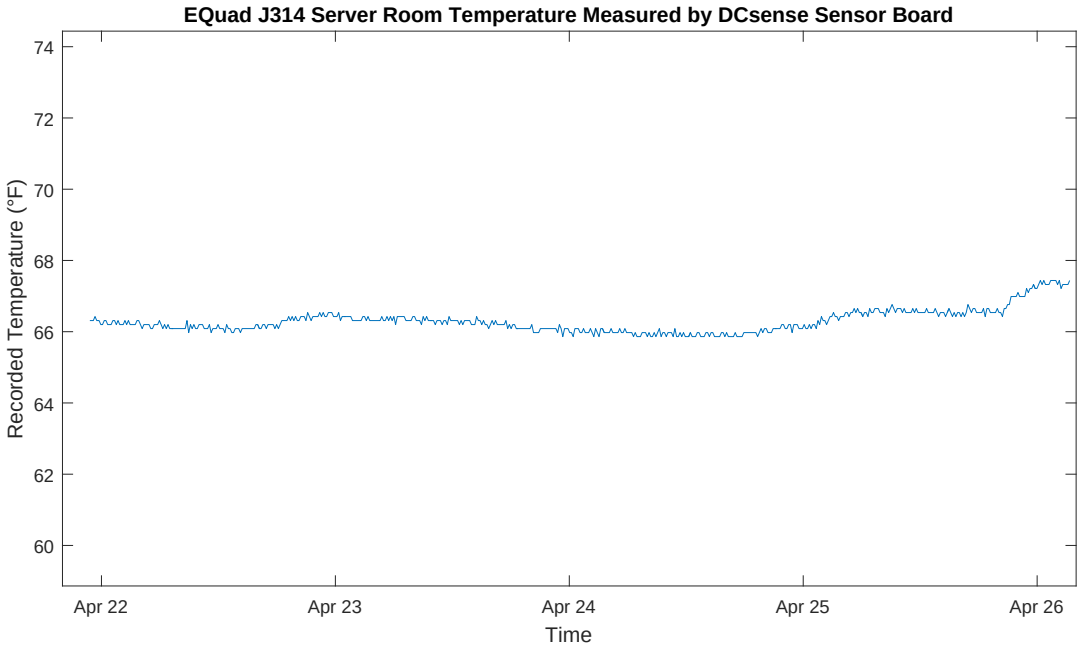


Figure E.1: Example data over a period of 4 days in a small temperature-controlled server room

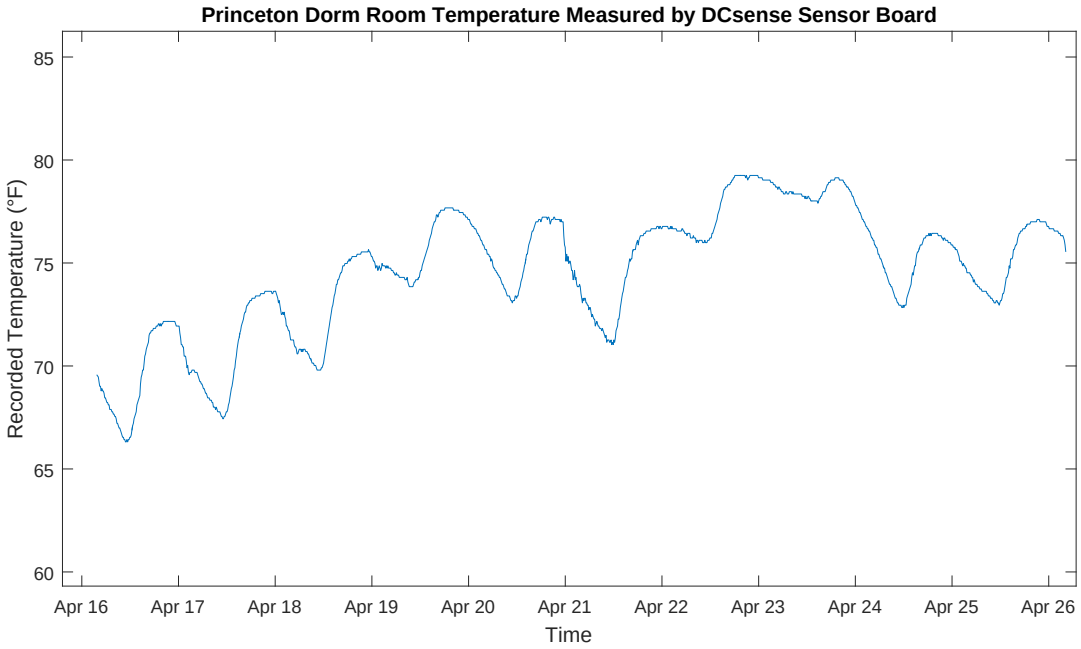


Figure E.2: Example data over a period of 10 days in a small dorm room at Princeton where temperature is not controlled. Note the daily temperature cycling.

References

[1] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," vol. 98, no. 2, pp. 253–266.

[2] J. G. Koomey, "Worldwide electricity used in data centers," vol. 3, no. 3, p. 034008. [Online]. Available: http://stacks.iop.org/1748-9326/3/i=3/a=034008

[3] M. Pedram, "Energy-efficient datacenters," vol. 31, no. 10, pp. 1465–1484.

[4] J. Koomey, "Growth in data center electricity use 2005 to 2010." [Online]. Available: http://www.analyticspress.com/datacenters.html

[5] Alliance to Save Energy, I. C. F. Incorporated, E. R. G. Incorporated, U. S. Environmental Protection Agency, R. E. Brown, R. Brown, E. Masanet, B. Nordman, B. Tschudi, A. Shehabi, J. Stanley, J. Koomey, D. Sartor, P. Chan, J. Loper, S. Capana, B. Hedman, R. Duff, E. Haines, D. Sass, and A. Fanara, "Report to congress on server and data center energy efficiency: Public law 109-431." [Online]. Available: http://www.osti.gov/scitech/biblio/929723

[6] J. B. Marcinichen, J. A. Olivier, and J. R. Thome, "On-chip two-phase cooling of datacenters: Cooling system and energy recovery evaluation," vol. 41, pp. 36–51. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1359431111007058

[7] R. Subramanian, "Design of an instrumentation circuit for datacenter temperature measurements."

[8] P. de Groot, "Data center instrumentation project: Final report." [Online]. Available: http://dataspace.princeton.edu/jspui/handle/88435/dsp01ww72bd80p

[9] K. Deland, J. Lenchner, J. Nelson, J. Connell, J. Thoensen, and J. O. Kephart, "A robot-in-residence for data center thermal monitoring and energy efficiency management." ACM Press, p. 381. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2070942.2071000

[10] Raritan. Data center environmental sensors | environmental monitoring. [Online]. Available: http://www.raritan.com/products/environmental-monitoring/environment-sensors

[11] Geist. Server & data center monitoring systems. [Online]. Available: http://www.geistglobal.com/products/monitor

[12] T.I. TMP175 | digital output | local temperature sensors | description & parametrics. [Online]. Available: http://www.ti.com/product/tmp175

[13] R. O'Shea. Git repository: data-center-monitoring. [Online]. Available: https://github.com/PrincetonUniversity/data-center-monitoring