Design and Proof of Concept of Parking Garage Capacity Network using Distributed Ultrasonic Devices Interfaced with MQTT Protocol

Erik Olsen

Justin Distaso

Project Advisor

Tina Smilkstein

Computer Engineering 463/464 Senior Project

California Polytechnic State University, San Luis Obispo

June 12th, 2017

Abstract

This paper overviews the Smart Structure project. We found a desire from the City of San Luis Obispo for a way to tell the availability of parking spaces in local parking garages. In addition to meeting this need the project aims to provide functionality and adaptability based on future "smart" devices and making the device fit into an Internet of Things (IoT) system. Currently drivers must manually navigate the parking garage to determine which floor has a parking spot. With our proposed solution drivers would be able to tell at a glance which floors had the most spots, as well as the relative fullness of each floor. Initial testing included a magnetic sensor, but it showed a lack of promise. The final solution is accomplished using a network of ultrasonic sensors. These sensors communicate to a database that is able to be accessed via a GUI. It is hoped that this project will provide the information needed for a real-world implementation of the network. This implementation could be done cheaply and with future device expansion built in.

Table of Contents

- 4 Introduction
- 6 Technical Background
- 13 Design
- 16 Testing
- 18 Conclusion

1. Introduction:

When deciding on a senior project we wanted to choose something that would be useful to somebody after the project was completed. Our advisor already had a passion for IoT capable devices so we looked in that direction. Since IoT works well with public infrastructure and community services (such as a smart city) we decided to meet with our city, San Luis Obispo, to discuss possible smart city implementations. Through our advisor we set a meeting Greg Herman, Assistant to the City Manager. At this meeting we established one way the city could be improved through an IoT implementation was in the field of parking, specifically the local parking garages.

When using a public parking structure certain floors have more spots than others. In fact some floors will not have any spots at all during busy times when the structure is at or near full capacity. Currently the only way to know is to drive your car in circles, searching for a spot. Depending on the layout of the structure you will need to decide if you will keep going round or try another floor. This wastes time and gas and is frustrating to those trying to park their cars.

Our project aims to solve this problem by providing a way to view how many spots are open on each floor. Using ultrasonic distance sensors interfaced with a network of raspberry pi computers we are able to track cars as they enter and exit each floor. By keeping track of how many vehicles are on each floor and knowing the total number of spots we are able to show the number of available spots on each floor. This might sound familiar, indeed some parking structures already have the number of spots on each floor displayed. Where our solution improves on existing systems is how the data is made available. By passing the data to a database using a lightweight protocol (MQTT), it is possible to view the number of open spots view an application or visually, depending on the structures implementation. In addition our solution was designed with future Internet of Things (IoT) development in mind, and is able to share its data with compatible devices to form a building block of a smart city IoT network.

Because this is a real world implementation we had to work under several technical constraints that would be less relevant in a purely theoretical design. As with any real world implementation, but especially for a non-necessary expenditure by the city, our project needed to be as low cost as possible. Specifically, on a per unit cost, since this project is a distributed system with many nodes, even a small decrease in the per unit price can have a significant impact on the total cost since the price scales per unit. Another part of the cost is in power usage, so we chose very low power devices and protocols. Ease of deployment was another factor we considered, if everything comes more or less put together it will save time and money when the units are deployed. Finally while our project is useful by itself, we made sure to design the architecture of the system with future expansion into the IoT field in mind. Our devices publish messages to a network and can receive from and push data to future yet to be added devices as the city expands upon its Smart City development.

2. Technical background

TCP/IP

TCP/IP is the basic communication protocol of the internet. Just about every computer comes loaded with the programs needed to access the internet. This collection of programs is often referred to as the communications “stack”. TCP/IP has two parts. The first layer is Transmission Control Protocol (TCP). TCP assembles the messages your device sends into smaller packets of data that are transmitted over the internet and received by the TCP layer of the receiving device. The lower layer, Internet Protocol (IP) handles the address part of each packet so that it gets to the right destination. Each computer on the way through the network checks this address to see where to forward to packet to get it to your specified destination. TCP/IP uses a client/server model of communication, meaning a user (the client) makes a request to the server, and the server provides the service (such as sending the web page data). Most common protocols such as Web browsing (HTTP/HTTPS) and email (SMTP) utilize TCP/IP to get to the internet.

