Zipcart

Ricardo Henriquez, EE Ryan P. Lagasse, CSE Jonathan Azevedo, CSE

***Abstract*— Checkout aisles in grocery stores are inefficient. There are better ways to process transactions that don't involve waiting in long lines for a cashier. We propose Zipcart: a system that employs computer vision techniques to keep track of the items in an order as a customer shops. It employs a centralized data store on public cloud infrastructure that contains order and item information. A smartphone interface is used by shoppers to audit their order and the balance of their selected items.**

I. Introduction

Of all parts of the grocery shopping experience, the checkout process is the most needlessly long and frustrating. Once shoppers are done selecting the items they wish to purchase, they queue up for service from cashier attendants. This entails iteratively aligning the barcodes of items with a scanner, bagging the groceries, and navigating the point of sale (POS) terminal's interface to process payment.

Maintaining order information and payment processing are both tasks of the digital system. The cashier's part in this is helping the system identify each item in the order and selecting the appropriate means for payment. Using computer vision techniques, this first task could also be automated, as a system can be trained to recognize the items that shoppers select through passive observation. The second task could be entrusted to the shopper, so long as the supported methods are digital-friendly (nowadays, even checks can be processed by digital systems using computer vision).

There are many competing solutions in this space. Amazon Go [1] uses a large amount of cameras embedded in their stores to track what customers take out of the store, charging them accordingly. This approach is very costly and requires the type of infrastructure and technical expertise that only a company like Amazon has. Peapod [2] is one of a class of websites that offer grocery delivery as a service. Consumers avoid trips to the store at the expense of service charges. Additionally, articulating *which* items (out of a similar class of items) can be difficult, given the user interface and things like sales and coupons for shoppers to consider. Both solutions require large changes to the landscape of the grocery store. For Amazon Go, it's the addition of cameras and sophisticated infrastructure to process video feeds. Sites like Peapod either need to employ warehouses for groceries or alter the layout of stores to make it more efficient for "grocery pickers" to sweep through the landscape and fulfill orders. Zipcart necessitates retrofitting shopping carts – not the store. This keeps the customer shopping experience relatively unchanged and makes the cost of renovation less expensive in terms of cost and time.

II. Design

A. Overview

Our project seeks to present a more efficient alternative to the checkout aisle. It starts with a system embedded onto every shopping cart, with a camera mounted onto it. That camera identifies items as they enter or exit the cart by reading their barcodes. Once the system reads the barcode, it passes that information onto a service running in public cloud infrastructure (e.g. Amazon Web Services), which keeps track of orders and item information in a database. The shopper will use an interface on their smartphones (e.g. an Android application) to audit the order balance and items selected as they shop. This also enable us to process payments through a PayPal service called Braintree, which gives the user multiple payment options.

There are some interesting challenges that arise when trying to utilize computer vision for this purpose. First, we need to scan the barcodes of items as they enter the cart. Since we do not require shoppers to put items into the basket in a specific way, they enter the cart at unpredictable angles and speeds, and yet we still must be able to read the barcodes with a high rate of accuracy. Furthermore, the barcodes must be read in a timely manner. Computer vision algorithms are highly resource-intensive. The system running these algorithms has the heavy workload of processing image frames and detecting not only the barcode but direction of each item for the duration of each shopping trip. This must all be done nearly in real-time, as large latencies may confuse or upset shoppers who expect to see the current state of their orders on the interface as soon as possible.

As we are not acting as store facilitators, we do not maintain our own database of items and their relevant information, yet we want our system to be able to scan and account for any item with a barcode. As such, another one of our challenges involves getting this information from an external source quickly. When the web service happens upon an item it has no information on, it must request for and cache it in a manner that complies with the real-time constraints of our system. Lastly, our systems must operate over the course of entire business days. To avoid

R. Henriquez Jr. <ricardohenri@umass.edu>
R. Lagasse <rplagasse@umass.edu>
J. Azevedo <jazevedo@umass.edu>

requiring maintainers to charge carts throughout the day, we utilize the mechanical energy exerted by shoppers pushing the cart to power our system. This necessitates the design of an efficient subsystem that can generate an adequate amount of power to sustain our system with the mechanical potential it sees.

