CSC 591/791, ECE 592

Group 6 Course Project

Automated Parking Reinforcement

Submitted 16 April 2025

TEAM MEMBERS

Akshay Kumar Joshi Appanna
Kristopher Stickney
Mohit Hosakatte Niranjana Murthy
Yi Zhang

TEAM MEMBERS CONTRIBUTIONS

Component	Weight	Akshay	Kristopher	Mohit	Yi
High Level Design	10.00%	25%	25%	25%	25%
Algorithm Development	20.00%	25%	25%	25%	25%
Coding	20.00%	25%	25%	25%	25%
Debugging	20.00%	25%	25%	25%	25%
Hardware Setup	15.00%	33%	33%	33%	0%
Cloud Setup	5.00%	0%	0%	0%	100%
Report	10.00%	25%	25%	25%	25%
Aggregate Contribution		**0.250**	**0.250**	**0.250**	**0.250**

Table 1: Team Contributions

INTRODUCTION

Urban areas worldwide face mounting challenges related to parking management, including traffic congestion, inefficient utilization of available spaces, security vulnerabilities, and outdated manual billing practices. Traditional parking systems often result in significant frustration for users due to prolonged wait times, unauthorized vehicle access, limited real-time information, and cumbersome billing procedures prone to human errors. These inefficiencies not only inconvenience users but also lead to operational bottlenecks and reduced profitability for parking management entities.

Recognizing these persistent issues, our team developed an innovative Automated Parking Reinforcement system leveraging advanced Internet of Things (IoT) technologies. Our solution integrates RFID-based vehicle identification systems, ultrasonic sensors, servo-controlled gates, and a robust cloud computing infrastructure to streamline the parking experience comprehensively. By employing RFID tags for automatic and secure vehicle identification, our system drastically reduces unauthorized access and enhances security through precise and verifiable entry and exit logs.

The core of our system lies in real-time cloud-based monitoring, which continuously tracks parking occupancy and vehicle status. This real-time capability allows for immediate updates regarding space availability, effectively reducing congestion and optimizing space utilization. Additionally, automated billing processes triggered by RFID detection streamline payment collection, significantly reducing billing errors and delays associated with traditional manual methods.

Barrier automation, implemented through servo motors controlled by cloud decisions, ensures seamless vehicle entry and exit only upon proper validation, further enhancing security and efficiency. By maintaining comprehensive data logs that include timestamps for all events, our system ensures complete transparency and traceability, critical for security auditing and dispute resolution.

This project underscores the practicality and benefits of deploying IoT-based automated solutions within urban parking infrastructures. Our solution provides significant improvements in operational efficiency, customer satisfaction, and overall management effectiveness, laying a solid foundation for future enhancements and potential widespread adoption in smart city initiatives.

DESIGN

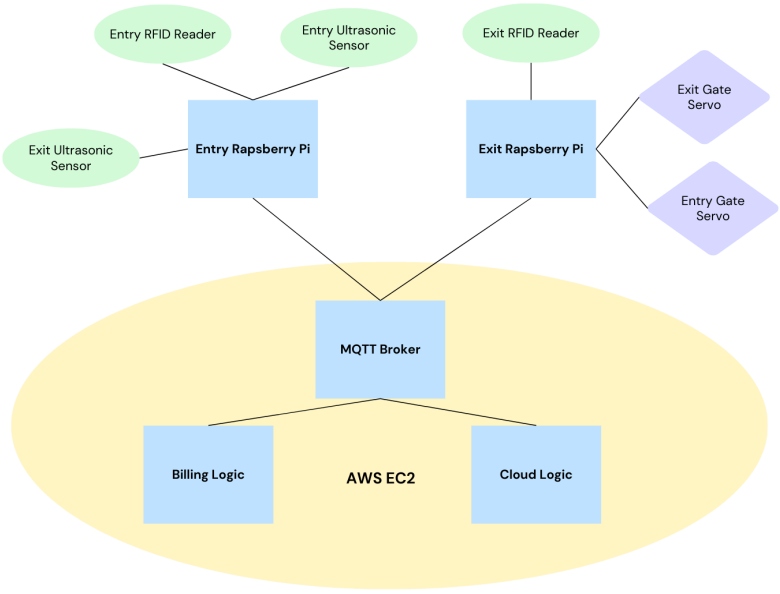

Image 1: System Diagram

The system consists of an entry and exit system, a system to control the gate, and a cloud system.

