ROBOTICS TECHNOLOGY

CS-3015

Semester Project Report

Fall-2024

“NANO NAVIGATORS: Mapping with Precision”

Submitted By

FAMIA MALIK , AWAIS KHAN, MOHAMMAD ROHAN
22I-2375 , 22I-2390, 22I-2327
SECTION: C

Supervisor

Miss Umarah Qaseem

DEPARTMENT OF COMPUTING, FAST-NUCES

ISLAMABAD

11/DECEMBER/2024

TABLE OF CONTENTS:

ABSTRACT:

Autonomous navigation in confined environments presents unique challenges, particularly in achieving precision, coordination, and adaptability. The Nano Navigators project addresses these challenges by developing a group of three nano robots - Robot A, Robot B, and Robot C - designed to operate collaboratively within a defined space. These robots are equipped with ultrasonic sensors to measure distances and communicate data in real time, enabling them to map their environment and dynamically adjust their movements. By leveraging advanced sensing and synchronization techniques, the project aims to explore the potential of collaborative robotics in complex scenarios.

I. INTRODUCTION:

Overview:

The Nano Navigators project is an innovative approach to addressing the challenges of autonomous navigation in confined and structured environments. By leveraging a team of three nano robots—Robot A, Robot B, and Robot C—equipped with advanced ultrasonic sensing and real-time communication capabilities, the project aims to create a synchronized robotic system capable of mapping, navigating, and adapting dynamically to its surroundings. This collaborative robotic system is designed to optimize spatial awareness and enhance navigation precision, demonstrating significant potential for real-world applications such as industrial automation and confined-space exploration.

Objective Statement:

The primary objective of the Nano Navigators project is to develop a coordinated system of nano robots that can:

1. Accurately measure distances within a confined space using ultrasonic sensors.

2. Synchronize and transmit real-time data to a central processing unit for collaborative mapping.

3. Generate a virtual map of the environment to determine the positions of robots and outline optimal paths.

4. Adapt dynamically to changes in the environment through real-time data collaboration, ensuring efficient and precise navigation.

Scope & Limitations:

1.The project demonstrates a proof of concept for synchronized robotics in confined spaces.

2.It showcases the integration of ultrasonic sensing, wireless communication, and real-time data processing.

3.The development board caused some limitations. The original plan was to use four ultrasonic sensors per robot, but the board did not support enough pins to accommodate them.

4.The wireless communication module on the board occasionally breaks connection due to its limited Wi-Fi capabilities, which affects real-time data transmission.

II. PROBLEM STATEMENT:

Statement:

In confined or structured environments, precise navigation and real-time spatial awareness are critical for autonomous robotic systems. Existing solutions often lack effective collaboration between multiple robots to dynamically adapt to environmental changes. The challenge lies in designing a system where multiple robots can accurately measure distances, communicate data, and synchronize their movements to create a comprehensive map of their surroundings. The goal of the Nano Navigators project is to develop a coordinated group of nano robots, equipped with ultrasonic sensors and real-time communication capabilities, to address this problem by enabling efficient and adaptive navigation in constrained spaces.

III. IMPLEMENTATION STRATEGY:

Explanation:

The Nano Navigators project involves the development of three nano robots—Robot A, Robot B, and Robot C—arranged in a defined space resembling a box. The implementation focuses on accurate distance measurement, synchronization, mapping, and real-time data collaboration to achieve precise navigation and dynamic adjustments.

System Layout:

- **Robot A**: Positioned at the leftmost wall.
- **Robot B**: Occupies the middle section.
- **Robot C**: Located at the rightmost wall.

Robot Functionality:

1. Distance Measurement

Each robot utilizes its ultrasonic sensors to measure distance.
The data from these measurements are continuously transmitted to a central system for processing.

2. Synchronization

To achieve synchronization among the robots, a laptop is employed as the central processing unit:

Laptop as Server:

- The laptop acts as a server, coordinating the operations of the robots (clients).
- It receives real-time data from all three robots wirelessly.

Data Collection:

- The laptop collects distance values from the ultrasonic sensors on the robots.

Synchronization Protocol:

- The server ensures accurate time-stamped measurements to maintain data consistency.

3. Mapping

Data Processing:

- The laptop processes the incoming distance values from all robots.
- Using these values, it calculates the relative positions of the robots within the defined space.

Virtual Map Generation:

- The laptop plots the positions of Robot A, Robot B, and Robot C on an x-y coordinate system.
- The virtual map outlines the precise locations of the robots and the surrounding environment.

Path Planning and Tricks:

- Based on the map, the laptop determines optimal paths for the robots to take.
- This mapping also enables the robots to perform synchronized tricks or movements by leveraging their coordinated positions.

4. Real-time Data Collaboration

Dynamic Updates:

- The robots continuously send updated distance measurements to the laptop.
- The laptop integrates these updates into the virtual map, allowing for real-time adjustments.

