**Naan Mudhalvan**

**Public Transportation Optimization**

**Phase-5**

**Problem definition:**

The objective of this project is to enhance the efficiency, reliability, and user experience of public transportation through the implementation of Internet of Things (IoT) technologies. By leveraging real-time data and smart devices, the aim is to optimize various aspects of public transport systems, including route planning, vehicle tracking, passenger experience, and overall operational efficiency.

**Detailed Explanation:**

* **Real-time Vehicle Tracking:** Implement GPS and IoT sensors on public transport vehicles to provide real-time location tracking. This allows for accurate arrival predictions.
* **Passenger Information Systems:** Develop a system to provide passengers with real-time information about bus/train locations, expected arrival times, and any delays. This can be accessible through mobile apps, digital displays at stops, or other communication channels.
* **Occupancy Monitoring:** Use sensors to monitor the occupancy of vehicles in real time. This information can be used to optimize routes, allocate resources efficiently, and enhance the overall passenger experience.

**Design thinking:**

**Project objectives:**

* To reduce delays, improves route efficiency, and provides real-time information, collectively reducing the overall travel time for passengers.

**IoT sensor design:**

* Generate ideas for a system that incorporates IR sensors, GPS module, Wi-Fi module, and an LCD interface.
* Create a physical prototype, connecting IR sensors for obstacle detection, GPS module for location tracking, and Wi-Fi module for data transmission.

**Real time transit Information platform:**

* The web-based real-time transit information platform aims to provide a comprehensive, user-friendly, and secure experience for passengers. Continuous improvement through iterative testing and user feedback ensures that the platform remains responsive to changing needs and technological advancements.

**Integration approach:**

* ThingSpeak is a popular Internet of Things (IoT) platform that allows users to easily build and control IoT projects through a user-friendly mobile app. It provides a simple way to connect various hardware devices, sensors, and microcontrollers to the internet and control them remotely.

**Work Flow:**

* When system is powered up, it goes through initializing phase during which it sets the baud rate for connected devices communicating a Universal Asynchronous Received Transmitted (UART) serial connected devices such as Wi-Fi module, GPS, and serial monitor as well initializes the LCD and DHT sensor.
* After that microcontroller scans all the sensors and reads both analogue and digital sensors.
* The microcontroller then performs analogue to digital conversion for all analogue read sensors. It then processes the data read and computes passenger count, calculate bus speed, compute GPS coordinates and passenger count.
* All computed parameters are displayed on the LCD locally.
* The same values are also sent to serial monitor for testing and debugging purpose.
* These parameters are sent to the ThingSpeak cloud over the internet using ESP32 module.

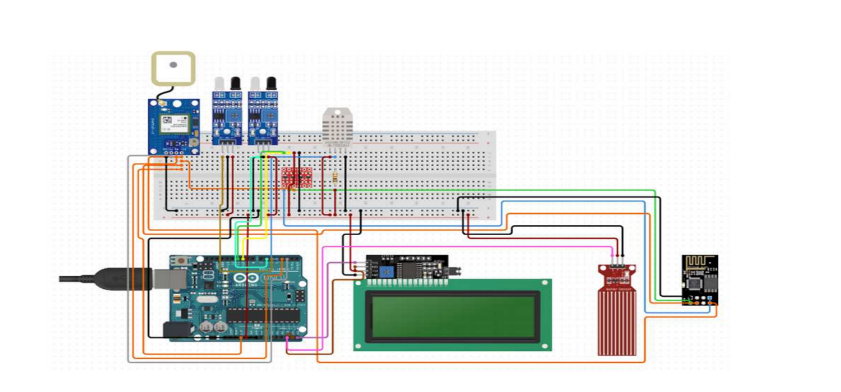
The data sent to cloud is fetched from the cloud and displayed in a web browser for user visualization. In the next step data is fetch from the cloud using channel ID and API key which is then analyzed using the MATLAB.

**Plan of action:**

The project focuses on the design and implementation of a comprehensive system for optimizing public transport, integrating hardware and software components. The system design diagram includes sensors, communication modules, and the ThingSpeak cloud.

1. **Hardware Implementation:**

In this project we use 2 Ultrasonic sensors which are connected to digital pins on the Arduino Uno. The Ultrasonic sensors count passengers entering and exiting the transport, providing essential data for optimizing Public Transport operations. The output pin of LM393 module is connected to the digital pin of the Arduino, which is used to determine the speed of the transport, providing crucial information for route optimization and scheduling. The NEO 6M GPS module is connected to the software serial ports on the Arduino Uno. Although, the recommended operating voltage is 3.3V but it is tolerant to 5V making it easier to connect to Arduino without using logic level converter. The module has four pins: GND, Vcc for powering chip, TxD (Transmitter) and RxD (Receiver) pins are used for serial communication. The Arduino pins 2 and 3 are configured as software serial receiver and transmitter respectively using software serial library. For local display of the data in the bus, we used a 20x4 Character I2C LCD, which is connected to microcontroller using only four wires instead of many thus simplifying the wiring. The four pins used are VCC, GND, SDA, and SCL. SDA and SCL are the serial data and the clock pins, respectively. There is a POT on the I2C Module. We can control the contrast of the LCD display by rotating this POT.