How TCP/IP Works

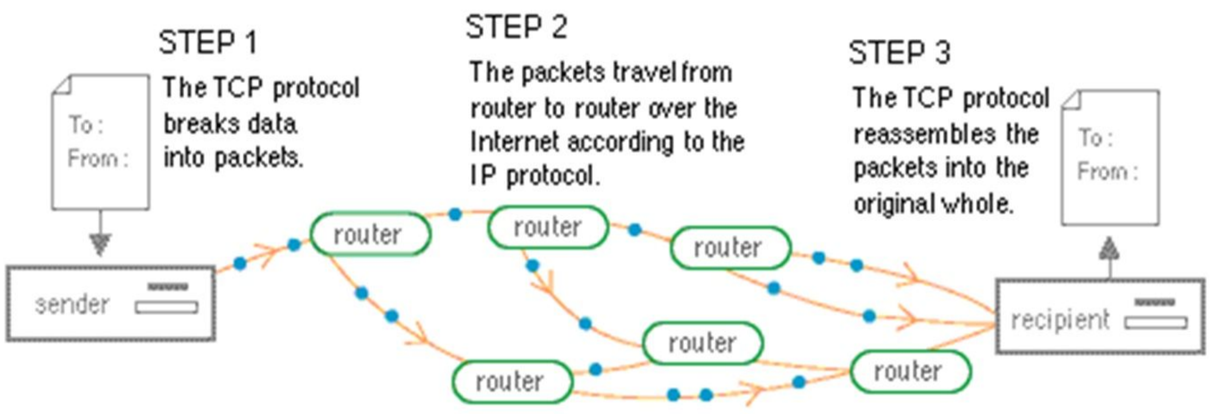

STEP 1
The TCP protocol breaks data into packets.
STEP 2
The packets travel from router to router over the Internet according to the IP protocol.
STEP 3
The TCP protocol reassembles the packets into the original whole.
To :
From :
sender
router
router
router
router
router
router
recipient
To :
From :

Figure 2. How data travels over the Net.

