Smart Park:

A Senior Design Project For the Wireless Communications

Capstone

By:

Patrick Olivero

Chris Cherin

Joluis Castellanos

Table of Contents:

1. Motivation

A parking deck is a symbol of efficient use of limited space to provide the most parking spots in a given area. As you add more and more spots, however, it becomes increasingly difficult for a driver to find the closest spot. This heightened difficulty necessitates driving in circles around the parking deck and looking for a spot all while decreasing our focus on safely driving in order to race in hopes of beating someone else. In today's world, the word "smart" is used to describe everything from cellphones to homes which all feature more communication and interactivity between the user and the product. If a home can be "smart" why can't a parking garage? This is the question our group has posed and it is a question we feel we have solved with a system we call "Smart Park".

2. Existing Solutions

Our group is not alone in our quest to create a more interactive parking experience and there exist systems already out there that attempt to tackle in many ways, each with pros and cons. One such system would be the robotic garage. The concept of the robot garage is to fully automate the placement of a car in an efficient manner without the aid of the driver. All that is required of the driver is that they drive up to the elevator or robotic stand at the entrance to the facility and then exit the vehicle. Sensors are used to confirm that there are no passengers left in the vehicle and then machines move the vehicle into a storage area.

Fig. 1. Computer generated model of a robotic parking structure (left) and a real world example of one such system (right)

This system has its merits, including being able to use less space to store vehicles, as well as isolating vehicles from potential break-ins and theft. That isolation can also be a major issue. What if once out and the vehicle is stowed away, the driver remembers they may have left something in the car itself? Would the garage waste energy getting the car back for the driver to retrieve the item in question then return it back once more to its holding area? Also these robotic garages require large, complicated, and expensive machines in order to operate. This point is made clear when comparing the cost per spot of a conventional garage (about $15,000) with that of a spot in a typical automated garage (around $30,000). While machines like the ones used by the company, Robotic Parking System, use components rated with L10 lifetimes of 40,000 hours, this number can easily decrease with heavy usage as well as simply fail altogether as was the case several times with the nation's first automated garage in Hoboken, NJ. There are many documented instances from 2002 thru 2007 of cars being trapped when the system could not retrieve them as well as incidents where vehicles were actually dropped by the machinery. Overall this system, while a promising idea, is costly both in the initial capital required to build the garage and in the upkeep of having staff on hand at all times to maintain the machines that are critical to the workings of the system itself.

Another example of a similar system would be systems that use embedded sensors or wires to detect the status of a parking spot. These systems look at their embedded nature as a means of integration into the parking garage while avoiding exposure to the outdoor environment. Most of these systems rely on screens that show the number of spots left on a given floor or a light board where each light represents the status of a spot, red for closed green for available. While the initial data gathering is done similar to the way we chose to, the display of this data to drivers is far different. Signs that display the number of open spots per floor are far too general and light boards showing all available spots too complicated for a driver to navigate in the moments they have to look at it before they begin to hold up other drivers waiting to enter the same parking garage. They leave much of the stress and guesswork of finding a spot

still in place since the driver either has to stop to frantically look at a big light board to determine which spot they want with no guarantees that when they get to said spot that it is still available. The same inconvenience can be said of the displays that show the number of spots on each floor. While less cluttering or confusing as the giant light board idea, it too has its limitations. Drivers still aren't given specifics and so still have to drive around finding a spot of their choice. Our system hopes to eliminate this issue.

Fig. 2. Example of display showing number of spots available on a specific parking garage floor

3. Goals and System Requirements

The system our team has created aims to improve on the communication of data to the driver in an efficient and helpful manner as well as manage the costs of upgrading to a "smart" parking deck. A main issue with many of the options for creating a more convenient solution for parking management is the upfront installation cost. As stated before the price per parking spot of a conventional parking deck is around $15,000. To create a compelling system, one must minimize the increase to this ratio of cost per spot. To this end after careful consideration of both the level of complexity, timeframe, and invasiveness of the kind of system we wished to make, a general consensus was reached. Our system

would start by using a device equipped with an ultrasonic detector to detect the presence of a vehicle in a parking space. We then decided that we wanted to implement a Wireless Sensor Network (WSN), forgoing using wired connections in order to reduce the material cost of the system as well as to test the capabilities of a WSN within a parking deck structure. We also knew that in order to differentiate our system design from others already out there, that we would need to take a new approach to communication with patrons of the parking deck to better inform them about the locations of potential available spots that would be of interest to the driver. To achieve this we would need to design and write our own program to not only interface with our WSN but that would also allow for the level of clear and precise communication between the parking structure and the driver. With the main goals of our system in place we divided the work into three parts: Hardware, Networking, and Software and began to create.

The system that our group has developed is relatively simple yet coupled with our software creates a whole new level of interactivity between drivers and the parking deck, effectively creating the "smart" parking experience. Our system has three steps to it: Detection, Transmission, and Suggestion. The first step is detection, where our sensor, using ultrasonics, measures the distance between the ceiling, where it is attached to, and the floor. This data is fed to our microcontroller, which takes the output signal of the ultrasonic detector and translates it into actual distance measurements. It then takes this measurement and compares them to the known distance from ceiling to floor to determine if a vehicle is occupying the spot. If the distance has decreased the sensor then waits and checks the distance measurement again to see if the spot is still taken or if instead an object, such as a person, simply was walking under it. These sensors would be grouped together in clusters with sensors transmitting their ID and spot state to clusterheads. These clusterheads would aggregate data and transmit it to a main "hub" which would collect the data and use it to suggest the nearest available spot to drivers coming in. Suggestion is a key word here. When our group polled with the following question:

> “For my senior design project, my group and I are planning on creating a system for private parking decks that allows for the detection of whether a parking spot is available or full. This would allow drivers to be more informed as to where there are open spots and whether or not the parking deck is even full or not thus making parking more swift, less stressful, and more efficient overall.
>
> If you were paying to park in a garage outfitted with the system described above, would you be willing to be told where to park according to parking spot availability regardless of location?”

