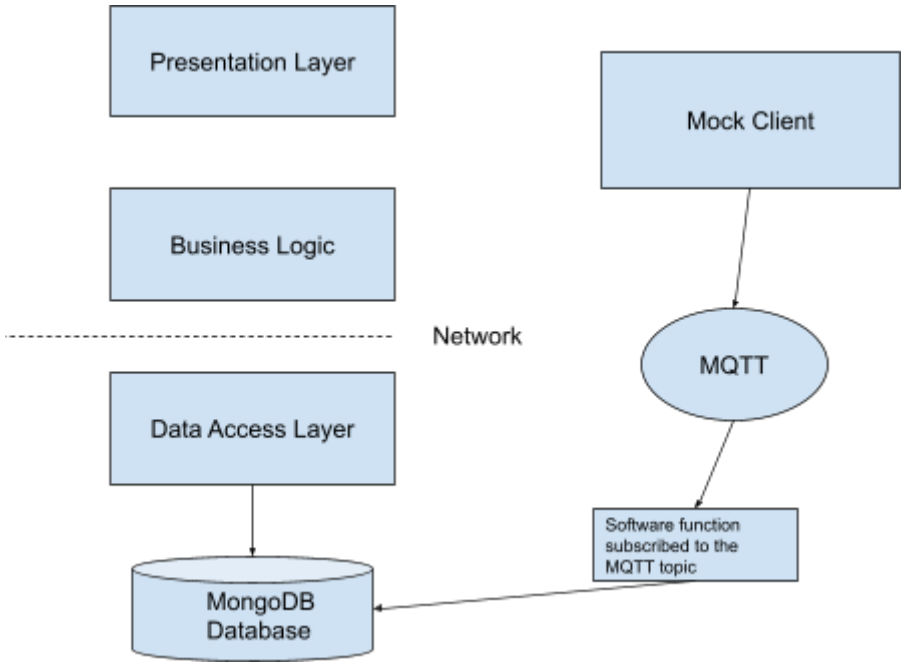
AGV Dashboards

I. Introduction:

The application I worked on is part of a project aiming to develop an MCU or SBC based software solution that controls an autonomous vehicle.
The goal behind the application I will work on will be querying and visualizing data of the Autonomous Vehicle in order to help in tracking and troubleshooting issues that may occur or analyzing and understanding the causes of accidents.
The data I will work with will be collected from the sensors attached to the vehicle.
The dashboard will have graphs and charts displaying the state of sensors and tracking the usage of resources across time and the users will have the possibility to get information coming from sensors at any particular specified date.
My work also included creating a mock client that imitates the work of the AGV, this mock client uses the MQTT protocol to send random sample data to our database.

II. Architecture:



The application is written with the Xamarin framework, it is a cross-platform mobile application that runs on Android and iOS.

The UI is based on Xamarin Forms, and the oxyplot framework was used to create the chart and graphs.
The APIs used by the front-end to fetch the data, and the data itself, coming from the AGV, are stored in MongoDB Database hosted locally and the APIs were created using Express which is a node js web application framework.

AGV Mock Client:

It is a Windows Forms App written with the .NET Framework. It is a simple app that generates or selects random data to imitate the work of the AGV itself. Sending the data will be performed.

III. List of classes in AGV Dashboards app:

I just used a single class for the application code, but this class is divided into 3 parts (partial classes)

The first partial class is in the file “MainPage.xaml.cs”

It contains the constructor that calls the functions

InitializeComponent();
LoadResources();
LoadWheelsData();
LoadLineTrackerData();
LoadImages();
LoadObstaclesImages();
LoadUltrasonicData();

It also contains the OnFilteredClicked(object sender, EventArgs e) event handler that will allow us to get the date-filtered data. This event handler calls the functions:

LoadFilteredWheelsData(selectedDate);
LoadFilteredResources(selectedDate);
LoadFilteredLineTrackerData(selectedDate);
LoadFilteredUltrasonicData(selectedDate);
LoadFilteredImages(selectedDate);
LoadFilteredObstaclesImages(selectedDate);

The second partial class is in the file “LoadingData.cs”, this file has the implementation of functions that are called by the constructor.
The third partial class is in the file “LoadingFilteredData.cs”, this file has the implementation of the functions that will retrieve the date-filtered data upon the call of the event handling function OnFilteredClicked(object sender, EventArgs e).