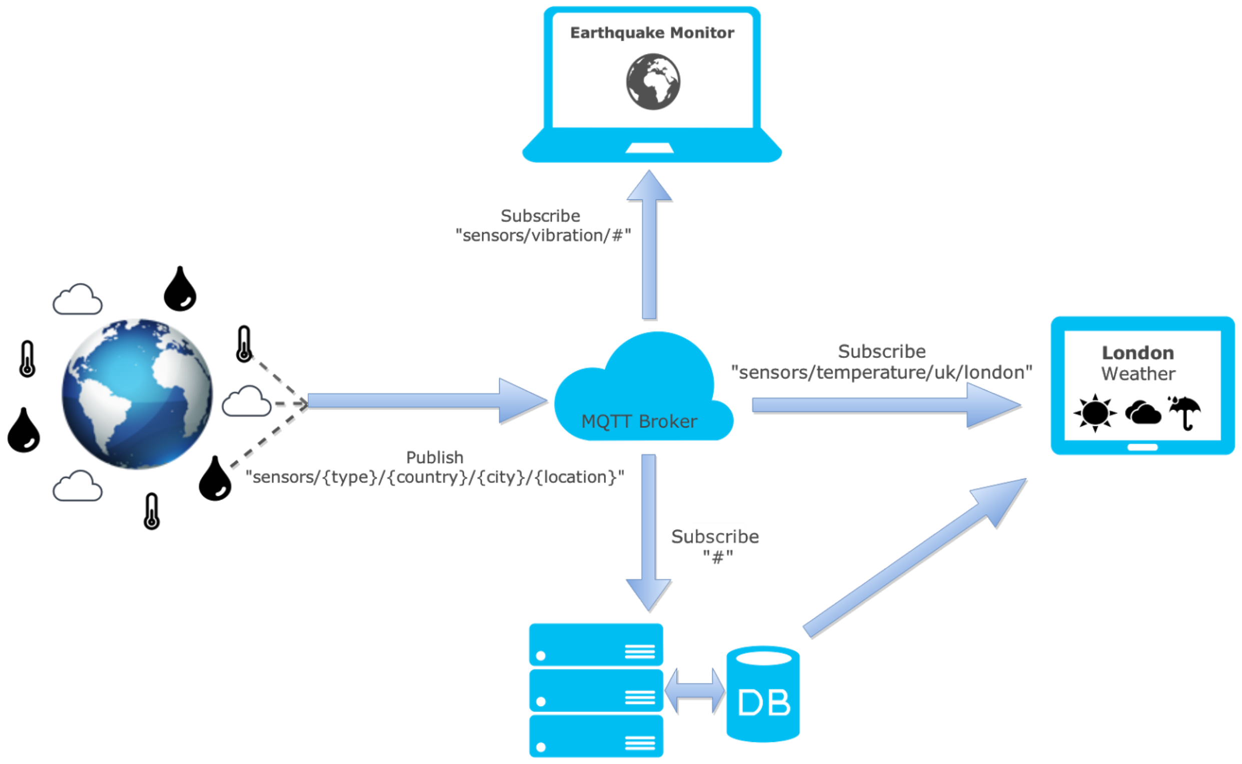
Dr. Vinton Cerf

MCI

MQTT and the publish-subscribe model

MQTT or Message Queue Telemetry Transport, is a standardized publish-subscribe "lightweight" protocol used for messaging on top of the TCP/IP stack. In the context of software, publish-subscribe is a messaging pattern where the senders of messages, called publishers, do not specify their messages to be sent to specific receivers, called subscribers. Instead they publish these messages into certain categories without knowledge of any subscribers. In a similar manner, the subscribers choose what categories of messages they are interested in, without specific knowledge of publishers themselves. This protocol is very useful for IoT communications / services, because new devices can both send and receive useful data to the network with minimal configuration, since the messages are origin and destination agnostic. The MQTT protocol was designed with connections in more remote locations where a "small code footprint" is desired, when bandwidth is limited.

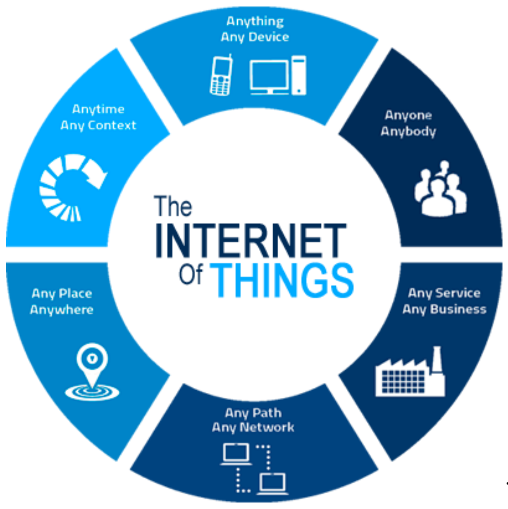
Since we designed the architecture of this project with future IoT compatibility in mind we decided on a publish-subscribe message pattern. Since ours is the first IoT project in the city


Earthquake Monitor
Subscribe "sensors/vibration/#"
MQTT Broker
Subscribe "sensors/temperature/uk/london"
London Weather
Publish "sensors/{type}/{country}/{city}/{location}"
Subscribe "#"
DB

there is not anything for our system to interact with yet. However when another system is implemented our system will be able to subscribe to relevant categorized data, and the other new systems will be able to read our systems publications. We chose the MQTT protocol specifically because the communications needed between sensors in a parking garage are very simple. By using a lighter messaging protocol it is possible to use smaller, minimally power consuming devices to communicate. In addition the dense materials and closed off shape of most parking structures can pose a problem when it comes to network connectivity. This makes limiting bandwidth usage more desirable to avoid delayed or even lost transmissions.

