Wireless Sensor Network for Wine Fermentation

Senior Project Report

Students
Kerry Scharfglass (CPE)
Andrew Lehmer (CPE)

Advisor
Dr. John Oliver

6 June 2012

Table of Contents

List of Figures

Acknowledgments

Caitlin Devaney — The materials engineering student who we teamed up with to design and construct an enclosure for our electronics and single-handedly brought the scientific process to this project.

Dr. John Oliver — Our advisor, this project's creator and initiator, and an amazingly hard-working professor who provided us with much needed criticism, motivation, inspiration, and comic relief.

Matt Brain — The Cal Poly pilot winery's cellarmaster who played the role of our client and whose feedback ultimately defined and improved the final product.

1. Abstract

This project implements an automated temperature monitoring system for wine fermentation which is affordable, easy to use, and scalable to typical small winery setups. To realize these requirements, we implemented the system as a wireless sensor network utilizing commercial off-the-shelf hardware. Temperature and system diagnostic information is communicated wirelessly in a peer-to-peer network topology such that all information flows toward an aggregating server. The server makes the temperature information available over the Internet via a web application and alerts the winemaker by email when the temperature has left acceptable bounds that the winemaker may configure. This project also involved materials selection and enclosure design performed cooperatively by or with a materials engineering student, which we briefly discuss. However, we focus mainly on the design, implementation, and assessment of the system's electronics, software, and network protocols.

2. Introduction

The fermentation process is critical to the science of winemaking. Fermentation chiefly determines the wine's ultimate alcohol content and other significant aspects of its flavor. The process is predominantly dependent on temperature because fermentation rate is directly proportional to temperature. Consequently, winemakers put a great deal of effort, or a sizeable sum of money, into keeping their vats of fermenting grapes at a constant temperature to more accurately determine the total fermentation time. Large-scale wineries typically have the funds to acquire automated fermentation systems that both monitor and control the temperature of the wine, but smaller wineries may opt to perform these tasks manually to save money.

However, monitoring the temperature manually is both error-prone and inconvenient. Every few hours, winemakers have to take the temperature of every fermentation vat which involves removing the lid, inserting a thermometer, waiting for thermal equilibrium, and reading the result. This causes the grapes to be exposed to the air, which is undesirable, and measurements are prone to inconsistency, especially if the responsibility is shared between multiple people. Readings also have to be taken in the middle of the night to maintain data resolution, which is tiresome over the typically two-week-long process.

These issues with accuracy, resolution, automation, and cost define an ideal application for a wireless sensor network. If each sensor node can be produced at a relatively low cost, a wireless network can create an affordable and scalable solution for small winemakers, which is the ultimate goal of this project.

3. Background

The original idea for this project is attributed to Dr. John Oliver, our senior project advisor. The concept materialized during a conversation with winemaker and Cal Poly alum Chris Turkovich who described the difficulties of temperature measurement during wine fermentation. Dr. Oliver informally started the project as a special assignment in his FPGA and Microcontroller Based System Design class, which was taken on by Kerry Scharfglass. He also contacted Dr. Kathy Chen, a materials engineering professor, about enclosure design as a possible parallel project, which was eventually taken on by her student, Caitlin Devaney. As a result, a multidisciplinary senior project between CPE and MATE was formed to create a fully integrated product involving materials, structure, electronics, and software for an end user.

Our target client for this project was the Cal Poly pilot winery. The Cal Poly pilot winery was founded in the fall quarter of 2008 for the study of wine and viticulture. It exists in a renovated space in the campus crops unit that houses a cool room, presses, a de-stemmer/elevator, steam generator, bottling line, fermentation tanks, and work areas. The fermentation equipment, most of which is donated, includes a

few high-tech but low capacity tanks that are capable of monitoring and controlling the temperature of the ferment, but most of the tanks are simple, large boxes made of high-density polyethylene (HDPE).

4. Requirements

We collected behavioral requirements for this project from Matt Brain, the cellarmaster for the Cal Poly pilot winery, and many functional requirements were provided by our advisor, Dr. John Oliver.

The single, foremost, overall requirement is that the complete system must be affordable. In concrete terms, each device must cost as little as possible and not more than $100.

The computer engineering aspect of this project was smoothly divided into two halves: the sensor network and the application layer above it. However, along with Caitlin, we were also responsible for designing and constructing an enclosure for the network electronics, which we consider more of a mechanical or manufacturing engineering problem.

4.1. Sensor Network

The sensor network portion of the project consists of both the hardware and firmware that are integrated with the enclosing materials into a physical product and must satisfy the following requirements:

- The network must run continuously without user interference for at least a month.
- The network must be scalable.
- The network must be fault-tolerant.
    - As long as an active node is within the communication range of a member of the network, it must be able to join the network.
- Every sensor node must periodically measure the temperature of fermenting wine.
    - Measurements must be taken every ten minutes at maximum.
    - Temperature readings must be accurate within 0.1˚C.
    - Two temperatures corresponding to different locations within the ferment (see §4.3) must be recorded at the same frequency.
- Sensor nodes must have no externally accessible user interface besides the insertion and removal of batteries.
    - They must automatically add themselves to the network when batteries are in place and are in range of another network member.
    - The network must automatically detect when a node has powered off.
- All data measured by a sensor node must be sent wirelessly to a single sink connected to a host that can aggregate the data and store it long-term.

4.2. Software Applications

The software applications portion of this project include both a Windows desktop application and a web application. Together, they must satisfy the following requirements:

- Both the desktop and web applications must automatically collect data without failure or user intervention for extended periods of time.
- All user interface components must be usable and configure by non-technical people.
    - There must be as little configuration as possible for the system to work.
- All data must be accessible from anywhere via an internet browser.
- Data must be clearly presented.
- All data must be downloadable for local analysis.

    - It should come in a common format which can be directly imported into other software, such as Microsoft Excel.
    - All data must be available in the manner.
- There must be a mechanism to notify the user if their ferment temperature is outside of desirable bounds.
    - There must be a mechanism for the user to set allowable temperature ranges in the web interface.
    - Notifications must be dispatched as soon as a problem is known.

4.3. Enclosure

The work involved to fulfill the enclosure's chemical requirements is fully contained in the related senior project carried out by materials engineering student Caitlin Devaney, so nothing further will be said about them. However, there are also mechanical and manufacturing requirements that define the extent of the collaboration between our two projects.

- The enclosure must reasonably protect the electronics from physical compromise.
    - It must be watertight.
    - It must be resistant to structural stresses resulting from handling errors such as impacts due to dropping.
- The enclosure must properly position the electronics responsible for probing temperature in two places:
    - Directly below the base of the cap of floating grape solids (approximately 18–24 inches below the surface).
    - Approximately 12–18 inches below the base of the cap.
- The enclosure must be permissive of 2.4GHz RF energy to minimize wireless attenuation.
- The enclosure must be manufacturable using tools available to Cal Poly students in the College of Engineering.

5. Design

For all three aspects of this project, we spent a considerable amount of time on design to ensure all of the requirements were met. In some cases, a particular design went through development or testing before we realized that the result was not going to satisfy the requirements, which required further iterations of design. Where appropriate, we will discuss earlier designs to document our overall process.

5.1. Sensor Network

The design of the sensor network involved both hardware selection and a network protocol scheme.

5.1.1. Hardware Selection

Texas Instruments' eZ430-RF2500 development kit was selected as the main control unit of the sensor node because of its wireless capabilities, low power consumption, and size. Additionally, it integrates an MSP430 microcontroller which contains an on-board thermistor suitable for measuring temperatures in the required range [3]. The kit also exposes several GPIO pins to allow for interfacing with peripherals and includes a USB-to-serial programmer that can also be used to communicate arbitrary data with a PC.

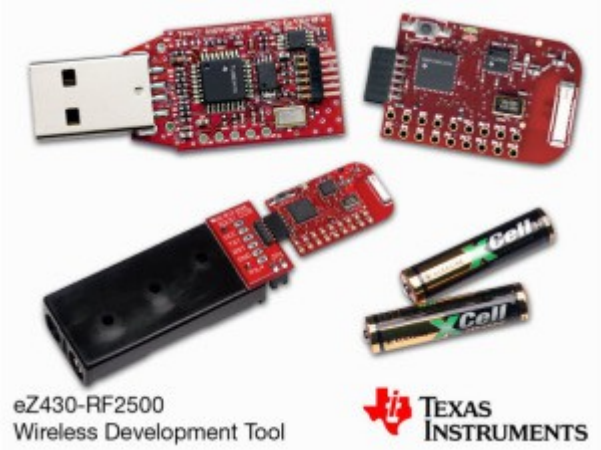


Figure 1: The eZ430-RF2500 development kit [4]

No other control hardware was considered because several eZ430 modules were already on-hand from previous work.

Originally, the requirements did not state that temperature had to be measured in two different physical locations in the fermenting volume, so we assumed no other hardware was needed. However, because of difficulties we were experiencing with designing an enclosure that effectively used the MSP430's thermistor, we needed a peripheral temperature probe that could be safely submerged in wine and still met our accuracy and precision requirements. Therefore, we selected a stainless steel encapsulated DS18B20 temperature sensor [2].