IV. Roles of functions in AGV Dashboards app:

- Get the speed and direction in a given day from the api and create two line graphs
  LoadFilteredWheelsData(selectedDate);

- Get the average usage of resources in a given day from the api and create two pie charts for the memory and a cpu consumption
  LoadFilteredResources(selectedDate);

- Get the data about the line tracker status in a given day from the api and create a line graph:
  LoadFilteredLineTrackerData(selectedDate);

- Get the data of the four ultrasonic sensors in a given day from the api and create a graph with four lines
  LoadFilteredUltrasonicData(selectedDate);

- Get the images taken by the camera in a given day from the api and add them to an horizontal scroll view
  LoadFilteredImages(selectedDate);

- Get the images of obstacles detected in a given day from the api and add them to an horizontal scroll view
  LoadFilteredObstaclesImages(selectedDate);

- Get the last data about the usage of resources from the api and create two pie charts for the memory and a cpu consumption
  LoadResources();

- Get the last data about speed and direction from the api and create two line graphs
  LoadWheelsData();

- Get the last data about the line tracker status from the api and create a line graph
  LoadLineTrackerData();

- Get the last images taken by the camera from the api and add them to an horizontal scroll view
  LoadImages();

- Get the last images of obstacles detected from the api and add them to an horizontal scroll view
  LoadObstaclesImages();

- Get the last data of the four ultrasonic sensors from the api and create a graph with four lines
  LoadUltrasonicData();

V. List of classes in AGV Mock Client:

I just used a single class for the application code, but this class is divided into 2 parts (partial classes)

The first partial class is in the file “Form1.cs”

It contains the constructor that calls the function

InitializeComponent();

It also contains the event handling functions that will allow us to call the functions responsible for sending random sample data to the database through MQTT after a button representing a certain sensor is clicked.

The second partial class is in the file “Mqtt Functions.cs”, this file has the implementation of functions that are responsible for sending the data.

VI. Roles of functions in AGV Mock Client:

- Subscribe to the MQTT topic of the appropriate sensor and sends the data to the MongoDB collection when received
  - ImageFromMqttTopicToMongoDB();
  - ObstacleFromMqttTopicToMongoDB();
  - LineTrackerStatusFromMqttTopicToMongoDB();
  - ResourcesFromMqttTopicToMongoDB();
  - UltrasonicFromMqttTopicToMongoDB();
  - WheelsDataFromMqttTopicToMongoDB();

- Send a random data sample to the Mosquitto server on the appropriate sensor’s MQTT topic
  - ImageToTopicThroughMQQTT();
  - LineTrackerStatusToTopicThroughMQQTT();

- ResourcesToTopicThroughMQQTT();
- IUltrasonicToTopicThroughMQQTT();
- WheelsDataToTopicThroughMQQTT();

VII. APIs:

The application uses a different api endpoint for each of the functions mentioned above.

Deploying APIs on an Express.js Server

Prerequisites

Before getting started, ensure that you have the following:

1. Node.js and npm (Node Package Manager) installed on your machine.

Step 1: Set Up a New Express.js Project

To begin, create a new directory for your project and navigate to it using the command line. Then, initialize a new Node.js project by running the following command:

```
npm init
```

This command will prompt you to provide information about your project and generate a `package.json` file.

Step 2: Install Express.js

Install Express.js as a dependency for your project by running the following command:

```
npm install express
```

This will download and install the latest version of Express.js.

Step 3: Create an Express.js Server

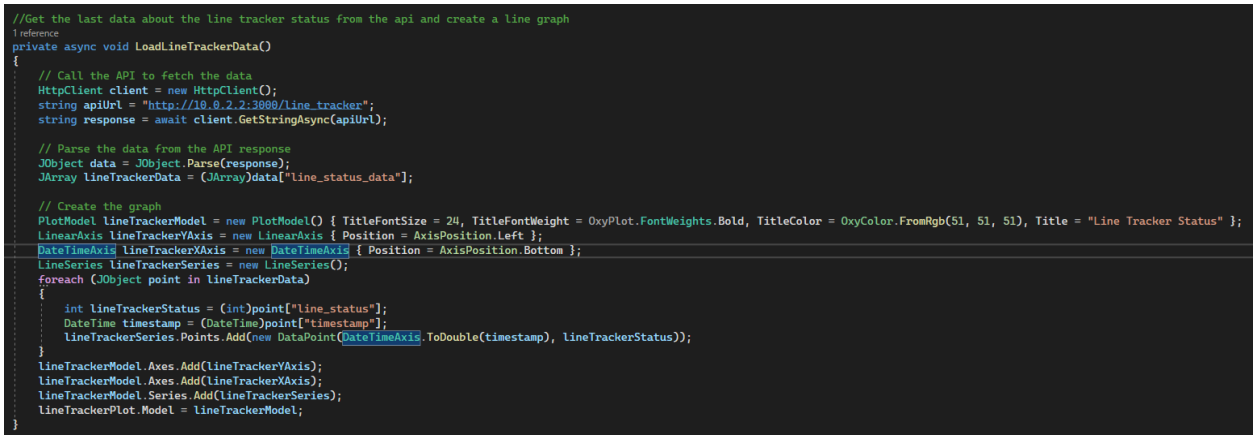
Create a new file, `server.js`, in your project directory and open it in a text editor. Add the following code to set up a basic Express.js server:

```
const express = require('express');

const app = express();

const port = 3000;


// Define your API routes here


app.listen(port);

});
```

This code imports the Express.js module, initializes an Express application, and defines the port on which the server will run. You will define your API routes in the section indicated.

