Team 26: EfficienSeat

Dennis Donoghue, CSE, Matthew Donnelly, EE, Kristina Georgadarellis EE, and Aarsh Jain, EE

Abstract—**The EfficienSeat system is proposed as a solution to address problems and inefficiencies presented by overcrowded dining halls. Through real time monitoring of individual seats via sleek, LED based, modular units on every table, users walking through the dining hall can easily ascertain seating occupancy at a glance and claim their seats with ease. Further, these units will ultimately communicate this information to users outside the dining hall via a phone application in the form of a real time map, giving a simple visual way to determine dining hall occupancy and viable seating locations.**

I. Introduction

Dining halls on our campus suffer from problems in seating that limit their overall efficiency. During busy times, incoming patrons require info on available seating. Without this information people travel up and down walkways searching for a potential open seat. Larger parties face more difficulty in finding seats. Patrons are then faced with the equally unfavorable choices of using their limited time walking throughout the entirety of the dining hall an unknown amount of times until they find a seat or leaving and wasting the meal swipe or money that they committed to enter the hall. This also inhibits staff ability to quickly refill food and dishware throughout the dining hall. These time delays, though potentially minimal in single instances, add up and are detrimental in a food service setting.

We sought to design a system that would decrease these frustrations and time delays and explored several methods of implementation. A big factor was whether to use an active or passive system. An active system completely controls where patrons sit while a passive system reacts to where patrons choose to sit. A completely active system would be akin to a restaurant reservation system. This system would be good for maximum seating efficiency but is not suitable for a dining hall environment. Patrons would be averse to the idea of being told where to sit, especially when the dining hall is relatively empty. The sheer number of seats to manage in a dining hall also makes this approach inefficient.

On the opposite end of the spectrum we contemplated an entirely passive system, one that indicates seat status based on the seats at which people choose to sit. This system senses when someone is sitting at a particular seat and then relays this information to patrons inside and outside the dining hall. This system is favorable because patrons will not have to do anything new, however, it would be expensive and complex to implement. Our solution was a hybrid of the two approaches. With the hybrid system in mind, we then considered the specific needs of our solution. The solution must improve time and travel efficiency for patrons and staff. To do this, it must accurately identify seating status and indicate this to incoming patrons. It must also be simple to service without interrupting the normal operation of the dining hall e.g. easy to install and requiring infrequent maintenance. The only maintenance that should be required would be battery replacement once a month for individual monitoring units. The system should be IPX4 [1] compliant to comply with food safety standards and to ensure reliable operation.

To satisfy these needs, the system will consist of nodes situated on every table in the dining hall, all reporting to a central hub. The nodes need to be small and self-sufficient in terms of power; this coincides with the low maintenance requirement, where routine recharging should be monthly. Every node will be connected to a central hub, creating a network that will reliably monitor the seating situation and ultimately report this to the user via a phone application. Table 1 quantifies these requirements.

TABLE I
Requirements and Specifications

Requirement	Specification
Table search functionality	Users can search for available seats by party size through a mobile app
App users will receive timely graphical response	Updated map displayed within 2 seconds
App users can find their tables from the map on the mobile app	Table locations will be accurate to ½ a table length
System can manage entire dining hall	Can support > 100 seats
Table unit is splash-proof and safe for use in a dining hall	Table unit is compliant with IPX4
System can accommodate non-app users	Patrons can claim seats by pressing a button on the Table Unit
Infrequent maintenance	Monthly battery replacement

II. Design

A. Overview

To solve this problem, we designed a network within a dining hall that has real time monitoring of seat occupancy and consistent reporting of this data to users. The data is accessible to users both inside and outside of the dining hall. As shown in the block diagram in Figure 1, this network consists of four subsystems: 1) the nodes - dubbed 'Table Units,' 2) a 'Sky Unit,' 3) a database, and 4) a phone application. The purpose of each subsystem and the interaction between them is outlined in the overview, with more detailed descriptions provided in the following sections.

D. Donoghue from Dover, Ma (e-mail: ddonoghue@umass.edu).
M. Donnelly from Springfield, Ma (e-mail: matthewdonne@umass.edu).
K. Georgadarellis from Dartmouth, Ma (e-mail: kgeorgadarel@umass.edu).
A. Jain from New Delhi, India (e-mail: aarshjain@umass.edu).

