UrbanSense

Team 20

Aatish Nayak □ Chris Wei □ Riya Savla □ Ridhi Surana

May 14, 2017

Abstract

The proliferation of AI powered traffic lights, internet connected streetlamps, solar powered roads, smart flood watch systems, among others, signals the movement towards smarter cities. As the first city with driverless cars and an endless pool of the world's best engineering talent, Pittsburgh is perfectly positioned to flourish in the 21st century. However, with aging road infrastructure and regulatory oversight, it's becoming even more important for municipalities to embrace smart decision making. Local governments are more and more starting to rely on data to drive their investment decisions in infrastructure, public transit, and real estate. UrbanSense takes advantage of the reach and regularity of public transportation to gather valuable metrics about a city. UrbanSense presents data about roads, air, and transit systems to city officials to assist them in making better decisions about the cities we live in everyday.

Table of Contents

Project Description

UrbanSense is an urban sensing platform that aims to take advantage of the reach and regularity of public transportation, like the Port Authority bus system in Pittsburgh, to collect data to extract meaningful metrics. Specifically, in our prototype, we plan to include pothole detection and noise-level monitoring. Our goal is to design a modular system that can seamlessly integrate additional sensors for new metrics. We will consolidate all of these sensors into a set of compact devices that can be attached to any vehicle.

Design Requirements

Requirements: UrbanSense needs to be a cost-effective solution that can scale vertically to large numbers. To achieve this, or solution needs to be highly energy efficient where possible and be able to grow without high cost of entry. Additionally, our solution must also handle a massive amount of data transfer and storage. Lastly, we need UrbanSense to scale horizontally, making the cost of adding on additional sensors minimal.

Cost-effective: The driving idea behind UrbanSense is that the more data the city has, the more actionable conclusions the city will be able to draw. As an example, if multiple buses report a pothole at a particular location then the city can be more and more certain that a pothole is indeed there and can be serviced if need be. To achieve this, we have designed a system uses inexpensive microprocessors and sensors to create a sophisticated system capable of collecting useful data. We also need to consider power sources that can be easily sustained and layout designs that minimizes the amount of service these microprocessors need. Nearly everything that can be automated should be automated, since human interaction greatly increases the cost of operation.

Diversification: To scale horizontally, we need to design our system to handle an arbitrary number of additional sensors (obviously limited by our choice of microprocessor). More specifically, we need to design our overall system to not discriminate between different types of data and make data from, for example, a pothole sensor and a pollution sensor treated exactly the same by the system. The point at which data gains meaning is when it is in the server and a job consumes the data to produce results. To achieve this, data is stored as generic rows of data that contain the location, the time, the type of sensor, and the measurement as

well as some other metadata that might become relevant. The microprocessor simply tags different data with different tags and those tags ascertain meaning when they are processed by some online job.

Architecture

Central Data Collection:
We choose a Atmega328P to interface with our sensors as it has dedicated hardware support for various communication protocols and an inbuilt analog-to-digital convertor. The Atmega328P has four dedicated ADC pins and one SPI channel for interfacing with peripherals. We chose to use an external 8-way multiplexer (the number is arbitrary) to be able to support a higher number of analog sensors than what the Atmega328P inherently allows.

The microcontroller software is structured to allow easy customization. The user instantiates a ‘ processor’ that takes care of setting up the required Atmega environment to enable SPI, I2C, ADCs etc. The user can register a maximum of eight analog devices and provide an ADC value to meaningful value convertor function (eg. IR sensor ADC values to distance in metric units). The user can also register one SPI device along with a start-up function and a read function specific to that device. Once the processor boots, it starts polling the sensors with a user-configurable frequency and stores collected values in memory until the central data collection unit asks for a set of values.

We used a Raspberry Pi as a central data collection unit. The Pi polls the Atmega at a predetermined frequency for a set of all sensor values that are attached to the Atmega. It doesn’t care about what the sensor values mean, simply identifying each sensor type with a sensor ID and each individual sensor value with a tag ID. The Pi talks to the Atmega using I2C, with the Pi acting as master and the Atmega as slave. This allows the Pi to communicate with a number of other MCUs (limited by the size of the I2C address), thus allowing the system to scale horizontally to include a large number of sensors.