VIII. API calls and response handling:

The following is one of the functions responsible for retrieving the data from the api endpoint, and making a line graph out of it.



```
//Get the last data about the line tracker status from the api and create a line graph
1 reference
private async void LoadLineTrackerData()
{
    // Call the API to fetch the data
    HttpClient client = new HttpClient();
    string apiUrl = "http://10.0.2.2:3000/line_tracker";
    string response = await client.GetStringAsync(apiUrl);

    // Parse the data from the API response
    JObject data = JObject.Parse(response);
    JArray lineTrackerData = (JArray)data["line_status_data"];

    // Create the graph
    PlotModel lineTrackerModel = new PlotModel() { TitleFontSize = 24, TitleFontWeight = OxyPlot.FontWeights.Bold, TitleColor = OxyColor.FromRgb(51, 51, 51), Title = "Line Tracker Status" };
    LinearAxis lineTrackerYAxis = new LinearAxis { Position = AxisPosition.Left };
    DateTimeAxis lineTrackerXAxis = new DateTimeAxis { Position = AxisPosition.Bottom };
    LineSeries lineTrackerSeries = new LineSeries();
    foreach (JObject point in lineTrackerData)
    {
        int lineTrackerStatus = (int)point["line_status"];
        DateTime timestamp = (DateTime)point["timestamp"];
        lineTrackerSeries.Points.Add(new DataPoint(DateTimeAxis.ToDouble(timestamp), lineTrackerStatus));
    }
    lineTrackerModel.Axes.Add(lineTrackerYAxis);
    lineTrackerModel.Axes.Add(lineTrackerXAxis);
    lineTrackerModel.Series.Add(lineTrackerSeries);
    lineTrackerPlot.Model = lineTrackerModel;
}
```

The response of the api is in JSON, we need to convert it to data we can use first, then feed it to the plot model we made. The plot model is made with the OxyPlot which is a cross-platform plotting library for .NET.

Sending a get request to the endpoint, triggers a function on an Express js server listening to the port 3000 on the localhost:

This is the function called by the API mentioned above:

```
const express = require('express');

const app = express();

const port = 3000;




const MongoClient = require('mongodb').MongoClient;

const url = 'mongodb://127.0.0.1:27017';



// Enable command monitoring for debugging

const client = new
MongoClient('mongodb://127.0.0.1:27017/?directConnection=true&server
SelectionTimeoutMS=2000&appName=mongosh+1.9.0');

 app.get('/line_tracker', async (req, res) => {

    try {

       const collection =
client.db('AGV').collection('agv_line_tracker');

       const projection = { line_status: 1, timestamp: 1, _id: 0 };



         // Use the aggregation pipeline to optimize performance

       const pipeline = [

       { $sort: { timestamp: -1 } },

       { $limit: 5 },

       { $project: projection }
```

```
      ];


      const data = await collection.aggregate(pipeline).toArray();


      const lineTrackerData = data.map(d => ({
        line_status: d.line_status,
        timestamp: new Date(d.timestamp)
      }));


      // Return the sensor data in the response
      return res.json({ line_status_data: lineTrackerData });
    } catch (error) {
      console.error(error);
      return res.status(500).send('Internal Server Error');
    }
  });
app.listen(port);
```

The function above is responsible for querying the data from the MongoDB collection stored in the Database ‘AGV’.

IX. Deploying an MQTT Mosquitto Server on Windows

Introduction

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol commonly used in IoT (Internet of Things) applications. Mosquitto is an open-source MQTT broker that allows you to set up and manage MQTT communication between devices. This guide will walk you through the process of deploying a Mosquitto server on a Windows machine for your MQTT communication needs.

Step 1: Install Mosquitto Server

1. Download the Mosquitto server installation package for Windows from the official Eclipse Mosquitto website: https://mosquitto.org/download/
2. Choose the appropriate version (32-bit or 64-bit) based on your system architecture.
3. Run the downloaded installer file and follow the installation wizard.
4. During the installation process, select the components you want to install, such as Mosquitto Broker and Mosquitto Clients.
5. Choose the destination folder for the installation and complete the installation process.

Step 2: Start the Mosquitto Server

1. Open the command prompt as an administrator.
2. Navigate to the Mosquitto installation directory using the `cd` command. For example:
   bash

   ```
   cd C:\Program Files\mosquitto
   ```

3. Start the broker and put it in verbose mode to enable debug messages, using the command

   ```
   mosquitto.exe -v
   ```