WeDoFootball!

Filippo Alzati 745495

Redon Kokaj 744959

Introduction

This project involves the development of a simulation system for tracking soccer players' movements, pressure, body temperature, heart rate, and calories consumed, using sensors integrated with digital devices for efficient data management and flow. The system includes a GUI that allows users to view real-time match information, including a heat map representing the players' movements on the field. This enables an immediate understanding of tactical dynamics and strategies. An integrated database system supports data analysis, providing storage, extraction, and advanced analytics to generate useful insights for coaches, players, and fans.

System Overview

The system is composed of the following main components:

1. **Simulators for Data Generation**
2. **MQTT Broker for Data Communication (check *MQTT Hierarchy.pdf*)**
3. **Node-RED for Data Retrieval and Dashboard Creation**
4. **MongoDB Integration for Data Storage and Analysis**

Simulators

The simulators are responsible for generating the data flow that simulates the sensors' behavior over a 90-second period (equivalent to a 90-minute game).

The *simulators.py* script generates data for various sensors and inserts them into the MQTT broker under specific topics.

MQTT

Configuration Variables

```
MQTT_BROKER = 'localhost'
MQTT_PORT = 1883
MQTT_TOPIC_TEMPLATE = 'football/players/{}/sensors'
MQTT_COORDINATES_TOPIC_TEMPLATE = 'football/players/{}/sens
ors/coordinates'
```

These variables define the configuration for the MQTT broker and the topic templates. *MQTT_BROKER* and *MQTT_PORT* specify the broker's address and port, while *MQTT_TOPIC_TEMPLATE* and *MQTT_COORDINATES_TOPIC_TEMPLATE* provide templates for constructing MQTT topics for player sensor data and coordinates.

on_connect Callback Function

```
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print(f"Connected to MQTT broker with result code
{rc}")
        # Subscribe to topics for all players
        for player_id in range(1, 12):
            client.subscribe(MQTT_TOPIC_TEMPLATE.format(pla
yer_id))
    else:
        print(f"Failed to connect to MQTT broker with resul
t code {rc}")
```

This function handles the event of connecting to the MQTT broker. It verifies a successful connection and subscribes to the relevant topics for all players to receive their sensor data. If the connection fails, it outputs an error message.

on_message Callback Function

```
def on_message(client, userdata, msg):
    for player_id, role in ROLES.items():
        topic = MQTT_TOPIC_TEMPLATE.format(player_id)
        if msg.topic == topic:
            data = json.loads(msg.payload.decode())
```

```
            # Create a simplified message with only x and y
coordinates
            simplified_data = {
                "player_id": data["player_id"],
                "role": data["role"],
                "gps": {
                    "x": data["gps"]["x"],
                    "y": data["gps"]["y"]
                }
            }
            # Publish simplified data to coordinates topic
            coordinates_topic = MQTT_COORDINATES_TOPIC_TEMP
LATE.format(player_id)
            client.publish(coordinates_topic, json.dumps(si
mplified_data))
            print(f"Published coordinates for Player {playe
r_id} to topic '{coordinates_topic}'")
```

This function processes incoming MQTT messages. It checks if the message topic matches any player's sensor data topic. If it does, it extracts and simplifies the data to include only essential GPS coordinates, then publishes this simplified data to a separate coordinates topic.

main Function

```
def main():
    mqtt_client = mqtt.Client(protocol=mqtt.MQTTv311)
    mqtt_client.on_connect = on_connect
    mqtt_client.on_message = on_message
    mqtt_client.connect(MQTT_BROKER, MQTT_PORT)

    mqtt_client.loop_start()

    try:
        elapsed_time = 0
        simulation_name = datetime.now(rome_timezone).strft
ime("%Y%m%d_%H%M%S")

        while elapsed_time <= 90:
```

```
            for player_id in range(1, 12):
                role = ROLES[player_id]
                payload = generate_metrics(player_id, role,
elapsed_time)

                mqtt_topic = MQTT_TOPIC_TEMPLATE.format(pla
yer_id)
                mqtt_client.publish(mqtt_topic, json.dumps
(payload))
                store_simulation_data(simulation_name, payl
oad)
                print(f"Published and stored data for Playe
r {player_id} ({role})")

            elapsed_time += 1
            time.sleep(1)

    except KeyboardInterrupt:
        print("\\nStopping sensor simulation...")
        mqtt_client.loop_stop()
        mqtt_client.disconnect()
```