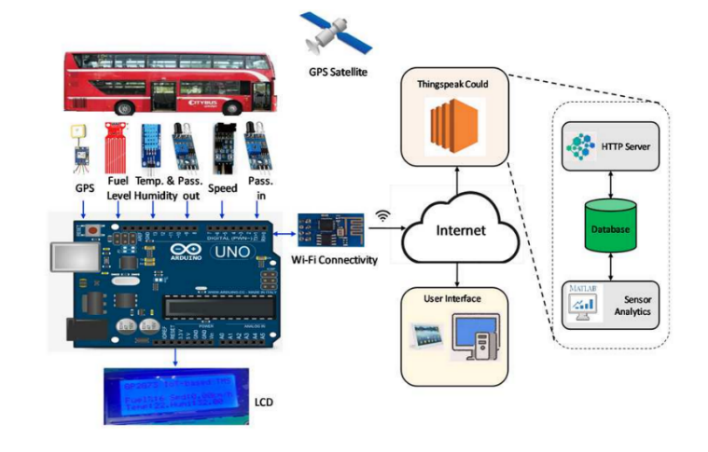


1. **Data transmission from Arduino to Wi-Fi module:**

The system is connected to the internet using ESP32 module, a low-cost Wi-Fi module that has an in-built microcontroller and a 1MB flash. The TCP/IP protocol stack allows the module to communicate with Wi-Fi signals. The module is connected to microcontroller using soft serial UART having a specified baud rate. Microcontroller communicates with the Wi-Fi module using a set of AT commands. On the Arduino side, the Serial library can be used to communicate over the hardware serial pins (usually pins 0 and 1). On the ESP32 side, Arduino libraries or the ESP32 core for Arduino can be used to receive data from the Arduino. The maximum working voltage of the module is 3.3 V and it has total of six pins: 3.3 V Power Pin, Ground Pin, Active Low Reset Pin, Active High Enable Pin, Serial Transmit Pin of UART, and Serial Receive Pin of UART.

1. **From Wi-Fi module to Real time Transit Information Platform:**

To link data from Wi-Fi module to a real time transit information platform, we select ThingSpeak platform. Then we configure ESP32 Wi-Fi module to connect to the chosen platform and provide the necessary credentials to establish secure connection. Next Program the ESP32 to collect relevant data from sensors (e.g., passenger count, GPS location) and transmit it to the IoT platform at regular intervals using protocols like HTTP or MQTT.