Of the 149 people who responded 80% said yes while 20% said no. In order to account for the 20% we decided to allow our system to put parking spots that were detected into a temporary array where after a few minutes the state of the suggested spot would be checked to see if the spot was indeed taken or not. If the spot was then it is deleted from the temporary array, however if the spot was not taken, then the spot is simply returned to the array of spots currently open. Implementing this has allowed our system to be much more lenient while simultaneously maintaining the precise spot locating we set as a requirement for our system.

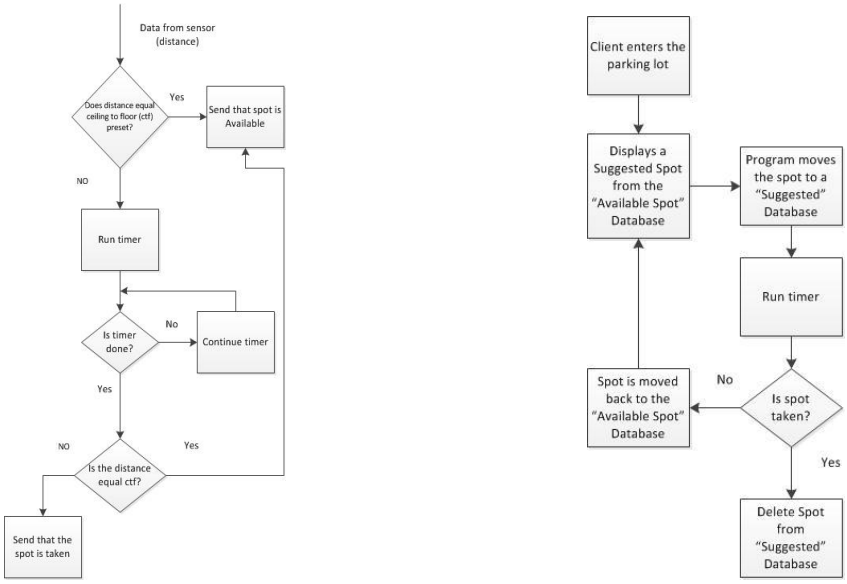

Fig. 3. Block diagrams for the programming of the space sensor (left) and the suggestion program (right)

4. Hardware

Creating the sensor itself for our system was, as it was with every part of this project, an iterative process. When first brainstorming the ideas that would lead to our system, we knew a microcontroller would be a necessity. It would give our sensor the processing power needed to control the ultrasonic detectors as well as power the XBee wireless modules. Being that this was going to be a wireless sensor we also wanted a microcontroller that was optimized for said tasks and would be able to handle them in an efficient manner. After more research, however, a compatibility issue was found and our focus had to shift to another more compatible version. The microcontroller we ended up with was the Arduino Uno.

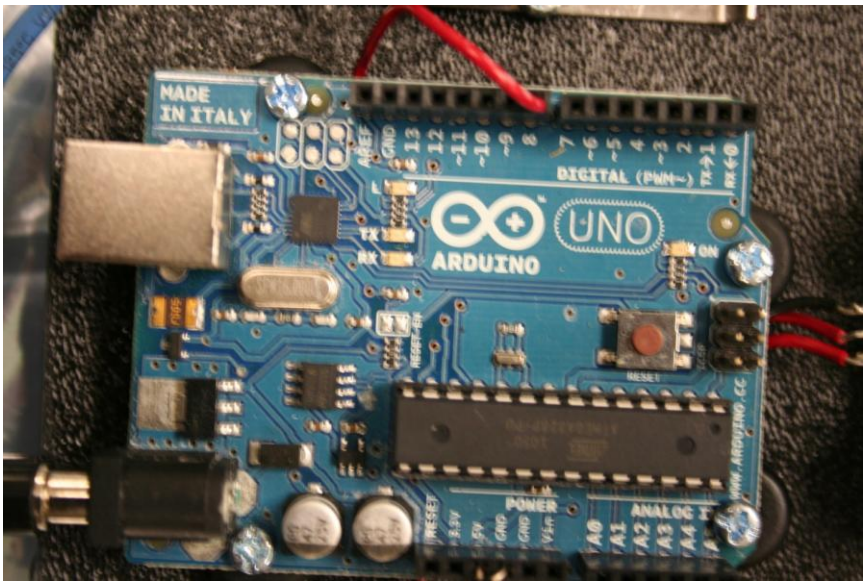

Fig. 4. Image of Arduino Uno microcontroller

Small enough in size yet still able to handle the calculations and logic we needed with ease, the Uno proved to be a simple and reliable piece of hardware to code for. The tradeoff came in the form of larger size than the first considered and wireless optimized Arduino Fio as well as larger power consumption at

234mW as the Uno was designed more for being plugged into a wall than into a battery. More information on battery life can be found in the testing section of this report.

Along with our chose of microcontroller, a choice of which ultrasonic detect to use needed to be made. Many of the ultrasonic detectors came in the form of having five connectors (5V, TRIG, SIG, GND, and a connector that was not usually supposed to be connected) and reliability as well as energy use often were issues. The decision was made to go with the Parallax PING))) sensor. This ultrasonic detector used a more “user-friendly” three-prong connector layout (5V, SIG, GND) and had power consumption that made it stand out from the rest at 100mW.