Internet of Things (IoT)

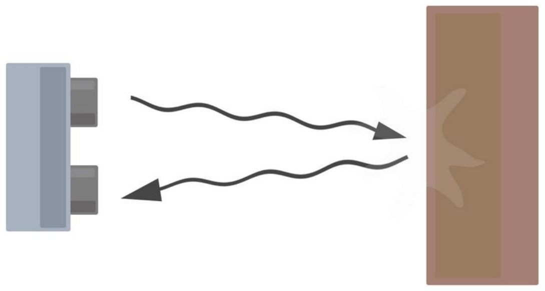
IoT is a broad concept without a standardized definition. It refers to the connection of vehicles, buildings, appliances, sensors, and other physical devices using web connectivity which will enable these devices to communicate with each other. With the recent rise of global web connectivity and the continuing fall of the price of devices the vision of a connected network of physical devices is becoming more and more possible. Still in the early stages, IoT has been theorized to be used to create things like smart homes (where appliances and utilities communicate to provide enhanced functionality) or even smart cities (where public infrastructure works together with other community services and users for a better quality of life). General connectivity is usually done from one machine directly to one machine, however IoT will offer enhanced connectivity that will allow “swarms” of devices to properly communicate. The devices on the IoT network collect useful data using existing sensor technologies and then autonomously flow the data between other devices.



The expansion into the IoT paradigm is expected to generate large amounts of data from many locations, and as such the need to store and process data effectively is one of the main concerns.

Ultrasonic Sensor

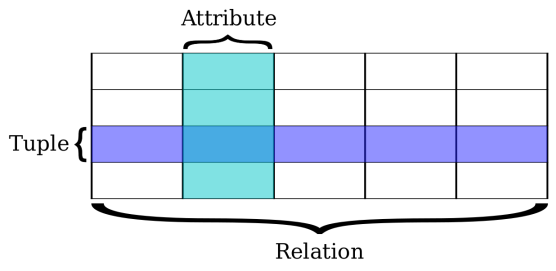
An ultrasonic sensor is a device that measures the distance to an object using sound waves. It measures distance by sending out a sound wave and listening for that same sound



wave to come back. Using the time it took for the sound wave to come back it is easy to determine the distance of the object the sound wave reflected from using the following formula: distance = (time taken * speed of sound) / 2. The speed of sound is known to travel 344 m/s (1129 ft./s), you can take the time for the sound to return and find the round-trip distance. Round-trip distance is double the distance to the object, as it includes distance to and from the object.

SQL Database

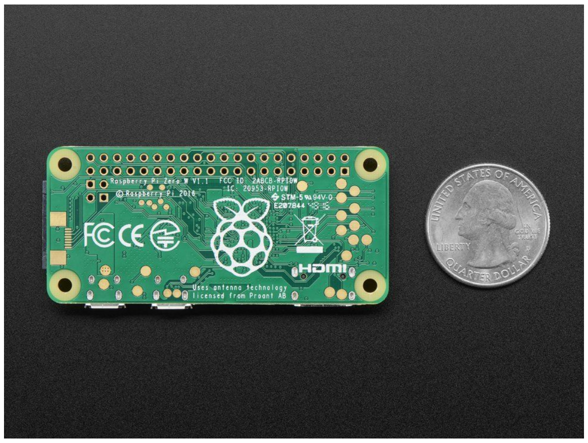
A database is a collection of data. Databases are often implemented as a relational model, which structures the data logically. Traditionally rows represent tuples which are an ordered list. This tuple holds all the data relating to one object. Columns hold the values for different attributes or characteristics. For example if an attribute was temperature then every value in a column would represent a temperature value. The entire table would hold related data, and certain attributes from one table are used to relate tuples between different tables. This structure is enables many logical and mathematical operations to be ran to process the data,



making it ideal for computers. Structured Query Language (SQL) is a specific computer language used to manage data in a database. SQL is used to insert, retrieve, update, and delete data from the database. SQL was one of the first commercial languages for relational databases, and remains one of the most widely used languages today.