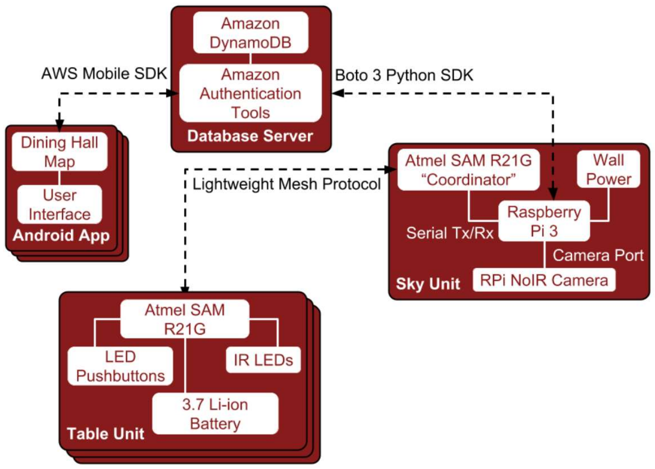


Fig 1. Block diagram of our system. There are four main blocks that make up our system: The Table Unit, Sky Unit, Database Server, and Android App.

Patrons of the dining hall, the users of our system, interact with it from two points: the phone application and the Table Units. Table Units populates the dining hall as compact units, one per table of four seats. These units provide the first point of user interaction within the dining hall: users claim seats by pressing a button, and the LEDs on the unit indicate seat status, providing a simple method of identifying occupancy at a glance. By having an easy-to-interpret system to claim seats, the issue of patrons marking their spots with valuable items like phones or keys is eliminated and seat status will be clearly marked via the LEDs, eliminating the ambiguity of a stray coat or left behind dishware.

Seat status is reported from each Table Unit to the Sky Unit, the central hub of the network. As the hub, it processes the Table Unit data and functions as the medium of communication between the Table Units and the database/app side of the system. Alongside this, the Sky Unit will also perform table localization; this information is used to accurately report seat status to users via the app.

From the Sky Unit, seat statuses are sent to the database. The database is constantly updated in real-time by both the Sky Unit and the phone app so that it contains up-to-date information of all seats within the dining hall.

The phone application provides the other point of user interaction with our system from outside of the dining hall. Namely, users can view a map of the dining hall with table occupancy, search for available seats by party size, and reserve seats, all from the convenience of their phone. The map of the dining hall is generated using information queried from the database for table occupancy and the localization data for the position of each table within the dining hall. With a basic outline of the system's functionality and interactions established, the following sections delve into each subsystem more, discussing design choices and alternatives, protocols, and component interactions.

B. *Table Unit*

The Table Unit is a small modular unit for user interaction at the table itself containing a visual LED and button pair for each of the table's four seats (See Figure 2). It also contains four IR LEDs, which will be discussed later regarding table localization. The visual LEDs are used to indicate the status of the seats to the user, namely whether the seat is taken or not. The buttons are used to confirm seat reservations made from the app and to occupy and vacate seats without the app.

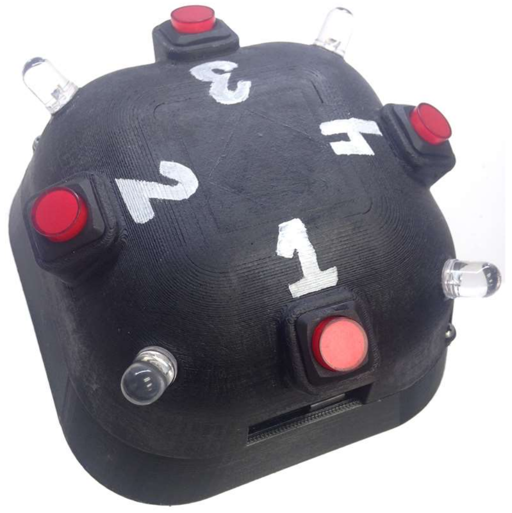


Fig 2. Picture of the final Table Unit Design. There are four LED push buttons on each side and four IR LEDs on each corner. The front panel houses the ON/OFF switch and microUSB port for charging.

These components are integrated with a battery power supply and the Atmel SAMR21G board by our custom PCB, which will be discussed below. The following table shows the seat state corresponding to the LED responses:

TABLE II
TABLE UNIT CHAIR STATES

Chair State	LED Response
Vacant	OFF
Occupied	ON
Reserved	Flashing
About to Expire	Flashing

The following bullets guided us on selecting the processor to be used on the Table Unit:

- The processor needs to support a wireless network with low bandwidth and energy.
- The processor shall be able to handle at least hundred nodes on a network.
- It should be low powered to keep the maintenance of the unit to a minimum.
- Needs to also take input from four buttons and output to eight LEDs.