Once the sensor data packets are received by the Pi, if it has internet connectivity (wifi), it will send the data to the preconfigured server URL. It also caches the data locally in Redis for redundant storage in the case that an internet connection cannot be established.

Pothole Detection - This subcomponent contains the IR sensor and accelerometer for pothole detection. Although the name suggests that this subsystem determines the presence of

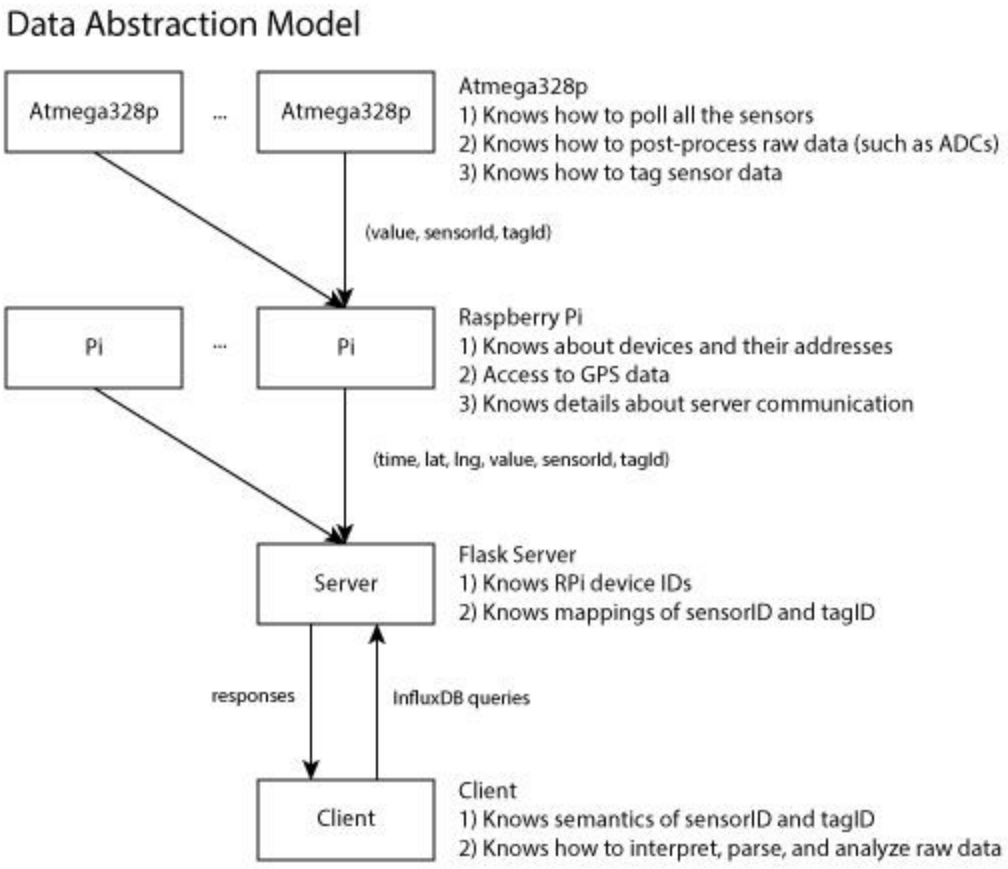
potholes, it only records raw sensor data and pushes it to the central data collection system. The actual detection of potholes will be determined on the cloud server discussed below. Another caveat is since this system will be connected to the central broadcast system via a some hardware interface, the central system will poll the IR sensor and accelerometer as opposed to the sensors broadcasting data.

GPS - This subcomponent contains the GPS module and Real Time Clock. The central data collection function will poll this component whenever it needs the current location and timestamp.