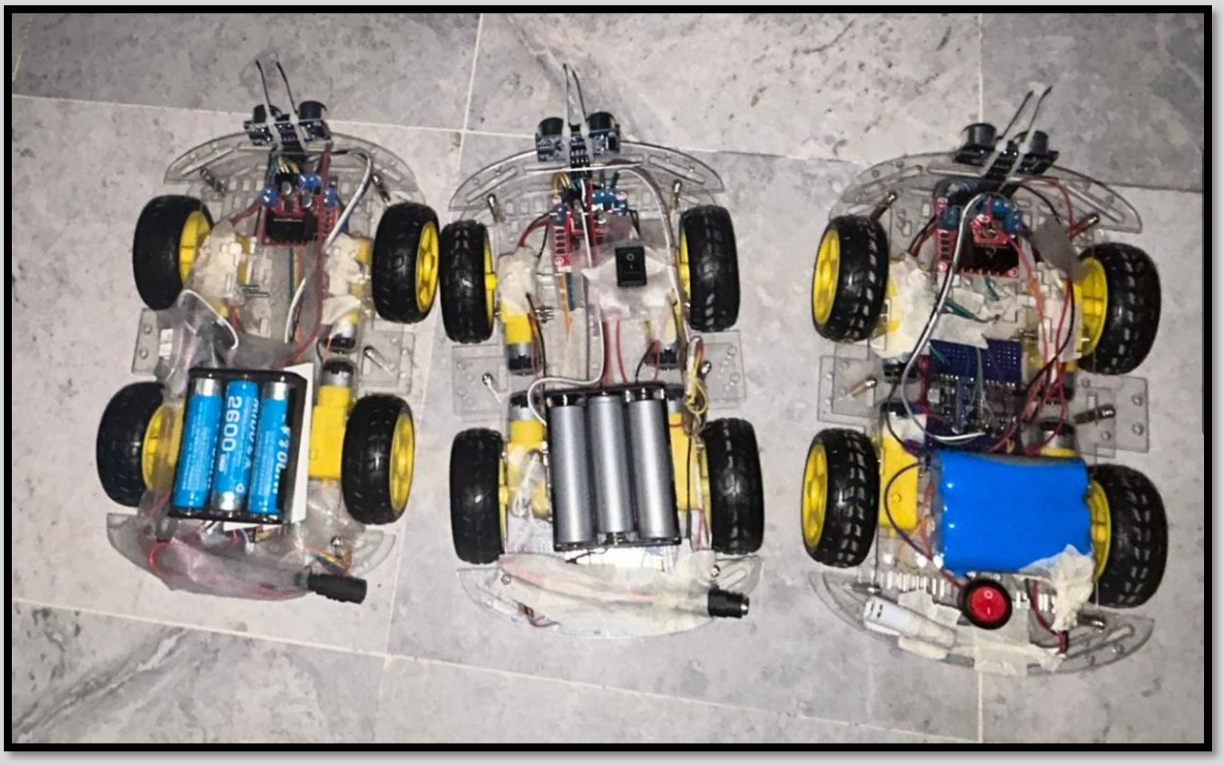
Collaborative Output:

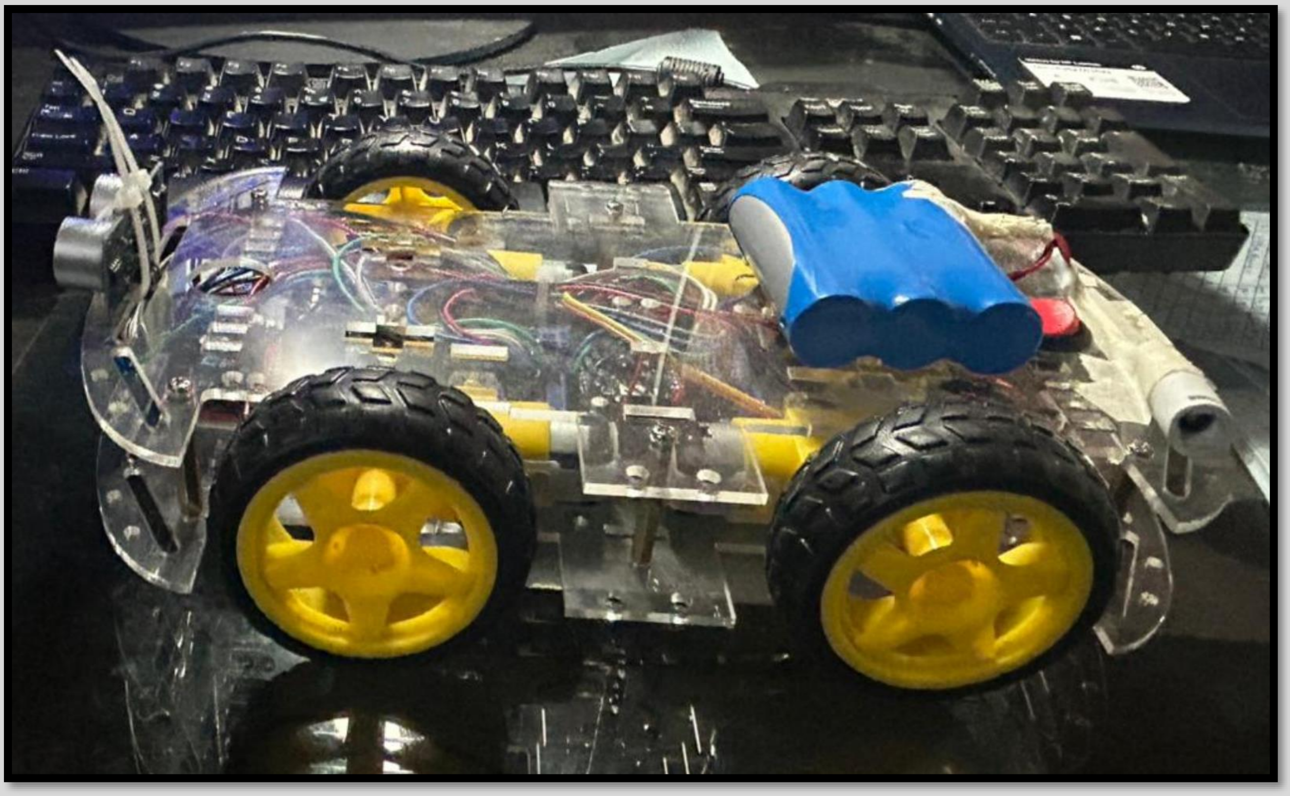
- By combining data from all three robots, the laptop provides a comprehensive view of the environment.
- This enables dynamic adjustments to the robots' movements, ensuring efficient navigation and enhanced collaboration.