The SAMR21 microcontroller [2] by Microchip Technology Inc. seemed to be the right choice. The microcontroller has a System on Chip to connect the Table Unit to a network based on IEEE 802.15.4 communication protocol [3]. This is a low bandwidth network protocol due to reduced power requirements to run the network. The Lightweight Mesh protocol provided by Microchip is well supported for this microcontroller [4]. Some of its features are:

- Network management of more than sixty-five thousand nodes.
- Queuing of incoming packets.
- Callbacks to confirm data received.

The project started with a few example codes in Atmel Studio Framework to better understand the overall code that would govern operation of the Table Unit. Simple network communication was established between two SAMR21 boards via their built-in RF antennas. The range of communication was established to be beyond 50 feet through testing with a wall barrier, this range is more than enough to communicate to any corner of a dining common from its center. Further testing was done including button debouncing optimization to give a smoother interface for the user. Typical Table Unit operation was tested using a default set of commands including the following responses:

- Button press shall toggle states between vacant and occupied.
- Message received from Sky Unit shall be processed and signified on LEDs.
- Reserved state shall go to occupied when button is pressed.
- Timer state shall give a pulse on the corresponding LED.
- Data shall be sent when a button is pressed at the Table Unit which should be properly received at the coordinator in Sky Unit.

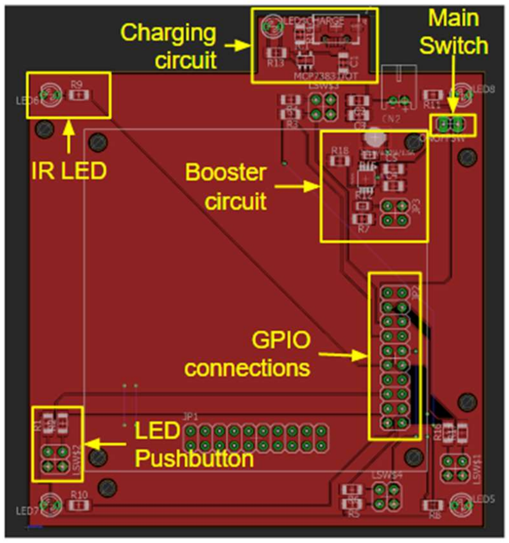


Fig 3. CAD Layout of the custom-built PCB. The board was built using the services provided by OSH Park and soldered using SDP soldering equipment.

The Table Unit uses a custom-built PCB to accommodate a few extra features for robustness, utility and ease of use for both the user and dining hall staff. The features included in the custom PCB are as follows:

- Booster circuit - supports up to four Panasonic Model 18650 batteries; total capacity = 13600 mAh.
- Charging circuit - charges the batteries at 500 mA with a status LED. The charging can be done using any standard micro USB type B port.
- Main switch to turn units off if not used, reducing the current to less than 10 µA.
- GPIO connections to breakout SAMR21 Xplained pro board to precisely fit IR-LEDs and buttons onto the case with their respective resistances.

In addition, the PCB stands as the main frame to hold the Xplained pro board with nylon standoffs. A second set of drills enable the board to easily fit with the custom-built case. All GPIO pins use standard double row male pin heads for maximum compatibility and easy swap of components.

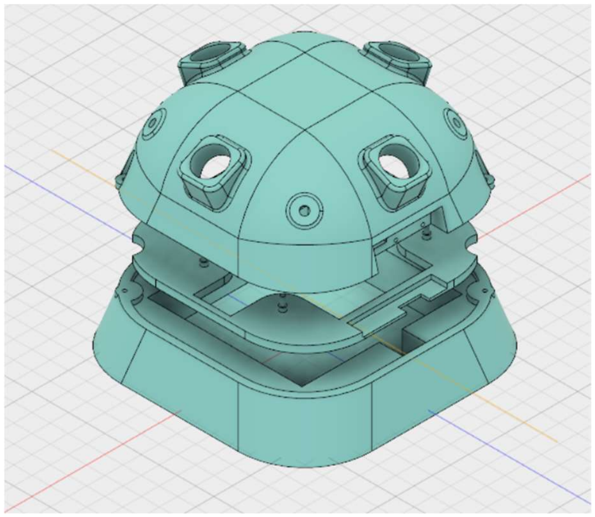


Fig 4. CAD model of the Table Unit case. The unit was designed in Autodesk Fusion 360 and 3D printed at M5 using both ABS and PLA.