For our final design, we used two waterproof DS18B20 temperature probes and an eZ430-RF2500 development board powered with two standard AAA batteries. The overall product also required two 4.7 kΩ resistors and a .1 μF capacitor which will be discussed further in later discussions on development and testing (§6.1 and §7.1).

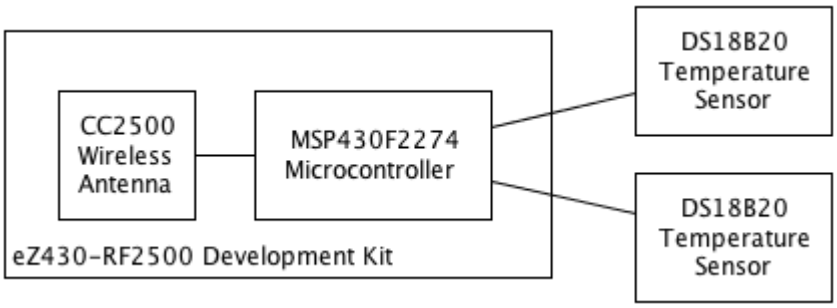


Figure 2: Hardware design block diagram

5.1.2. Network Protocol Design

The protocol design was driven as much by the capabilities of the selected hardware as the requirements. According to the embedded network stack provided by TI, SimpliciTI, a node can take on one of three roles: an endpoint, a range extender, or an access point [5]. The access point is the designated sink that all endpoints connect to by default under the SimpliciTI API, and if an endpoint is out of range of the access point, it will automatically channel communications through the nearest range extender. At an abstract level, this defines an extended star topology for the network.

Although this initially appeared to be a simple, functional way to define the network, we found a couple of very serious problems. First, the network stack has a hard limit for the number of range extenders that can be chained from the access point. This severely limits the scale of the network. Second, the node's

role can only be determined at programming time. How would the user know where to put the range extenders, and what happens if any of them move? In the end, this design was simply too limited and rigid for general use.

Instead of using the existing SimpliciTI protocol, we created a more scalable peer-to-peer (P2P) communication protocol. Our network, therefore, is mostly homogeneous, but still involves two different member role definitions. The members that measure temperature are termed *sensor nodes*, and the member that sinks all of the temperature data and transfers them to a PC is termed the *access point*.

In computer networking terms, the protocol is defined entirely at the internetworking layer because the SimpliciTI network stack effectively negotiates all physical and link layer communication. This means we did not have to design our own carrier-sense multiple-access collision avoidance (CSMA/CA) mechanism or other physical layer and data link layer mechanisms for reliable, wireless communication.

Our protocol defines a minimal connection state for each member of the network. Each sensor node must keep track of its hop count (HC) from the access point, a number of downstream connections, and a single upstream connection. The access point only maintains downstream connections in its protocol state. *Downstream* refers to connections leading away from the access point, and *upstream* refers to connections leading toward the access point. This asymmetric state definition produces a tree topology where the access point is the root of the tree.

Our protocol also defines a minimum set of communication types: data, hop count advertisement, and acknowledgment. Though we will not specify the format for these here (see §7.1.2), we will explain their general purpose. Data packets contain a sensor node's temperature measurements and are always sent or forwarded upstream. Hop count advertisements contain a sensor node's hop count, or zero for the access point, and are only sent downstream in response to a link request. Acknowledgments are only sent by the access point in response to data packets and are forwarded downstream. Hop count advertisements and acknowledgments support network efficiency and fault tolerance respectively.

As described here, the protocol design is complete and satisfies the requirements, though many details are left to the implementation.

5.2. Network-to-Application Interface

The manner in which data was communicated from the network to the supporting software application necessitated its own design decision. To eliminate the need for data processing in the network altogether and further simplify the firmware, which is difficult to debug, we decided that it was in our best interest for the network to initiate all communication with the application and transfer the data in the network-native format. Therefore, all of the processing responsibilities for the interface were pushed to the much more powerful PC application at the cost of requiring changes to the application whenever the network data format changed.

5.3. Software Applications

The non-embedded software design required the construction of a desktop application and a web application.

5.3.1. Desktop Application

Choosing the platform for our desktop application was a decision which was essentially made for us. While the eZ430-RF2500 development boards had a serial passthrough that could be used to communicate between the connected computer and the microcontroller it used specialized drivers which were only available for Windows. As such, the final desktop application had to run on Windows and there was no point in making it compatible with any other systems. Based on this, it was decided to use C# and

Visual Studio 2010 Express, as this is what we had experience with and it would be the simplest path to writing a Windows desktop application.

5.3.2. Web Application

Selecting a platform for the web application was a complex process. It was clear that to be feature complete, the website could not merely be a web server, but must also include a database of some sort to store readings and potentially other infrastructure. As a convenience, the Computer Science (CSC) department offers all of its students an outward facing URL they can host a website at. After a few minutes of exploration on our part, it was discovered that this free hosting had a functional installation of PHP and MySQL, both extremely common foundations for writing web applications of all types, though in declining popularity. As these are the web technologies that we were initially familiar with, it made sense to use them. However, after doing more research and beginning the design process it was decided there may be a better choice; Google's App Engine.

Google App Engine (GAE) is a service Google provides which allows developers to write complex web applications using their back end. The developer can choose between writing their web servers in Python versions 2.5 or 2.7, Java, or Google Go. One of the selling points of GAE is that the developer does not need to worry about configuring any servers or databases and instead uses the framework which Google provides. In return, as load on a webapp increases the developer can pay to allow it can scale to meet demand, as it using Google's presumably enormous pool of resources. Additionally, GAE allows developers to easily integrate with Google's login service, allowing webapps to differentiate between users based on their Google accounts.

It would be unreasonable to expect that our senior project would need to scale to more than the pilot winery, but if our sample were to be used as the basis for a real product, then this might be a relevant issue. Along these lines, a webapp written on GAE can have its ownership transferred, which will allow for someone to carry on our work without disrupting service to the pilot winery. Plus, they provide a free outward facing URL with every application. Moreover, GAE is free as long as you stay within certain well defined usage quotas. These quotas cover such parameters as as the maximum number of queries to each webapp, the size of the Datastore, and the number of emails sent. Based on these traits, we decided to develop our web application using Google App Engine instead of our CSC website.

5.4. Enclosure

The enclosure we cooperatively designed between all three members of our team, and went through a number of revisions before we landed on what became our final design.

As stated earlier, during early tests we discovered that the MSP430 included on our development boards had an internal thermistor which provided fairly accurate temperature measurements. Based on this, our first enclosure idea aimed to minimize external hardware by utilizing it. In order to do this, the plan was to place the entire node, including power source inside the same container, and use some sort of thermally conductive material to connect the external environment to the actual surface of the MSP430 IC, where the thermistor was located. While this design would be supremely easy to use (it could simply be tossed into the box of fermenting grapes), we realized that such a design would be non-ideal for a few reasons. It would be very difficult to build an enclosure with the necessary thermal properties, both because mechanically connecting the top of the MSP430 to the outside was difficult and because the more intermediate layers between the thermistor itself and the outside, the less accurate a reading we would get. Additionally, there would be significant RF attenuation by having our antenna (which is attached to the node PCB) submerged in liquid. This design was scrapped.

The next idea became the root of our final design. We decided that an external temperature probe would be necessary in order to easily interact with the fermenting grapes without needing a thermal conductor. In order to keep the node out of the attenuating liquid, it would be placed inside of a housing of some type, above the fluid. The result was the diagram shown in Figure 3 below.

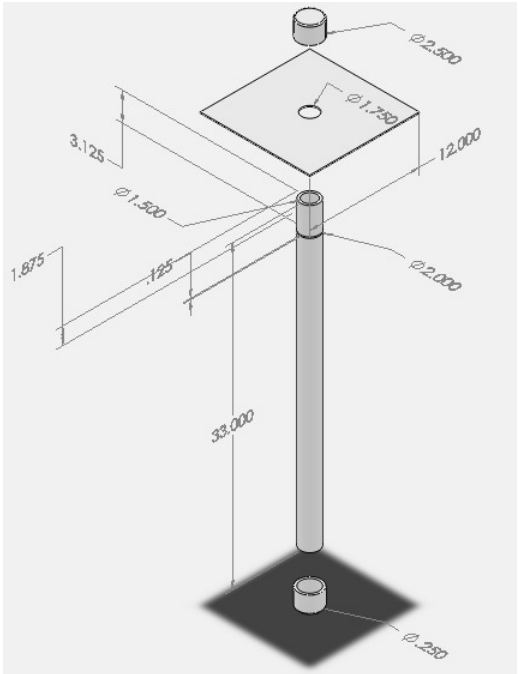


Figure 3: A CAD drawing of the entire node

This design is essentially a cylindrical tube of plastic which would be sealed on each end, with a flat plastic plate attached near the top. The waterproof temperature probes we found are entirely closed inside stainless steel, and can be directly immersed in liquid without worry about corrosion, so one would protrude from the bottom of the tube, and the node and batteries would be hung from the top. The HDPE plastic we chose for the body of the node is less dense than water, so it naturally tends to float. This combined with the plate would allow the node to sit upright in the grapes, with the top above the cap and the rest submerged.