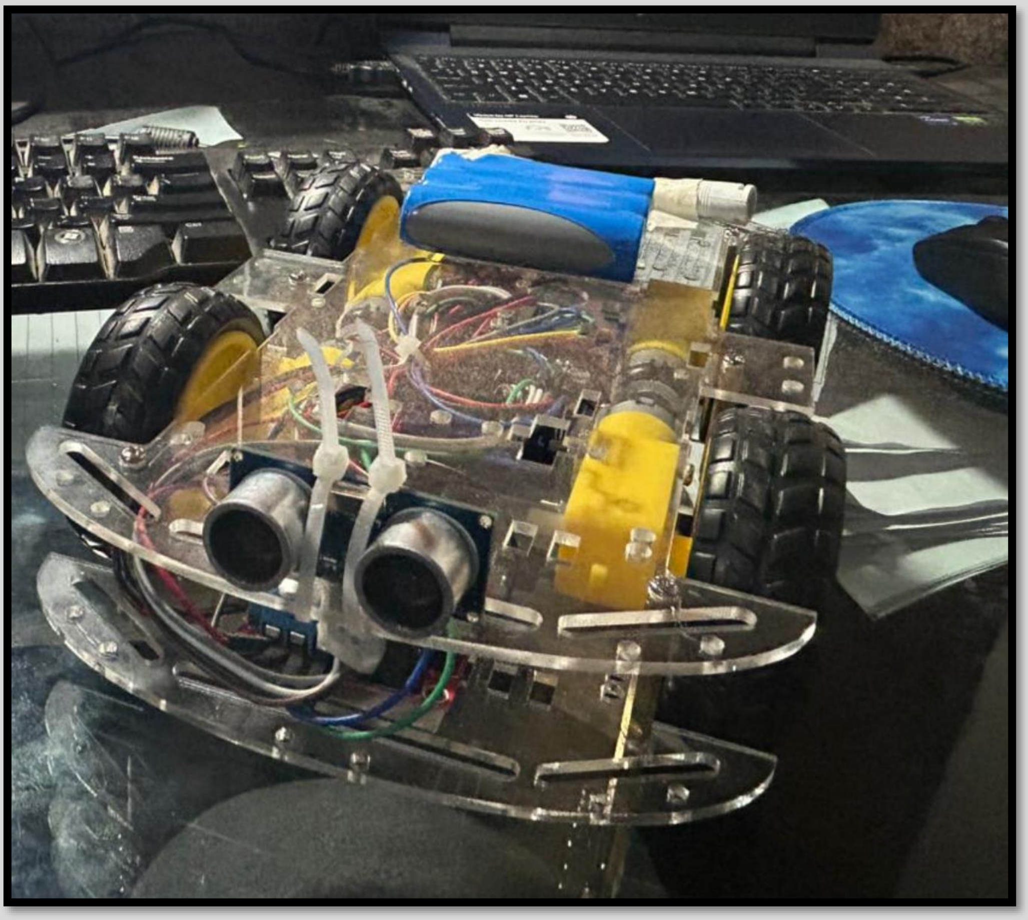
Supplies Needed:

1. 3x car chassis kit
2. 3x l298n motor driver
3. 3x small bread boards
4. 3x bunch of jumper male to male
5. 3x male to female
6. 3x female to female
7. 3x ultra sonics
8. 3x mpu 6050 (gyro accelerometer and temperature)
9. 3x button
10. 12 DCV Adpter
11. Cable