Feature Collection and Storage - This subsystem harbors the server and time series database that stores the data received from the Pi. Since at this point the data is still generic, the server also fetches the sensor id and tag id configuration for each data point before storing in the server. This ensures it does not need to know which sensor the data came from or what the data represents. Simply that the sensor id and tag id have entries in the configuration lookup table.

Feature Extraction and Visualization - This is the most interesting system in that it actually does processing and visualization of the sensor data to gather meaningful insights. Independant jobs written for each type of sensor (accelerometer, ir, and sound in our case) do a series of anomaly detections using an exponential moving average. Each anomaly is flagged and saved in a separate DB. Then the client server extracts data from this DB and visualizes it on a map and time series graphs. Looking forward, the correct authorities such as city and transportation planning officials can analyze the maps and graphs to determine the conditions of roads and noise pollution throughout the city.

Design Trade Studies



Two chip design - We added a Raspberry Pi to our design for a couple of reasons. First, we needed a processor to handle threading, which would allow us to make progress even when an instruction blocks (such as checking for internet connection by pinging google.com, sending a POST request, waiting for a sensor to produce a value). Secondly, we also needed the hardware to enable WiFi and Bluetooth (which we later decided was not necessary) - the Pi already had all this hardware and had many nice properties that would be hard to implement by ourselves, including caching previously used WiFi connections so that the Pi can connect automatically when a remembered network is available. We could have directly used the Pi's GPIO pins to poll sensors directly, but we decided on using a Atmega328p MCU to poll sensors directly, which would allow for multiple MCUs to be distributed across the bus, and a single Pi to poll all of them, as seen in the Data Abstraction Model diagram.

PCB Design - We wanted our overall product to be compact so our PCB was designed to be a shield that would sit on top of the Pi, taking up as little room as possible. This constraint forces our PCB to be a certain dimension in order to fit on top of a Pi, and thus we are physically constrained when deciding how many headers to put on our board. Primarily, we wanted to support 8 ADC devices on our MCU which exceeds the total number of ADC ports our MCU had. We placed an onboard multiplexer in order to support all 8 channels - we could have used all 16 channels of the multiplexer, but due to limited space, we settled with 8. In addition to the ADCs, we left room for the GPS chip, which is connected directly to the Pi, as well as an accelerometer chip. Power to the whole board is supplied by the Pi's 5V output.

GPS - We decided to use a breakout board GPS from Adafruit since we did not want to introduce unnecessary complexity to our board. Instead, we simply added several pins that allowed the GPS to be swapped in and out. Most (possibly all) GPS chips communicate serially through UART and through a protocol called NMEA, which sends data as a comma separated strings and requires a lot of cycles string parsing. The GPS can be polled for new positions at a set frequency which blocks whatever device polling the GPS blocks when no new positional data is available, thus prevents progress on the other sensors. Due to the need for fast string parsing and the need for threads to allow for progress, we opted to move the GPS module to be connected directly to the Raspberry Pi's UART pins, which gives it a more heavy-weight environment that can parse strings faster and allow for threading.

Protocol Selection - To show the flexibility of the system, we opted to use several different types of sensors that measure different metrics and communicate using different protocols. To start with, we had four protocols to poll data from the MCU - SPI, UART, I2C, and ADC. We chose to use I2C to communicate between the MCU and the Pi since UART on the Pi was being used to poll the GPS - this also freed up the UART channel on the MCU which allowed us to use that channel for debugging. Unfortunately, no existing code to set the Raspberry Pi as a I2C slave existed (and we did not have the resources to implement I2C slave code for the Pi) so we chose to use the MCU as a slave which introduced a couple of issues. First, the MCU could no longer poll any other I2C sensors since it must be registered as an I2C slave; possibly, the Pi could directly poll the other sensors, but this would break our interface since it would require the Pi to know specific details about specific sensors. Secondly, the MCU must wait for the Pi to poll it in order to send any data, thus the MCU must coordinate with the Pi to poll data only when the MCU is ready with a new batch of data. However, since the Pi acts as the master, this allows for it to communicate to multiple MCUs which might be something we would want in the future.