****

**Deployed IOT devices:**

1. **GPS Module**

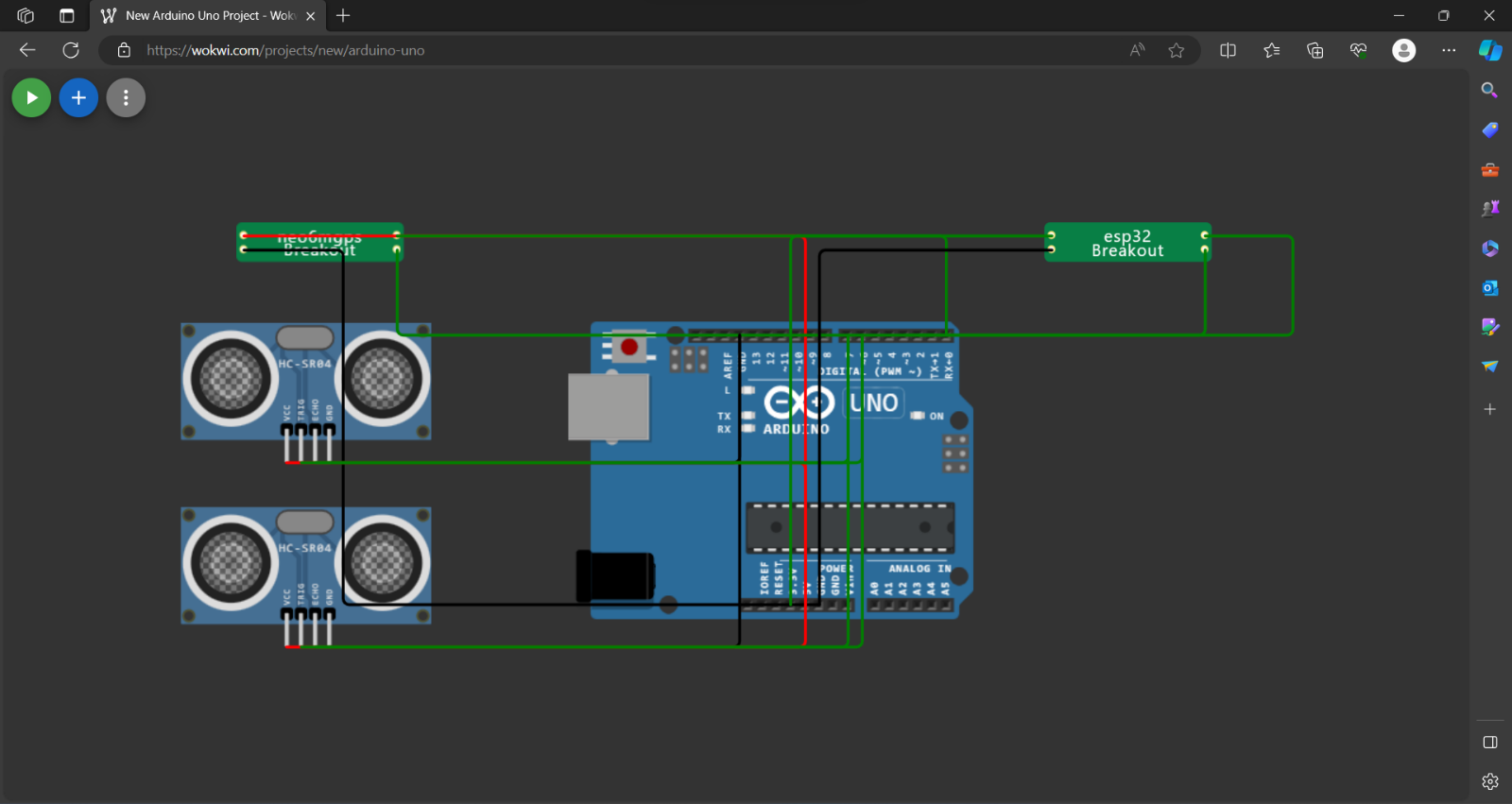
GPS (Global Position System) used for positioning and tracking buses based on satellite communication. GPS satellites cover the entire earth at all times. To get accurate GPS location data, there should be a minimum of three satellites. The NEO-6M GPS module used in the proposed system is small and works on very low power, making it ideal for tracking applications. The GPS module operates at 3.3 V, as a result, powered by connecting the GPS module to the 3.3 V pin of the ESP32.

1. **ESP32 Microcontroller**

The ESP32 is a microcontroller with a Wi-Fi module, an open-source IoT platform that is characterized by low-cost and low-power system-on-a-chip (SOC). An ESP32 has a dual-core structure and internal modules such as Wi-Fi, Bluetooth, and many Peripheral Interfaces such as IR, SPI, CAN, Ethernet, and temperature sensors

1. **Ultrasonic Sensor**

Ultrasonic sensors are commonly used with Arduino to measure distance by sending and receiving ultrasonic waves. Connect the VCC pin of the ultrasonic sensor to the 5V pin on the Arduino. Connect the GND pin of the ultrasonic sensor to the GND pin on the Arduino. Connect the TRIG pin of the ultrasonic sensor to a digital pin (e.g., Pin 7) on the Arduino. Connect the ECHO pin of the ultrasonic sensor to another digital pin (e.g., Pin 6) on the Arduino.



**ARDUINO CODE FOR ESP32 MICROCONTROLLER:**

This code enables an ESP32 to interact with Blynk, read distances from two ultrasonic sensors, and update the Blynk application with information on people entering and leaving a defined area. It also controls an LED based on these conditions. The distances are measured using the Haversine formula, which is common for calculating distances between latitude and longitude coordinates.

#define BLYNK\_TEMPLATE\_ID "TMPL26V4fGv5q"

#define BLYNK\_TEMPLATE\_NAME "Test"

#define BLYNK\_AUTH\_TOKEN "XEHxNF\_Ur1Nt2p7wB5B20dNI1ZUwj34P"

#include <WiFi.h>

#include <WiFiClient.h>

#include <BlynkSimpleEsp32.h>

int duration1 = 0;

int distance1 = 0;

int duration2 = 0;

int distance2 = 0;

int dis1 = 0;

int dis2 = 0;

int dis\_new1 = 0;

int dis\_new2 = 0;

int entered = 0;

int left = 0;

int inside = 0;

#define LED 2

#define PIN\_TRIG1 15

#define PIN\_ECHO1 14

#define PIN\_TRIG2 13

#define PIN\_ECHO2 12

BlynkTimer timer;

char auth[] = BLYNK\_AUTH\_TOKEN;

char ssid[] = "Wokwi-GUEST";   // your network SSID (name)

char pass[] = "";