Raspberry Pi Zero

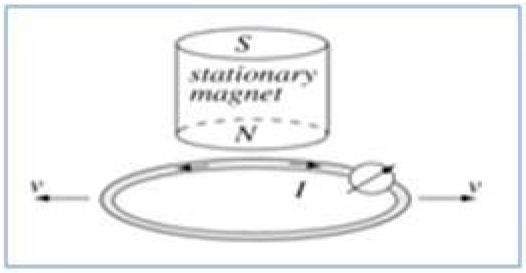
The processing power for our project comes from a line of small single-board computers, Raspberry Pi. This series of single-board computers are popular for their low price point and high utility. Originally developed for teaching basic computer science in schools and developing countries, it took off in popularity outside its original audience. Robotics and computer technology enthusiasts have led the Pi to over 5 million sales worldwide.



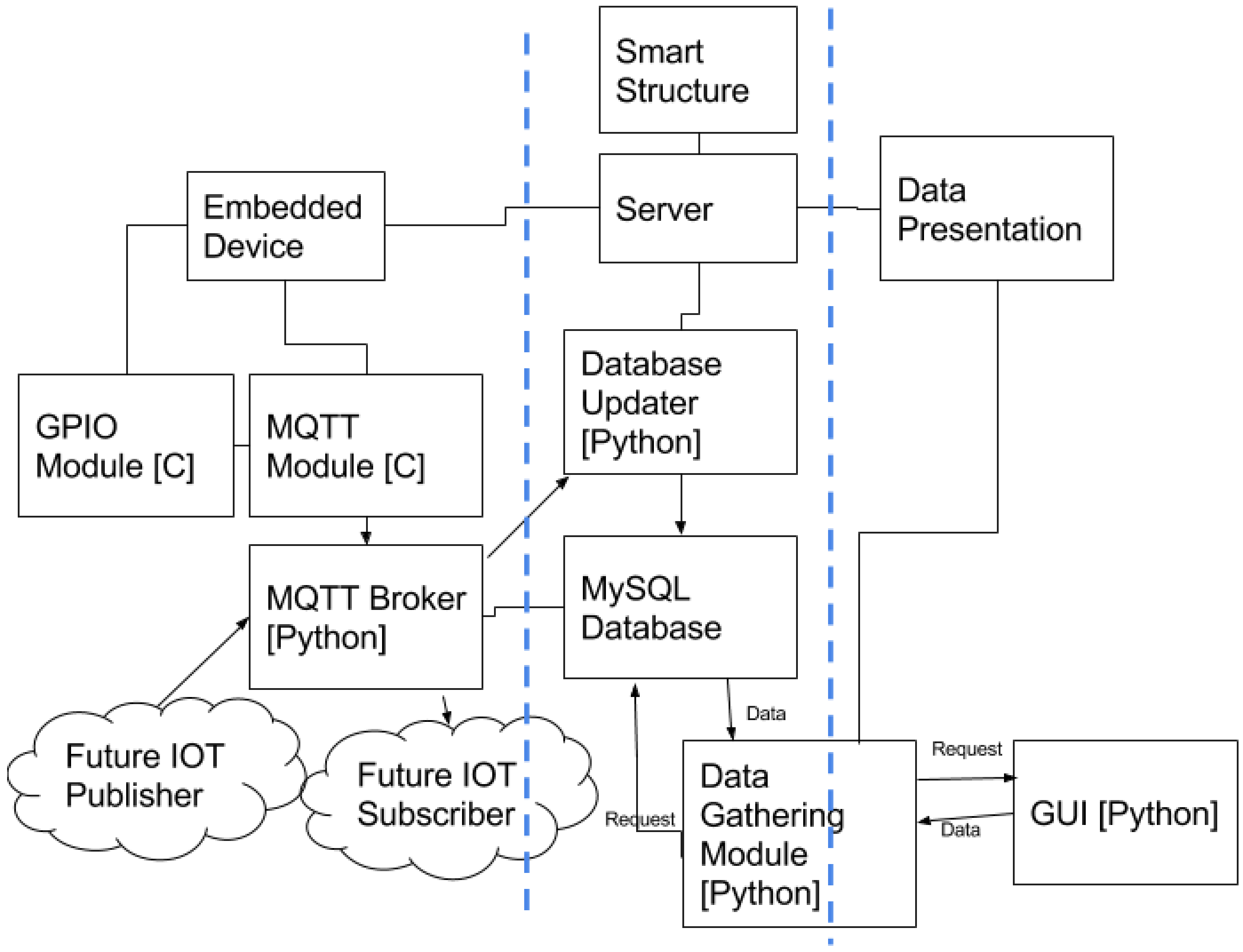
The specific model used in this case is the Raspberry Pi Zero W. The Zero is half the size of the main model with built in WIFI and Bluetooth, available at the time of this report for $20. This model was chosen for several reasons. One motivation in our project was keeping the price point low to increase ease of implementation. The Pi Zero W is also power efficient using on average under 200mA while WIFI is turned on. Having built in WIFI means this device has everything it needs out of the box to be able to connect to an IoT network.