To meet the expectations of each of our stakeholders, we set the following project requirements:

1. Recognize barcode as item is placed in cart
2. Detect when item is removed from cart
3. Display item list and current balances
4. Detect unscanned items to prevent theft
5. Sustain power for a full business day

To create this design, we set a number of assumptions about our project. These are our specifications:

1. One item entered or removed per two-second interval
2. Barcode surface must be reasonably flat
3. Maximum system latency of four seconds
4. Eighteen hours of continuous operation

Residing in the appendix is an enhanced block diagram made up of two figures that illustrate the topology of our system in addition to the function of each component and the relationships between them.

B. Power

We want our system to be self-sufficient to avoid having to plug it into a wall outlet – having to do so multiple times over the course of a day would be impractical. Our approach is to take advantage of the mechanical motion of the shopping cart and convert that into electrical energy which in turn will charge our lithium ion battery. The project requirements, state that our system must operate for a full business day, which means the power generated must be greater than the power consumed. Figure 1 shows the schematic diagram of our power circuit. The diagram is separated into four pieces which are the stepper motor, full-wave rectifier, voltage regulator, and the Adafruit Powerboost 1000C [3].

The first piece is the Vexta PX245-02B-C8 stepper motor which is two-phase and rated each at 6V and 0.8A. We decided to use a stepper motor because they work best at low speeds and in our application, the average walking speed of a customer is 3mph which equates to roughly 200RPM. Heading into MDR, we only use one stepper motor as a proof of concept, which plays the role of converting mechanical energy into electrical energy. The power generated by a stepper motor is proportional to the rotational speed of the motor shaft in the form of AC.

The second piece is converting the AC generated into DC and we do this through a full-wave rectifier made up of 1N5818 schottky diodes [4]. Our stepper motor has two phases, each phase must be rectified which is shown in Figure 1. We added a 100uF capacitor at the output of the rectifier to help reduce voltage variations.

This rectified voltage then feeds into our DE-SWADJ switching regulator [5] which takes the input voltage between the acceptable voltage range of 3V-30V and efficiently converts it into a stabilized 5V output. We chose this regulator because it has the capability to step-up or step-down the input voltage, which makes it an ideal choice in our application since the voltage the stepper motor generates varies dependent on speed. This regulator can source up to 1.5A which gives us plenty of headway, since we do not expect to generate more than 1A. Figure 2 shows the efficiency of this regulator sourcing current at different input voltages. In our case, the efficiency depends on how much voltage we generate at the input which is dependent on the speed of the shopping cart.

The last piece is the Adafruit Powerboost circuit, which has several necessary features. It has a built-in load-sharing battery charger circuit which allows the Raspberry Pi to run while charging the lithium-ion batteries. It also features a built-in battery protection circuit which is necessary when charging lithium-ion batteries to protect from overcharging them. This circuit can recharge at a max rate of 1A and allows the battery to output more than 1A if required.

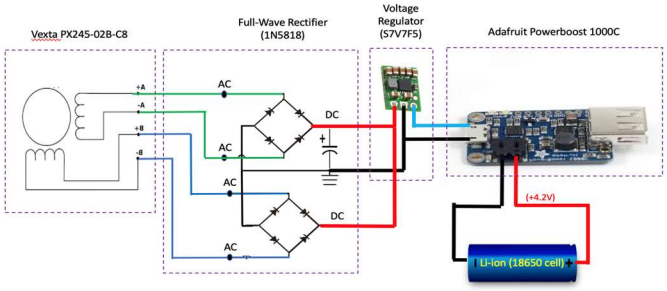

Figure I: Power circuit diagram

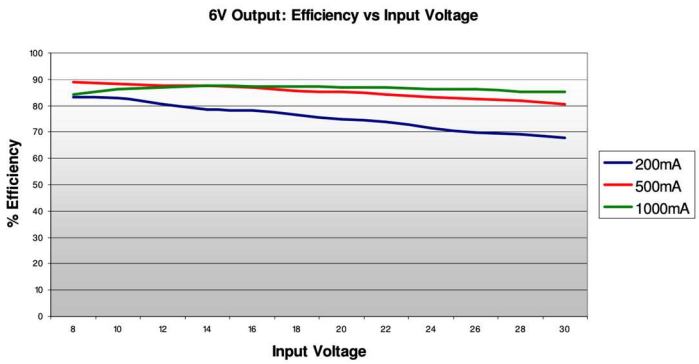

Figure II: DE-SWADJ Switching Regulator Efficiency

To build this power generating circuit, we had to leverage the information we learned in ECE 211 and 212 to understand the general rules of putting together a circuit and the role each element plays. ECE323 and ECE324 equipped us with the knowledge to understand full-wave rectifiers and voltage regulators.