As discussed above, at some point in our communications with Matt Brain, it was determined that having a second temperature probe would be beneficial. As we were fairly confident with our previous design, we decided that the best solution for adding the second probe was to drill a hole in the body tube below the flat plate, far enough down that the protruding probe would be just below the cap. This was the final design.

6. Test Plans

Because the entire project was designed to be almost purely automated, most of our tests were also designed to be automated. As a general rule, tests were conducted in an iterative manner parallel to development, although development itself was by no means test-driven. Since we were not responsible for testing the enclosure, we outline our subsystem and integrated test plans here.

6.1. Sensor Network

The difficulty with testing any embedded system is the typical lack of directly observable output mechanisms by which to verify correct behavior. Because of this, we heavily utilized black box testing with no input. The only testing aids available to us in this context were two LEDs on the development kits and serial output from the access point. These limited indicators made tests that are usually dependent on wireless range much more complex.

6.1.1. Temperature Probe Test

This test verifies if the temperature probes are correctly reading and reporting their measurements. This involves programming the sensor node to directly output the data over serial in a human-readable format.

1. Plug in the sensor node and open up a serial console by which to observe the probe data.
2. Verify that the reported temperatures from the probes match within an acceptable margin by submerging them in water at a known temperature.
3. Submerge one probe in warmer water and verify that the corresponding reported temperature increases to the correct temperature.
4. Submerge the other probe in cooler water and verify that the corresponding reported temperature decreases to the correct temperature.

6.1.2. Single Connection Test

This test was performed after a basic sensor node and access point were implemented.

1. Plug in the access point and open up a serial console by which to observe incoming data.
2. Power on a single sensor node.

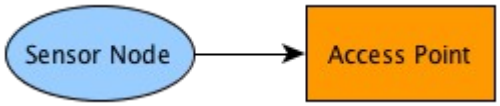


Figure 4: Single Connection Test setup

Expected Behavior: Shortly after powering on the node, the access point receives a data packet and prints its contents to the console in a human-readable format for debugging purposes. Periodically, the access point receives another data packet from the sensor node.

6.1.3. Multiple Connection Test

This test is the same as the Single Connection Test (§6.1.2), but powers on two sensor nodes instead of one. The access point should periodically receive data packets directly from both nodes.

Figure 5: Multiple Connection Test setup

6.1.4. Talk-through Test

This test required programming one sensor node to explicitly *not* directly connect to the access point. Instead, it must send its data through another sensor node. For this test, the node that is not directly connected to the access point is called the *distal node*, and the node it talks through is called the *talk-through node*.

1. Plug in the access point and open up a serial console by which to observe incoming data.
2. Power on the talk-through node and wait for an indication that it has successfully connected to the access point.
3. Power on the distal node.

Figure 6: Talk-through Test setup

Expected Behavior: Shortly after powering on the distal node, the access point receives a data packet from it. The access point should continue periodically receiving data from both sensor nodes without interruption.

6.1.5. Fault Tolerance & Efficiency Test

The setup for this test is the same as the Talk-through Test (§6.1.4), except every sensor node is programmed to illuminate an LED for every downstream connection they obtain.

1. Plug in the access point and open up a serial console by which to observe incoming data.
2. Power on one talk-through node. This node will be referred to later as TT1.
3. Power on the distal node.
4. Power on the other talk-through node. This node will be referred to later as TT2.
5. Power off TT1.

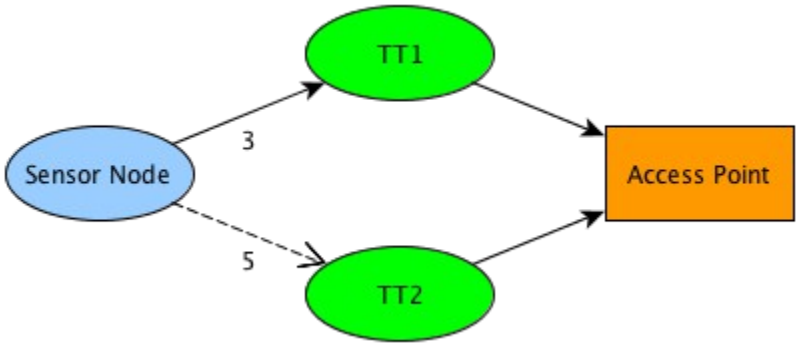


Figure 7: Fault Tolerance & Efficiency Test setup

Expected Behavior: When the distal node is powered on, TT1 illuminates a single LED. When TT2 is powered on, the LED state of all of the nodes does not change. Before TT1 is powered off, all nodes have successfully sent data to the access point. After TT1 is powered off, TT2 illuminates a single LED to indicate that the distal node has reconnected to the network through it. It is acceptable if the access point failed to received data packets from the distal node for at most two measurement periods.

6.1.6. Range Test

To speed up this test, the sensor node was programmed to send data every five seconds, and the access point was programmed to toggle an LED every time a data packet was received.

1. Plug in the access point.
2. Power on a sensor node and position it one foot away from the access point.
3. Verify that the access point receives data from the sensor node.
4. Move the sensor node one foot further away from the access point.
5. Repeat steps 3 and 4 until the access point no longer receives data from the sensor node.

Expected Result: The effective range is greater than or equal to 20 feet.

6.1.7. Scalability Test

The scalability test is an extension of the Talk-through Test (§6.1.4) that determines the maximum chain length of sensor nodes from the access point. To ensure the chain length increases, each additional sensor node is programmed to join the network with a higher hop count than the last. This test should continue until either a noticeable failure is observed or there is no more available hardware.

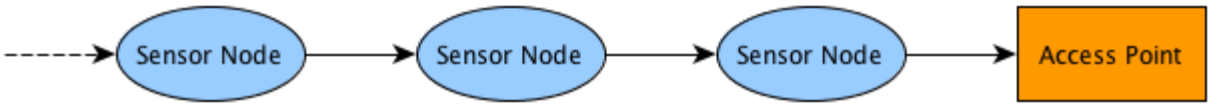


Figure 8: Scalability Test setup

6.1.8. Access Point Fan-out Test

This test verifies that the access point can support a certain maximum number of direct connections, which is called its *fan-out*. This test is important for cases where many of the sensor nodes are able to be in relatively close proximity to the access point such that the network tree is broader than it is deep. This is the dual of the Scalability Test (§6.1.7). To support this test, all sensor nodes are programmed to illuminate an LED if they have a downstream connection.

1. Plug in the access point and open up a serial console by which to observe incoming data.
2. Power on a number of sensor nodes corresponding to the access point's fan-out.
3. Power on an additional sensor node.

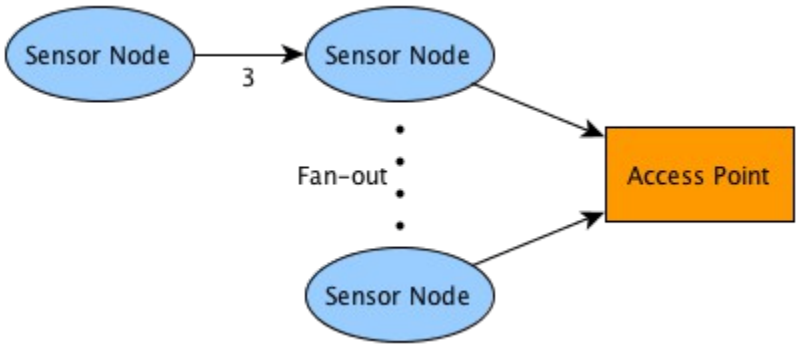


Figure 9: Access Point Fan-out Test setup

Expected Behavior: After step 2, none of the sensor nodes have any downstream connections, and the access point is able to receive data from all of the sensor nodes. After step 3, exactly one sensor node has a downstream connection, and the access point is still able to receive data from all members of the network.

6.1.9. Sensor Node Fan-out Test

This test is exactly like the Access Point Fan-out Test (§6.1.8) except the fan-out nodes must be programmed to only connect to a parent with a hop count greater than zero. Only one node is not programmed this way, and it is referred to as the *parent node*.

1. Plug in the access point and open up a serial console by which to observe incoming data.
2. Power on the parent node.
3. Power on a number of sensor nodes corresponding to the fan-out of a sensor node.
4. Power on an additional sensor node.