The PCB, SAMR21 board, and batteries are housed inside the Table Unit case. The case was designed in Autodesk Fusion 360 and 3D printed at M5 using both ABS and PLA. In practice, these units would be made from a more durable and smooth material like acrylic or nylon. The table unit has three parts, a top piece for mounting the push buttons and IR LEDs, as well as providing access to the microUSB port and ON/OFF switch, a bottom piece is where the Li-ion batteries reside, and a middle piece to mount the PCB on for stability as well as placement in the case. The case is held together by two screws at diagonal corners.

As it is, this case is not IPX4 compliant or bacteria resistant. Our plan was to create a mold for the case so that we could make a food-grade silicone casing to cover it, but this unfortunately did not come to fruition due to time constraints and our inexperience in the area.

C. Sky Unit

The Sky Unit is composed of the Atmel SAMR21G processor and Raspberry Pi 3 board [5]. The prototype for the complete unit is shown in Figure 5. For table localization, a Raspberry Pi NoIR camera module [6] is used as well as a lens that filters out the visible spectrum. This wall mounted unit operates from wall power. The Sky Unit functions as a central hub for all the Table Units, and as an intermediate link between the Table Units and the database/app. The SAMR21G board is used to communicate wirelessly with the Table Units via the Lightweight Mesh Protocol, as explained in the Table Unit section. The Raspberry Pi was chosen as the central processing system for the Sky Unit due to its power and versatility as a computer coupled with easy-to-use functionality and an abundance of available documentation. For our database needs, we ultimately decided upon Amazon Web Services (AWS) - which is discussed in more detail in Section II.D - and this influenced the functionality of our Pi. Python code using Boto

3, Amazon's own SDK for Python [7], [8], is used on the Pi to communicate data between the Table Units and the database. Given the plentiful documentation for Boto 3 from AWS, as well as the convenient use of Python scripts aboard the Pi, this communication method proved to be the most efficient way of both writing to and reading from the database with the Pi. By utilizing the Lightweight Mesh and the Boto 3 SDK, we have successful communication between the Table Units and the database.

TABLE III
PROTOCOL

Address	Command	Data
2 Bytes	1 Byte	4 Bytes

Alternatives for interfacing the Pi with the database were explored as per the database chosen; Boto 3 was our final choice as we settled on Amazon DynamoDB [9] with which Boto 3 is closely coupled, but a previous option was Amazon RDS [10] - the choice to switch databases is discussed in detail in the following section.

We designed a communication protocol for seating information passed between the Sky Unit and Table Unit. To inform our design choice, we identified the information both units would need to receive in order to function properly: Table ID, Command Type, and Data. The resultant protocol is a 7 byte string, as shown in Table III. The first two bytes correspond to the Table ID, which is unique and is assigned to each table during system calibration. The next byte identifies whether the command being sent corresponds to the seat LEDs or the IR LEDs; this is only relevant for messages sent to the Table Unit and is ignored for messages sent to the Sky Unit. The last 4 bytes indicate whether the corresponding LED is to turn on or off. Possible values and their matching information are defined in Appendix A.

An example command sent from the Sky Unit to the Table unit would be "01 1 0 1 0 0." This corresponds to a command sent to Table 1 (ID 01) to change the seat states (command byte 1) as indicated by the data bytes. This particular message would turn seat 2's LED on (occupied) and seat 1, 3, and 4's LEDs off (vacant).

To test this subsystem in terms of the Sky Unit's ability to process and transmit data to and from the server, we used our protocol to send example data to the database from the Raspberry Pi. Our Python program automatically handles database connections and reading/writing through Boto 3. Incoming Table Unit messages are translated into the necessary DynamoDB format and the database is updated. In the same fashion, the database can be accessed readily and information can be read, translated into our protocol, and quickly sent to a Table Unit. To ensure accuracy and agreement between the database and the actual Table Unit states, conditional writing was also implemented. An "image" of the database is created locally on the Pi and is updated with every message that is sent. Any mismatches between the local image and the database are found and corrected.

Our system should function as close to real time as possible so that both the phone app and the Table Units convey the same information to users at a given time. To ensure this functionality we utilized DynamoDB's Stream application, which provides chronological records of database changes [9]. To efficiently process stream records, we used the abstraction layer Bloop [10]. With Bloop, our Python code could easily access and process stream records produced by AWS and propagate this information down to the Table Units in a timely manner.