A requirement we had to satisfy was designing and using a printed circuit board. We designed our printed circuit board to wire four stepper motors in parallel in case more power was

needed but we produced plenty of power using only two motors. Refer to Figure 3 to see top view of our PCB design.

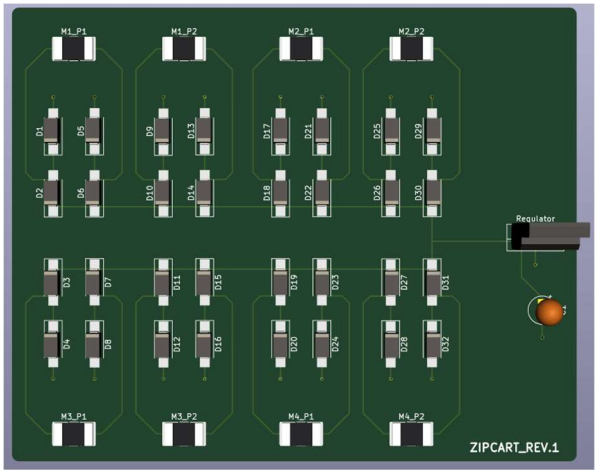

Figure III: PCB Design

For demo day, after all pieces were soldered onto the PCB, we generated 3.08W with two motors in parallel. See Figure IV to see how much power is generated at certain speeds.

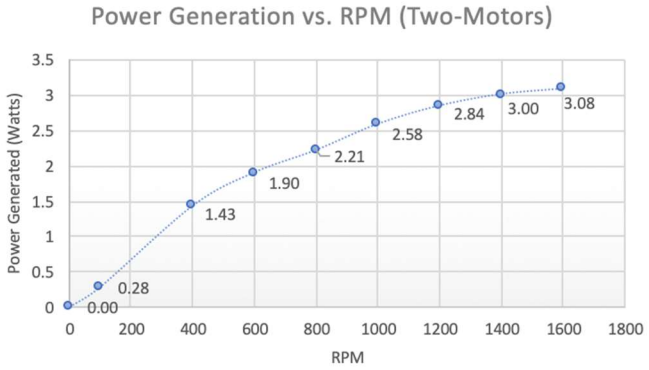

Figure IV: Power Generated using Two-Motors

It is worth mentioning that the Raspberry Pi consumes 1.4W/h while idling and 2.4W/h under a normal workload, refer to Figure V. We obtained these values by taking measurements with a KILL-A-WATT power strip. Overall, we generate more power than what is consumed. Also, we decided to use a switching regulator because it is more efficient at converting high input voltages and stepping down to 5V output. Figure VI shows a significant difference in the amount of current sourced. Although both produced same amount of power, more current is important because it charges the battery at a faster rate.

Pi State	Power Consumption
Idle	260mA (1.4W)
ab –n 100 –c 10 (uncached)	480mA (2.4W)
400% CPU load	730mA (3.7W)

Figure V: Raspberry Pi Power Consumption

	Voltage	Current	Power
Linear Regulator	18V	170mA	3.06W
Switching Regulator	5.63V	535mA	3.01W

Figure VI: Regulator Comparison

C. Optics & Detection

The detection subsystem is the main portion of our embedded system, which reads the barcodes of items as they enter the cart. The detection subsystem consists of two parts: optics and software. The optical part includes a camera placed on the far side of the cart facing inwards. Once an item enters the system and is seen by the camera, the detection algorithm processes the video input, using image processing techniques to locate the barcode in the image and extract the UPC code from it.

The optics system is the most important part of the detection subsystem as it is the greatest bottleneck. The camera must be able to capture the barcode in its frame no matter orientation or distance in respect to the camera. Cameras have a limited field-of-view (FOV), which the barcode must be within to have a chance of being recognized. The camera must also have a high enough resolution to detect the spacing between the lines of the barcode. Since it has to do this at variable lenses it must also have an adjustable focus. Autofocus cameras take too long to find the correct focal length, and an adjustable focus gives us control in the area of the cart we want to focus on. Another requirement for the camera is to be compatible with our software libraries, which all cameras connecting to the Raspberry Pis CSI interface should be. When using the Raspberry Pi there are limited camera alternatives for its CSI interface, because of it we chose the best one we saw fit which was the Kuman 5MP camera [11] with adjustable focus that can stream 1080p at 30 frames-per-second (FPS).