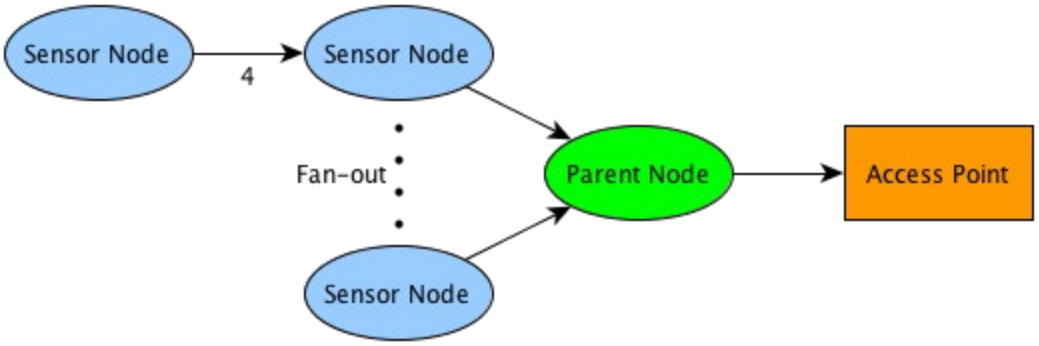


Figure 10: Sensor Node Fan-out Test setup

Expected Behavior: After step 3, all of the sensor nodes except for the parent node have no downstream connections. After step 4, only the parent node and one other sensor node have a downstream connection. In all cases, the access point is able to receive data from all members of the network.

6.1.10. Longevity Test

This test simply tries to determine the battery life and long-term reliability of a sensor node.

1. Plug in the access point and open up a serial console by which to observe the time incoming data arrives.
2. Power on a sensor node and note the time.
3. Regularly observe the data received by the access point and note the time data stops arriving.

6.2. Software Applications

In contrast to the embedded system, while testing the desktop and web software a very functional debugging environment exists, and testing was conducted without the inconvenience of a black box.
In order to ease development, Google App Engine has a local development server that can be used to run a webapp from a local machine instead of Google's infrastructure. This test server simulates all functions of the real environment, but does not have maximum quotas. As such, this local website was used in all tests to verify styling, layout, and functionality without cutting into the usage quota.

6.2.1. Access Point to Desktop Communication Test

This test determines if the desktop application can correctly receive sensor readings being relayed by the access point. In order to perform this test, a "dummy" access point was created. Each time a new revision of the real access point code was created, it was copied. This copy had all radio communication functionality and other extraneous functionality stripped out of it. A single fake sensor reading with constant, known values was created, and the firmware was modified to toggle an LED and send this at a regular interval similar to the rate at which real sensor readings would come in.

1. Connect the dummy node to the host computer.
2. Ensure that the LED on the dummy node is toggling, indicating that it is sending data.
3. Start the desktop application.

Expected Behavior: Each time the LED on the dummy node toggles, the desktop application should update the printout of the last recorded readings, as well as provide any indication in the debugger that it is sending data to the webapp. If it is submitting data and the webapp is running, it should be visible there as well.

6.2.2. Desktop Application Data Submission Test

To test the output of the desktop application, the query string which would typically be sent via POST request to the webapp could be printed to the debugger.

1. Connect the dummy node to the host computer.
2. Ensure that the LED on the dummy node is toggling, indicating that it is sending data.
3. Start the desktop application.

Expected Behavior: Because all valued from the dummy node are known, the POST request can be examined and verified to be correct. For each fake sensor reading submitted, the POST request was scrutinized to make sure it was correct. This usually did not need to happen for more than a few readings.

6.2.3. Web Application Data Reception Test

Testing the data submission URL of the webapp requires submitting known data via a manually generated POST request and observing the results in the webapp. There are two versions of this test, the single point and multiple point tests.

The single point test is a one line bash script which takes in a pair of temperatures (one for each temperature probe) and uses a utility called cURL to POST them to a particular URL. This works well for initial small-scale testing.

1. In a terminal emulator, navigate to the directory with the test script (called `submitLocalData.sh`).
2. Execute "`./submitLocalData-One.sh X Y`". Where X is the temperature of the upper sensor, and Y is the temperature of the lower sensor.

Expected Behavior: The word "`good`" should print out on the terminal (this is from the webapp). Each submitted data point should be immediately visible in the graph on the webapp, as well as in the CSV.

The multiple point test is very similar as the single point. It is about fifteen lines of Python, and uses the same hard coded query string as the bash script, but takes in the number of readings to generate. When it is run, the script generates the specified number of fake sensor readings by taking a base temperature, adding a randomly generated offset to it twice (once for each temperature probe), and cURLing it at the data submission URL. This makes it easy to generate tens or hundreds of readings at once.

1. In a terminal emulator, navigate to the directory with the test script (called `submitMany.sh`).
2. Execute "`./python submitMany.py X`". Where X is the number of random readings to submit.

Expected Behavior: The specified number of readings should be immediately visible in the graph on the webapp, as well as in the CSV.

6.3. Integration

The key point of integration between the two software halves of the project is the serial interface between the access point and desktop application. This, as well as the link between the desktop and web application was tested.

6.3.1. Sensor Network and Desktop Application Test

This test determines if multiple nodes in the sensor network can successfully communicate to the desktop application.

1. Connect the access point to the host computer.

2. Start the desktop application.
3. Power on more than one sensor node within range of either the access point, or the access point and each other.

Expected Result: Each time a new packet comes in, the box in the desktop application should display new sensor data. If there are multiple nodes attached, the node address should change each time.

6.3.2. Desktop and Web Application Interface Test

This test determines if the desktop application can successfully communicate with the webapp. If the desktop and web application individual test have been completed, this should be redundant.

1. Connect the dummy node to the host computer.
2. Ensure that the LED on the dummy node is toggling, indicating that it is sending data.
3. Start the desktop application.

Expected Result: The fake sensor readings from the dummy node should be immediately visible in the graph on the webapp, as well as in the CSV.

6.3.3. Full Integration Test

In order to test the entire system, all components must be connected and observed.

1. Connect an access point to the host computer.
2. Start the desktop application.
3. Power on a single sensor node within range of the access point.

Expected Result: As the sensor node starts sending data, the desktop application should begin to display sensor readings. As this happens, the sensor readings should be immediately visible in the graph on the webapp, as well as in the CSV.

7. Development

In an effort to follow our test plans as closely as possible, we also developed iteratively and made sure small portions of the overall functionality met specifications before proceeding to integrate them into the whole. Because the development process was very long, we seek to summarize the pitfalls and triumphs we experienced along the way as well as how the system actually meets the original requirements.

7.1. Sensor Network

One quarter prior to the start of our senior project, Kerry worked on the first prototype implementation of the sensor network using the eZ430-RF2500 development tool. As a result of his experience, firmware development went much more smoothly. The goal of our implementation, then, was to realize the network protocol design and interface with two DS18B20 sensor probes.

7.1.1. Interfacing with the temperature probes

The stainless steel encapsulated DS18B20 temperature sensors we selected became the source of a particularly embarrassing hardware-related pitfall during development. When we originally connected the probes to the board, we soldered the wires directly to their corresponding GPIO and voltage reference pads. During testing, we observed temperature readings that seemed to indicate an error. However, we incorrectly diagnosed that the problem originated in the firmware and continued refining the interface until the code was as small as possible. Only in our desperation did we return to the sensor’s data sheet to see if we had connected it improperly. The data sheet contained a large circuit diagram that clearly showed the position and value of the required pull-up resistor and decoupling capacitor. Once these

passive components were acquired and properly installed, our tests began passing. Figure 11 shows how we ultimately connected the probes, and Figure 12 depicts the surprisingly difficult mechanical interface.