IV. CAR DESIGN:







V. CODE IMPLEMENTATION:

ARDUINO CODE:

```
#include <ESP8266WiFi.h>
#include <Wire.h>
#include <MPU6050_6Axis_MotionApps20.h>

// ********** USER CONFIGURATION **********
// Replace with your Wi-Fi credentials and server IP
const char* ssid = "Server";
const char* password = "apple123";
const char* serverIP = "192.168.43.64"; // Laptop server IP
const int serverPort = 12345;
// ***************************************

// Motor Control Pins
#define IN1 D8
#define IN2 D7
#define IN3 D4
#define IN4 D3

// Ultrasonic Sensor Pins
#define TRIG_PIN D6
#define ECHO_PIN D5

WiFiClient client;
MPU6050 mpu;

// Variables
volatile long duration = 0;
volatile bool echoReceived = false;
long distance = -1;
unsigned long lastMeasureTime = 0;

// MPU6050 DMP variables
uint8_t fifoBuffer[64];
Quaternion q;
VectorFloat gravity;
float ypr[3];
float estimated_yaw = 0.0;
float estimated_pitch = 0.0;
float estimated_roll = 0.0;

// Moving Average Filter for yaw
#define YAW_BUFFER_SIZE 10
float yawBuffer[YAW_BUFFER_SIZE];
int yawBufferIndex = 0;
float yawSum = 0.0;
```

```
// Interrupt Service Routine for ECHO_PIN
ICACHE_RAM_ATTR void echoISR() {
  static unsigned long startTime = 0;
  if (digitalRead(ECHO_PIN) == HIGH) {
    startTime = micros();
  } else {
    duration = micros() - startTime;
    echoReceived = true;
  }
}

void setup() {
  Serial.begin(9600);
  Serial.println("ESP8266 Car Initialized.");

  // Initialize Motor Control Pins
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  pinMode(IN3, OUTPUT);
  pinMode(IN4, OUTPUT);
  stopCar();

  // Initialize Ultrasonic Sensor Pins
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  digitalWrite(TRIG_PIN, LOW);
  attachInterrupt(digitalPinToInterrupt(ECHO_PIN), echoISR, CHANGE);

  // Connect to Wi-Fi
  Serial.print("Connecting to WiFi...");
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("\nWiFi connected.");
  Serial.print("IP: "); Serial.println(WiFi.localIP());

  // Connect to TCP server
  Serial.print("Connecting to server ");
  Serial.print(serverIP); Serial.print(":"); Serial.println(serverPort);
  while (!client.connect(serverIP, serverPort)) {
    Serial.println("Connection to server failed, retrying...");
    delay(2000);
  }
  Serial.println("Connected to server.");

  // Initialize MPU6050 with DMP
  Wire.begin(D2, D1);  // SDA to D2, SCL to D1
  mpu.initialize();
  if (mpu.testConnection()) {
    Serial.println("MPU6050 connection successful");
  } else {
    Serial.println("MPU6050 connection failed");
  }

  uint8_t devStatus = mpu.dmpInitialize();
  mpu.setXAccelOffset(-2291);
  mpu.setYAccelOffset(-1602);
  mpu.setZAccelOffset(1228);
  mpu.setXGyroOffset(87);
  mpu.setYGyroOffset(-72);
  mpu.setZGyroOffset(8);

  if (devStatus == 0) {
    mpu.setDMPEnabled(true);
    Serial.println("DMP ready");
    mpu.setRate(19);  // ~10 Hz
  } else {
    Serial.print("DMP Initialization failed (code ");
    Serial.print(devStatus);
    Serial.println(")");
  }

  // Initialize yaw buffer
  for (int i = 0; i < YAW_BUFFER_SIZE; i++) {
    yawBuffer[i] = 0.0;
  }
}

void loop() {
  // Non-blocking distance measurement
  if (millis() - lastMeasureTime >= 500) {
    lastMeasureTime = millis();
    if (echoReceived) {
      if (duration != 0) {
        distance = (duration / 2) / 29.1;
      } else {
        distance = -1;
      }
      echoReceived = false;
    }
    // Trigger new measurement
    digitalWrite(TRIG_PIN, LOW);
    delayMicroseconds(2);
    digitalWrite(TRIG_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRIG_PIN, LOW);
  }

  // Read MPU6050 data
  if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
    float raw_yaw = ypr[0] * 180 / PI;
    estimated_pitch = ypr[1] * 180 / PI;
    estimated_roll = ypr[2] * 180 / PI;
    if (raw_yaw < 0) raw_yaw += 360.0;
    estimated_yaw = raw_yaw;

    // Apply moving average filter to yaw
    yawSum -= yawBuffer[yawBufferIndex];
    yawBuffer[yawBufferIndex] = estimated_yaw;
    yawSum += estimated_yaw;
    yawBufferIndex = (yawBufferIndex + 1) % YAW_BUFFER_SIZE;
    estimated_yaw = yawSum / YAW_BUFFER_SIZE;
  }

  // Check if data is available from server
  if (client.available()) {
    String cmd = client.readStringUntil('\n');
    cmd.trim();
    Serial.print("Command from server: "); Serial.println(cmd);
    handleCommand(cmd);
  }

  // (Optional) send sensor data periodically
  static unsigned long lastSend = 0;
  if (millis() - lastSend > 1000) {
    lastSend = millis();
    String data = "DIST:" + String(distance) + " P:" +
String(estimated_pitch,2) +
                  " R:" + String(estimated_roll,2) + " Y:" +
String(estimated_yaw,2) + "\n";
    client.print(data);
  }

  // If disconnected, try to reconnect
  if (!client.connected()) {
    Serial.println("Disconnected from server, retrying...");
    while (!client.connect(serverIP, serverPort)) {
      Serial.println("Failed to reconnect. Retrying...");
      delay(2000);
    }
    Serial.println("Reconnected to server.");
  }
}

// Motor Control Functions
void stopCar() {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, LOW);
  Serial.println("Car Stopped.");
}

void moveForward() {
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  Serial.println("Moving Forward.");
}

void moveBackward() {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  Serial.println("Moving Backward.");
}

void turnLeft() {
  digitalWrite(IN1, HIGH);
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, HIGH);
  digitalWrite(IN4, LOW);
  Serial.println("Turning Left.");
}

void turnRight() {
  digitalWrite(IN1, LOW);
  digitalWrite(IN2, HIGH);
  digitalWrite(IN3, LOW);
  digitalWrite(IN4, HIGH);
  Serial.println("Turning Right.");
}

void rotateAroundLeft() {
  digitalWrite(IN1, HIGH);  // Right Motor Forward
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);   // Left Motor Stopped
  digitalWrite(IN4, LOW);
  Serial.println("Rotating Around Left Wheels.");
}

void rotateAroundRight() {
  digitalWrite(IN1, LOW);   // Right Motor Stopped
  digitalWrite(IN2, LOW);
  digitalWrite(IN3, LOW);   // Left Motor Forward
  digitalWrite(IN4, HIGH);
  Serial.println("Rotating Around Right Wheels.");
}

// Handle Commands
```