For implementing computer vision, we use two libraries jointly. The first being OpenCV [12] an extensive and powerful open source computer vision library which provides us with a plethora of visual processing tools. The second is ZBar [13] - the library we use jointly with OpenCV to detect barcodes entering our video frame and to extract the UPC code to send to our database.

Detection Procedure

1. Request an order to be made in the database
2. While True:
   a. Detect barcode in video stream from optics system
   b. Extract UPC code from barcode using PyZBar
   c. Sends UPC code to cloud service

To get the detection system working as close to real-time and as efficiently as possible, we made a number of key revisions to it after MDR. The most notable was spending the week of spring break to rewrite the entire detection system in C++ from Python 3. We did this because Python programs cannot be parallelized on multiple CPU cores due to an implementation mechanism called the Global Interpreter Lock (GIL). Luckily, the same software stack was compatible with C++ so we were able to reuse the code we wrote in spirit.

In order to simplify development on some features, we continued to use Python for some system functions. Using interprocess communication, the C++ program would call one

of our Python "submodules" to make a network request or manipulate the LEDs, for instance. Closer to FPR, we made combined our individual submodules into a server that would listen on a specified port for requests by the C++ client. The server would parse them, and make network requests and actuate the LEDs in a dynamic manner instead of calling each submodule separately in order to create the same effect. A larger reason for this change of implementation was to make the submodule stateful. For instance, we solved our "double-scan" issue (when multiple threads pick up the same barcode in consecutive frames and try to make multiple requests) by maintaining a timestamp of the latest scan for each barcode, and only marking scans of the same barcode as valid if they were made some arbitrary time (a threshold value) after the latest one. By maintaining state, we were also able to make features such as flashing green or red depending on the success of network requests easier to develop.

With the advantage of parallel processing came the issue of synchronizing multi-threaded operations. Through trial and error, we discovered that using a queue to store camera frames for processing meant that the system would begin to process older and older data until it was totally out of sync. In order to combat this, we used a cyclic buffer instead. The size of the buffer is determined by the number of threads used to consume the video frames. Endlessly, the producer thread would grab frames from the camera and overwrite the next spot in the buffer in a cyclic manner. When all the frames from the old buffer had been consumed, the new buffer would be pushed for consumption. This would mean that our system would only operate on the most recent frames produced from the video stream.

In the end, we weren't entirely satisfied with our system performance. Processing the frames of an HD quality video stream to detect barcodes is a fairly intense workload. Even by leveraging all the cores on our system, we had a fair bit of latency that mainly required better hardware to decrease. The Raspberry Pi is an inexpensive, commodity computing platform. We identified other ARM-based platforms that would be compatible with our system but did not have enough time to purchase and test a new board. In a comparison done with an average, current-generation laptop, we found the performance of our system to triple (see the figure below). With GPU-optimized code (and the hardware to support it), we would expect to see the performance scale even greater.

Max Frames Processed/Second by Platform	
Platform	Maximum FPS
Raspberry Pi 3 Model B	5.35
Dell Inspiron i5 Laptop	15.51

Table I: Rate of frame processing by computing platform (same code, separate systems)

Lessons learned from ECE 570: System Software Design were immensely useful in helping us synchronize the detection system. The first half of the course directly taught us about concurrency and synchronization, and in retrospect, it feels like this could have been a personal project for the course.

The following table shows how successful our system is at detecting items at various entry speeds. When the item is held still until the system provides a feedback flash, it always recognizes the barcode (however in one trial, the barcode was not read correctly). The faster we enter the item into the cart, the less likely it is that the barcode will be detected by the system. This is directly due to the system latency caused by our less than optimal performance. With better hardware and an implementation written to optimize it, we would expect better results.

Detection Status vs. Item Entry Speed			
	Still	Slow	Normal
Correct Barcode	98%	64%	26%
Incorrect Barcode	2%	0%	0%
No Detection	0%	36%	74%
Total Trials	50	50	50

Table II: Detection status with respect to item entry speed

D. Cloud Infrastructure

The web service for this project takes barcodes from the embedded system as input and uses them to manage both orders and item information in a database. To build this service, we leveraged the offerings on Amazon Web Services' public cloud infrastructure [6]. Web requests made to the service and responses from it pass through the API Gateway, which handles and routes this information with the context it is given. Our system logic operates on Lambda [7], a "serverless" computational platform. It has a Python 3 runtime just like our embedded system. Lambda manages DynamoDB [8] (our non-relational database platform) and makes requests to an external barcode API. To interact with DynamoDB, we use a specially-designed object mapping library called Bloop [9]. At the moment, we are utilizing an API called Barcode Lookup [10], although our design allows us to change vendors with minimal effort.