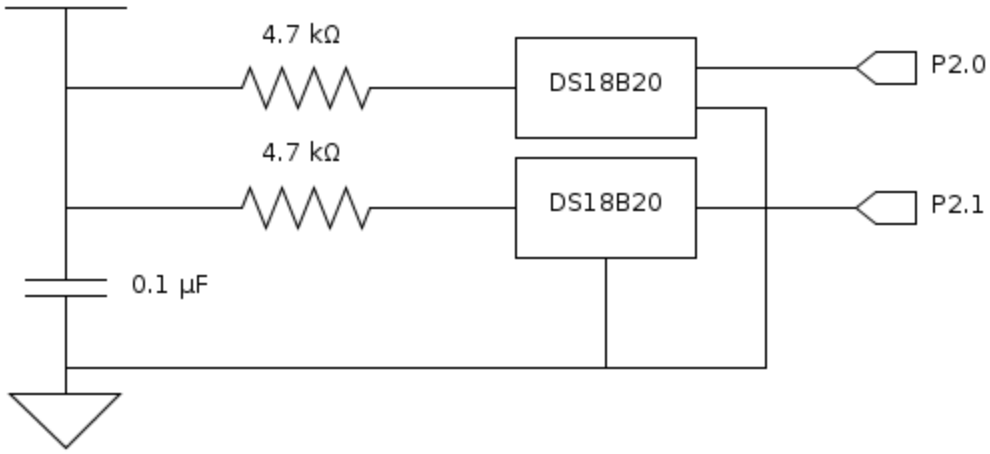


Figure 11: Temperature probe connection circuit diagram

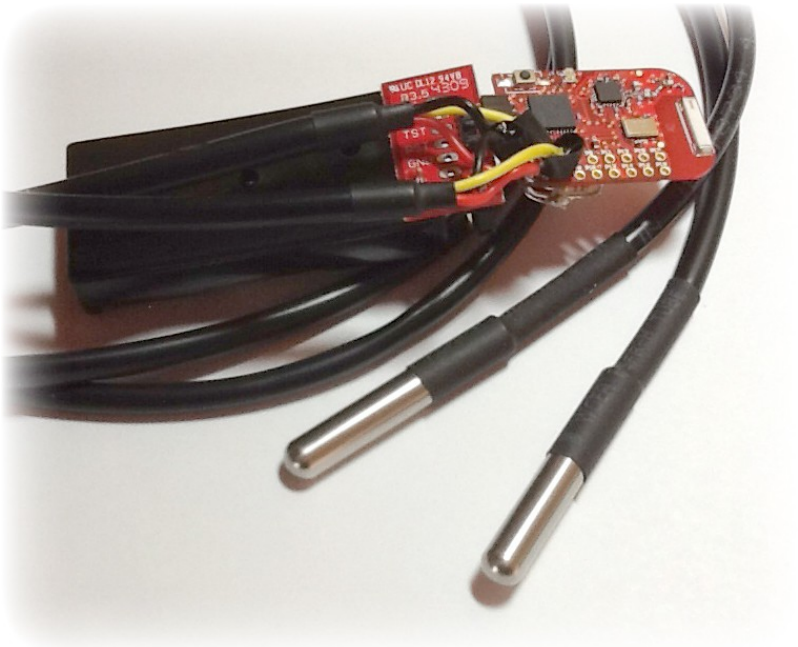


Figure 12: Final sensor node electronics construction

The DS18B20 communicates over the Dallas One-Wire serial protocol, which has very strict timing requirements. Fortunately, like many popular wired protocols, the open source community had already written a library for it on Arduino called OneWire. For this application, the only work that was technically required was to port the library from C++ to embedded C. However, while iterating under the influence of the previously mentioned hardware bug, the port became much more lean and specific to the capabilities of the MSP430F2274. The entire firmware interface consists of two functions: one to initialize the probes, and one to read the temperature in Celsius from a particular probe.

7.1.2. Implementing the network protocol

The network protocol was implemented on top of the existing SimpliciTI network stack. SimpliciTI provides abstractions for configuring the radio, creating wireless connections, and sending and receiving packets [5]. This allowed for us to completely control the content and semantics of our packets.

Every packet is prefaced with two bytes that specify the total length of the packet, including the byte itself, and the type of the packet. As explained in the design, packet types correspond to data, hop count advertisements, and acknowledgments.

Data packets contain the following information:

- Source Address — Every sensor node has a unique 4-byte identifier. This is used to differentiate data received from different nodes.
- Upper Probe Temperature (˚C) — Temperatures are reported as IEEE-754 floating-point values.
- Lower Probe Temperature (˚C)
- Battery Voltage — A 1-byte value that indicates the voltage level of the power source. This can be used to estimate remaining battery life.
- Relative Signal Strength Index

When a data packet is sent by its source sensor node, it only contains the information described above. However, since sensor nodes are only aware of their immediate connections, it is necessary to append source routing information to the packet at each intermediate sensor node on the way to the access point so that an acknowledgment can be routed back. To do this, each intermediate sensor node appends the downstream link identifier of the connection that sent or forwarded the packet to the end of the packet before forwarding it upstream.

When the access point finally receives the data packet, it replies with an acknowledgment (ACK) and includes the list of link identifiers that arrived with the data. The ACK continues to be forwarded downstream by popping link identifiers off the end of the list until it arrives at a sensor node with an empty list. The ACK only contains a destination address which is the same as the source address of the original data packet, and if the address matches, the node has confirmation that its data was received successfully.

The last packet type is used when a sensor node is attempting to join or reconnect to the network. It simply contains a source address and a hop count.

Hop count advertisements are used in the join process to allow the joining sensor node to select the optimum network member it can communicate with as its upstream connection. When a node is trying to join the network, it sends link requests over layer two and waits for a hop count advertisement from a single neighbor. To keep a single neighbor from repeatedly responding to link requests, nodes stop listening for link requests for a while after sending a hop count advertisement.

Once the joining node has received advertisements from all of its neighbors, it picks the neighbor with the minimum hop count as its upstream connection and calculates its own hop count by adding one. Before the join process is complete, the newly joined node tears down all of the other connections it made. Therefore, whenever a sensor node joins the network, it is a leaf in the tree. If the access point happens to be a neighbor during the join process, it advertises a hop count of zero and short-circuits the new node’s join process because a direct connection to the access point is always the most efficient.

7.1.3. Fault tolerance

We used two different mechanisms to implement fault tolerance: downstream time-to-live counters and preemptive upstream unlinking. By using two mutually supportive mechanisms, the network reacted much more quickly to moving nodes and sudden inner node losses.

Time-to-live (TTL) counters were associated with each downstream link such that if a data packet had not been received from the downstream link within two periods of the last data packet, the downstream link would be unlinked and freed up for a new connection. This mechanism brought about a particularly brutal pitfall during development. When a node checks its connection state for queued packets from downstream, it identifies live connections by checking for a non-zero TTL. However, we had somehow forgotten to set the TTL after the downstream node joined the network, so even though packets were

queued, we never retrieved them. This caused the network to behave as though the tree could not attain a depth greater than one, which would have been a significant obstacle to scalability.

The other mechanism is preemptive upstream unlinking. If a downstream connection either (1) has not received an acknowledgment for the past two data packets or (2) detected a layer two transmission failure to its upstream connection, it will immediately tear down all of its connection state and try to reconnect to the network. Without this mechanism, it would be possible for a subtree to become disconnected from the network until all of the subtree's sensor nodes were power cycled.

7.1.4. Implementation overview

The final sensor node and access point implementations are illustrated below in Figure 13 and Figure 14, respectively.