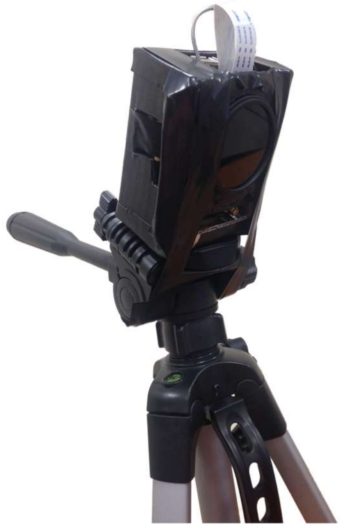


Fig 5. Sky Unit prototype used for Demo day. In practice, the unit would be mounted on the wall.

The code also handles Table Localization during normal runtime through an interrupt-based system. Normal operation is interrupted periodically so that table locations may be acquired and updated if need be. This process is done infrequently to not disturb normal operation often and for extended periods of time.

Our system needs to determine the position and orientation of each Table Unit in the dining hall to provide an accurate mapping of the tables and seats on the mobile app. Without it, our system can tell the app user that a particular table and seat is open but not where the table is or which of the four seats are available. we wanted our system to have adaptability in case tables were moved, e.g., if patrons pushed two tables together to seat eight people. The criteria for our table localization system is as follows:

- Tables identified within at least half a table length
- Periodic updates every hour
- Image processing does not disrupt entire system

Although we explored many options as discussed in length in our Mid-Semester report, the best choice for us was implementing an IR camera system where the camera on the Sky Unit takes pictures of the room and image processing is used to filter out the IR LEDs on the Table Units.

For our system to work, the camera undergoes a one-time calibration before dining hall operation begins. This is done by placing a Table Unit at four known locations in the dining hall at table height e.g. placing the unit at each corner of the room. The IR camera takes a picture of each location individually and then uses the same image processing for position and location to find each unit. Lastly, these new points are mapped to a 2D perspective, yielding a grid on which the units can be found. This is discussed in further detail in Appendix C.

For finding position, the Sky Unit sends three commands to one Table Unit to flash the IR LEDs two times, taking a picture in between each command i.e. obtaining three images depicting the LEDs on, LEDs off, then LEDs on. These images are then processed using OpenCV for Python [11], with information

used from Adrian Rosebrock's website, pyimagesearch to use the Raspberry Pi Camera with OpenCV [12].
The image processing used a four-step process:

- Filter image using a Gaussian Blur (to filter out noise), and a Binary Threshold to isolate pixels that are the same intensity as the IR LEDs (Fig 6)
- Use a Blob Detection to isolate the IR LEDs on the first image (Fig 6).
- Compare detected pixels to the other two images, and keep pixels that match pattern of ON-OFF-ON sequence
- The remaining points are averaged, and the table center coordinates are sent to the database

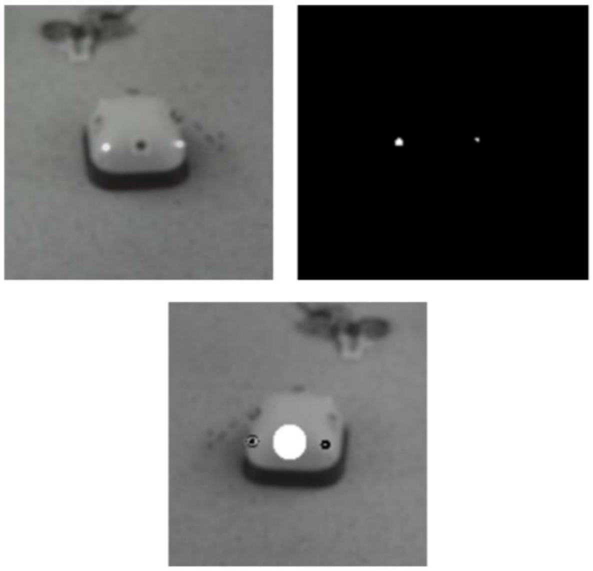


Fig 6. *Top Left:* Original capture of Table Unit with IR LEDs ON. *Top Right:* Filtering done on image. *Bottom Middle:* Blob Detection (denoted by the black dots) with the center of table marked by the white circle.