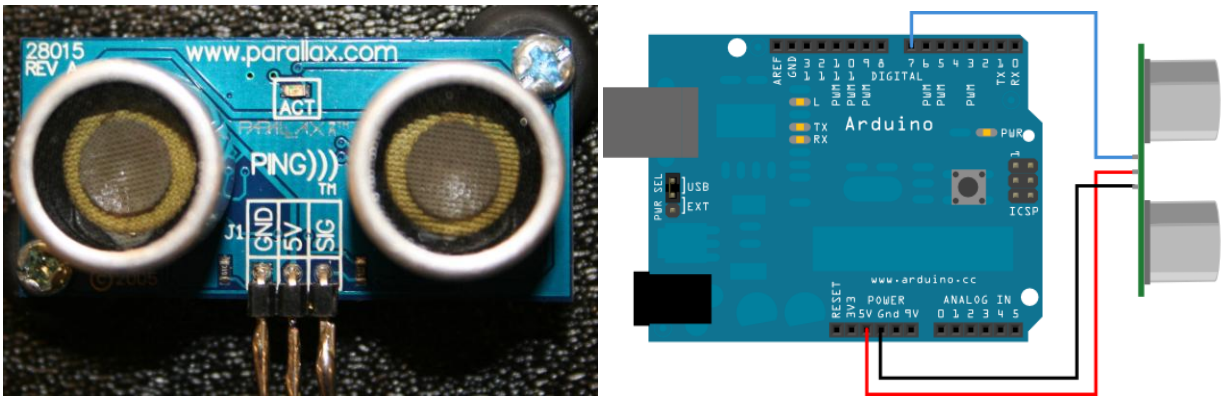

Fig. 5. Parallax PING))) ultrasonic detector (left) and a basic circuit schematic for interfacing with Arduino

Controlling the PING sensor was done using the Uno microcontroller, which would trigger the sensor via a “LOW, HIGH, LOW” pulse to the SIG connector. The sensor would then send out three short pulses of sound at a frequency of 40kHz and time how long the echo took to return in microseconds. The sound emitted has a spread of 15^{o} from center. If the average parking deck is approximately 114 inches this works to a circular detection area of diameter 61.1”. The microcontroller then receives that data from the SIG connector and changes that time into a distance. It does this by dividing the time first by 74 because on average it takes sound waves 74 microseconds to travel an inch. This gives the total travel distance from the PING sensor to the object and back so it is necessary to divide this value in half to get

the distance to the object. The Uno then makes a quick logic comparison between the newly found distance and the known distance from floor to ceiling to determine if the spot is taken and changes it state as outlined earlier.

From the beginning we knew we would need to build a wireless sensor network (WSN) in order to connect all these sensors. The Xbee modules were an obvious choice for our group considering they were built around the Zigbee standard and thus has its own protocol already in place. To go along with this was an Xbee shield designed to create the circuitry necessary for the Arduino and XBee to communicate. The Xbee itself was designed to draw 35mA while transmitting at 3.3V. More information on the Xbees and the Zigbee standard can be found in our networking section of this report.

Fig. 6. Xbee Series 2.5 module (right) and Xbee shield (left)

When all of these pieces were brought together our sensor was created. All that was needed was a power source and our “hub”. With a sensor ping cost of 115.5mW +100mW+234mW = 449.5mW. The Duracell 9V battery provides on average around 500mWh and for the purposes of making a proof of

concept worked out perfectly fine. We were even able to use the regular wall adapter plug built into the Uno so that we could use the built in voltage regulator circuit.

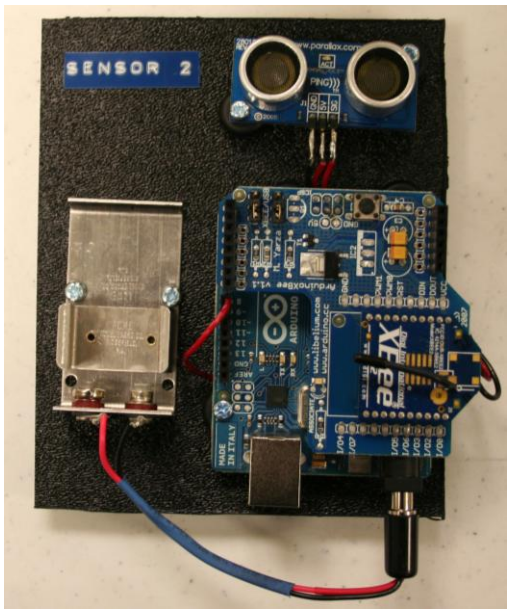

Fig. 7. A completed ceiling sensor for our network

Our “hub” needed to be nothing more than another Xbee, which acted as our receiver connected to a computer, which aggregated all the data being sent from the sensors.

Fig. 8. The receiver for our “hub”

5. Networking

In terms of networking, the “Smart Park” system implements a unique and highly efficient networking scheme in order to avoid the seemingly obvious troubles within a concrete parking garage structure. In fact, the first major hurdle with the design concept was the issue of transmitter to receiver communication (router to central HUB). In order to bypass the need for long range communication and avoid the various complications of environmental blockades, we examined the United States Military’s current use of “Wireless Mesh Networks”.

A Wireless Mesh Network is a network composed of various radio nodes which have the ability to communicate from one node to another in a rather quick and simple fashion. The information will ‘bounce’ from router to router until the data is finally received by the primary coordinator which then delivers the data to the user. In theory, a full implemented multi story parking garage system would include approximately one or two coordinators on each garage floor. The coordinator would then gather all the data from the parking space routers and would relay the total data to the floor below. This topology will continue to snake its way down until it reaches the central processing HUB in the lobby. For demonstrative purposes our final hardware setup is composed of five routers which send the parking status data directly to a main coordinator. Smart Park’s final topology is called a “Star” type topology because the coordinator radio sits at the center of the network while being circled by surrounding router node.