Sensor Protocol Selection - Since I2C was used to mediate communication between the MCU and the Raspberry Pi, we are left with three protocols to poll data - SPI, UART, and ADC. We knew that we wanted to use an accelerometer, and virtually all accelerometers on the market use SPI or I2C. This required us to use the only SPI port on the MCU to poll the accelerometer. The remaining sensors were all chosen to use ADC due to the design of the PCB.

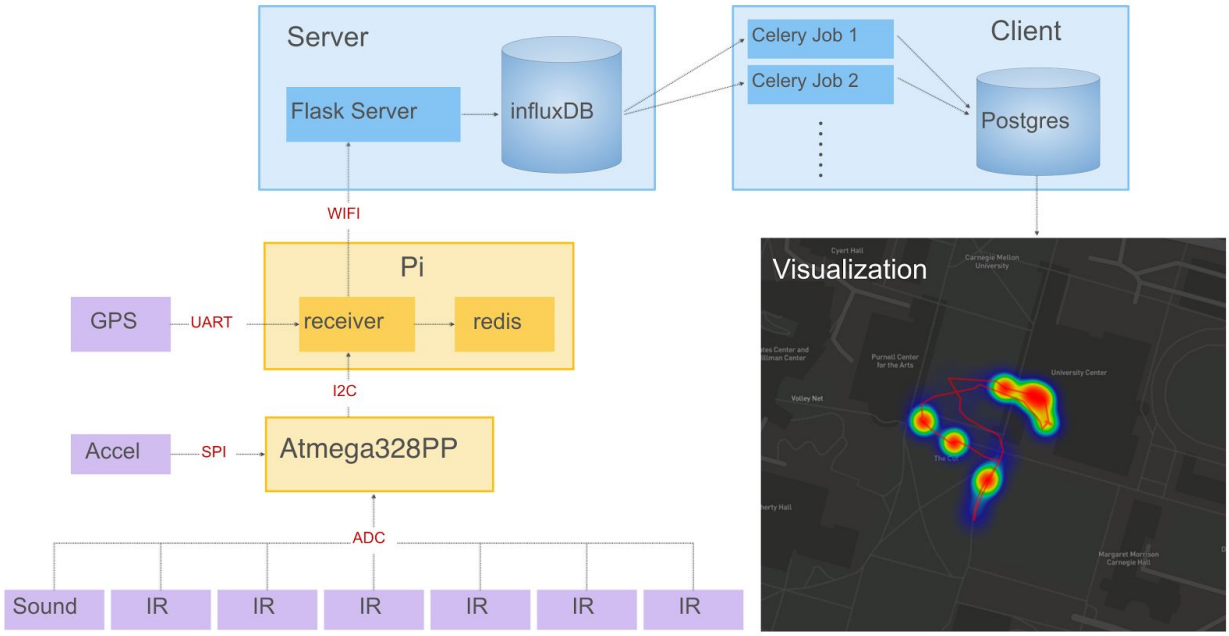
Voltage issues - One design choice that plagued us throughout development was the choice of using 5V as opposed to 3.3V as our power source. The Raspberry Pi had both voltages available, and choosing 5V over 3.3V means that all of the MCU GPIO outputs would be 5V. Virtually all accelerometer candidates operated safely in the 3.3V range but not at 5V - this required us to use stick a logic level shifter into the PCB design which would take up a lot of room. On the flip side, when running at 3.3V our IR sensors no longer operated, and at this point of the project, we had already made a hefty $90 investment in the chosen IR sensors. This forced us to stay with 5V supply, and to move the accelerometer off the chip and onto a breadboard. Further down the road, the issue with supporting both 3.3V and 5V sensors could be solved by having two MCUs, one powered with 3.3V and the other with 5V, and have both be polled by the Pi through I2C.