Triple Axis Magnetometer

A magnetometer is a device that measures magnetism (i.e. changes in the direction, strength of a magnetic field at a particular location. A compass is an example of a very simple magnetometer, one that measures the direction of the magnetic field. The Triple Axis prefix means that the magnetometer can measure strength in all three directions (XYZ). Magnetometers work on the principle of Faraday's law, which states that an induced electromagnetic field in a closed coil is proportional to the rate of change of magnetic flux through a circuit. Put more simply, when a magnetic field changes it produces a current in a coil of wire. This current is then measured to determine the magnetic field.



The Magnetometer used in our project is the Freescale MAG3110. This sensor is a low-power digital 3-axis magnetometer. It interfaces using standard I2C serial interface which works easily with the Raspberry PI and most other embedded device platforms.



Chapter 3: Design

Top level down

The Smart Structure project was designed very modular in nature. This was very helpful when we had to switch sensors in the project. Switching meant acquiring a new device for the embedded Device and writing a new GPIO Module, but the rest of the architecture was able to be reused. This is a good example of how are system is able to be used as a framework with future IoT devices. The modules represented by clouds are where systems that do not yet exist would fit into our system.

- Sensor - This is the device used to count cars, in our case it is the ultrasonic sensor.

- GPIO Module - Written in C, this is the part of the code that interfaces with the device and interprets the data. This code uses the pins on the Raspberry Pi to communicate with the ultrasonic sensor.
- MQTT Module - Written in C, this is the part of the code that is used to interface with the MQTT Broker. It functions as a go between C and Python code, and formats the data collected by the GPIO module for the Broker.
- MQTT Broker - Written in Python, this module is able to publish (send) and subscribe (receive) relevant data. The broker is not bound to specific sources or destinations, but instead operates on categories of data. This means it does not differentiate between our MQTT module and a future implementation. It gets its data from the MQTT module currently.
- Database Updater - Written in Python, this module converts the data received from the MQTT Broker module into a format able to be accepted by the SQL database, and then makes the SQL calls to update the database.
- MySQL Database - This is the actual database that holds the data for the system. It gets its information from the SQL Broker and sends that data to the data gathering module.
- Data Gathering Module - Written in Python, this is the backend for the client facing part of the system. It requests data from the server using SQL calls and formats it for the GUI.
- GUI - Graphic User Interface, written in Python, this is a simple interface for displaying information to users in graphical form. Because it pulls from a modularized python backend somebody could write a different app, or an app for a different system, that would be able to access the same data.
- Future IoT Subscriber - This is where a yet to be implemented device that wanted to use our data would sit. One example would be an app onboard your car that tells you which parking garage to go to. By subscribing to our data the new IoT device would be able to tell which garages had more or less space and navigate accordingly.
- Future IoT Publisher - This is where a yet to be implemented device that possessed relevant data for our project would fit in. Since its data would be categorized as desirable by our broker would take in this data and make appropriate changes. One

example would be a device onboard your car that communicates directly with parking garages. It could send a message directly to the system to improve the accuracy of only using an ultrasonic sensor.

Error Correction

Our testing showed us two things; that our system is generally accurate, and that errors are inevitable. Because no system is free from error we have theorized several ways to recover after an undetected error occurs. These methods may be used in conjunction, or individually, depending on the specific constraints of the situation.

