**INTERNET OF THINGS (IOT)**

**SMART PARKING**

**PHASE 1 : PROJECT DEFINITION AND DESIGN THINKING**

**PROJECT DEFINITION:**

**Smart parking in IoT (Internet of Things)** refers to the use of interconnected devices and technology to efficiently manage and optimize parking spaces. It involves the integration of various sensors, data collection systems, and communication networks to provide real-time information about parking availability, thereby enhancing the overall parking experience for both drivers and parking facility operators.

Key components of a smart parking system in IoT typically include:

* **Parking Sensors:** These are deployed in individual parking spaces to detect the presence or absence of vehicles. Common types of sensors include ultrasonic sensors, magnetic sensors, or infrared sensors. They collect data on parking space occupancy.
* **Data Communication:** Parking sensors are connected to a network infrastructure, often using wireless technologies like Wi-Fi, Bluetooth, LoRaWAN, or cellular networks. This allows for real-time data transmission to a central server or cloud platform.
* **Data Processing and Analysis:** The data collected from parking sensors is processed and analyzed to determine parking space availability, occupancy trends, and other relevant information. Machine learning and predictive algorithms may be used to optimize parking management.
* **User Interfaces:** Mobile apps, websites, or digital signage are used to provide drivers with real-time information about available parking spaces and directions to them. This enables drivers to make informed decisions about where to park.
* **Payment Systems:** Smart parking systems often include digital payment options, allowing users to pay for parking using mobile apps or contactless payment methods.
* **Parking Enforcement:** Some systems may integrate with enforcement tools, such as license plate recognition cameras, to monitor parking duration and enforce parking rules.
* **Analytics and Reporting:** Parking facility operators can use data analytics to gain insights into parking utilization, revenue generation, and operational efficiency. This information can help in making informed decisions to improve parking management.

Benefits of smart parking in IoT include reduced traffic congestion, decreased fuel consumption, improved air quality, increased revenue for parking operators, and enhanced convenience for drivers. It also contributes to the overall development of smart cities by optimizing urban mobility and reducing the environmental impact of urban transportation.

In summary, smart parking in IoT leverages interconnected devices and data-driven technologies to optimize parking space usage, enhance user experience, and contribute to more sustainable and efficient urban environments.

**DESIGN THINKING:**

**1.Project Objectives: Define specific objectives such as real-time parking space monitoring, mobile app integration, and efficient parking guidance.**

**Real-time Parking Space Monitoring**:

* Implement a system that continuously monitors the availability of

parking spaces in a designated area.

* Utilize sensors, cameras, or other technologies to provide real-time

data on vacant and occupied parking spaces.

* Ensure accuracy and reliability of data to minimize false notifications.

**Mobile App Integration:**

* Develop a user-friendly mobile application that allows drivers to

access real-time parking information.

* Enable users to search for available parking spaces nearby or at a

specific location.

* Integrate navigation features to guide users to their selected parking

space.

* Provide reservation and payment options through the mobile app for

added convenience.

**Efficient Parking Guidance:**

* Implement a dynamic parking guidance system that directs drivers to

the nearest available parking spaces.

* Utilize digital signage, mobile app notifications, and in-car navigation

to guide drivers effectively.

* Optimize traffic flow within the parking facility to reduce congestion

and save time for users.

**2. IoT Sensor Design: Plan the design and deployment of IoT sensors in parking spaces to detect occupancy and availability.**

Designing and deploying IoT sensors to detect parking space occupancy and availability involves

several steps, including hardware selection, sensor configuration, data transmission, and data

processing. Here is the high-level plan for designing and deploying IoT parking sensors using

Python

**Step 1: Hardware Selection Choose appropriate IoT hardware components for your parking space sensors. Common components include:**

* Ultrasonic distance sensors or PIR (Passive Infrared) sensors for occupancy detection.
* Microcontrollers like Raspberry Pi, Arduino, or ESP8266/ESP32 for dataprocessing.
* Wi-Fi or cellular modules for data connectivity.
* Power source (batteries, solar panels, or wired power).

**Step 2: Sensor Configuration Write Python code to configure and read data from the selected sensors. Below is an example using a Raspberry Pi and an ultrasonic distance sensor to detect occupancy:**

Isensor configuration

import RPi.GPIO as GPIO

import time

**# Set GPIO mode and pins**

GPIO.setmode(GPIO.BOARD)

trigger\_pin = 11

echo\_pin = 13

GPIO.setup(trigger\_pin, GPIO.OUT)

GPIO.setup(echo\_pin, GPIO.IN)

def distance\_measurement():

**# Send a short pulse to trigger the sensor**

GPIO.output(trigger\_pin, True)

time.sleep(0.00001)

GPIO.output(trigger\_pin, False)

start\_time = time.time()

stop\_time = time.time()

**# Record the start time when the echo pin goes high**

while GPIO.input(echo\_pin) == 0:

start\_time = time.time()

**# Record the stop time when the echo pin goes low**

while GPIO.input(echo\_pin) == 1:

stop\_time = time.time()

**# Calculate distance based on the time it took for the pulse to return**

elapsed\_time = stop\_time - start\_time

distance = (elapsed\_time \* 34300) / 2 # Speed of sound is 343 m/s

return distance

try:

while True:

distance = distance\_measurement()

if distance &lt; 30: # Adjust this threshold for your parking space

print(&quot;Parking space occupied&quot;)

else:

print(&quot;Parking space available&quot;)

time.sleep(2) # Adjust the interval as needed

except KeyboardInterrupt:

GPIO.cleanup()

**Step 3:**

Data Transmission For data transmission, you can use Wi-Fi or cellular connectivity to

send occupancy data to a central server or cloud platform. Python libraries like requests or

MQTT libraries can be used for this purpose.