The entry and exit system consist of a raspberry pi with an RFID reader, an ultrasonic sensor, and a gate. The RFID reader is placed before the gate and senses the car as it drives up to the gate, and the ultrasonic sensor is placed right after the gate to verify if the car actually entered/left the parking lot.

For technical purposes, the exit ultrasonic sensor had to be put on the entry Raspberry Pi and the values ported over via MQTT. In addition, the exit Raspberry Pi is also running the code to control the

actual gate servos, but receives commands from the entry/exit system via MQTT. You can see this in the system diagram.

Cloud

The smart parking system comprises two main cloud components: a cloud-based decision engine (cloud_system.py) running on an AWS EC2 instance and a monitoring client (sub.py) that provides real-time system visualization. These components communicate via an MQTT broker (Mosquitto) to manage parking lot operations efficiently. The system enforces strict business rules for vehicle access while maintaining comprehensive records of all parking events.

The cloud system serves as the central decision-making hub of the parking solution. It receives vehicle UIDs from entry and exit RFID systems through MQTT topics and applies business logic to determine access permissions. For entry requests, the system verifies two conditions: parking lot capacity (rejecting requests when occupancy reaches the 4-vehicle limit) and vehicle state (preventing duplicate entries). Exit requests are validated by checking whether the vehicle has a corresponding entry record. The cloud maintains real-time state tracking through a vehicle dictionary that records check-in/check-out times and current status (OUT_LOT, IN_LOT, etc.). All state changes trigger system status updates published to MQTT, including current occupancy, capacity, and timestamps in EDT timezone.

Beyond access control, the cloud system handles comprehensive event processing. It timestamps and logs all critical operations, including entry attempts, successful transactions, and gate operations. The billing module automatically calculates parking fees using a tiered pricing model ($2 for the first hour plus $1 for each additional 30-minute interval). Each exit transaction generates a detailed billing record containing entry/exit times, duration, calculated charge, and session IDs. These records are published to the billing/transactions topic while maintaining transaction integrity through locking mechanisms that prevent duplicate processing.

Client

The monitoring client subscribes to all system topics, providing operators with a real-time dashboard of parking activities. It displays human-readable event streams with EDT timestamps, including gate operations, vehicle movements, and system alerts. The client implements special handling for billing transactions, formatting receipts with clear breakdowns of parking durations and charges while detecting and ignoring duplicate messages. A rolling buffer maintains the 100 most recent events for immediate review, accessible through simple console commands.

The component interaction follows a well-defined sequence: Entry systems first request access approval from the cloud, which responds with permission grants or denials based on current conditions. Upon successful entry or exit, validation requests trigger cloud state updates and subsequent status publications. The monitoring client observes these transactions, presenting them to operators while maintaining historical records. This decoupled architecture allows each component to specialize in its domain while ensuring system-wide consistency through MQTT's publish-subscribe model.

System Flow

As seen in Image 2, an RFID tag is sensed at the entrance gate. The system sends the UID to the broker which sends it to the cloud system. The cloud system will determine if the car can enter or not and send the decision back. If the car cannot enter, nothing will happen. If the car can enter, the system will send an MQTT message to the entry gate to open. The gate system opens the gate. The ultrasonic sensor waits for the car to enter its field of view for at least a specified length of time and then leave its view for at least that same amount. If this happens, the entry system will send the UID to the cloud system via MQTT on a special topic which denotes that that car entered successfully. If the car does not successfully enter, the entry system will timeout and reset the gate. Since no entry verification is sent, the car is considered as not having entered.