This function sets up and runs the main simulation loop. It creates and configures the MQTT client, including defining the connection and message handling callbacks. After connecting to the broker, it starts the MQTT event loop. The main loop simulates a football match, generating and publishing sensor data for each player every second. If the simulation is interrupted, it stops the MQTT loop and disconnects the client gracefully.

MongoDB

Configuration Variables

```
MONGO_URI = "mongodb://localhost:27017/"
DATABASE_NAME = "footballDB"
BASE_COLLECTION_NAME = "simulations"
```

These variables set up the connection details for MongoDB. *MONGO_URI* defines the address of the MongoDB server. *DATABASE_NAME* specifies the database to use, and *BASE_COLLECTION_NAME* provides a base name for collections that will store simulation data.

MongoDB Client and Database Initialization

```
client = MongoClient(MONGO_URI)
db = client[DATABASE_NAME]
```

This snippet creates a connection to the MongoDB server using the provided URI and selects the specified database. If the database doesn't exist, it will be created when data is inserted.

Function to Store Simulation Data

```
def store_simulation_data(simulation_name, data):
    collection_name = f"{BASE_COLLECTION_NAME}_{simulation_
name}"
    collection = db[collection_name]
    try:
        collection.insert_one(data)
        print(f"Data inserted into MongoDB collection '{col
lection_name}': {data}")
    except Exception as e:
        print(f"Error inserting data into MongoDB: {e}")
```

This function handles the insertion of simulation data into a MongoDB collection. It dynamically constructs the collection name using the base name and simulation name, inserts the data into this collection, and provides feedback on the success or failure of the operation. Each simulation run gets its own unique collection, ensuring organized data storage.

Generation of data

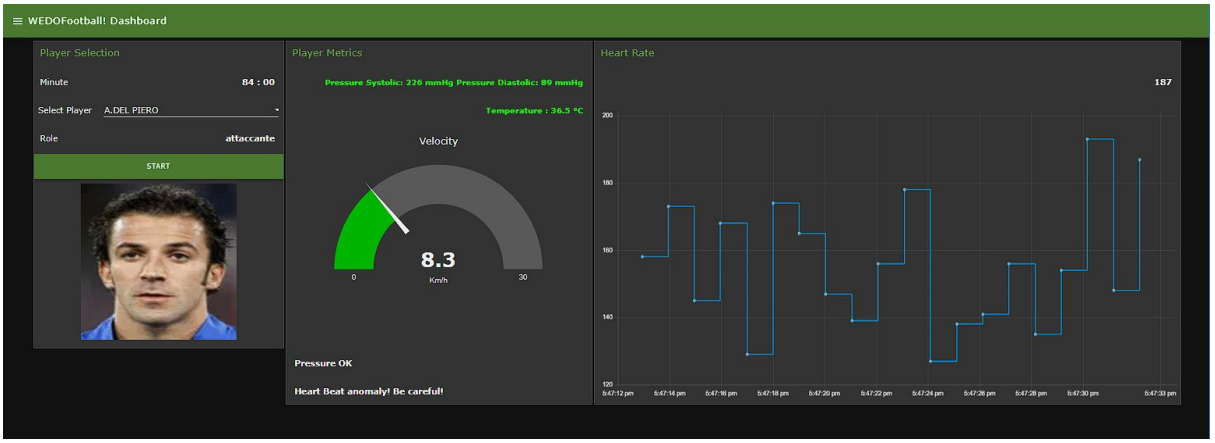
```
def generate_metrics(player_id, role, elapsed_time):
    # Coefficients to modify player behavior towards the end
    if elapsed_time > 80:
```

```
        heart_rate_coefficient = 1.1  # Increase heart rate
        gps_velocity_coefficient = 0.5  # Reduce movement spe
    else:
        heart_rate_coefficient = 1.0
        gps_velocity_coefficient = 1.0

    if role == 'attaccante':
        return {
            "player_id": player_id,
            "role": role,
            "heart_rate": {"heart_rate": int(random.randint(1
            "temperature": {"body_temperature": round(random.
            "blood_pressure": {"systolic": random.randint(170
            "calories_consumed": {"calories": round(random.un
            "gps": {"x": random.randint(50, 105), "y": random
            "timestamp": datetime.now(rome_timezone).isoforma
            "elapsed_time": elapsed_time
        }
     elif role = '...'
```