Local Redundant Caching - As mentioned previously, if the Pi cannot establish a network connection to offload the data, it stores it locally in Redis. Redis was chosen as a caching agent because it has different configurable levels of on disk persistence. With This ensures that the Pi does not run out of RAM in the case that it collects data for over an hour. Additionally, since Redis has a key value storage design, it allows us to store schemaless data. We considered other options including MongoDB however, we chose Redis since it has a low memory footprint and one of our team members had extensive experience working with it.

Data Collection Server Design - The server communicates with the Pi by exposing a HTTP endpoint that a Pi can POST data. We decided to use Flask for the server over alternatives like Django and Node.js. Flask is generally used for APIs containing little to no html templates, and since it's implemented in Python, has extensive support for the plethora of useful python libraries. Since we did not need templating and an extensive ORM like Django provides, Flask did the job. Additionally, we needed a write optimized database to store time and geostamped data while also allowing for efficient time based aggregating queries. After looking for a database that supported our needs, we discovered InfluxDB, an open source time series database best suited for IoT device monitoring allowing millions of writes per second. Most importantly, it uses a SQL-like query language with support for mean, standard deviation, selecting my date range with nanosecond precision, the applications of which will be described in the Data Processing and Visualization section.

Data Processing and Visualization - Once the server writes data to InfluxDB, the anomaly extraction process begins. We wanted to make the processing jobs asynchronous and scalable to many different type of sensor values. In other words, whenever new data is avaliable the job should start processing it. We decided to use Celery, an asynchronous task queue based on distributed message passing. Additionally, Flask has a plugin for celery allowing scheduling to independant jobs for each sensor value type to run periodically. For the actual anomaly detection, we decided to use a derivative detector algorithm, similar to various algorithms used for edge detection in computer vision, because it highlights abrupt changes in value. Since the algorithm involves finding a moving exponential average, I experimented with test data to determine the best smoothing factor for each type of sensor value. In the end, the ideal smoothing factors for each sensor were 0.2, 0.3, and 0.6, for IR, sound, and accelerometer data, respectively. After flagging each data value as an anomaly or not, the refined post processed data was written to PostgresQL. From there, the user could visualize the data on a real city map overlaid with a heatmap as well as a time series graph. For rendering the map, we decided to use Leaflet + Mapbox over Google Maps. Leaflet supports adding multiple layers on top of the map as well as rendering thousands of map markers and polygon elements for visualizing a path simultaneously. For the graph component, we decided to use Chronograf, the recommended visualizer for InfluxDB data. It exposes a customizable dashboard to write custom Influx queries to generate graphs. Through the use of these two tools, city officials could get meaningful insights from the sensor data.

System Description/Depiction



Project Management

Week #	Deliverables
1	• **System Demo #1** • All parts in hand, set up required RPi environment. • Set up SPI communication with Accelerometer
2	• Initial server architecture was in progress. • Tested IR sensors and map ADC values to distance measurements
3	• **System Demo #2** • Decided on a protocol for sensor data communication between the Atmega and the Pi.

	● Test and debug Atmega <-> Pi communication via UART. ● Worked on the Pi data caching system.
4	● Worked on getting the Pi to poll the GPS to geostamp and timestamp the data collected ● Made Pi offload data to server ● Initial test run to get an idea of what real data looked like
5	● **System Demo #3** ● Visualization for sensor data in place ● Switched to I2C communication between Atmega and the Pi
6	● **System Demo #4** ● Design final PCB version so that it sits right on top of the Pi as a backpack ● Tested and calibrated Sound sensor
7	● **System Demo #5** ● Restructured the Atmega software to be make it more modular ● Worked on physical casing to create an self-contained product.
8	● **System Demo #6** ● Tested the entire system together and collected real data by attaching the rig to a car ● Worked on Demo.