This process is repeated for each Table Unit. As the database is updated, so is the rendering of the map on the mobile app.

Additionally, the Table Unit orientation is needed to determine which button and LED corresponds to which seat, since the Table Unit is not fixed to the table (for cleaning purposes). The Sky Unit finds orientation in a similar process to position. Instead of turning all four IR LEDs on at once, each IR LED is turned on one at a time. The camera captures an image for each LED and determines if the LED is found using the filtering and blob detection for position-finding. Based on the LEDs that the Sky Unit "sees", the orientation can be found relative to Seat 1 i.e. the orientation is 0-degrees if Seat 1 can be seen.

D. *Database/Server*

Our system features a set of databases, each of which stores table data corresponding to a specific dining hall. These databases are populated with information obtained from the Sky Unit regarding each Table Unit, as explained in the previous sections, corresponding to table layout and orientation, and seating occupancy status. Our app fetches data from the database corresponding to the user-selected dining hall and uses it to populate the app's dining hall map and respond to user searches. To this end, we decided to use Amazon's DynamoDB. DynamoDB is fast, flexible, and highly scalable NoSQL database hosted by Amazon Web Services. DynamoDB is fully integrated with the full suite of AWS services, including identity management, update streaming, and access security. The techniques needed to produce this part of our project were acquired both in coding classes and through work experience. A test of this part of our system would likely start with the basic CRUD functionality, then test conditional writing, simulating data mismatches, and then test the integration of the database with our greater system. If we can perform CRUD operations properly, detect data mismatches and correctly connect the parts of the system together, then the test will have passed.

Before using DynamoDB, Amazon RDS was also explored as a potential option, however it proved difficult to interact with in the way we needed. MySQL is available for simple Pi-database interaction using RDS, but at the cost of security; to successfully interact with the database, a lot of the security authentications need to be removed. With DynamoDB, initially connecting to the database involved a bit more work but the overall process of interacting is simpler with respect to the Pi, and more secure in terms of the connection. This led us to choose DynamoDB for our project.

E. *Android App*

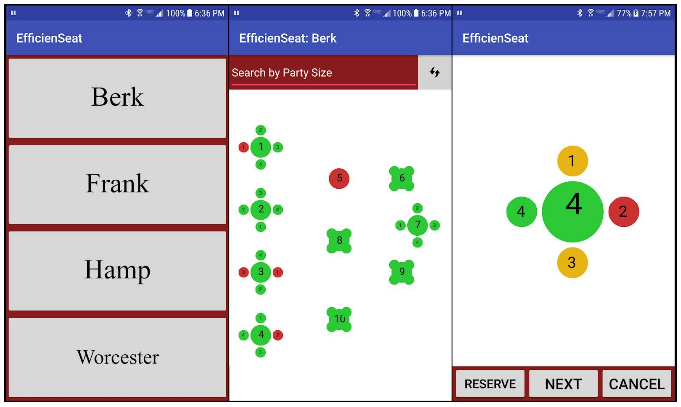


Fig 7. Screenshots of our Android app. *Left:* Users can select their dining hall. *Middle:* A map of tables and their occupancy is presented. *Right:* Results from user query based on party size.

Our app is one half of the outward-facing part of our system. The app allows the user to select the dining hall they wish to interact with, to browse a seating map of the selected dining hall, to search the available tables by party size, to reserve and to re-reserve seats and tables, as shown in Figure 7. The Android Studio IDE and Java were utilized to write and build the app. A testing suite for this app would be used to initially test the user's ability to select a dining hall, the app's ability to update the table map based on database changes, the app's ability to communicate with the Pi, and the app's ability to search the table data by party size. Proper outcomes from these tests confirmed app functionality before integration with the rest of the system.

III. Project Management

EfficienSeat was brought to reality through the effective communication and organization of our team. We continuously discussed each component as they pertained to the whole of the project, not just in terms of functionality but also higher-level application, since this would be used heavily within a social setting; we wanted our system to make sense to users and staff. Weekly meetings with our advisor, Professor Wolf, helped keep us on track and able to prioritize the different components of our project.