#define BLYNK\_PRINT **Serial**

long get\_distance1() {

  // Start a new measurement:

  digitalWrite(PIN\_TRIG1, HIGH);

  delayMicroseconds(10);

  digitalWrite(PIN\_TRIG1, LOW);

  // Read the result:

  duration1 = pulseIn(PIN\_ECHO1, HIGH);

  distance1 = duration1 / 58;

  return distance1;

}

long get\_distance2() {

  // Start a new measurement:

  digitalWrite(PIN\_TRIG2, HIGH);

  delayMicroseconds(10);

  digitalWrite(PIN\_TRIG2, LOW);

  // Read the result:

  duration2 = pulseIn(PIN\_ECHO2, HIGH);

  distance2 = duration2 / 58;

  return distance2;

}

int count = 0;

void myTimer() {

  dis\_new1 = get\_distance1();

  dis\_new2 = get\_distance2();

  if (dis\_new1<100){

    count++;

**Serial**.println("Number of passengers inside the bus:");

**Serial**.println(count);

  }

  if (dis\_new2<100){

    if(count<=0){

      count=0;

    }

    else{

      count--;

    }

**Serial**.println("Number of passengers inside the bus:");

**Serial**.println(count);

  }

}

 void setup() {

**Serial**.begin(115200);

  pinMode(LED, OUTPUT);

  pinMode(PIN\_TRIG1, OUTPUT);

  pinMode(PIN\_ECHO1, INPUT);

  pinMode(PIN\_TRIG2, OUTPUT);

  pinMode(PIN\_ECHO2, INPUT);

  Blynk.begin(auth, ssid, pass, "blynk.cloud", 8080);

  timer.setInterval(1000L, myTimer);

}

void loop() {

  Blynk.run();

  timer.run();

}

**DISTANCE CALCULATION:**

The Haversine formula was adopted to calculate the distance that will appear in the Android app. It calculates the distance between the passenger and the bus location using the latitude and longitude of the bus and the passenger who is at home, work, or at the bus stop. The following equations can be used to calculate the distance.

a=sin^2(x3/2) +cos(x1). cos(x2). sin^2(y3/2)

c= 2. atan2(sqrt(a). sqrt(1-a))

d= R\*c

where,

x=latitude;

y=longitude;

x3= latitude2 – latidue1;

y3= longitude2 – longitude1;

R = Earth radius (6400 km);

d = distance between two locations;

**Python Script:**

import math

def haversine (lat1, lon1, lat2, lon2):

    # Convert latitude and longitude from degrees to radians

    lat1 = math. radians(lat1)

    lon1 = math.radians(lon1)

    lat2 = math.radians(lat2)

    lon2 = math.radians(lon2)

    # Differences in latitude and longitude

    dlat = lat2 - lat1

    dlon = lon2 - lon1

    # Haversine formula

    a = math.sin(dlat/2)\*\*2 + math.cos(lat1) \* math.cos(lat2) \* math.sin(dlon/2)\*\*2

    c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1-a))

    # Radius of the Earth (in kilometers)

    R = 6371

    # Calculate the distance

    distance = R \* c

    return distance

# Example Usage

latitude1 = 40.7128  # Latitude of point 1 (in degrees)

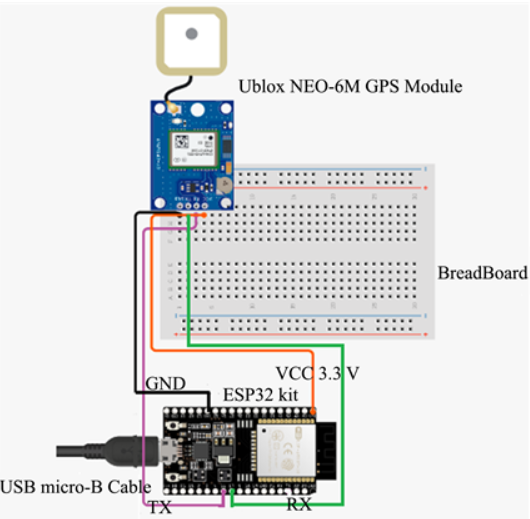
longitude1 = -74.0060  # Longitude of point 1 (in degrees)

latitude2 = 34.0522  # Latitude of point 2 (in degrees)

longitude2 = -118.2437  # Longitude of point 2 (in degrees)

distance = haversine(latitude1, longitude1, latitude2, longitude2)

print(f"The distance between the two locations is approximately {distance} kilometers.")