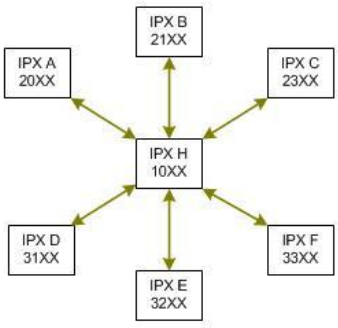

Fig. 9. Visualization of Network Design

The FCC certified mode of enabling such a network is called IEEE 802.15.4, which is also known as “ZigBee”. IEEE 802.15.4 radios operate in unlicensed bands including 2.4 GHz, 900 MHz and 868 MHz. The most compatible and documented modern hardware utilizing this ZigBee standard is Digi’s line of ‘Xbee’ communication modules. Operating at 2.4 GHz operating frequency with an output of 2mW, the Xbees have to ability to communicate within a range of about 120 meters. Communications between Xbee modules vary in two primary command sets, and programming of the modules allows for 128-bit encryption. For our system “Xbee ZNet 2.5 Antenna (XB24-B)” modules are used. Each device includes a small omnidirectional antenna which allows for physical space minimization and the ability to avoid potentially bulky antennas. Omnidirectional meaning that when the wire is fixed upwards perpendicularly the max transmission distance will be the same in all directions.

Fig. 10. Image of XBee Series 2.5 module

In order to fine tune the module for the system’s particular needs, a program called “X-CTU” is used to manually program individual features of the Xbee’s firmware. Programming involves the use of a single USB board which includes the proper pins for easy hardware reading; this USB board avoids breakout boards and messing wiring techniques for managing the device’s firmware. Xbee firmware includes seven primary categories of options including: Networking, RF interlacing, Serial interlacing, sleep modes, IO settings, diagnostic commands, and AT command settings. In this particular system the most important elements of the Xbee’s firmware is the PAN address (a universal address for every

device in the entire network which creates an appropriate channel for node communication), the 16-bit/64-bit destination address (how the parking spot detector devices will know to specifically communicate with its required end device), and baud rate (the rate of pulses per second sent or received by the module). Each Xbee is read and modified by properly setting the COM port assigned to the module by a PC and by choosing the appropriate "baud rate" of the Xbee (which is almost always 9600 pulses per second by default). For our design, Xbees used for reading parking spaces will be synced with Arduino microcontrollers, so this requires the baud rate for every Xbee (including the coordinator) to be set at 57600 pulses per second. When X-CTU software verifies that the module can be read, the " XB24-B - ZNET 2.5 Router/End Device AT" firmware can be uploaded to the device. This gives one the ability to set the device to specifically send "AT commands" to the specified coordinator HUB Xbee. The coordinator will be programmed to explicitly receive AT commands from these surrounding Xbee router nodes. AT commands refer to an interactive setup descending from the classic Hayes command set. In AT mode the device can only transmit AT data packets to one other device, as each device will share each other's destination addresses, enabling directed one to one communication.

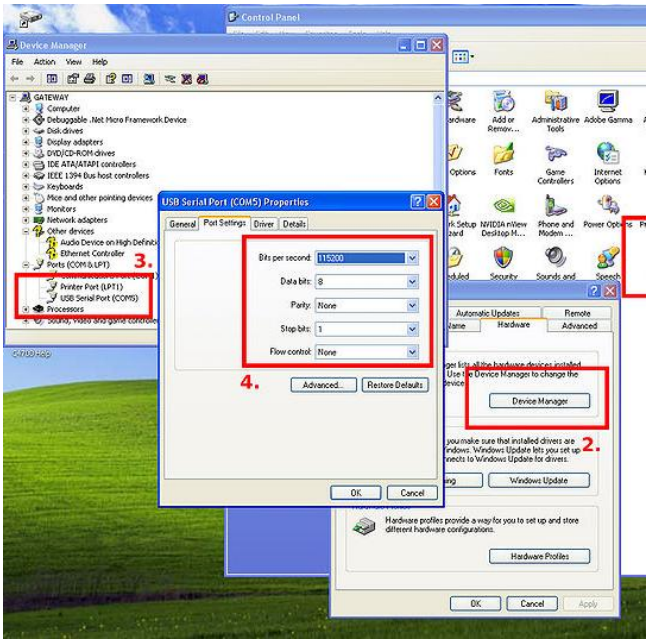

Fig. 11. Screencap of setting up COM port

In the “Smart Park” system, Xbees programmed as “Routers” serve to specifically create a wireless COM channel which transmits spot availability data directly to its programmed “Coordinator” destination address. Each Xbee module is hardwired with a predetermined Address LOW and Address HIGH, and these addresses can be discovered by either reading the firmware using X-CTU, or by manually reading the printed address on the backside of the module. Because the coordinator in our star configured system is required to read from ALL router nodes in the network, it must be programmed in another command set called “API mode”. ‘API’ stands for application programming interface and is a set of standard interfaces which allows the device to read from more than one module. The coordinator Xbee programmed in API mode (XB24-B - ZNET 2.5 Coordinator API) does not include destination addresses for it is designed to receive data from all devices with the same PAN ID in the network. Each router node however MUST be required to be programmed with the coordinators address HIGH and LOW.