The project was neatly divided into four main components, as reflected in our block diagram, which we were able to divide up amongst our team. Though everyone had an assigned focus on different aspects of the project, there was great team involvement to help flesh out the most efficient approach and to ensure that everyone was consistently on the same page. Aarsh was our Table Unit expert, managing the innards including PCB design and circuitry linking the PCB, Atmel processor, and external periphery. He also coded the Atmel processors on both the Table and Sky Units. Matthew managed the Sky Unit, handling the main code ensuring database and Table Unit communication and message processing. Kristina developed our table localization method and helped integrate it with the main code on the Sky Unit. She also designed the CAD models for the Table Unit casing. Dennis handled phone app development and integration with the AWS server. As previously mentioned, each component had substantial input from the team to ensure that the best method to solve the problem and implement the solution was found and agreed upon.

IV. Conclusion

TABLE IV
FPR Deliverables

Deliverable	Percent Complete
Deployable Table Unit • Case, PCB built and fully integrated • Case protects from food/spills	80
Demonstrate Table Localization ability • Improve algorithm for large scale implementation	100
Complete and robust system operation • All parts fully integrated • Complete user App experience • Reservation/claim timers implemented	100

Our project functions as intended, allowing users to claim seats via the Table Units and reserve them via the phone app. Information is communicated reliably and quickly across the system. We did not yet ensure IPx4 compliance for our Table Units due to time constraints and inexperience in making silicone casings.

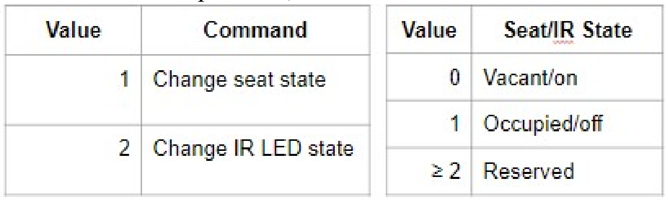
Features that we wanted but were unable to get into the final version of our project include batch multi-reserving and full streaming capabilities. Currently, our app allows for users to reserve seats one at a time by hand or several at once through our search function. However, each of these reservations is done individually. Given enough time, we would have preferred a system where we could select seats to add to a reservation queue, which could be reserved or re-reserved as a batch. This would allow the user to more easily re-reserve his seats, as well as to make it easier to select and unselect seats before making a reservation. Currently, reserving several seats in a short stretch of time creates streaming records we cannot process on the Pi, causing delays, bugs, and/or crashes. If we had more time, we would have liked to implement database streaming on the app side and improved the robustness of all our streaming capabilities. To this end, I think choosing another database service would be wise if we intended to work on this project long-term.

We would also like to redesign the booster circuit to minimize our power consumption. Due to not being able to test multiple designs of our PCB because of cost, our initial design had an unintended issue causing more power to be consumed than was initially planned.

Appendix

A. Communication Protocol Definitions

Below are the definitions for each byte of our Table unit to Sky Unit communication protocol, as discussed in Section I.C.

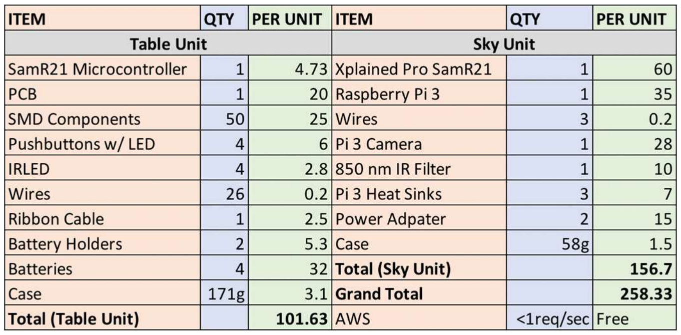


Value	Command
1	Change seat state
2	Change IR LED state

Value	Seat/IR State
0	Vacant/on
1	Occupied/off
≥ 2	Reserved

B. System Costs

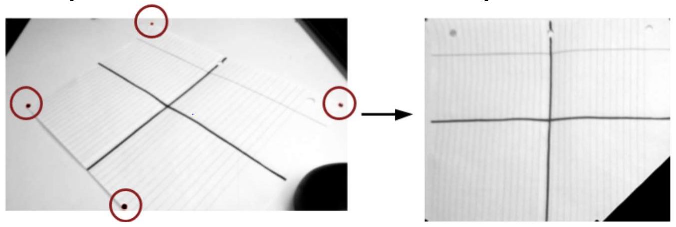
Below are the costs of the system per Table Unit and Sky Unit.