**ARRIVAL TIME CALCULATION:**

**Python script:**

class ArrivalTimeCalculator:

    def \_\_init\_\_(self):

        self.prev\_coordinate = (0, 0)  # Initialize previous coordinate to (0, 0)

    def calculate\_arrival\_time(self, distance, average\_speed):

        # Calculate arrival time in minutes

        arrival\_time = (distance / average\_speed) \* 60

        return arrival\_time

    def process\_new\_coordinates(self, new\_coordinate, distance, average\_speed):

        if self.prev\_coordinate == new\_coordinate:

            # Start timer and calculate delay time

            delay\_time = self.start\_timer\_and\_get\_delay()

            # Calculate final arrival time

            final\_arrival\_time = self.calculate\_arrival\_time(distance, average\_speed) + delay\_time

            self.prev\_coordinate = new\_coordinate  # Update prev coordinate

            return final\_arrival\_time

        else:

            # Process new coordinates and calculate estimated arrival time

            estimated\_arrival\_time = self.calculate\_arrival\_time(distance, average\_speed)

            # Update prev coordinate

            self.prev\_coordinate = new\_coordinate

            return estimated\_arrival\_time

    def start\_timer\_and\_get\_delay(self):

        # Placeholder function for starting timer and getting delay time

        # You should implement a timer function based on your specific environment

        delay\_time = 5  # Placeholder value, replace with actual delay time calculation

        return delay\_time

# Example Usage

arrival\_time\_calculator = ArrivalTimeCalculator()

# Example coordinates

new\_coordinate1 = (12.34, 56.78)  # Example new coordinate (latitude, longitude)

new\_coordinate2 = (12.34, 56.78)  # Example new coordinate (latitude, longitude)

distance = 10  # Example distance in kilometers

average\_speed = 40  # Example average speed in km/h

final\_arrival\_time1 = arrival\_time\_calculator.process\_new\_coordinates(new\_coordinate1, distance, average\_speed)

print(f"The final arrival time for first coordinate is approximately {final\_arrival\_time1} minutes.")

final\_arrival\_time2 = arrival\_time\_calculator.process\_new\_coordinates(new\_coordinate2, distance, average\_speed)

print(f"The final arrival time for second coordinate is approximately {final\_arrival\_time2} minutes.")

**Python script on the IoT sensors to send real-time location and ridership data to the transit information platform:**

import paho.mqtt.client as mqtt

import json

import time

from your\_sensor\_module import get\_real\_sensor\_data

# MQTT broker information

broker\_address = "mqtt.eclipse.org"

broker\_port = 1883

topic = "transit\_data"

def generate\_real\_data():

    location\_data = gps\_module.get\_location\_data()

    ridership\_data = ultrasonic\_sensor\_module.get\_ridership\_data()

    return {

        "location": location\_data,

        "ridership": ridership\_data

    }

# MQTT client setup

client = mqtt.Client()

client.connect(broker\_address, broker\_port, 60)

try:

    while True:

        # Generate real sensor data

        data = generate\_real\_data()

        # Convert data to JSON

        payload = json.dumps(data)

        # Publish data to the topic

        client.publish(topic, payload)

        # Print for verification

        print("Published:", payload)

        # Adjust the frequency based on your requirements

        time.sleep(10)  # Wait for 10 seconds before sending the next data

except KeyboardInterrupt:

    print("Script terminated by user.")

finally:

    # Disconnect from the broker

    client.disconnect()

**About ThingSpeak:**

In the endeavor to optimize public transport, the integration of real-time transit information on ThingSpeak is paramount. This process involves the amalgamation of hardware components, including GPS modules and passenger sensors installed on each bus. These devices work together to acquire and process critical data, such as location coordinates and passenger count. Subsequently, the meticulously processed data is transmitted to ThingSpeak channels via Wi-Fi or cellular networks.

A dedicated ThingSpeak channel is established to receive and display this real-time information. The channel is configured with a unique API key, ensuring secure communication between the microcontroller and the ThingSpeak platform. Through this seamless interaction, passengers and transit authorities gain access to a dynamic display of bus locations, passenger count, etc.

**Code to display Real time Transit Information on Thingspeak:**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Transit Information</title>

    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

    <style>

        body {

            font-family: Arial, sans-serif;

            margin: 20px;

        }

        #transit-info {

            border: 1px solid #ccc;

            padding: 20px;

        }

    </style>

</head>

<body>

    <h1>Real-Time Transit Information</h1>

    <div id="transit-info"></div>

    <script>

        const apiKey = 'HSVQUBXHFT54BK1V';

        function updateTransitInfo() {

            $.getJSON(`https://api.thingspeak.com/channels/ 2320111/feeds.json?api\_key=${apiKey}&results=1`, function(data) {

                const transitInfoDiv = document.getElementById('transit-info');

                transitInfoDiv.innerHTML = ''; // Clear previous content

                if (data && data.feeds && data.feeds.length > 0) {

                    const latestEntry = data.feeds[0];

                    const transitData = {

latitude: latestEntry.field1,

                        longitude: latestEntry.field2,

                        speed: latestEntry.field3,

                        directions: latestEntry.field4,

arrival time: latestEntry.field5,

passenger count: latestEntry.field6

                    };

                    const div = document.createElement('div');

                    div.innerHTML = `<strong>${transitData.route}</strong> | Location: ${transitData.location} | Passengers: ${transitData.passengers}`;

                    transitInfoDiv.appendChild(div);