**Step 4:**

Data Processing On the central server or cloud platform, you can use Python to

process and store the sensor data. You may use frameworks like Flask or Django for building a

server or cloud functions.

**Step 5:**

Data Visualization and Notification Develop a Python-based dashboard or mobile app

to visualize parking space availability in real-time. Send notifications to users through email,

SMS, or a mobile app when parking spaces are available or occupied.

Remember to handle data security and privacy concerns, and consider power management for

long-term deployments of IoT sensors, especially if using battery power.

This plan provides a high-level overview of designing and deploying IoT parking sensors with

Python. Detailed implementation may vary depending on the specific hardware and software

requirements of your project.

**3. Real-Time Transit Information Platform: Design a mobile app interface that displays real-time parking availability to users.**

from kivy.app import App

from kivy.uix.boxlayout import BoxLayout

from kivy.uix.label import Label

from kivy.uix.button import Button

class ParkingApp(App):

def build(self):

**# Create the main layout**

layout = BoxLayout(orientation='vertical', padding=10, spacing=10)

**# Add a label for the app title**

title\_label = Label(text="Real-Time Parking Availability")

layout.add\_widget(title\_label)

**# Add a button to refresh parking data**

refresh\_button = Button(text="Refresh Parking Availability")

refresh\_button.bind(on\_press=self.refresh\_parking)

layout.add\_widget(refresh\_button)

**# Add a label to display parking availability**

self.availability\_label = Label(text="Available Parking Spaces: 0")

layout.add\_widget(self.availability\_label)

return layout

def refresh\_parking(self, instance):

**# In a real application, you would fetch real-time parking data here**

available\_spaces = 10 # Replace with actual data retrieval

self.availability\_label.text = f"Available Parking Spaces: {available\_spaces}"

if \_\_name\_\_ == '\_\_main\_\_':

ParkingApp().run()

In this code:

* We create a basic Kivy app with a vertical layout.
* The app displays a title label, a button to refresh parking data (which currently simulates the data), and a label to display the parking availability.
* When the "Refresh Parking Availability" button is pressed, it updates the availability label with the current availability (simulated in this example).

**4.Integration Approach: Determine how Raspberry Pi will collect data from sensors and update the mobile app.**

To integrate Raspberry Pi with sensors and update a mobile app, you&#39;ll need a multi-step

approach involving hardware connections, sensor data processing, and communication with

the mobile app. Here&#39;s a step-by-step guide:

Hardware Connections:

**Connect Sensors to Raspberry Pi:**

* Connect the sensors (e.g., ultrasonic sensors for parking space monitoring) to the GPIO pins of the Raspberry Pi.
* Ensure you have the necessary resistors, power supply, and wiring for the sensors.

**Power Supply:**

* Provide power to the Raspberry Pi, either through a micro USB power adapter or

alternative power source like a battery or solar panel, depending on your deployment

environment.

**Write Python Code for Sensor Data Collection:**

* Develop Python scripts to read data from the connected sensors. Use appropriate

libraries for the specific sensors you&#39;re using.

**pythonCopy code**

import RPi.GPIO as GPIO

import time

# Sensor initialization and data collection logic here

try:

while

True: # Read sensor data # Process data as needed except Keyboard

Interrupt: GPIO.cleanup()

**Data Processing and Logic:**

* Process the sensor data based on your requirements. For parking availability, you

might define thresholds (e.g., distance &lt; 30cm indicates occupied space).

Integration with Mobile App:

**Data Transmission:**

* Set up a communication protocol to send sensor data from the Raspberry Pi to your

backend server. Options include HTTP requests, MQTT, WebSocket, etc.

**pythonCopy code**

import requests

# Send data to your backend server

url = &#39;http://your-backend-server-endpoint&#39;

data ={

&#39;parking\_space\_1&#39;: &#39;occupied&#39;, &#39;parking\_space\_2&#39;: &#39;available&#39;

}response = requests.post(url,json=data)

if response.status\_code == 200:

print(&#39;Data sent successfully&#39;)

else:

print(&#39;Error sending

data&#39;)

**Backend Server:**

* Create a backend server (could be hosted on a cloud service or a local server) to

receive and process the data from the Raspberry Pi.

**Data Processing on Backend:**

* On the backend, process the received data, perform any necessary computations or

filtering, and prepare it for sending to the mobile app.

**API for Mobile App:**

* Create an API endpoint on your backend server to which the mobile ap**p can send**

requests to retrieve real-time parking availability data.

**Mobile App Integration:**

* In your mobile app, use the appropriate SDK or libraries (e.g., Retrofit for Android,

Alamofire for iOS) to make HTTP requests to the API endpoint on the backend server.

**Update Mobile App UI:**

* Once the mobile app receives the data from the backend server, update the UI to

display the real-time parking availability information.

**Security and Error Handling:**

* Implement security measures such as encryption and authentication to protect the data intransit.
* Add error handling mechanisms on both the Raspberry Pi and backend server to handle network failures, sensor malfunctions, and other potential issues.

**Continuous Monitoring and Maintenance:**

* Implement logging and monitoring systems to keep track of sensor data and system health.
* Regularly test and maintain the hardware components to ensure they&#39;re functioning properly.