The exit system works the same way as the entry gate, just for exiting. But once the verification is sent that the car actually left the parking lot (via sensing by the ultrasonic sensor), the cloud system will put a bill out for that car (which would be sent to the car's owner or specified user) which specifies the entrance and exit timestamps, the amount of time spent in the parking lot, and the associated cost.

System Flow Diagrams

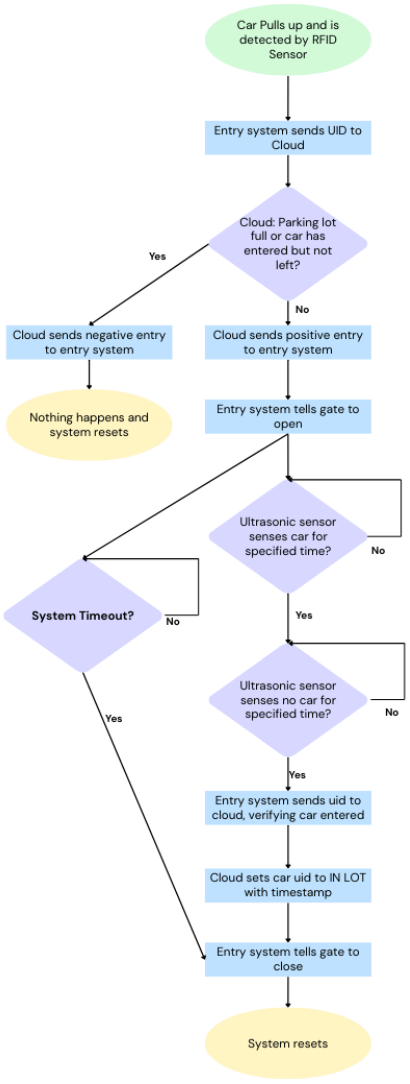

Image 2: Entry Flow Graph

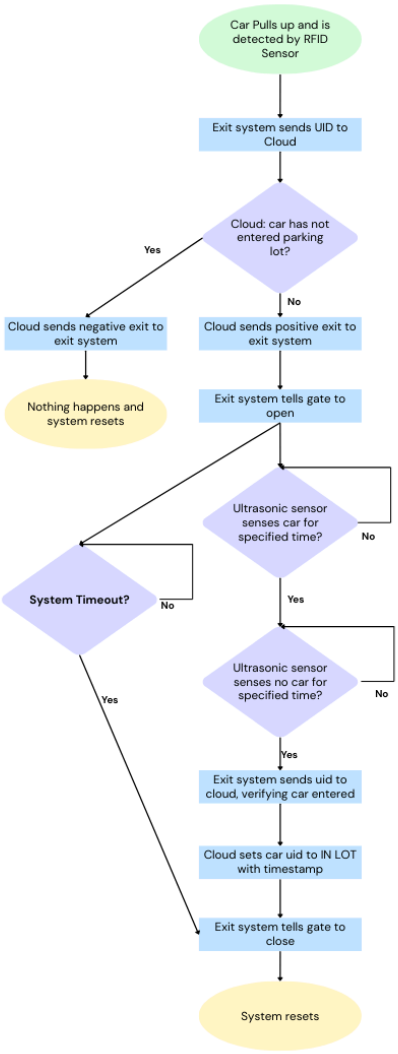

Image 3: Exit Flow Graph

IMPLEMENTATION

We decided to write all our code in Python since it is a very simple language to easily prototype, and there were bindings for all the sensors, motors, and protocols we wanted to use.

We used an AWS EC2 with a Ubuntu 22.04 image to run our cloud services since it is very straightforward and easy to set up and configure an instance, as well as it is easy to set up new ones in case of failure. We also used Ubuntu because development is easier on it and the tools we used, such as Python and MQTT, were easily installed or configured.

We chose to run Mosquitto MQTT on the EC2 instance because it is a well-known, well-documented, and well-established MQTT broker and it has worked well for us previous projects.

The Raspberry Pis are running Raspberry Pi's Bullseye operating system. This OS interacts with the GPIO pins the way we needed them to.