                }

            });

        }

        // Update transit information every 5 seconds (adjust the interval as needed)

        setInterval(updateTransitInfo, 5000);

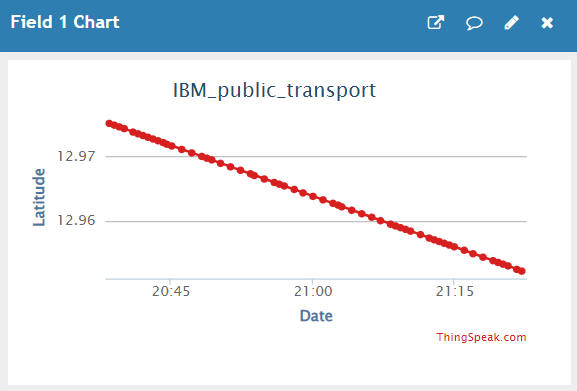
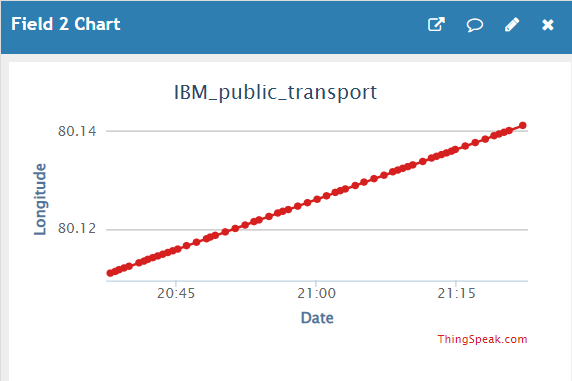
        // Initial update

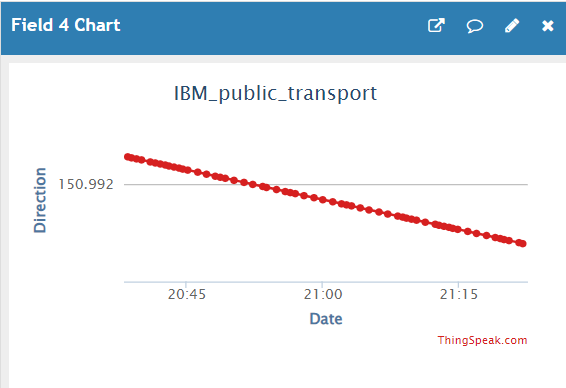
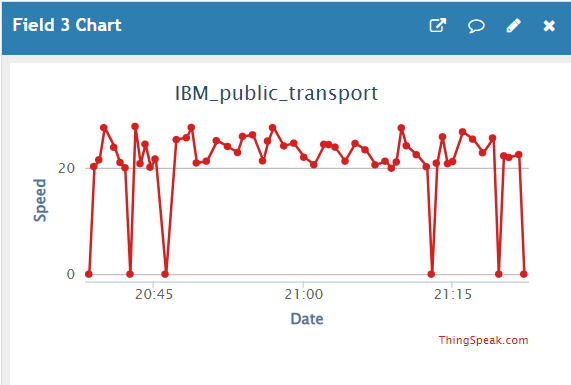
        updateTransitInfo();

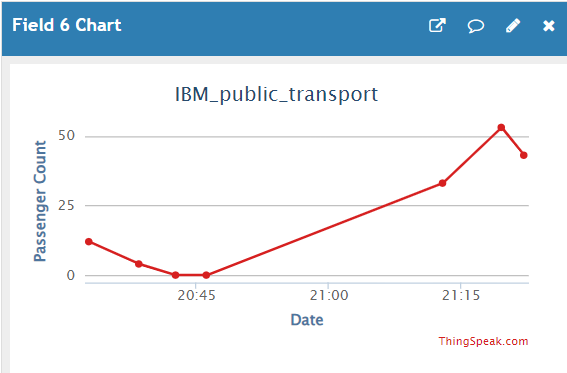
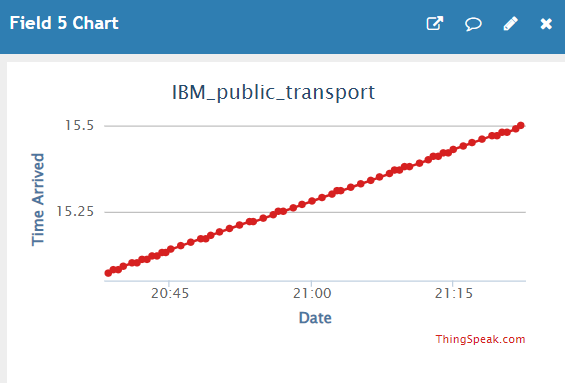
    </script>