One way of keeping system accuracy is non-specifying the data. Users care about how the number of spots will affect their experience more than the actual number of spots. Without knowing how many there are total the number of spots becomes less helpful. Instead of displaying the actual number in the database it could be non-specified as a rough percent. (Ex. Multiples of 10%). Another way of non-specificity would be arbitrary fullness labels (i.e. Full, Almost Full, Semi-Full, Mostly Empty). These labels would mean that as long as the data was in general bounds the information would remain useful to the user. This method of non-specificity might not be able to be used, for example in instances where a more exact number is desired.

A second method of error correction is considering the logical bounds of the data. For example a 100 spot per floor garage will never have over 100 available spots per floor. Similarly it is very unlikely that the same garage would have 115 cars on a floor. (It is however possible to have a few extra cars relative to spots if people enter when it is at capacity) By putting bounds on the database errors we can prevent the extent of the error offset. While this only prevents errors when errors are already present it does keep them from snowballing out of control.

Testing

Testing the Magnetic Sensor

Our initial design did not use an ultrasonic sensor at all. Instead it used a Triple-Axis Magnetometer. The reason for this is we wanted to avoid as many false reading as possible, especially false positives. Since it is possible for a person, bike, or other non-space using object to move by a distance sensor and trigger a response we went with the Magnetometer. Initial tests looked promising, the sensor had a quick update speed and was able to accurately detect things such as the magnet inside our computer. After implementing the majority of the system we took the unit to the downtown parking garage on Marsh Street to do a live test. We stood on the side of one of the ramps of the garage and aimed the sensor, observing a live stream of the magnetic data fed to our computer. When the sensor was stationary we received fairly consistent data. However the sensor was very susceptible to movement, if the sensor moved position the magnetic data showed a large change. We theorized this would not cause a problem in the actual implementation since the location of the sensor would be static. It's likely these changes were caused by the large amount of metal materials used in the construction of the garage itself. In addition all of the electrical wiring goes through the walls of the structure. Just as changes in magnetic field induce a current, a current will also cause changes in the local magnetic field.

As the cars drove by we kept observing the stream of readings from the magnetometer. As time went by we realized the cars had very little effect on the magnetic field detected by the sensor. So little that the effect the cars had was comparable to the amount of static noise in the ambient environment. We realized that a magnetometer would not be able to accurately detect a car driving by, so we would have to use another kind of sensor to detect a passing vehicle.

Testing the Ultrasonic Sensor

Our next session of testing came after we completed work on the ultrasonic sensor implementation. Luckily we designed our system in a modular way so the back end used by the magnetometer could still be used with the ultrasonic sensor. Initial testing of the ultrasonic sensor showed that it would definitely be capable of detecting a car passing. The sensor does not discriminate based on the type of object however, so we would need to put in an error correction algorithm to stop the sensor from marking a person walking by as a car. Because this process involves changing the code many times to precisely calibrate we conducted this testing on-site so we could observe our code changes right away. During this testing phase we encountered several obstacles.

Error Detection

The first was due to the non-real time nature of the Operating System (OS) on the Raspberry Pi. While we wanted our sensor to be available and scanning constantly the OS would randomly switch resources away from our program. These switches are extremely short on the human awareness time scale, but could cause the sensor to return a non-expected value. To fix this issue we coded in some buffer room, if a very low amount of unexpected results are returned we know that is the OS and we can discard those values without impacting the accuracy of the sensor.

Another thing that required calibration was how to determine when one car passed and another left. Since there might be a situation where a car would stop in front of the sensor (if the line of cars was moving slowly). To account for this we implemented a flag variable, the flag gets set when an object is detected, and a car is not counted until there are consecutive readings affirming that the car has left. Because of the fast scan rate of our sensor cars driving close by do not pose much of a problem. This is because the time between cars is very large from a programmatic timescale. Even a fraction of a second if enough time for many readings to be taken.

The final calibration made was to stop false positives, where a person or other object passed in front of the sensor. We only wanted to count cars, not a person walking by. To account for this likely possibility we trigger the flag for a car after a higher number of reads is detected, this way when a person walks by briefly, there will not be enough reads to count as a car.