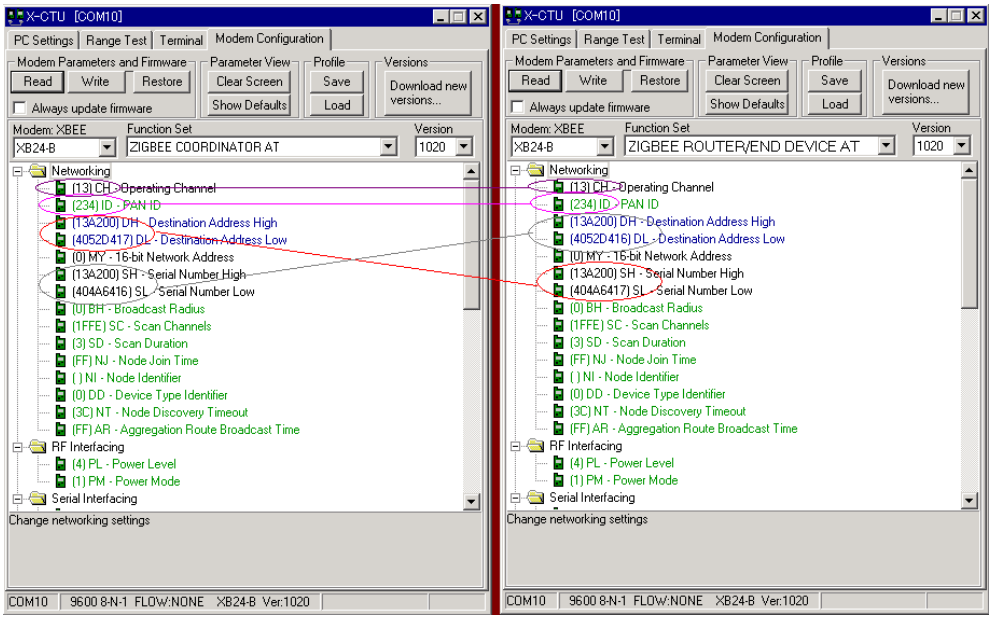

Fig. 12. Screencap of configuration screen for coordinator(left) and router/end device (right)

So overall, Xbees connected to adruino microcontrollers via specialized hardware interface shields (Routers/End Devices) communicate in AT command sets sent directly to an API configured coordinator HUB, which is physically connected to a PC via USB. The API configured coordinator knows to receive data from any router node which has its same programmed PAN ID and that data is then read from the appropriate COM port. For testing purposes a program called Tera-Term was used to verify received data.

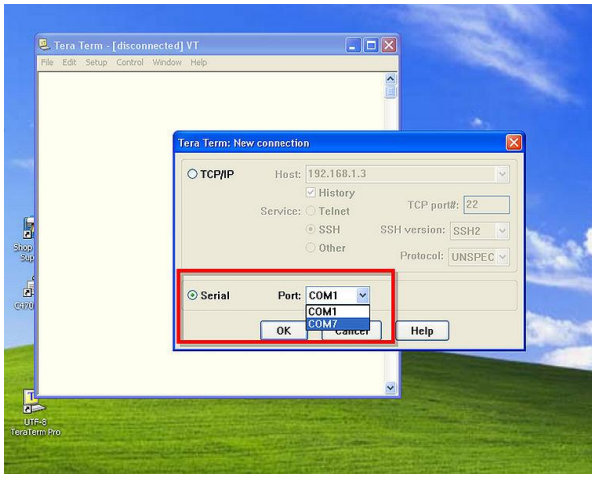

Fig. 13. Screencap of setting up Tera-Term

6. Software

Software Design

Our software has to be able to interpret the data stream incoming from the CommPort. The incoming data may have some other information and errors that we need to process and ignore because we are only interested in the information about each spot in the parking lot. After the information is processed the useful data will be store in the backend which will contain all the information about the parking spots. This backend is updated continuously as soon as useful data is in the CommPort. The program reads the data from the CommPort as long as it is ready. It has a listener waiting for data to be

ready and that triggers the software to read the datastream and update the array if necessary. For the matter of testing we have the basic command prompt design and two interfaces which use GUI.

Programming Language

Java Programming Language is the one selected to handle the information incoming from the CommPort and to display the results.

Why Java Language?

-Java Language is simple

-It is object-orientated which gives us a lot of flexibility in how to implement and solve the problem.

-It is also secure, the information can be private and protected from third parties.

-Java is reliable. A lot of testing can be performed. Debugging is easy and this allows us to attack the main source of any problem.

-Platform-Independent, this is one of the best advantages that java brings. We can basically “Write Once, Run Anywhere”.

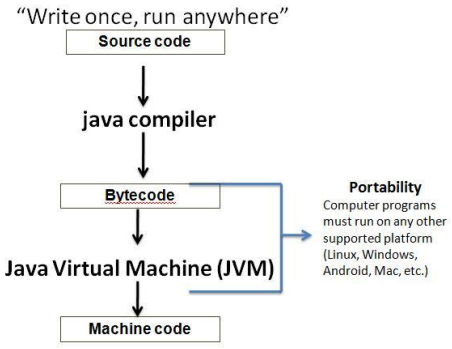

Fig. 14. Steps to convert source code into machine code

The source code in java can be written in any type of Operating System and be compiled in another one besides the first one used. The ‘java source’ is converted into ‘byte code’ through the compiler. Then this bytecode is translated into Machine code using the Java Virtual Machine (JVM). This is the key step because the JVM is independent from the Operating System where it is. Instead of using the CPU to obtain the Machine code the JVM generates the machine code for a specific Operating system using as a source the same bytecode that would be generated in other Operating Systems.

Software Architecture

Since engineering and life changes daily we would like to look for an architecture that allows us to modify our software without doing too many changes in the user interface. This project has commercial purposes, which means that any update in the software should not bother the customer by installing a new version. That is why the user interface most of the time should be the same and all updates and changes should happen in the back.