Concepts from *Software Intensive Engineering* course were beneficial in the process of designing this subsystem. In order meet our latency specification, we evaluated a number of ways to tackle this problem. Leveraging public cloud infrastructure was a glaringly obvious solution, but what remained to be seen was the manner in which we would instrument platforms to meet our requirement. Through experimentation, in consideration with what we had learned, we determined an optimal solution for our needs. We also found the mindset of writing good software – gained from various assignments – to be useful in our implementation. Taking care to handle cases

for errors and exceptions, considering performance and scalability, testing code well, and writing good documentation were all practices that improved the development process of this subsystem.

When we perform our integration tests on AWS infrastructure, it returns the runtime of the code in milliseconds. By performing numerous trials, we determined the average runtime of our system for both a cache-hit and cache-miss (when we are required to get info from the barcode API), finding that the durations of both fit well within our specification. On cache-misses, the expected runtime of our system (updating both the item information cache and order table) is 2.446 seconds.

E. User Interface

The User Interface is the portion of our system which will actively communicate with the shopper to relay important information like whether an item was successfully scanned. It will also allow the user to view and manage their orders.

The feedback system consists of two meters of RGB 60 LED Dotstar LEDs [14] placed along the inner, top perimeter of the cart. The LEDS will be persistently yellow till a QR code is scanned and the embedded system is synced to user application, The LEDs will flash purple when it has scanned and is attempting a network request then flash green or red depending on the status of the request (signifying success or failure, respectively).

We also chose to create an Android application as it is the most popular OS worldwide and the easiest to work with. Having the ability to easily download and access the Android Studio IDE gives us freedom and flexibility to take the app where we want to. The Android application enables the user to view their balance and the list of items currently in their cart nearly in real time. In addition, it provides us with a method to integrate payment processing into the system.

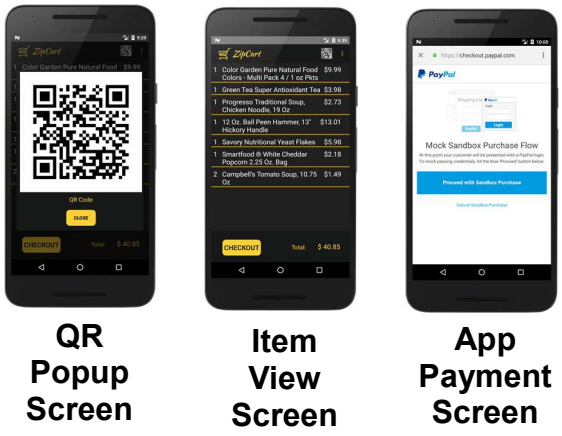

Figure IV: Android Application Screens

To create the application, we used Java and Android Studio [15] which is the most popular developing environment for android apps. By choosing the most popular platform and developing environment we had access to lots of documentation. No class taken by any members of the team helped in this development directly as mobile app development is something completely new to us. There was a steep learning curve, especially in how to use Android Studio and designing the mobile app hierarchy (or app flow).

The QR code shown on the app in the figure above is used to integrate the embedded camera system on the cart to the application. The QR encodes a unique order ID which was created once the user entered the main item view screen. When the camera scans this QR code it extracts the order ID to add the items to during the shopping process. Instead of creating another addition to integrate the app and camera system on the cart, we took advantage of the fact our system is already able to read different barcode types.

From the item view screen, the user can view an items quantity, name, and price, along with the total balance. The user is also given options to open QR popup and cancel order (top right), and the option to pay for the order. To meet our specifications the app checks for new items every half-second so that the wait time between the UI and the database is minimal.

To process order payments, we integrate Braintree [16], a PayPal service, into our application. Braintree provides an easy drop in UI from a developer standpoint that allows us to integrate multiple payment platforms. Braintree supports credit and debit cards, PayPal, Venmo, Apply Pay, and Google Pay; for demonstration we setup a PayPal sandbox as shown in the figure above. It also gives an easy desktop environment to track/manage all order transactions and refunds. When the user completes the payment process the LEDs cycle green on the cart to signify the order was paid for and the user may leave. On the app side the ‘CHECKOUT’ button changes to ‘PAID’ and the user can stay on this screen as a receipt of the order.

To fully integrate Braintree, we had to add necessary components to the application but also setup our own payment server. The payment server oversees initializing users with client tokens and securely passing payment information to Braintree’s main servers (as illustrated in figure below), This service is also hosted by AWS as an EC2 instance.