</body>

</html>

Output:





Code for Interfacing Data with Thingspeak:

import geopy.distance

import requests

import time

import math

import random

import numpy as np

from datetime import datetime, timedelta

# Replace with your ThingSpeak API key and the data you want to send

api\_key = 'HSVQUBXHFT54BK1V'

# Kundrathur coordinates

kundrathur\_coords = (12.9790, 80.1064)

# Chromepet coordinates

chromepet\_coords = (12.9518, 80.1415)

# Bus stops along the path

bus\_stops = [

    {"coords": (12.9790, 80.1064), "name": "Stop 1", "passenger\_change": 10},

    {"coords": (12.9720, 80.1100), "name": "Stop 2", "passenger\_change": -5},

    {"coords": (12.9680, 80.1165), "name": "Stop 3", "passenger\_change": 15},

    {"coords": (12.9650, 80.1200), "name": "Stop 4", "passenger\_change": -8},

    {"coords": (12.9620, 80.1250), "name": "Stop 5", "passenger\_change": 12},

    {"coords": (12.9600, 80.1280), "name": "Stop 6", "passenger\_change": -7},

    {"coords": (12.9580, 80.1320), "name": "Stop 7", "passenger\_change": 20},

    {"coords": (12.9550, 80.1350), "name": "Stop 8", "passenger\_change": -10},

    {"coords": (12.9520, 80.1390), "name": "Stop 9", "passenger\_change": 8},

    {"coords": (12.9490, 80.1420), "name": "Stop 10", "passenger\_change": -12},

]

# Generate coordinates along the path

def generate\_path(start\_coords, end\_coords, num\_points):

    path = []

    for i in range(num\_points + 1):

        fraction = i / num\_points

        lat = start\_coords[0] + fraction \* (end\_coords[0] - start\_coords[0])

        lon = start\_coords[1] + fraction \* (end\_coords[1] - start\_coords[1])

        path.append((lat, lon))

    return path

# Generate a path with 100 points

selected\_stops = random.sample(path\_coords, len(bus\_stops))

# Calculate distances between consecutive points to simulate speed

speed\_values = [20]  # Initial speed

for i in range(1, len(path\_coords)):

    distance\_km = geopy.distance.distance(path\_coords[i - 1], path\_coords[i]).km

    time\_hours = distance\_km / speed\_values[-1]  # Assume constant speed

    speed\_values.append(distance\_km / time\_hours)

# Calculate bearing and map to 8 directions

def calculate\_bearing(point1, point2):

    lat1, lon1 = point1

    lat2, lon2 = point2

    delta\_lon = lon2 - lon1

    x = math.cos(math.radians(lat2)) \* math.sin(math.radians(delta\_lon))

    y = math.cos(math.radians(lat1)) \* math.sin(math.radians(lat2)) - \

        math.sin(math.radians(lat1)) \* math.cos(math.radians(lat2)) \* math.cos(math.radians(delta\_lon))

    bearing = math.atan2(x, y)

    bearing = math.degrees(bearing)

    bearing = (bearing + 360) % 360  # Ensure bearing is between 0 and 360 degrees

    return bearing

# Calculate bearings between consecutive points

bearings = [calculate\_bearing(path\_coords[i - 1], path\_coords[i]) for i in range(1, len(path\_coords))]

# Map bearings to 8 directions

directions = []

for bearing in bearings:

    angle = bearing + 22.5  # Offset by half the angle to get more accurate direction

    if angle < 0:

        angle += 360

    direction\_index = int(angle / 45) % 8

    direction = ["N", "NE", "E", "SE", "S", "SW", "W", "NW"][direction\_index]

    directions.append(angle)

# Extract latitude and longitude values

latitude\_values, longitude\_values = zip(\*path\_coords)

# Simulate passenger count and stops

passenger\_count = 0

stop\_interval = 5  # Time interval between bus stops in minutes

update\_interval = 30  # Update interval in seconds

# Generate timestamps

start\_timestamp = datetime.now()

timestamps = [start\_timestamp + timedelta(seconds=i \* update\_interval) for i in range(len(speed\_values))]

# Display the first few values

print("Latitude values:", latitude\_values[:5])

print("Longitude values:", longitude\_values[:5])

print("Speed values:", speed\_values[:5])

print("Directions:", directions[:5])

for i in range(len(speed\_values)):

    data = {

        'field1': str(latitude\_values[i]),

        'field2': str(longitude\_values[i]),

        'field3': str(max(0, speed\_values[i] + 10 - random.uniform(2, 10))),  # Speed is non-negative