For the RFID readers, we chose the MRF522. These are cheap and readily available readers, and they also are well documented and have been used in many Raspberry Pi projects and have a good Python library.

For the ultrasonic sensors, we went with the HC-S04 sensors for the same reasons and the RFID readers–they are cheap, readily available, are well used and documented, and have a good Python library.

We ended up putting the entry and exit RFID readers on two separate Raspberry Pi's due to the fact that having both on the same Pi was causing trouble.

In addition, the ultrasonic sensors would not work on the Raspberry Pi which controlled the exit system, so we put it on the entry Pi and sent the values via MQTT.

Source Files

Cloud_system.py
This file is the core logic engine that manages parking occupancy, validates vehicle access, and generates billing based on session data.

sub.py
A real-time MQTT subscriber client that monitors and displays gate activity, vehicle movement, billing receipts, and system status updates.

cloud_mqtt_client.py
An MQTT interface layer that connects the cloud system logic to the broker, handling topic subscriptions, message routing, and publishing.

servo_motor.py
An MQTT client that controls the entry and exit gate servo motors. Receives commands from publishers and opens and closes the gates accordingly.

Sensor_system_entry.py
The logic for the entry system. It sends the RFID tag to the cloud and opens and closes the entry gate according to the decision received. It also determines if the car actually entered the parking lot or not. It reads from the RFID reader as well as both ultrasonic sensors. It sends data from the RFID reader and the exit ultrasonic sensor to the MQTT broker and receives feedback from the cloud system.

sensor_system_exit.py
The logic for the exit system. It sends the RFID tag to the cloud and opens and closes the exit gate accordingly. It also determines if the car actually exited the lot. It receives data from the cloud system as well as the exit ultrasonic sensor, which had to be attached to the entry system.