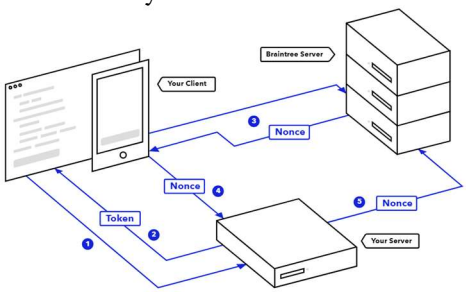

Figure V: Braintree Payment Processing

Software Intensive Engineering has proven to be very helpful in this part of the project as well as we need to create clean, concise, sustainable code. Creating an application can be a hard process and bad code practice will only server to delay us more and cause more issues. It was very important for the feedback system, as there were multiple programmers working on it, that we write clean, well commented, functional code and practice good software development techniques like correct usage of version control to complete this subsystem.

III. Project Management

#	Deliverable	Goal	Status
1	Detect barcode around front-face of camera perspective	30″	Works up to 20″
2	Display Order Info to User	✔	✔
3	Scan-to-UI Latency	4 sec	≈ 2.5 sec (with cache)
4	Power Generation	2.4w	≈ 3.1 W
5	Continuous Operation	18 hours	24 hours

Table III: Status of FPR deliverables

Our team consists of two Computer Systems majors (Ryan and Jonathan) and one Electrical major (Ricardo). As a result, we split the deliverables amongst the group by major, with the CSEs taking deliverables one through three and Ricardo taking deliverable four. As a small team of three, we had agreed that each of us would work primarily alone in order to make progress, and stay in close communication with the team with status and updates. This would allow us to each be flexible with our schedules and not have to meet as a group to make progress, while also allowing for the opportunity to meet when needed to collaborate or solicit help.

Ricardo worked on the power subsystem and his deliverable mostly alone and with great success. When issues arose or he needed validation for his work before continuing onwards, he arranged meetings with Professor Robert Jackson to talk about the circuits and electronic components of his design. In creating his demo, Ryan assisted him by loaning an Arduino and helping him out with the code.

Ryan also worked mostly unsupervised, completing the second deliverable in its entirety and that of deliverables one and three with some assistance from Jonathan. This included writing the detection code and integrating AWS. Ryan also completed the website in its entirety, in addition to mounting the non-power components of the system to the cart. Getting the feedback system operational involved creating a circuit and writing code, both of which Jonathan helped to debug.

Jonathan worked on the user application and payment system Independently. This included creating the user application in android studio, UI design, designing integration with order on embedded system, and fully integrating payment system.

Ricardo and Jonathan worked on setting up our system on a shopping cart. They created a chassis for the motor system and affixed it to the bottom of the cart. Jonathan designed the gears for the power system and Ricky mounted them to the wheels. Both worked on further securing the motors to the wheels and making sure the chassis would not move during use.

IV. Conclusion

Although we were not able to complete each requirement as we had initially planned (namely indirect detection around the object), Team Zipcart was able to construct a system that we believe has exceeded the base requirements and expectations of our stakeholders (e.g. store owners and shoppers). Barcode detection, power generation, the application interface, and payment have all been demonstrated to work as expected for normal use while grocery shopping. Along the way, we've learned a lot and received practical advice from many people that we can use in future endeavors, whether this may (or may not) be the end for Zipcart.

Acknowledgment

Team Zipcart would like to thank our advisor, Professor Wolf, for making the time out of his busy schedule to meet with us weekly and provide us with invaluable advice. We also want to thank our evaluators, Professor Krishna and Professor Aksamija, for critiquing our project and supplying us with thoughtful feedback. We would also like to thank Francis Caron, Professors Hollot, Jackson, Goeckel, and Irwin, and Shira Epstein for their valued assistance.

References

[1] Amazon.com, Inc., *Amazon Go.* [Online]. Available: https://www.amazon.com/b?node=16008589011

[2] Peapod, LLC. *Peapod*. [Online]. Available: https://www.peapod.com/

[3] ada, l. (2018). *Adafruit Powerboost 1000C*. [online] Cdn-learn.adafruit.com. Available at: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-powerboost-1000c-load-share-usb-charge-boost.pdf [Accessed 20 Dec. 2018].

[4] Vishay.com. (2018). *1N5818*. [online] Available at: https://www.vishay.com/docs/88525/1n5817.pdf [Accessed 20 Dec. 2018].