Team Member Responsibilities -

Person	**Primary Responsibilities**	**Secondary Responsibilities**
Aatish	Cloud server and accompanying infrastructure set up	RPi data caching
Chris	RPi data collection and GPS software	PCB design
Riya	MCU <-> RPi communication	Packaging
Ridhi	MCU <-> Sensors interfacing	Sensor calibration

Budget

Part Name / Description	Link to Part Website	Qt.	Unit Price	Total Price
Raspberry Pi	https://www.adafruit.com/products/3055	2	$40	$80
32 GB SD Card	https://www.amazon.com/Samsung-Class-Adapter-MB-MP32DA-AM/dp/B00IVPU786/	2	$10.99	$21.98
				$0.00
Coin Cell for GPS RTC (5-pack)	https://www.amazon.com/Maxell-CR1220-Lithium-Batteries-5-Pack/dp/B0014WUYWC?th=1	1	$4.20	$4.20
USB <-> TTL Cable	https://www.adafruit.com/products/954	1	$9.95	$9.95
IR Sensor Long range	https://www.sparkfun.com/products/8958	3	$14.95	$44.85
Accelerometer	https://www.sparkfun.com/products/11446	1	$14.95	$14.95
32 GB SD Card	https://www.amazon.com/Samsung-Class-Adapter-MB-MP32DA-AM/dp/B00IVPU786/	1	$11.95	$11.95
F/F Jumper Wires	https://www.sparkfun.com/products/12796	2	$1.95	$3.90
JST Connector Wire	https://www.sparkfun.com/products/8733	3	$1.50	$4.50
Break Away Headers	https://www.sparkfun.com/products/116	2	$1.50	$3.00
Logic Level Shifter	https://www.sparkfun.com/products/12009	1	$2.95	$2.95
16 Channel Multiplexer	https://www.sparkfun.com/products/299	1	$0.95	$0.95
Velcro Tape	https://www.amazon.com/dp/B00114LOMM?ref=emc_b_5_i&th=1	1	$28.35	$28.35
RPi Case with Heat Sinks	https://www.amazon.com/Smraza-Raspberry-Supply-Heatsinks-Switch/dp/B01I1OESI6	1	$12.99	$12.99
Zip Ties	https://www.amazon.com/TR-Industrial-TR88302-Multi-Purpose-Cable/dp/B01018DC96/ref=sr_1_2?ie=UTF8&qid=1492109237&sr=8-2&keywords=zip+ties	1	$5.99	$5.99
Sound Detector	https://www.sparkfun.com/products/12642	1	$10.95	$10.95
Gas Sensor Breakout	https://www.adafruit.com/product/3199	1	$14.95	$14.95

Risk Management

Risk Type	Risk	Mitigation Strategy
Technical	BLE bandwidth cannot handle the amount of data we upload to the beacons	Remove Beacon as the middleman, attach 4G or Wifi chip onto on-bus device and broadcast data straight to server
Schedule/ Technical	Setting up server/sensors proves to be more challenging than expected	In addition to removing the beacon (same as above), we will prioritize our sensor implementations and stick two one or two core features (e.g. pothole and noise level monitoring)
Technical	RPi and sensors consume too much power from onboard battery	Use voltage regulator and vehicle's battery to power
Technical	Due to lack of proper set up, battery fries sensors	Buying extra sensors so progress is not halted
Budget	The per unit cost of a sensor system becomes unscalable.	Change launch plan to only deploy system to 1-2 busses per route

Evaluation

- **Power -** We used a portable battery to power our system via the RPi. However, in the future versions we would need a longer-lasting power source as explained in the conclusions sections below.
- **Range -** To be able to detect unusual depths on the road, we looked at long range IR sensors and LIDAR. While LIDAR sensors have a better accuracy, they are much more expensive and hence we went with the IR sensors. They have a good range of 20 cm - 2m and would help us detect anomalies.
- **Sampling Rate -** The GPS updated location every half-a-second and hence we decided that sensor polling rate be the same. As a result, our data polling rate was a little slower than would be ideal to derive more meaningful metrics.

Lessons Learned

- **Parts/Sensor Selection** - One of the key lessons we learned is that parts selection must be done keeping the operating conditions of the selected sensors in mind. The IR sensors and the accelerometers worked with operating voltages of 5v and 3.3v respectively and we spent a lot of time and effort interfacing both the sensors with the same MCU.
- **Migration to a different operating voltage** - On deciding to a shift to different operating voltage, we must make sure that all our parts would work with the newer voltage.