ITEM	QTY	PER UNIT	ITEM	QTY	PER UNIT
Table Unit			Sky Unit		
SamR21 Microcontroller	1	4.73	Xplained Pro SamR21	1	60
PCB	1	20	Raspberry Pi 3	1	35
SMD Components	50	25	Wires	3	0.2
Pushbuttons w/ LED	4	6	Pi 3 Camera	1	28
IRLED	4	2.8	850 nm IR Filter	1	10
Wires	26	0.2	Pi 3 Heat Sinks	3	7
Ribbon Cable	1	2.5	Power Adpater	2	15
Battery Holders	2	5.3	Case	58g	1.5
Batteries	4	32	**Total (Sky Unit)**		**156.7**
Case	171g	3.1	**Grand Total**		**258.33**
Total (Table Unit)		**101.63**	AWS	<1req/sec	Free

C. Table Localization Calibration

Concept for the table localization calibration process.



Left: The paper represents the floor of a room and the black dots represent four known locations. These are detected with the image processing, as circled in red. *Right:* These points are then warped to a 2D perspective using a Perspective Transform in OpenCV.

Acknowledgment

We would like to thank our advisor Professor Tilman Wolf and evaluators Professors Eric Polizzi and Csaba Moritz for their very useful critiques and insightful advice that helped guide our project. We would also like to thank Professors Patrick Kelly and Dennis Goeckel for taking the time to meet with us and discuss potential options for a part of our project. Special thanks to Shira Epstein and M5 for 3D printing resources. Dennis Donoghue would like to thank the creators and users of the website Stack Exchange for boundless advice and help.

References

[1] "IP Rating Chart | DSMT.com," DSMT.com, 2018. [Online]. [Accessed 2017].

[2] "ATSAMR21G18A - Wireless - Wireless Modules," Microchip.com, 2018. [Online]. Available: http://www.microchip.com/wwwproducts/en/ATSAMR21G18A. [Accessed 2017].

[3] "IEEE 802.15.4-2015 - IEEE Standard for Low-Rate Wireless Networks," Standards.ieee.org, 2018. [Online]. Available: https://standards.ieee.org/findstds/standard/802.15.4-2015.html.

[4] "Atmel Lightweight Mesh - Atmel Lightweight Mesh | Microchip Technology Inc.", *Microchip.com*, 2018. [Online]. Available: http://www.microchip.com/DevelopmentTools/ProductDetails.aspx?PartNO=Atmel%20Lightweight%20Mesh. [Accessed: 2018].

[5] "Raspberry Pi 3 Model B," www.rs-components.com, 2018. [Online]. Available: http://docs-europe.electrocomponents.com/webdocs/14ba/0900766b814ba5fd.pdf.

[6] "Pi NoIR Camera V2 - Raspberry Pi," Raspberry Pi, 2018. [Online]. Available: https://www.raspberrypi.org/products/pi-noir-camera-v2/. [Accessed 2018].

[7] "boto/boto3 Repository," GitHub, 2018. [Online]. Available: https://github.com/boto/boto3. [Accessed 2018].

[8] "AWS SDK for Python," Amazon Web Services, Inc., 2018. [Online]. Available: https://aws.amazon.com/sdk-for-python/.

[9] "Capturing Table Activity with DynamoDB Streams - Amazon DynamoDB", *Docs.aws.amazon.com*, 2018. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html. [Accessed: 18].

[10] "Amazon DynamoDB Product Details - Amazon Web Services," Amazon Web Services, Inc., 2018. [Online]. Available: https://aws.amazon.com/dynamodb/details/. [Accessed 2017].

[11] "Amazon RDS Product Details - Amazon Web Services (AWS)," Amazon Web Services, Inc., 2018. [Online]. Available: https://aws.amazon.com/rds/details. [Accessed 2017].

[12] J. Cross, "Bloop: DynamoDB Modeling — bloop 1.0.0 documentation", *Pythonhosted.org*, 2018. [Online]. Available: https://pythonhosted.org/bloop/#. [Accessed 2018].

[13] "OpenCV-Python Tutorials — OpenCV-Python Tutorials 1 documentation", Opencv-python-tutroals.readthedocs.io, 2018. [Online]. Available: http://opencv-python-tutroals.readthedocs.io/en/stable/py_tutorials/py_tutorials.html. [Accessed: 2018].

[14] A. Rosebrock, "Accessing the Raspberry Pi Camera with OpenCV and Python - PyImageSearch", PyImageSearch, 2018. [Online]. Available: https://www.pyimagesearch.com/2015/03/30/accessing-the-raspberry-pi-camera-with-opencv-and-python/. [Accessed: 2018].