[5] Superdroidrobots.com. (2019). *Adjustable 10W Step Down Switching Regulator*. [online] Available at: https://www.superdroidrobots.com/shop/item.aspx/adjustable-10w-step-down-switching-regulator/824/ [Accessed 14 May 2019].

[6] Amazon Web Services, Inc., *What is AWS*. [Online]. Available: https://aws.amazon.com/what-is-aws/.

[7] Amazon Web Services, Inc., *AWS Lambda*. [Online]. Available: https://aws.amazon.com/lambda/.

[8] Amazon Web Services, Inc., *AWS DynamoDB*. [Online]. Available: https://aws.amazon.com/dynamodb/.

[9] numberoverzero, "Bloop: DynamoDB Modeling." [Online]. Available: https://bloop.readthedocs.io/en/latest/

[10] "Barcode Lookup Homepage." [Online]. Available: https://www.barcodelookup.com/

[11] "Kuman 5MP 1080p HD Camera Module for Raspberry Pi For Raspberry Pi 3 model B B A RPi 2 1 SC15." [Online]. Available: http://www.kumantech.com/kuman-5mp-1080p-hd-camera-module-for-raspberry-pi-for-raspberry-pi-3-model-b-b-a-rpi-2-1-sc15_p0063.html.

[12] OpenCV library. [Online]. Available: https://opencv.org/.

[13] ZBar barcode reader. [Online]. Available: http://zbar.sourceforge.net/.

[14] iPixel LED Shiji Lighting "APA102 Data Sheet." Adafruit. [Online].Available: https://cdn-shop.adafruit.com/datasheets/APA102.pdf

[15] Android Developers. (2019). Documentation. [online] Available at: https://developer.android.com/docs

[16] Braintree Developer Documentation. [online] Available: https://developers.braintreepayments.com/