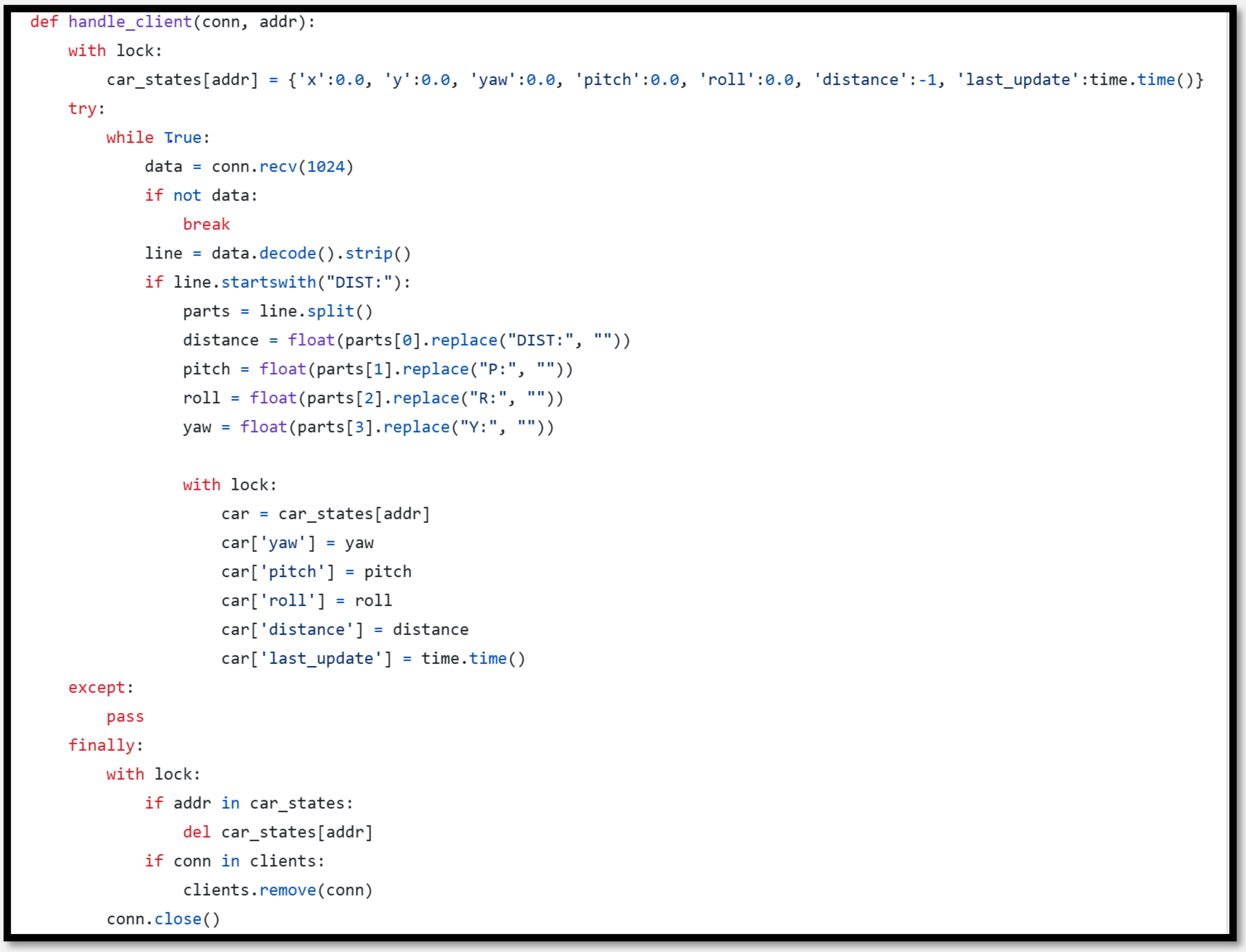
```
void handleCommand(String cmd) {
  if (cmd == "F") {
    moveForward();
  } else if (cmd == "B") {
    moveBackward();
  } else if (cmd == "L") {
    turnLeft();
  } else if (cmd == "R") {
    turnRight();
  } else if (cmd == "S") {
    stopCar();
  } else if (cmd == "RL") {
    rotateAroundLeft();
  } else if (cmd == "RR") {
    rotateAroundRight();
  } else {
    Serial.println("Unknown Command.");
  }
}
```