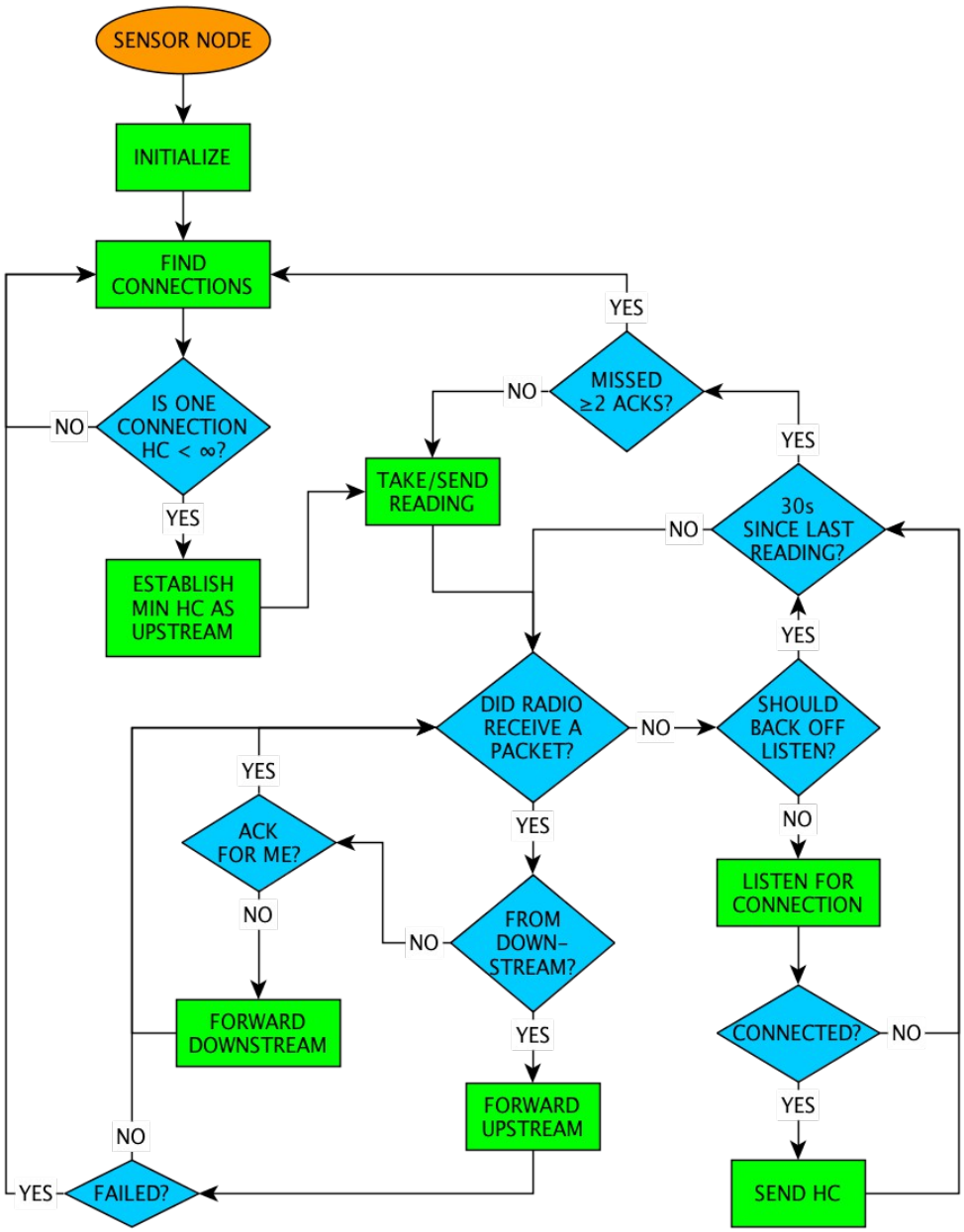


Figure 13: Sensor node implementation

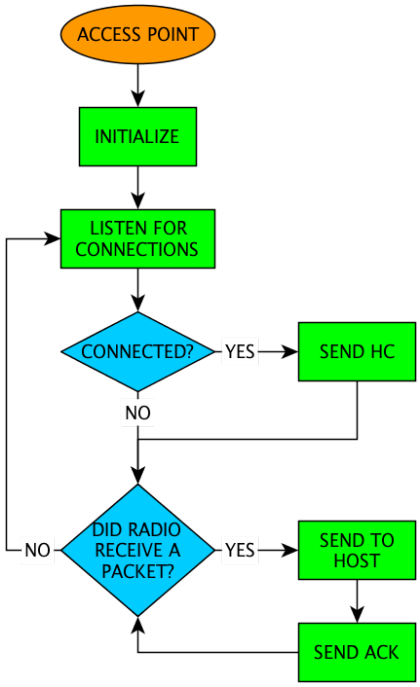


Figure 14: Access point implementation

We would like to share one last pitfall we encountered during the development of the sensor network. The SimpliciTI endpoint model only transmits and does not receive, so the radio's receive functionality is turned off by default. This saves a great deal of energy, but it makes it impossible to implement a P2P network of endpoints. To get around this, we had to explicitly turn on the radio's receive function using the SimpliciTI IOCTL API. However, if the IOCTL call is made before the SimpliciTI API is initialized, it silently fails. Unfortunately, this problem could have been avoided by carefully reading the API documentation in the first place.

7.2. Windows Application

The windows application came together with few unforeseen difficulties. Visual Studio makes it very simple to design user interfaces, so that component did not take very long. Otherwise, the pre existing `System.IO.Ports.SerialPort` serial port interface was used to communicate with the serial passthrough device. Once the data was read into a byte array from the serial port, it was converted from the network-native format the Access Point transferred it in into something which could be more easily worked with. To do this, the raw byte array was processed using C#'s `BitConverter` class and offsets based on the sizes of each field in the C struct used in the embedded code. At this point, the sensor sample was now stored in the desktop application and could be easily manipulated.

Each time new data is observed on the serial port, the desktop application fires a callback. This callback gets one struct's worth of bytes, decompose the raw bytes into a sensor sample object as described above, serializes it into an HTML form encoded string, and POSTs it at the data submission URL provided by the webapp. If the webapp received good data, it replies with '`good`', and the desktop application returns to its idle state. Each time this cycle occurs, a field in the user interface is updated with the hardware address of the last node to submit data, as well as the reported temperatures.

When connected to an access point and after receiving data, the desktop application looks as it does in Figure 15.

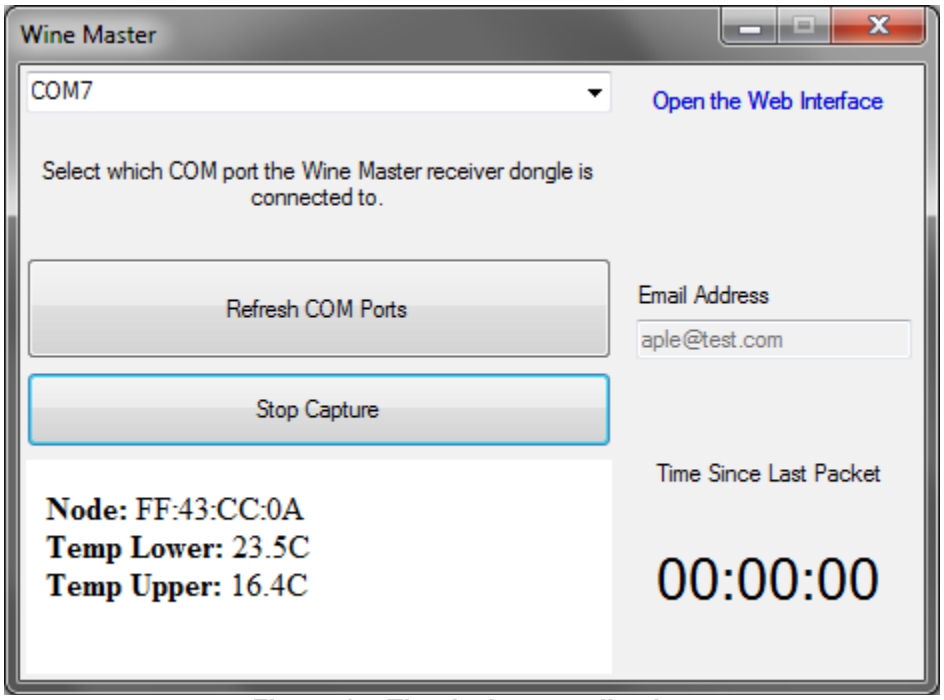


Figure 15: The desktop application

7.3. Web Application

7.3.1. Revision 1

Like the other components of this project, the webapp was iteratively developed in pieces. The first revision was as simple as possible, and had no concept of users, graphs, or CSV files; it was simply a raw printout of all submitted sensor readings. In order to get it to this point, a number of important elements had to be in place. These included the data submission URL, the sensor reading objects for the database (or Datastore in GAE parlance), and an interface to iterate through all elements in the Datastore.

The submit URL is the address that new sensor readings are submitted to in order to be entered into the database. It takes data in the form of a POST request which includes a certain set of parameters, unpacks the request into an object in Python, validates the incoming values, and writes them to the Datastore. Each POST request looks something like this:

```
nodeID=33:FF:55:DD&senseType=temp&senseUpper=28.433&senseLower=24.4
                &senseUnit=C&email=User@example.com
```

The parameters encoded in the request are:

- nodeID: The hardware ID of the node submitting the sensor reading (not the access point). This allows nodes to be unique and allows the user to differentiate between boxes of grapes.
- senseType: The type of reading being recorded. In the case of this project this is always set to ‘temp’, however if this webapp were used to store and report on different kinds of data (even if it was mixed together), this would allow it to be differentiated by type. Though each sensor node can only submit one type of data per report.
- senseUpper: This is the value of the upper sensor on the node.
- senseLower: This is the value of the lower sensor on the node.
- senseUnit: This is the unit of measurement for whatever data is being recorded. In the spirit of keeping it somewhat generic, this allows different types of data to be properly represented.

- email: The email address of the user who should be associated with the data being submitted. If this user has not previously logged into the webapp, the behavior is undefined.

The Datastore is a database in the sense that it stores data, but it is not a traditional relational database like MySQL or PostgreSQL. It does not organize data into tables, but is something closer to a key-value store which stores *entities* with *properties* such as strings, integers, and binary blobs. From a developers perspective using the default Python APIs (which is the path we chose), most simple interactions do not require the use of GQL (Google's alternative to SQL), they can query based on certain properties in other ways. In our usage case, to store an object such as a sensor reading, in order to store it, all that we needed to write was something along the lines of "`sensorReading.put()`", and it was stored.

An individual sensor reading is composed of a number of properties:

- date: The date and time stamp of when this particular reading was received by the webapp. This is stored in UTC.
- author: The User object for the user who owns this reading. GAE provides APIs for converting an email address into the User object associated with that email. This functionality is used on all incoming sensor readings in order to associate those readings with users.
- nodeID: The hardware ID of the submitting node.
- sensorType: The type of sensor readings encoded in this entity.
- sensorReadingUpper: The actual reading for the upper probe.
- sensorReadingLower: The actual reading for the lower probe.
- sensorReadingUnit: The unit of measure for this sensor reading.

It is a fairly direct mapping between what data is submitted by the desktop application to the submit URL and what gets added to a single sensor reading object and inserted into the Datastore.

7.3.2. Revision 2

The second revision of the website added the rest of the features necessary to achieve full functionality, including attractive graphs, data downloads, user accounts, and notifications. After these were added, the webapp had reached its final form and was ready for more testing and release.

The first addition was the concept of users. In order to allow more than one winery or group to use the webapp at once, reported data needed to be divided into segments based on who owned the submission. There were a number of steps which were completed to make this apply site wide. The first was to add a landing page at the base URL and move the page for browsing readings farther inside the site. App Engine allows the developer to specify regions of their site which require a user to be logged in to view. By using the mechanism, the page to browse readings was able to guarantee that the only users who saw it were those who were logged in. Once this assumption could be made, the browse page was modified so that, instead of displaying all readings of any kind which were in the Datastore, it would query for only those which were tied to the current logged in user.