This function handles the generation of data for the players (in this case, the forward). Every role has different generation logics, depending on different variables, simulating the most realistic data for each player.

Data Retrieval and Dashboard



WeDoFootball! main dashboard

Node-RED is used to create an interactive dashboard that retrieves data from the MQTT broker and displays it in real-time.

The *flows.json* file contains the Node-RED configuration, which includes nodes to retrieve data from the MQTT broker and show it on the dashboard.

- **MQTT Nodes**: These nodes subscribe to specific topics and receive real-time data.
- **Dashboard Nodes**: These nodes display the data in various formats, including text, charts, and heatmaps.

```
{
        "id": "6dbd40de520d852a",
        "type": "ui_tab",
        "name": "WEDOFootball! Dashboard",
        "icon": "dashboard",
        "order": 1,
        "disabled": false,
        "hidden": false
},
```

Real-Time Data Display

The Node-RED configuration includes various UI groups and tabs for different players and metrics, ensuring comprehensive and organized data visualization

```
...

{
        "id": "751ca1a09dce3fc1",
        "type": "function",
        "z": "fdcff9d7ca80b512",
        "name": "Pressure",
        "func": "var player_topic = flow.get('player_topi
c');\nif (msg.topic === player_topic) {\n    var systolic =
msg.payload.blood_pressure.systolic;\n    var diastolic = m
sg.payload.blood_pressure.diastolic;\n\n// Round systolic a
nd diastolic values to the nearest integer\nvar roundedSyst
olic = Math.round(systolic);\nvar roundedDiastolic = Math.r
```

```
ound(diastolic);\n\n// Create a new object with rounded val
ues\nmsg.payload = {\n    blood_pressure: {\n        systol
ic: roundedSystolic,\n        diastolic: roundedDiastolic\n
}\n};\n\nreturn msg;\n} else {\n    return null;\n}\n",
        "outputs": 1,
        "timeout": 0,
        "noerr": 0,
        "initialize": "",
        "finalize": "",
        "libs": [],
        "x": 600,
        "y": 580,
        "wires": [
            [
                "fd1c983964ba1810",
                "b16dbda3bf6f6363"
            ]
        ]
    },

...
```

Database Integration

The project integrates MongoDB to handle data storage and retrieval for the simulation. MongoDB is used to store real-time data generated by the simulated sensors and to facilitate advanced data analysis. The integration involves setting up MongoDB connections, defining database and collection structures, and implementing data insertion and retrieval functionalities.

MongoDB Configuration and Initialization

```
MONGO_URI = "mongodb://localhost:27017/"
DATABASE_NAME = "footballDB"
BASE_COLLECTION_NAME = "simulations"
```

```
client = MongoClient(MONGO_URI)
db = client[DATABASE_NAME]
```

These snippets configure the connection to the MongoDB server and initialize the database. The *MONGO_URI* specifies the address of the MongoDB server, while *DATABASE_NAME* indicates the name of the database to use. The *BASE_COLLECTION_NAME* provides a base name for collections that store simulation data. The *MongoClient* establishes the connection, and the *db* variable is used to interact with the database.