PYTON CODE:

```
import socket

import threading

import math

import time

from flask import Flask, jsonify, render_template, request

from functools import partial


# ---------- CONFIG ----------

HOST = '0.0.0.0'

PORT = 12345

# ----------------------------


clients = []

lock = threading.Lock()


car_states = {}


# Define more robust tricks with carefully timed steps.

# Each trick is a list of (command, duration) steps.

# After each step, a 'S' (stop) command is issued before moving to the next.

TRICKS = {

    "SPIN": [

        ("RL", 2.5) # Rotate left for 2.5 seconds, then stop

    ],

    "FLIP": [
        ("R", 0.5),

        ("F", 0.5),

        ("B", 0.5)

    ],

    "CIRCLE": [  # Move in a circle-like pattern by rotating and going 
forward

        ("RL", 0.5),

        ("F", 1.0),

        ("RL", 0.5),

        ("F", 1.0),

        ("RL", 0.5),

        ("F", 1.0),

        ("RL", 0.5),

        ("F", 1.0)

    ],

    "DANCE": [  # A playful pattern of movements

        ("F", 0.5),

        ("S", 0.2),

        ("B", 0.5),

        ("S", 0.2),

        ("L", 0.5),

        ("S", 0.2),

        ("R", 0.5)

    ],

    "ZIGZAG": [ # Zigzag pattern: forward+left, stop, forward+right, stop

        ("F", 0.5),

        ("L", 0.5),
        ("S", 0.2),

        ("F", 0.5),

        ("R", 0.5)

    ]

}


def handle_client(conn, addr):

    with lock:

        car_states[addr] = {'x':0.0, 'y':0.0, 'yaw':0.0, 'pitch':0.0,
'roll':0.0, 'distance':-1, 'last_update':time.time()}

    try:

        while True:

            data = conn.recv(1024)

            if not data:

                break

            line = data.decode().strip()

            if line.startswith("DIST:"):

                parts = line.split()

                distance = float(parts[0].replace("DIST:", ""))

                pitch = float(parts[1].replace("P:", ""))

                roll = float(parts[2].replace("R:", ""))

                yaw = float(parts[3].replace("Y:", ""))


                with lock:

                    car = car_states[addr]

                    car['yaw'] = yaw

                    car['pitch'] = pitch
                    car['roll'] = roll

                    car['distance'] = distance

                    car['last_update'] = time.time()

    except:

        pass

    finally:

        with lock:

            if addr in car_states:

                del car_states[addr]

            if conn in clients:

                clients.remove(conn)

        conn.close()


def accept_clients(server_socket):

    while True:

        conn, addr = server_socket.accept()

        with lock:

            clients.append(conn)

        thread = threading.Thread(target=handle_client, args=(conn, addr),
daemon=True)

        thread.start()


def broadcast_command(cmd, target=None):

    with lock:

        to_remove = []

        for c in clients:

            c_addr = c.getpeername()
            c_addr_str = f"{c_addr[0]}:{c_addr[1]}"

            if target is None or target == c_addr_str:

                try:

                    c.sendall((cmd+"\n").encode())

                except:

                    to_remove.append(c)

        for c in to_remove:

            clients.remove(c)


def perform_trick(trick_name, target=None):

    steps = TRICKS.get(trick_name, [])

    for (cmd, duration) in steps:

        # If the step is a move command, send it and wait

        if cmd != "S":

            broadcast_command(cmd, target)

            time.sleep(duration)

            broadcast_command("S", target)  # Stop after each movement step

        else:

            # If the step is 'S' itself, just wait the duration

            broadcast_command("S", target)

            time.sleep(duration)


app = Flask(__name__)


@app.route("/")

def index():
    return render_template("index.html")


@app.route("/cars")
def get_cars():
    with lock:
        data = []
        for addr, state in car_states.items():
            addr_str = f"{addr[0]}:{addr[1]}"
            data.append({
                'addr': addr_str,
                'x': state['x'],
                'y': state['y'],
                'yaw': state['yaw'],
                'pitch': state['pitch'],
                'roll': state['roll'],
                'distance': state['distance']
            })
        return jsonify(data)


@app.route("/command", methods=['POST'])
def send_command():
    cmd = request.form.get('cmd', '').strip().upper()
    target = request.form.get('target', '')
    if target == 'all':
        target = None
    if cmd in TRICKS.keys():
        threading.Thread(target=perform_trick, args=(cmd, target),
daemon=True).start()

    else:

        broadcast_command(cmd, target)

    return "OK"



def start_flask():

    app.run(host='0.0.0.0', port=5000, debug=False, use_reloader=False)



def main():

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server_socket.bind((HOST, PORT))

    server_socket.listen(5)



    accept_thread = threading.Thread(target=accept_clients,
args=(server_socket,), daemon=True)

    accept_thread.start()



    flask_thread = threading.Thread(target=start_flask, daemon=True)

    flask_thread.start()



    try:

        while True:

            cmd = input("Enter Command: ").strip()

            if cmd:

                if cmd in TRICKS.keys():

                    perform_trick(cmd)
                else:
                    broadcast_command(cmd)
    except KeyboardInterrupt:
        pass
    finally:
        server_socket.close()


if __name__ == "__main__":
    main()
```