  In order to interface the accelerometer directly with the MCU, we decided to shift the PCB to a 3.3 operating voltage. This would have impacted the performance of the IR sensors as they only worked within 4.5v-5.5v; however our final PCB was connected to RPi's 5V and hence the IR sensors still worked with the new PCB. This meant that the accelerometer could no longer be mounted on the PCB and had to be shifted to a breadboard.
- **Time spent on hardware components** - When deciding on the overall schedule of the project, some extra time must be allocated as buffer time in making the hardware synchronize properly with the rest of the system. We spent a lot of our time and budget on making the accelerometer work with our PCB. Although it was required, it would have been useful to have some buffer time in our schedule just for this purpose as we would have structured our goals differently.

What would you do differently if you could start from scratch?

- **Lab 3 sensors** - We should have chosen the sensors that we were thinking of using in the final projects instead of the heartbeat sensor. This would have given us an opportunity to work with those parts beforehand and would have allowed us to make any necessary changes in our parts selection process.
- **Comprehensive System Design** - If starting from scratch, we would design the entire system together and pick hardware and sensors that complemented each other better. This time, we picked parts individually, without putting them into the context of the

entire system, and that led to a few problems we've already talked about (eg. the 3.3 vs 5 V voltage issue).

Future Work

- **Compaction** - A good starting point would be to work on making our prototype as compact as possible. The IR sensors would still have to scale out from the prototype to cover as many potholes as possible but the accelerometer could be moved off the breadboard to sit on the PCB. There are two ways to achieve this - solder both the logic level shifter and the accelerometer on the 5v PCB although we are limited by size here. Otherwise, we could move the accelerometer to second 3.3v PCB and make the RPi poll both the MCUs instead of just one to collect sensor data. Second option seems more scalable here since there will always will be a limit on how many sensors we can attach to a single PCB compared to having RPi poll multiple MCUs each specialized for a different type of sensor(s).
- **Power** - At the moment, we are powering the RPi through a portable battery. To be able to attach the prototype to a bus and collect data from it for 10 - 12 hours, we would have to provide a more stable, long-lasting power source. The preliminary plan would explore the following options - power the system with the portable battery and the portable battery with the bus battery. Alternatively, we can directly power the system with the bus battery. We do realize that there are safety issues to consider here and our plan would best adhere to the regulations issued by the city authorities.
- **Energy-efficient:** Similar to cost-effectiveness, these devices should not place a huge burden on any particular power source. Particularly, the embedded devices should be able to run long times without changing batteries (this becomes irrelevant when the devices are powered by the car battery in v2.0). We have designed a system to be overall energy efficient. This means reducing the amount of power used by microprocessors by using an interrupt system that allows the microprocessor to sleep when no useful data is being collected. In contrast, the beacon must be less power efficient because it has to handle massive amount of data coming in from multiple processors and upload to the server via WiFi or 4G. This is unavoidable but the best solution in terms of energy efficiency as a whole.

Related Work (Competition)

2017 Ford Fusion V6 -
http://www.motortrend.com/news/2017-ford-fusion-v6-sport-features-pothole-detection-system/
Jaguar Land Rover -
https://www.theengineer.co.uk/issues/june-2015-online/jaguar-land-rover-unveils-pothole-detection-technology/

University of Colombo Research Paper -
https://pdfs.semanticscholar.org/6b44/e5504dd23ae33484eee01bb356eb0c61d3a7.pdf

Air Quality Sensors -
MIT Lab - http://news.mit.edu/2016/air-quality-sensors-track-pollution-0615

Noise Reduction System -
Sonitus Systems - http://www.sonitussystems.com/applications/environmental

References

Edge Detection Algorithm -
https://en.wikipedia.org/wiki/Edge_detection

InfluxDB documentation -
https://docs.influxdata.com/influxdb/v1.2/

Flask Documentation and Examples -
http://flask.pocoo.org/docs/0.12/