With users came a new kind of entity, the UserPreference object. This object was used to store each user's maximum and minimum temperature preferences for the upper and lower probe. It will be discussed in more detail shortly.

The landing page exists to provide an introduction to what the site actually is, and suggest to the user that they are going to be forced to log in to view the rest of the site. That is its only purpose. It is shown in Figure 16 below.

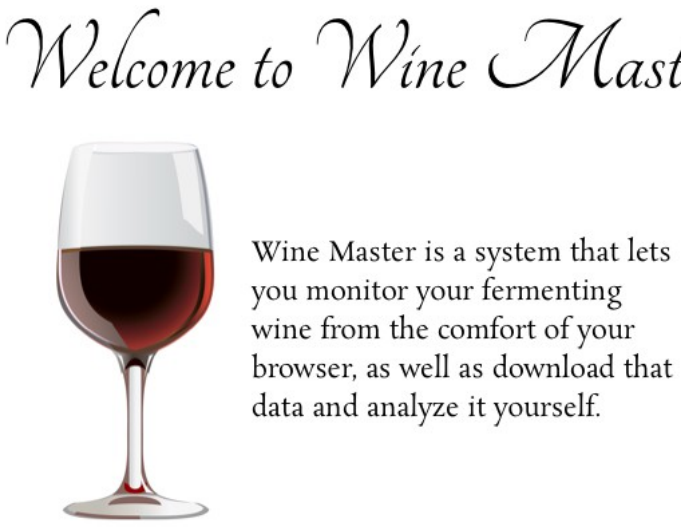


Figure 16: The webapp landing page

Once users were in place, graphs were added. One of the most useful and visible features of the webapp was the ability to view graphs of the recorded data directly on the web page. In order to provide this functionality, an open source Javascript graphing library called ico [1] was used. It provides a variety of graphs which are dynamically drawn based on arrays of data passed into certain Javascript APIs it provides. We had never written any Javascript before, but for this simple use it was not difficult to learn enough to make it work.

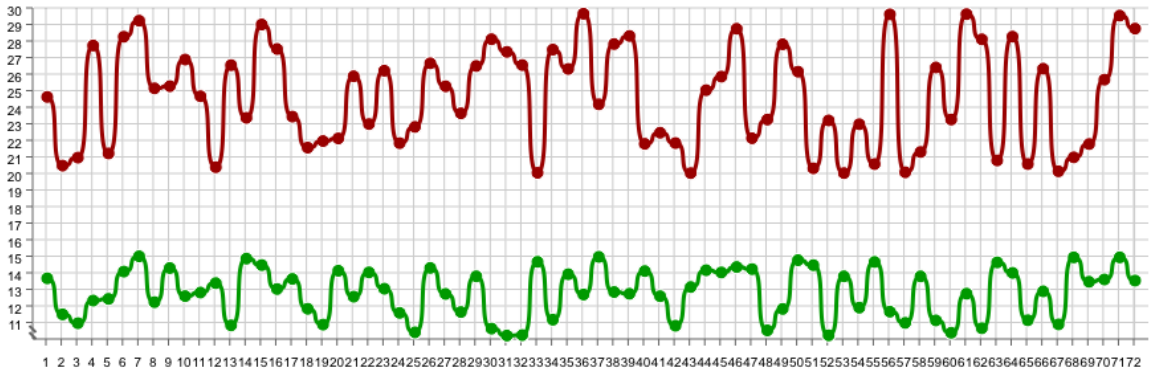
After graphs the ability to download data for offline analysis was added. A link was added to the browse screen allowing the user to download all of their data at once in a CSV file. Besides being somewhat human readable, this allows the data to be imported directly into Microsoft Excel or most other spreadsheet software.

It was important to graph each node on its own graph with separate plots for each temperature probe. To accomplish this, when the webapp pulls all the sensor readings for a particular user, it then divides them into arrays based on their node ID's. Each of these arrays gets drawn on to a graph, with the lower and upper probes each getting a trace. In order to not overwhelm the graphs with data points, we decided to only graph the last 6 hours of data, but provide the complete set for download. These graphs, as well as the CSV download link, were placed on the central browse page. An example of this page can be seen in Figure 17 below.

Hello, Iborgel (logout here).
To download all (your) readings in a CSV: here
To set max and min temperatures, go to settings

These are the datapoints that were collected for all nodes attached to your account in the last six hours.

Here is A Graph of Node 33:FF:55:DD from 2012-06-06 17:00:10.853379 - 2012-06-06 17:00:14.105618



The upper probe is red, the lower probe is green. Data gets older to the right.

Figure 17: The webapp's browse page

Finally, the webapp needed to provide some way of notifying the user if their fermenting grapes got too warm or too cold. A user preferences page was added in order to allow the desired maximum and minimum temperatures for the upper and lower probes to be specified. However, as of the most recent version of the webapp, these are applied to all sensor nodes and are not differentiated by node ID. If a user has preferences set, when data is submitted with their email address, the webapp pulls their preferences out of the Datastore and checks to make sure the incoming temperatures are within acceptable levels. If they are not, it utilizes the final feature; user notifications.

GAE is able to send emails to any address, though like all other resources it is limited by quotas. When the submission process determines that an incoming sensor reading is out of bounds, it uses a hook to send the user submitting the data a notification that something is amiss. In this case, it only sends a canned email, which includes the last temperature readings for their upper and lower probes, a URL to the webapp, and some explanation. These last two features are both of the preferences page, an image of which is below, in Figure 18.

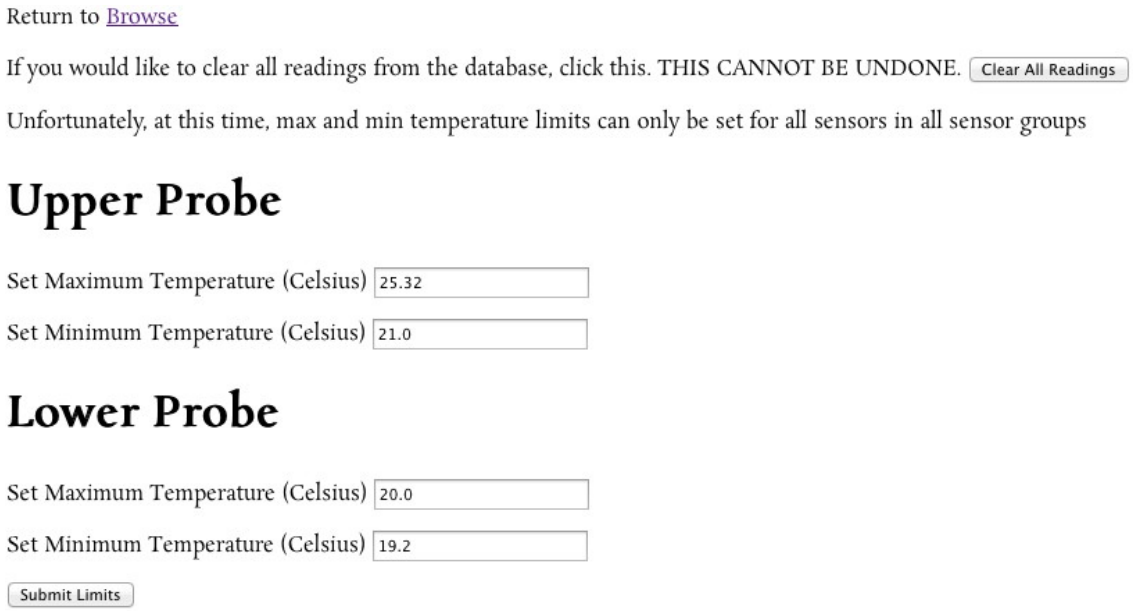


Figure 18: The user preference page of the webapp

Hierarchically, the webapp is organized as follows. Everything under `/internal` is marked as "secure" in GAE and requires the user to be logged in.

- `/` - The landing page
    - `/internal/browse` - The browse page
    - `/internal/settings` - The user settings page
    - `/internal/browse/data.csv` - The data download link (linked to, but not directly exposed)

The public URL of the website is available at [6]. Anyone with a Google account can log in, though without any data there is little to see or do.

7.4. Enclosure

Manufacturing enclosure prototypes ended up being more complex than originally thought. Despite spending considerable time attempting to design something which would be trivial to fabricate, little headway was made. In total, one prototype was constructed following the second design outlined above. We lacked precisely the correct tools to build it, so there was a large amount of filing necessary to make the hole in the center of the flat plate the correct diameter to snugly encircle the body tube. Additionally, we did not have any end caps, so a rough plug was formed for the end with the single temperature probe. To fix the plate and plug in place on the body, silicone caulk was applied liberally around the joints. The results of this piecemeal construction process were a single, rough-looking, fragile sensor node which took roughly 5 hours to build. An image of it under initial testing is visible in Figure 19.