Server Socket Function

CODE:



```python
def handle_client(conn, addr):
    with lock:
        car_states[addr] = {'x':0.0, 'y':0.0, 'yaw':0.0, 'pitch':0.0, 'roll':0.0, 'distance':-1, 'last_update':time.time()}
    try:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            line = data.decode().strip()
            if line.startswith("DIST:"):
                parts = line.split()
                distance = float(parts[0].replace("DIST:", ""))
                pitch = float(parts[1].replace("P:", ""))
                roll = float(parts[2].replace("R:", ""))
                yaw = float(parts[3].replace("Y:", ""))

                with lock:
                    car = car_states[addr]
                    car['yaw'] = yaw
                    car['pitch'] = pitch
                    car['roll'] = roll
                    car['distance'] = distance
                    car['last_update'] = time.time()
    except:
        pass
    finally:
        with lock:
            if addr in car_states:
                del car_states[addr]
            if conn in clients:
                clients.remove(conn)
        conn.close()
```

EXPLINATION:

Purpose: Processes incoming data from a connected car.

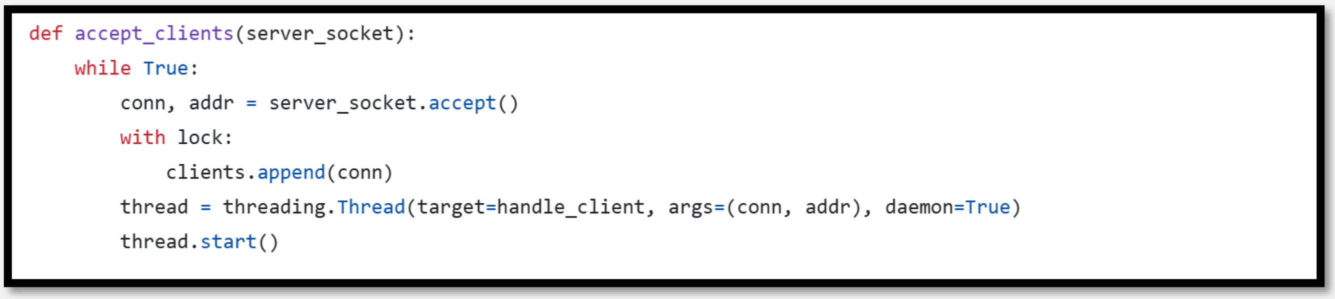
Steps:

Parses received data to extract distance, pitch, roll, and yaw.

Updates the car's state in car_states.

Handles disconnections by cleaning up the car's entry from car_states and clients

CODE:



```python
def accept_clients(server_socket):
    while True:
        conn, addr = server_socket.accept()
        with lock:
            clients.append(conn)
        thread = threading.Thread(target=handle_client, args=(conn, addr), daemon=True)
        thread.start()
```

EXPLINATION:

Purpose: Listens for new client connections.

Steps:

Accepts a connection and starts a new thread to handle the client.

Adds the client to the clients list for tracking.

CODE:

```
def broadcast_command(cmd, target=None):
    with lock:
        to_remove = []
        for c in clients:
            c_addr = c.getpeername()
            c_addr_str = f"{c_addr[0]}:{c_addr[1]}"
            if target is None or target == c_addr_str:
                try:
                    c.sendall((cmd+"\n").encode())
                except:
                    to_remove.append(c)
        for c in to_remove:
            clients.remove(c)
```

EXPLINATION:

Purpose: Sends commands to all cars or a specific target car.

Steps:

Iterates through connected clients and sends the command.

Cleans up disconnected clients from the clients list.