        'field4': directions[i],

        'field5': timestamps[i].strftime("%H.%M"),  # Format timestamp as string

    }

    print(timestamps[i].strftime("%H.%M"))

    # Simulate stops

    for stop,passenger in zip(selected\_stops, bus\_stops):

        if stop == (latitude\_values[i], longitude\_values[i]):

            print("bus stop")

            passenger\_count = max(0, passenger\_count + passenger["passenger\_change"])  # Ensure passenger count is non-negative

            data['field6'] = str(passenger\_count)

            data['field3'] = '0'  # Speed is 0 at stops

            print(f"Bus stopped at {passenger['name']}. Passenger count: {passenger\_count}")

    # Update ThingSpeak with a delay

    time.sleep(update\_interval)

    # ThingSpeak update URL

    url = f'https://api.thingspeak.com/update?api\_key={api\_key}'

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

    if response.status\_code == 200:

        print("Data uploaded successfully to ThingSpeak")

    else:

        print("Failed to upload data")

**MOBILE APPLICATION:**

The application MIT App Inventor plays a crucial role in the seamless operation of this system. It serves as the vital link between passengers and the server, offering essential information about theLatitude, longitude, speed, distance, arrival time and Passenger count . The application is crafted using the Blynk platform, an innovative IoT platform renowned for its ability to swiftly create impressive applications for both Android and iOS smartphones.

The architecture of the Blynk platform encompasses Blynk libraries, the Blynk server, and Blynk apps. Within the Android application it supplies passengers with estimated arrival times of the buses, reports the speed of the bus in transit, and presents the nearest available bus based on calculations of proximity between the passenger and the bus's location.

1. ThingSpeak Configuration:

* Data, such as real-time transit information, was collected and stored on ThingSpeak channels.
* Each data point included relevant fields such as route information, location, and passenger count.

1. MIT App Inventor Setup:

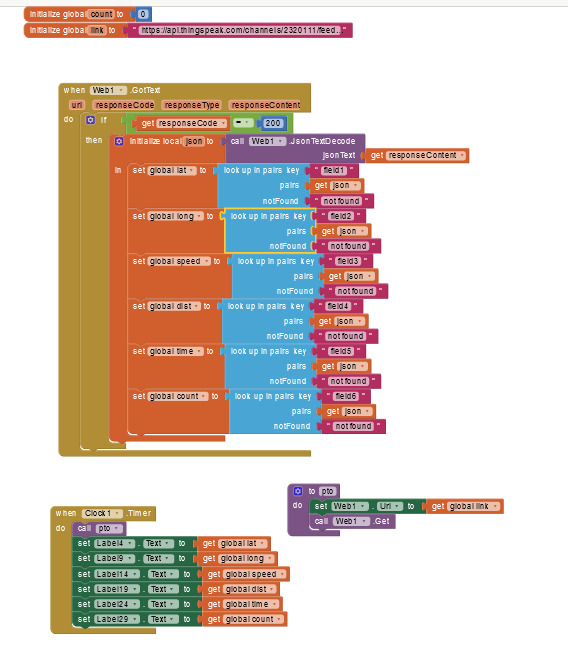
* MIT App Inventor's visual interface was utilized for app development.
* Components such as labels, text boxes, and web components were employed to design the app's user interface.

1. Data Retrieval:

* The MIT App Inventor app was configured to retrieve data from ThingSpeak channels using the Web component.
* API calls were made to ThingSpeak to fetch the latest data entries.

1. Real-Time Display:

* The retrieved data, including route information, location, and passenger count, was dynamically displayed on the MIT App Inventor app's interface.



**Innovation:**

1. **Dynamic Traffic Signal Coordination:**

* Adaptive Traffic Control:

1. Implement IoT sensors at traffic intersections to collect real-time data on traffic flow.

* Congestion Alerts:

1. Implement a system that uses IoT data to provide real-time congestion alerts to public transport operators and commuters.
2. Optimize routes dynamically to avoid congested areas.
3. **Smart Ticketing and Boarding:**

* Passenger Movement Tracking:

1. Utilize IoT sensors to track passenger movement within public transport systems.
2. Optimize boarding and alighting processes based on real-time data.
3. **Predictive Maintenance for Vehicles:**

* IoT Sensors for Real-time Monitoring:

1. Equip public transport vehicles with IoT sensors to monitor various components in real time.
2. Detect potential issues before they lead to breakdowns.

Integrate IoT-enabled traffic monitoring systems along public transport routes to gather real-time data on traffic congestion, road closures, and other factors influencing travel times

**Team members:**

Aishwarya I (2021504005)

Hayagreevan S (2021504009)

Janani M (2021504010)

Naresh Kumar S (2021504025)

Shohail Shahabudeen (2021504040)