Model-View-Controller (MVC) Architecture

The MVC architecture is independent and allows our program to be flexible. Since all parts are independent they can be updated easily and errors can be found in a fast way. We basically organize the software in three parts and each one has its own objective and functionality.

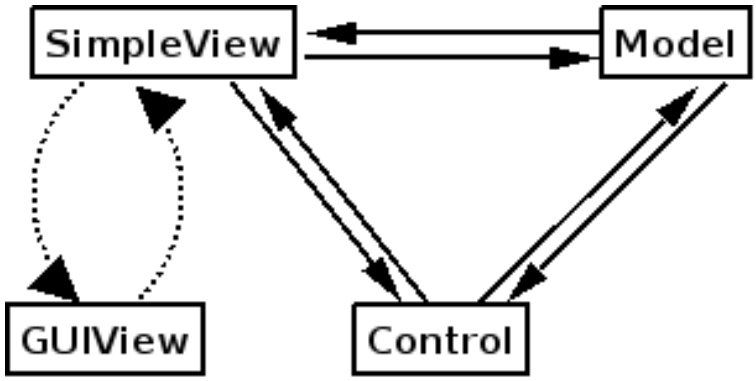

Fig. 15. MVC architecture graphical representation

Model: The representation and management of the system's data

View: The user's interface to the system

Control: The algorithmic logic, decision-making and data manipulation processes

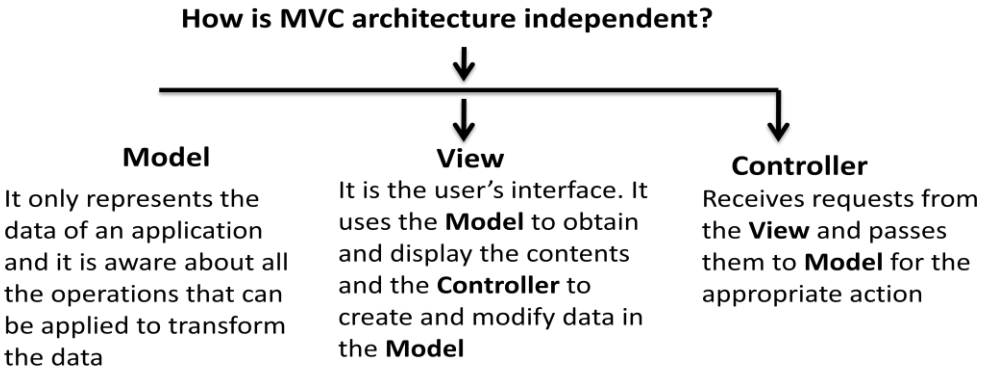

Fig. 16. Independency of the MVC

The MVC software architecture gives us extensibility, robustness and code irredundancy. The view will only access the backend to read and write data. All other functions have to go through the Control Part which will be the bridge for almost everything between the View and Backend part.

In our case Fig#4 represents our final design. As we may notice the backend contains the Data, which implements the data structure where the spots are going to be store. The class Spot will contain all the information about each spot, such as name, availability, location, etc. In the Control part we have the ControlTodo class which will handle all the methods and connect the View and backend using the ComControl class which reads the information from the CommPort. The util package is a common package that can be used by any component to read and display information in the console. The View Part contains all the classes that conform the GUIViews and the test class is the simpleView.

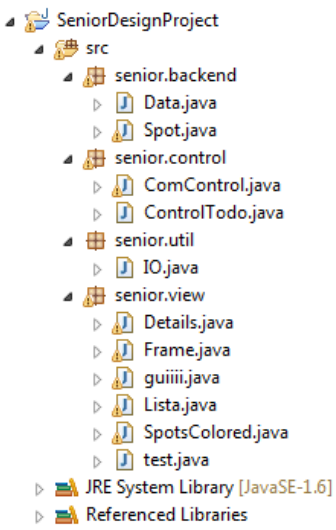

Fig. 17. Our MVC design

Data Stream Processing

We first need to create a virtual ComPort from where to read the information coming from the Xbees. After the port is initialized and open we are able to read any incoming information in bytes.

As we agreed every sensor located above each parking spot will send the information about the availability of its spot. The format chosen is “SIDXXX Y”, where ‘XXX’ represents the spot number and if Y=1 the spot is available otherwise if Y=0 the spot is currently taken. As the information reaches the port our program will read it and remove it from the buffer of the CommPort.

Other Data Incoming (Garbage)

Since there is noise and other factors in the environment that will affect the data stream incoming we have create methods to eliminate the extra information. Our program will grab each byte

of information and only when the exact sequence “SID XXX Y” is found it will update the specific spot status.

Incomplete Data Stream

When there is information in the CommPort ready to be read the programs grabs it and processes it. Often the data stream may cut the expected format “SIDXXX Y” in two pieces since there are a lot of other bytes (garbage) also in the buffer. In order to avoid missing this format at this time our software will be sensing the data stream and once it finds the letter “S” a counter will begin in order to hold the first part of the format. Then when the second data stream is ready in the buffer the program checks if the counter was initialized before and it appends at the beginning of the new data stream the last few bytes from the first one. This way we will not lose the any possible right format incoming if it is shop at the buffer. In order to update the spots status the format has to be exactly SIDXXX Y.

Data Stream Analyzer

The first approach implemented was to read byte per byte from the buffer and pass each one to the method that is looking for the right incoming format. This approach failed since when we have more than one sensor the amount of information increases and the programs cannot catch up with the actual status of the spots.

As a second and more effective approach we have read all the information from the port and store it into an array of bytes. So now we have access to more bytes in a faster way and we processes it quickly since the array allows us to access to each byte at O(1) times in terms of Big O.