CODE:

```
def perform_trick(trick_name, target=None):
    steps = TRICKS.get(trick_name, [])
    for (cmd, duration) in steps:
        # If the step is a move command, send it and wait
        if cmd != "S":
            broadcast_command(cmd, target)
            time.sleep(duration)
            broadcast_command("S", target)  # Stop after each movement step
        else:
            # If the step is 'S' itself, just wait the duration
            broadcast_command("S", target)
            time.sleep(duration)
```

EXPLINATION:

Purpose: Executes a predefined sequence of commands (trick).

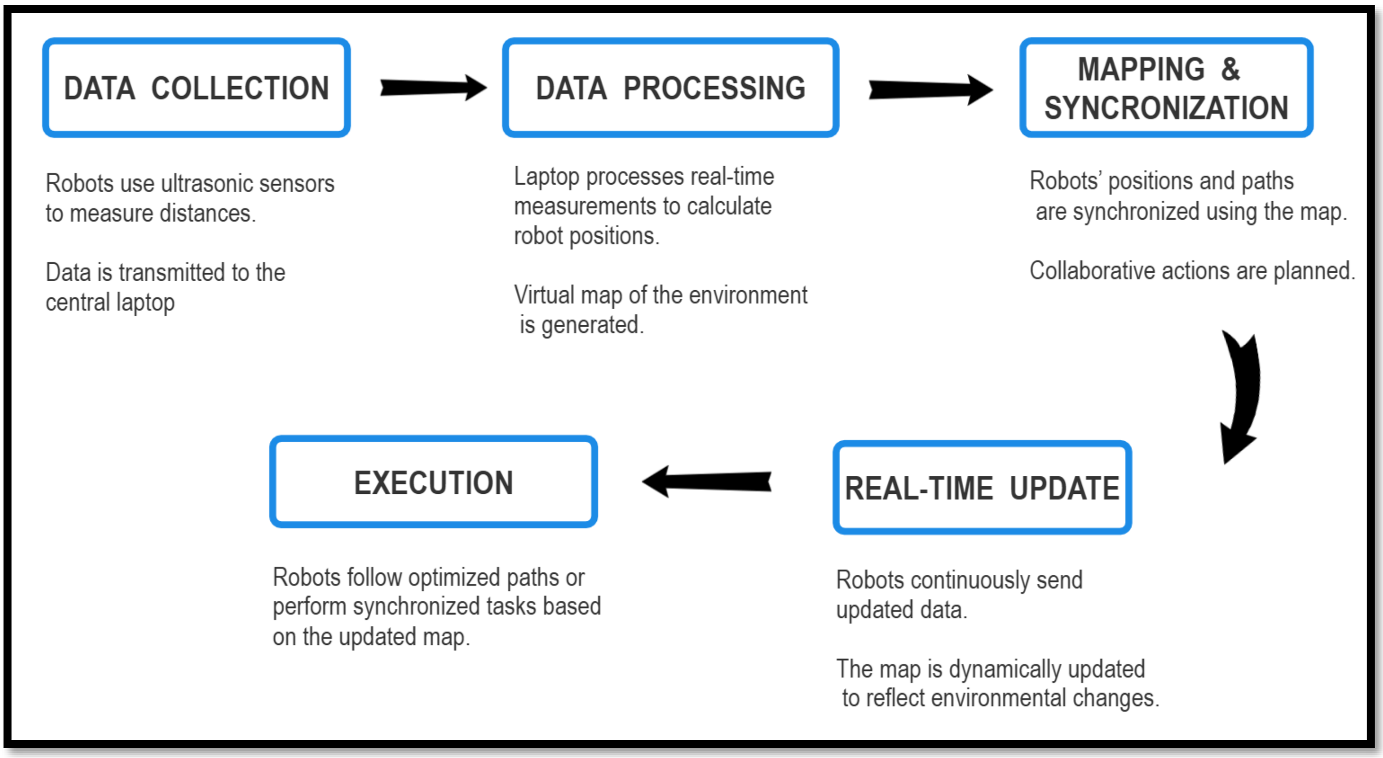
Steps:

Fetches the steps of the trick from TRICKS.

Sends commands sequentially to the target car, pausing between steps for the specified duration.

Sends a "Stop" (S) command after each step for safety.

VI. WORK FLOW DIAGRAM:


DATA COLLECTION
Robots use ultrasonic sensors to measure distances.
Data is transmitted to the central laptop
DATA PROCESSING
Laptop processes real-time measurements to calculate robot positions.
Virtual map of the environment is generated.
MAPPING & SYNCRONIZATION
Robots' positions and paths are synchronized using the map.
Collaborative actions are planned.
REAL-TIME UPDATE
Robots continuously send updated data.
The map is dynamically updated to reflect environmental changes.
EXECUTION
Robots follow optimized paths or perform synchronized tasks based on the updated map.

VII. CONCLUSION:

Explanation:

The Nano Navigators project demonstrates the potential for collaborative robotics in confined environments. By integrating distance measurement, synchronization, mapping, and real-time collaboration, the system ensures precise navigation and adaptability. The laptop, acting as a server, effectively maps the environment and coordinates the robots' movements, enabling them to perform synchronized tricks and tasks. This implementation lays the foundation for further advancements in autonomous robotic systems, showcasing their potential in both practical and creative applications.