Schematics

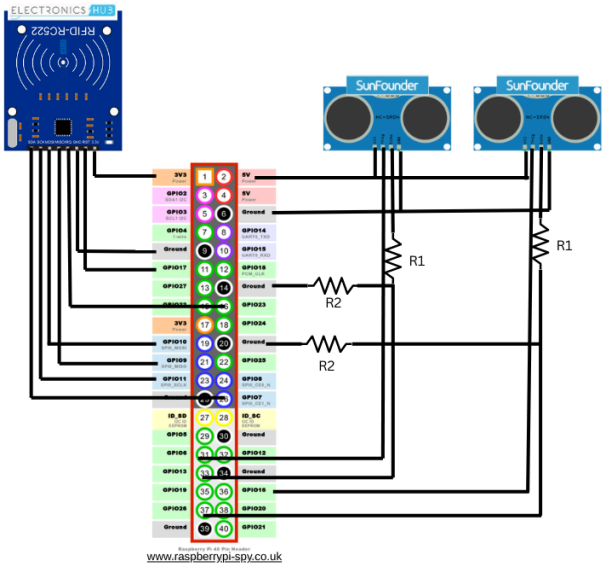

Diagram 1: Entry RPi Schematic

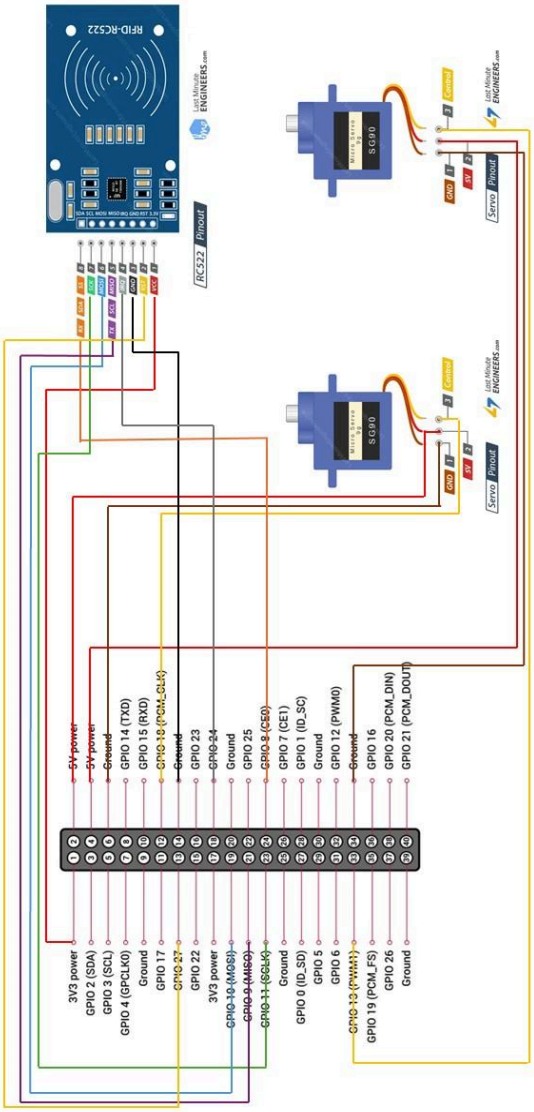

Diagram 2: Exit RPi Schematic

RESULTS AND DISCUSSION

During the implementation and testing phases, our Automated Parking Reinforcement system demonstrated robust performance and met most of the intended operational goals. Real-time cloud-based monitoring significantly improved space utilization by providing accurate and timely updates on parking space availability.

The RFID-based access control proved to be highly reliable, correctly identifying authorized vehicles and preventing unauthorized access, thereby substantially improving overall security. The automated billing system worked seamlessly, accurately calculating parking fees based on real-time occupancy data and significantly reducing the billing errors and delays typically associated with manual processes.

However, several challenges were encountered during the deployment phase. Initially, integrating multiple RFID readers on a single Raspberry Pi caused performance issues, prompting us to deploy separate Raspberry Pi units for entry and exit systems. Additionally, the ultrasonic sensors faced compatibility issues on the exit Raspberry Pi, requiring us to reroute data via MQTT from the entry Raspberry Pi, highlighting the need for careful hardware compatibility considerations in future designs.

Another area identified for improvement was the robustness of the MQTT broker. While Mosquitto performed reliably, ensuring redundancy and failover capabilities in a production environment would be essential for maintaining continuous operations.

Future Improvements

Several key features would significantly enhance the current system:

1. **Scalability:** Enhancing system scalability to efficiently handle an increased number of vehicles simultaneously. This could involve incorporating load balancing and distributed computing techniques.
2. **Redundancy and Reliability:** Implementing redundancy mechanisms for critical hardware and software components, such as MQTT brokers and cloud servers, to ensure continuous operation and data integrity.
3. **Security Enhancements:** Strengthening security through advanced encryption protocols and multi-factor authentication to protect against unauthorized access and cyber threats.

4. **Advanced Analytics:** Integrating advanced analytics and predictive modeling capabilities to further optimize parking space usage, forecast peak usage periods, and dynamically manage resources.
5. **Manual Override Function:** Introducing a manual override feature is necessary for scenarios where RFID scans or automated processes fail. This capability would allow authorized personnel to manually control gate operations, ensuring system reliability and uninterrupted service. Currently, this feature is not implemented and should be prioritized for future development.

RELATED WORKS AND REFERENCES

"Use an RFID reader with the Raspberry Pi.": https://howtoraspberrypi.com/rfid-raspberry-pi-2

"HC-SR04 Ultrasonic Range Sensor on the Raspberry Pi": https://thepihut.com/blogs/raspberry-pi-tutorials/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi

"Parking Lot Finder App Pitch Deck": https://slidesgo.com/theme/parking-lot-finder-app-pitch-deck

"How to Control Servo Motors with a Raspberry Pi": https://www.digikey.com/en/maker/tutorials/2021/how-to-control-servo-motors-with-a-raspberry-pi#:~:text=To%20make%20a%20Raspberry%20Pi,the%20power%20supply%20as%20well.

"Technical Explanation for Servomotors and Servo Drives": https://www.ia.omron.com/data_pdf/guide/14/servo_tg_e_1_1.pdf