Conclusion:

By situating an ultrasonic sensor at every ramp in the parking structure we are able to tell when a car leaves or exits each floor. Each sensor sends its data via a lightweight MQTT protocol to a central database, which aggregates all of the data. Using the database we can then say how many cars are on each floor. This data can be accessed via our Python GUI or through a custom python implementation, or directly from our MQTT broker. The system is reasonably accurate, with built in error detection (for example when a person instead of a car) walks in front of the sensor, as well as a variety of error correction models that can be applied, including bounded database, manual calibration. Over a long period of time our system is able to improve its accuracy through skew analysis. This means theoretically the system gets more accurate the longer it runs.

The requirements for an implementation of our system is a WiFi network for the sensors to connect to, so that they can communicate. A WiFi or other network is vital to any IoT implementation. In regards to future development of this project there are several ways to progress. One promising method that unfortunately we did not have time to test is program a false reading drift. (For our system to work with errors the number of false positives and negatives need to be roughly the same, that way they will balance each other out. This is fairly unlikely to happen naturally. Over a long period of time there will be more false positives than negatives, or vice versa, depending on the conditions. This ratio can be measured in terms of the number of readings or over a period of time. For example every day on average every 1000 readings there is one more car in the database than there should be. In the program after 1000 readings a false negative would purposely be introduced. This would offset the skew factor and keep the database consistent. To calculate the skew two manual surveys of the lot would be required. In this survey somebody would need to manually count the number of cars on each floor. The first survey is necessary for our project to be implemented, it is the starting value of the database. The second survey would be taken after a long period of time. The difference from reality and the database could then be used in conjunction with the following formula used to calculate the skew.

Skew Factor = (Number of Cars In Database - Number of Cars actually in Lot) / Period Of Time.

If the Skew Factor is positive then there are too many false positives, and Skew Factor number of cars need to be subtracted from the database every period of time. The opposite holds true if the skew factor is calculated to be negative. Ideally this database accounting would be spread out over the period of time. So if the Skew is +12/yr then one car would be subtracted each month. Although this skew would not be able to be accounted for until a longer period of time had passed, once an accurate skew factor was calculated as long as conditions did not change the error rate of the system would be almost completely eliminated!

The skew method touches on something that is important to the system, manual surveying for calibration. The system is able to be manually changed (by sending crafted MQTT messages or changing the database directly). So if a discrepancy is noticed between the lot and database it can be fixed manually.

From a hardware standpoint a second sensor would lead to vastly increased accuracy. Taking measurements on two axis would reduce false readings. Another way multiple sensor could be used is an offset of two parallel sensors. These pair of sensors could be calibrated to the length of a car, so that only an object with the appropriate length would trigger the system. From a software standpoint the Error Correction methods outlined above could be implemented to reduce the impact of inevitable errors.

The Smart Structure design and proof of concept were successfully completed. Unfortunately the City of San Luis Obispo is not interested in pursuing implementation due to time and budget constraints. The system works with an acceptable accuracy to be used in a real-world implementation. Each unit costs approximately $30 for processor and sensor.

Appendix:

A. ABET

B. Links to source code

Embedded code: **https://github.com/ecolsen7/smart-structure-embedded**

Server-side code:

https://pastebin.com/AJM4iBXA

Display-side code:

https://pastebin.com/FKLzhXHT

C . MQTT:

- "MQTT 3.1.1 specification". OASIS. December 10, 2015. Retrieved April 25, 2017.
- Erl, Thomas (2005). *Service Oriented Architecture: Concepts, Technology, and Design*. Indiana: Pearson Education. p. 171. ISBN 0-13-185858-0.
- https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

D. TCP/IP

http://searchnetworking.techtarget.com/definition/TCP-IP

http://www.billslater.com/tcpip.pdf

E. IoT

"An Introduction to the Internet of Things (IoT)" (PDF). *Cisco.com*. San Francisco, California: Lopez Research. November 2013. Retrieved 23 October 2016.

Ultrasonic Sensor

http://education.rec.ri.cmu.edu/content/electronics/boe/ultrasonic_sensor/1.html

F. Raspberry Pi Zero W

http://raspi.tv/2017/how-much-power-does-pi-zero-w-use

https://www.raspberrypi.org/blog/raspberry-pi-zero-w-joins-family/

G. Magnetometer

https://www.engineersgarage.com/articles/magnetometer