Multi-Threads

Our software is basically two programs in one that share the same information about the spots, stored either in an array or another type of data structure.

Both threads are totally independent except when one of them stops which causes the other one to stop as well. One thread will be constantly listening the CommPort waiting for information to arrive. When the information is ready it triggers this thread to process the incoming data stream. Then if it is the case, the data structure holding the spots information is updated. It may be updated with the same status that it was before. It would be a waste of time to just update the information in those that change the status because that will introduce an-if statement. It is faster just to replace the status of the spot with the incoming value without looking at the previous value. The user never interacts with this thread. It runs in the back and it stops when the second thread stops, which is the user interface.

The second thread will contain the friendly information about the spots that the user will see. When the user enters the garage the second thread generates a GUI interface in a screen which tells the user the nearest or all the spots available in the building. Different set ups and version of the GUIView can be used without changing the Control and Model part. This is the great advantage that the MVC architecture offers to us.

Model, View and Control

SimpleView

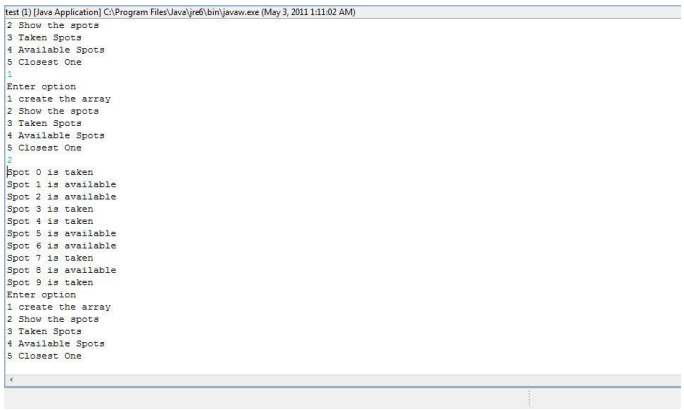

Fig. 18. SimpleView, command prompt

We first implemented a command prompt view, Fig. 18, where we created randomly the data structure containing the spots statuses. The spots were generated randomly and the options were limited. Yet, this helped us to start building the MVC architecture.

GUIViews

Since we are using the MVC architecture it is easy for us to replace the previous simpleView with any other View. For this project we created a more sophisticated GUIView which will be able to display the status of all the spots available, the ones taken, the nearest spot and a small description of the location of them.

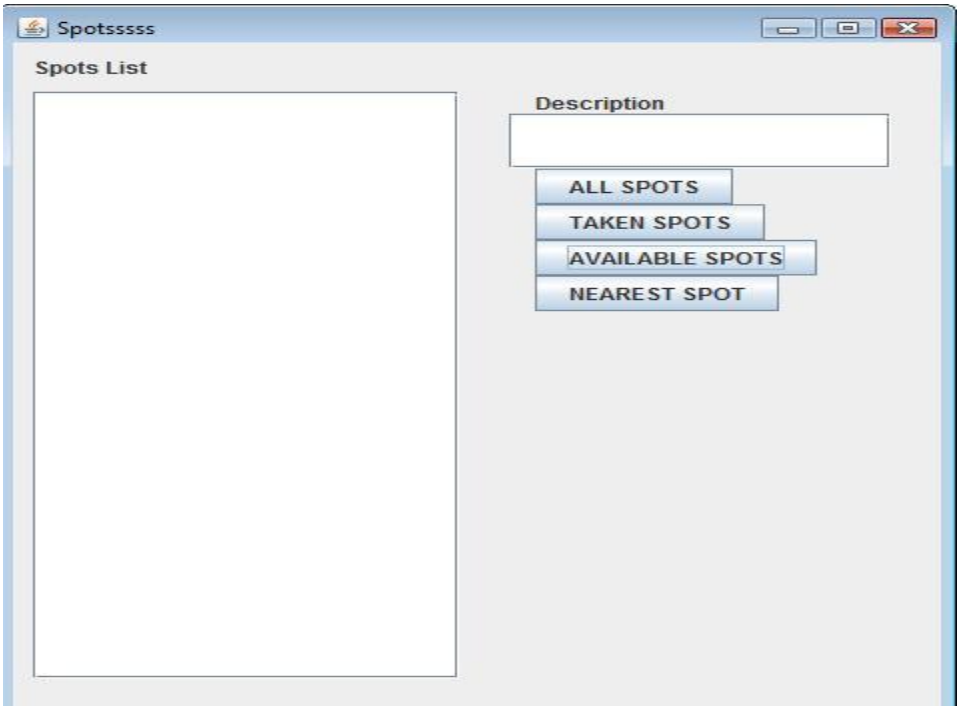

Fig. 19. GUIView with all the options.

As a second approached since we had 5 sensors we created a specific view for those five which turned red if the spot was taken otherwise green. This was the first time we had tested the program with more than 2 sensors and it worked well. The reaction was almost instantaneous and the program had no

problem at all when reading a higher amount of data from the port.