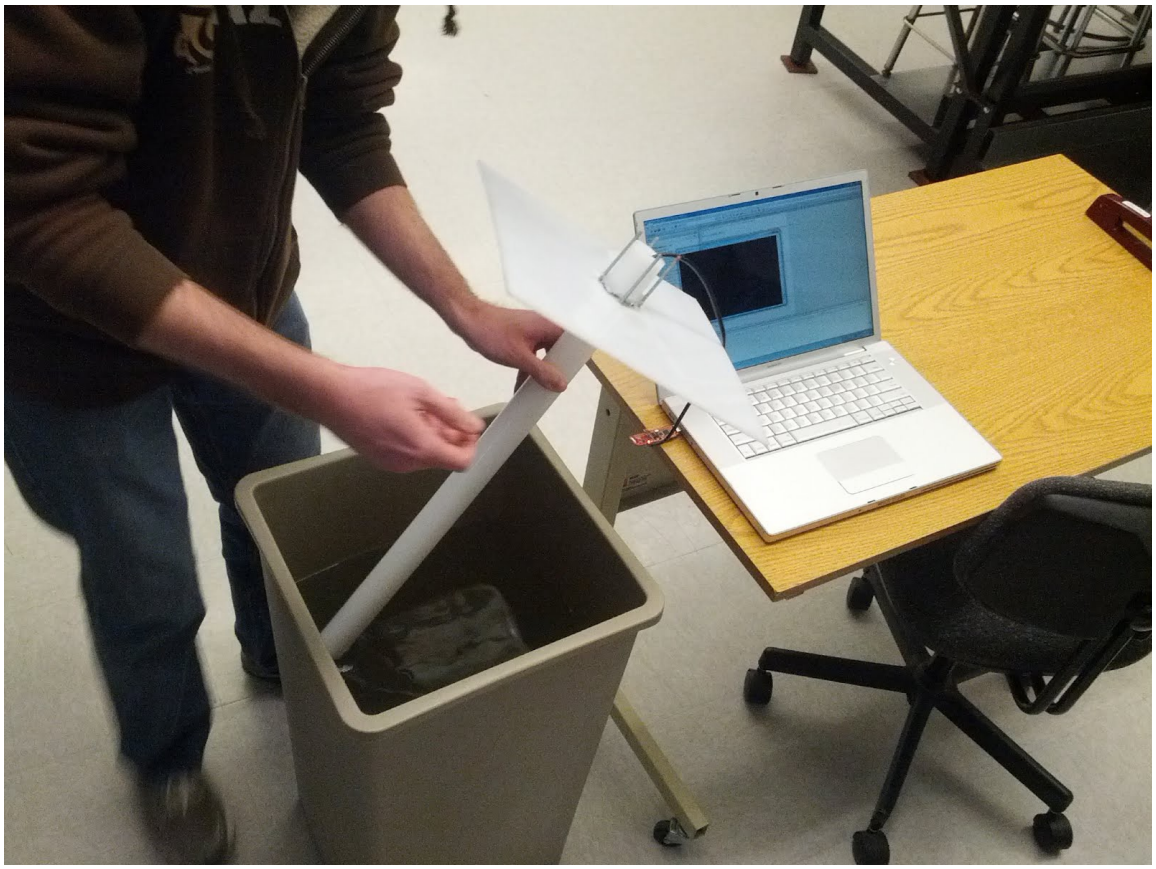


Figure 19: The first prototype under test

The components for a larger second prototype, which would have represented the third design from above were purchased but ultimately not used. This time, we obtained appropriately sized end caps to attach to either end of the body tube, but they needed their internal threads removed to fit correctly. We intended to use more appropriate machine tools (such as a lathe) to modify the end caps, as well as to cut a groove into the body tube to hold the plastic plate and obviate the need for caulk. However we discovered that we did not have sufficient machine shop certification to use the lathe, and our body tube was curved which prevented us from machining it. These are both problems which could be overcome, given more time.

8. Conclusion

Overall this project can be considered nearly a success. In terms of the computer engineering requirements, all goals were met or exceeded, though the enclosure is not yet complete in a final form. The resulting system is a collection of reproducible hardware and software which can be used to create a self-organizing wireless sensor network to measure temperature (or other parameters) and report it to a remotely accessible web interface. The desktop application and hardware is easy to setup, configure, and use, and should be functional for extended periods of time. The web application can handle an arbitrary number of users, presenting each with their own sensor data and allowing them to download it for local analysis.

There is room for future development in all areas of this project. Most obviously a prototype of the third design enclosure could be built, which would allow the entire system to be tested (or used) in situ. This is the critical last step in actually producing an entirely working system, which at this point has not been

done. However, even if this prototype were to be constructed it would still suffer from high fabrication complexity, and it would not be possible to test in situ until the fall when fermentation happens.

There was actually an additional prototype which was constructed out of line with the others and thus was not mentioned above. This one, which we will refer to as Design 1.5, was constructed entirely out of threaded and smooth PVC pipe purchased from a hardware store. It followed the same principles of operation, but cost substantially less, could be easily obtained, was assembled in few minutes instead of hours, and was even reconfigurable. But this prototype was not food safe and thus could not be used. An ideal final enclosure design would exhibit all these characteristics but be food safe as well. Unfortunately, quick research did not find a source for food-safe PVC and Design 1.5 was abandoned.

Perhaps the most significant potential improvement to the sensor nodes themselves would be to reduce their duty cycle to conserve power. Currently, the radio must always be awake in order to receive data packets at asynchronous intervals. However, the microcontroller spends most of its effort in a run loop that multiplexes the handling of interrupts for the radio and various timers. It is possible to put the microcontroller to sleep when it has nothing to do such that the timer and radio interrupts can wake it up only when absolutely needed [3]. Based on the current implementation, it may be feasible to reduce the microcontroller's duty cycle to as low as 20%, which would noticeably improve battery life.

At the outset of this project, we considered sensors that would measure other useful parameters for winemakers including pH, sugar content, and CO_2. After some preliminary research, we found that these sensors were either too large, too expensive, or, in the case of sugar sensors, did not exist.

The webapp, while functionally complete has plenty of room for enhancement. The most significant improvement would be to add any sort of security to the system. The secure portions of the site automatically use HTTPS to obscure any transactions, but this is not the vulnerable area. Anyone who knows the submission URL or has a copy of the desktop application can submit any amount of data at any rate to any user's account. Besides filling the target user's pool of sensor readings with garbage data, this could actually cost the owner of the webapp money, as it could exceed billing quotas.

There are two easy solutions to this. One is to have the webapp put its own quotas on data submission. While it would still consume quota to have the webapp spin up merely to reject incoming sensor readings and then return to idle, it would save space and queries in the Datastore, which are both held under a quota. The second solution is to force every user to use a password when they submit data, and then use HTTP Basic Authentication or something similar each time the desktop application submits data. Under this scheme, the user would have to log into the webapp and set a password before submitting data. Then they would have to enter this password in addition to their user account's email address into the desktop application before it would submit data.

In the realm of additional features, there are a wide array which could be applied to the webapp. Structurally, the source should be modified to use something called a "templating engine", which allows a developer to completely separate the data to be displayed on a web page from the raw HTML which is used to display it. When the page is loaded, the HTML is programmatically generated based on style guidelines the developer sets. This is considered good practice, as it makes complex webapps much easier to maintain.

Functionally, it would be ideal for the user to be able to configure many types of notifications. This would be most easily accomplished by attaching the webapp to a service such as ifttt (which stands for If This Then That). Ifttt lets non-programmers build something akin to a script which takes certain input (like an incoming email, a post to Facebook, a Tweet, or the time of day) and takes some action (sending an email, making a phone call, sending an SMS message, posting to Facebook, Tweeting, and much more). ifttt would let the users of the webapp completely customize their notification methods and enable a very wide range of possibilities, including text messages, emails, Tweets, phone calls, blog posts, and more.

In terms of user interface, there are a number of small tweaks which would make it an overall more pleasant experience. Users should be able to group nodes together, name them, and name the groups.

This would allow for some sort of organization, as well as the concept of “wineries” and user simplicity (by allowing them to choose nodes by name). The browse page should clearly show what the most recent observed temperatures for each node were, and how long it has been since they last reported in. The user should be able to download CSV files of particular nodes as well as all at once.

References

[1] alexyoung. “ico (Javascript graphing library).” <http://alexyoung.github.com/ico/>.

[2] Maxim. “DS18B20 Programmable Resolution 1-Wire Digital Thermometer.” <http://datasheets.maxim-ic.com/en/ds/DS18B20.pdf>.

[3] Texas Instruments. “MSP430F2274 Mixed Signal Microcontroller.” <http://www.ti.com/lit/ds/slas504f/slas504f.pdf>.

[4] Texas Instruments. “MSP430 Wireless Development Tool.” <http://www.ti.com/tool/ez430-rf2500>.

[5] Texas Instruments. “SimpliciTI Application Programming Interface.” <http://www.cs.washington.edu/education/courses/cse466/10au/pdfs/SimpliciTI%20docs/SimpliciTI%20API.pdf>.

[6] “Wine Master Webapp.” <http://cpwinemaster.appspot.com/>.