Storing Simulation Data

```
def store_simulation_data(simulation_name, data):
    collection_name = f"{BASE_COLLECTION_NAME}_{simulation_
name}"
    collection = db[collection_name]
    try:
        collection.insert_one(data)
        print(f"Data inserted into MongoDB collection '{col
lection_name}': {data}")
    except Exception as e:
        print(f"Error inserting data into MongoDB: {e}")
```

This function is responsible for storing simulation data in MongoDB. It constructs a collection name using the base collection name and the simulation name, ensuring each simulation run has a unique collection. The data is then inserted into the collection, and the function provides feedback on the success or failure of the insertion.

Retrieving and Calculating Metrics

1. **Getting the Latest Collection:**

```
def get_latest_collection():
    collection_names = db.list_collection_names()
    simulation_collections = [name for name in collectio
n_names if name.startswith(BASE_COLLECTION_NAME)]
    if not simulation_collections:
        print("No simulation data found.")
```

```
        return None
    latest_collection = max(simulation_collections)
    return latest_collection
```

This function retrieves the latest or the current simulation collection from MongoDB by listing all collection names, filtering for those that start with the base collection name, and selecting the most recent one.

2. **Calculating Metrics:**

```
def calculate_metrics(collection_name):
    try:
        collection = db[collection_name]
        metrics = {}
        for player_id in range(1, 12):
            pipeline_avg_velocity = [
                {"$match": {"player_id": player_id}},
                {"$group": {"_id": None, "avg_velocity":
{"$avg": "$gps.velocity"}}}
            ]
            avg_velocity_result = list(collection.aggreg
ate(pipeline_avg_velocity))
            avg_velocity = avg_velocity_result[0]["avg_v
elocity"] if avg_velocity_result else 0.0
            ...
        return metrics
    except Exception as e:
        print(f"Error calculating metrics: {e}")
        return {}
```

This function calculates metrics such as the calories consumed at that time and the average velocity for each player. Based on this average value, we calculate the distance traveled of each player. It uses MongoDB aggregation pipelines to compute these metrics from the data stored in the latest simulation collection. The function handles potential errors and returns the calculated metrics.

Conclusion

The successful development and integration of the simulation system for tracking soccer players' movements, vital signs, and energy expenditure have demonstrated the power and versatility of modern IoT technologies and data management solutions. By leveraging MQTT for efficient real-time data communication, Node-RED for interactive data visualization, and MongoDB for robust data storage and analysis, this project showcases a comprehensive approach to sports analytics that can significantly benefit coaches, players, and fans.

Key Achievements:

- **Scalability and Flexibility:** The modular architecture of the system, including separate components for data generation, communication, visualization, and storage, ensures scalability and flexibility. New functionalities can be added with minimal disruption, and the system can be scaled to accommodate more players or different sports.
- **Real-Time Processing and Feedback:** The system's real-time capabilities mean that users can receive immediate feedback on the match and individual player performance. This is crucial for making informed decisions during a game and for post-match analysis.
- **User-Friendly Interface:** The GUI, powered by Node-RED, ensures that complex data is presented in an accessible and user-friendly manner. Coaches and analysts can easily interpret the data, making it actionable and relevant for training and strategy development.

Future Directions:

- **Enhanced Analytics:** Future enhancements could include more advanced analytical models and machine learning algorithms to predict player performance and potential injuries, providing proactive insights.
- **Integration with Wearable Technologies:** Integrating with wearable sensors can provide even more detailed and accurate data, enhancing the realism and utility of the simulation.

In conclusion, this project stands as a testament to the transformative potential of integrating IoT, real-time data processing, and advanced analytics in sports. It opens new directions for performance optimization, strategic planning, and fan engagement, ultimately contributing to the evolution of sports technology. The combination of tools and careful design ensures that this system is not

only a technological achievement but also a practical solution that meets the needs of modern sports analytics.