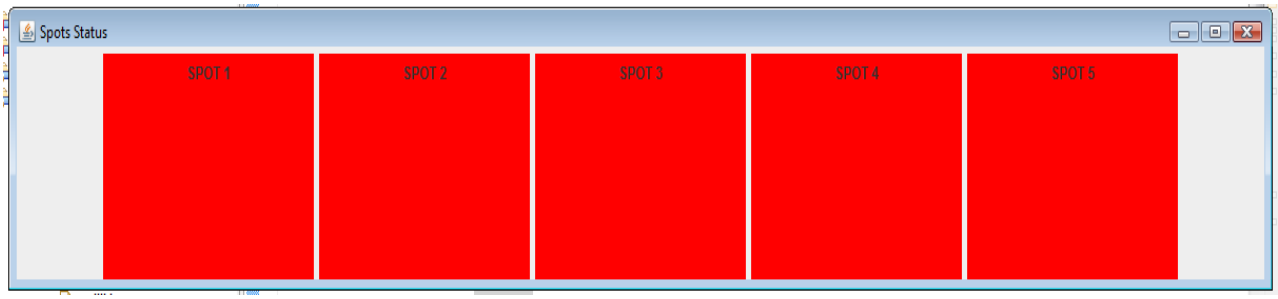

Fig. 20. GUIView of 5 spots

Each GUIView replaced and the SimpleView and they were reading the status of the spots from the backend. Then the next step was to update the backend with the real information. So far the Control part was generating the spots randomly which updated the backend with these random values.

CommPort Reading, Control

Reading from the CommPort was the key part of the programming part of the project. Java does not have a built in library for this. An external library was added 'comm.jar' which allowed us to open the port and read the data stream. This is the main stone of the Control part which will interact with the backend updating each spot's status.

The View part accesses the data structure and gets the spot status while this data is being updated by the Control part using the information at the CommPort.

7. Testing

Testing of the hardware of the system was an ongoing process. It began with simply testing to see if the original Arduino units found were in working order or not. This led to testing of a simply rangefinder which was simply an Arduino with a PING sensor interfaced. This first true test consisted of

two parts. The first was to see if the schematic found online was accurate and would indeed work. The second part was to test the small program or sketch that had been uploaded to see if it indeed was correctly giving the distance from an object. When these tests were complete, testing of the network modules had to occur. Upon setting up the initial Xbee modules provided to us, we began to notice hardware issues that manifested in the form of a highly unstable connection between units in a simple one to one connection topology. When indeed it was confirmed that our tests had been correct new parts were ordered and retesting began once the new parts arrived. While our parts all seemed to check out when run through a diagnostic test by Mark Sproul we could not achieve communication via the Xbee modules. Many tests were run to rule out several factors including wifi interference, power source issues, proper calibration, and coding of the Arduino to name just a few. Eventually a breakthrough occurred when it was found that the baud rate we had been using was far too low at 9600. When set to 57600 our network came to life and network testing could begin.

Testing of the XBee hardware range was done for both line of sight distance and distance with obstacles in the way. For the line of sight tests networks held on for far more distance than was expected, matching the specifications marked on the datasheet. In the case of testing with obstacles, a concrete wall within an apartment building was used. This wall was about six inches thick and network stability held on until we were about 20 ft away from our “hub”. A second test was done showing that the Xbee could indeed talk through the floor of a concrete structure. This time the floor of an apartment building was used and while a connection was made proper RSSI figures could not be recorded. In testing within a parking deck environment ranges seemed to fair much better when it was between two points on the general same side of the parking deck floor. Communication between floors was not possible most likely due to the amount of reinforcement used within the concrete creating a virtual Faraday cage. In these settings the information on the datasheet showing a range of about 100m within a structure seemed to be accurate. When multiple sensors could be connected to the same receiver,

testing was done to see the effects of collisions within the network and to see how the Zigbee protocol would handle it. The Xbees showed that they were able to initiate backoff protocols on their own detecting and preventing collisions. Within the XCTU program used to program the XBees there is an option to change the intervals of backoff to various milliseconds. Next, the software was tested with all sensors connected to ensure that the software was fast and responsive enough to keep up with the flow of data from the sensors. The software proved to be up to the challenge keeping up to date on the statuses of each of the sensors we had deployed during the test and successfully suggested spots based off current availability. Finally battery testing was done to see how long our sensors could last on a fresh battery. Testing showed that after 5hrs 30min our 9V battery was no longer producing enough current to sustain the entire unit. This was evident given the shutdown of first the PING sensor, followed by the XBee units. This short lifespan can be attributed to the fact that the sensors were kept on virtually non stop with delays on the sensors for two seconds with an additional second delay when the sensor was poling the distance again to see if it was still less than the known distance from ceiling to ground as well as the fact that the parts used were on a whole not meant for serious long periods of wireless usage. Ways of fixing this including reevaluation of currently used parts in the hopes of finding parts that better fit with the purpose of wireless sensor networks, a less aggressive delay as increasing the delay reduces the amount of times the sensor is transmitting. Also implementing sleep/wake cycles and having the system be activated only when a vehicle enters a parking deck could go far towards extending the life of the sensors.

8. What's Next?

With the successful construction and testing of our proof of concept design there are many items that would be on a check list of what would be next. First would be a reevaluation of the parts

used for the sensor as a whole. More research needs to be done on more energy efficient parts. Acquiring these new parts would increase the life of the sensor automatically. Next would be communication of sensors between floors of a parking deck. There isn't much use for a system if it can't get data past the first floor. After that would come figuring out how to set up the Xbees to act as relays so that cluster heads can send data to the "hub" in a multi-hop setup. Such steps would only be the beginning of a long road to getting a system such as this up and officially working within a parking deck but it is a vision that we all believe would be worth the effort to bring a system with such potential into practical use.

9. References

http://www.parallax.com/Portals/0/Downloads/docs/prod/acc/28015-PING-v1.6.pdf Parallax, PING datasheet

http://arduino.cc/en/ Arduino Documentation

http://arduino.cc/en/Tutorial/Ping?from=Tutorial.UltrasoundSensor Arduino How-to Guide for PING

Faludi, Robert.Building Wireless Sensor Networks. O'Rielly Media: CA, 2011. Print.

http://ftp1.digi.com/support/documentation/90001003_a.pdf X-CTU User's Guide