V. Appendix

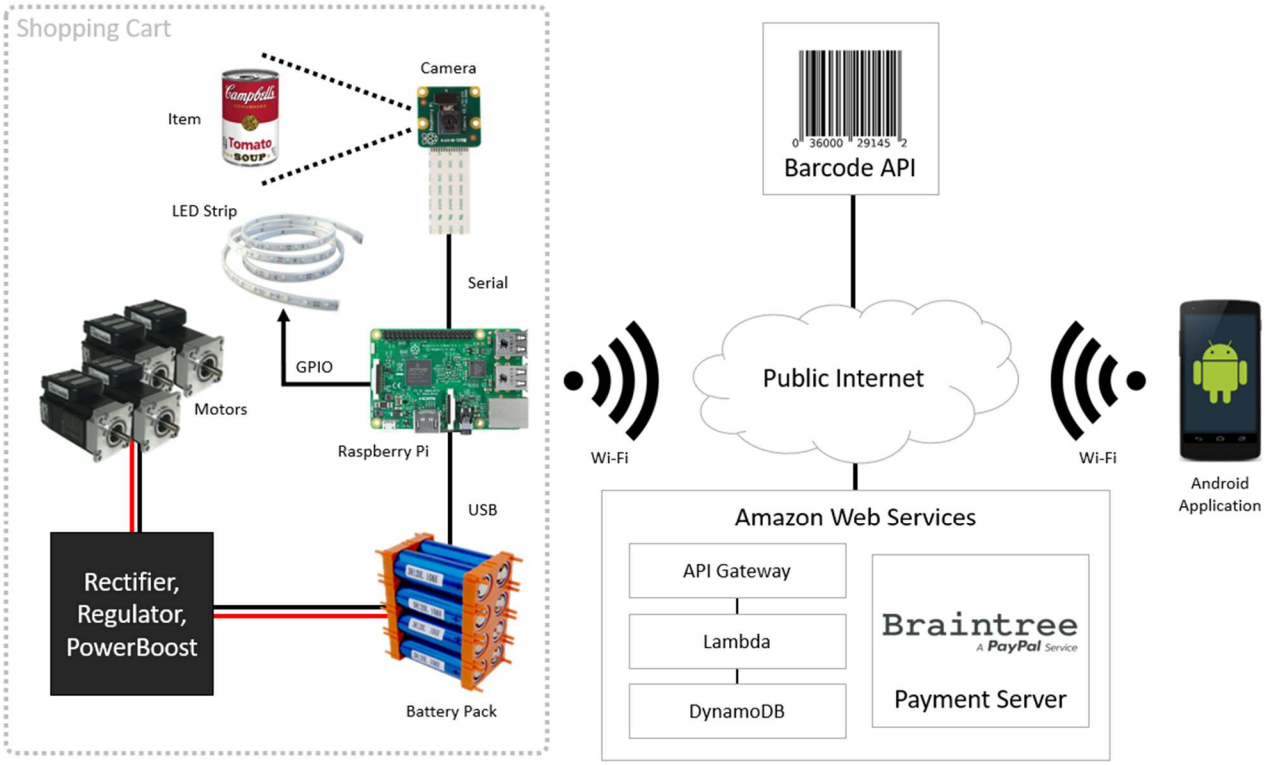

Figure V: System Topology

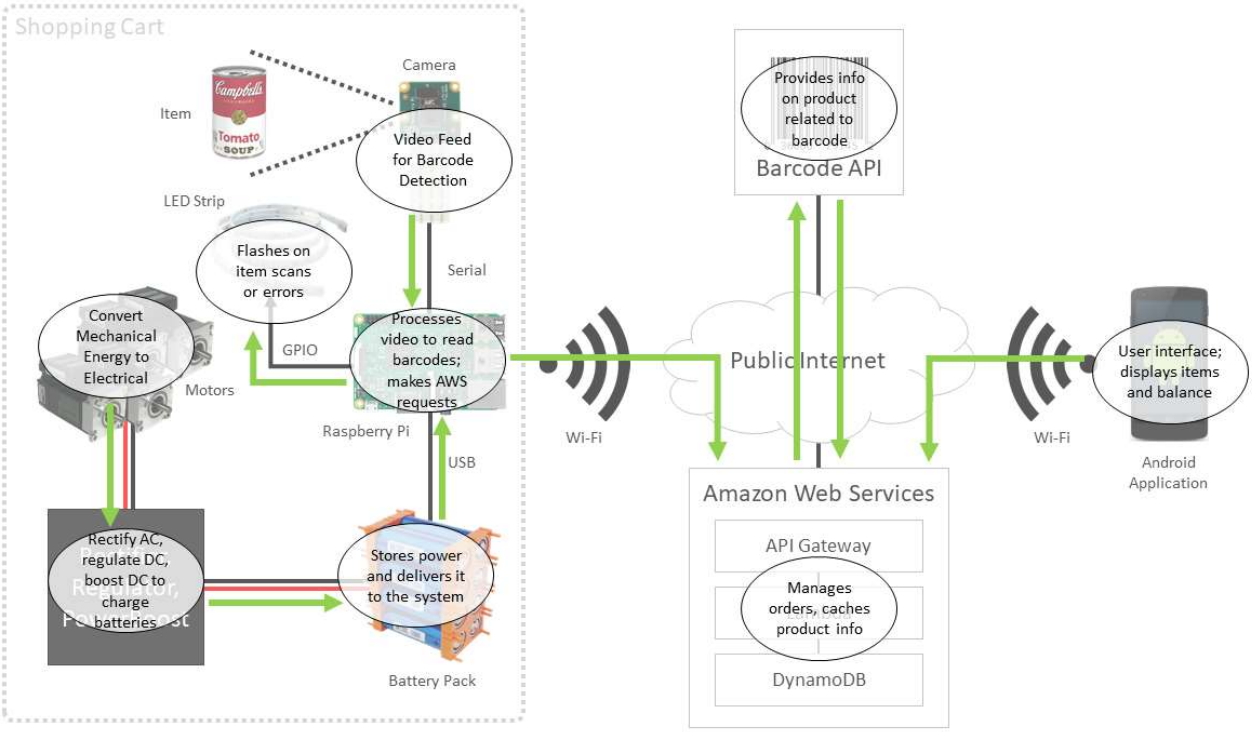

Figure VI: Component Functionalities & Relationships

Part	**QTY**	**Development**	**Production**
Shopping Cart	1	$62.99	$58.49
Raspberry Pi 3	1	$35.68	$35.00
Camera	1	$25.00	$22.50
Ribbon Cable	1	$3.95	$3.16
Stepper Motor	2	$58.62	$44.00
Adafruit Powerboost 1000C	1	$19.95	$15.96
Switching Regulator	1	$14.95	$12.93
Samsung Li-Ion 18650 Cells	4	$15.96	$11.00
PCB	1	$1.00	$0.77
Schottky Diode	16	$7.68	$2.72
Push Buttons	2	$2.18	$1.36
Voltage Level Shifter	1	$2.95	$2.51
Total		**$250.91**	**$210.40**

Table IV: Zipcart Cost Analysis