IoT Codes

Practical 1 : IoT Device Communication using MQTT or HTTP

AIM: To simulate IoT device communication using MQTT or HTTP protocols.
Objective: Demonstrate message publishing and subscribing using paho-mqtt in Google Colab.

```
!pip install paho-mqtt

import paho.mqtt.client as mqtt
import time

# Callback when connected to the broker
def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe("iot/test")

# Callback when a message is received
def on_message(client, userdata, msg):
    print(f"Message received: {msg.payload.decode()}")

# Create MQTT client instance
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

# Connect to public MQTT broker
client.connect("broker.hivemq.com", 1883, 60)

# Start loop to process network traffic
client.loop_start()

# Publish a test message
client.publish("iot/test", "Hello IoT")

# Stop the loop after short delay (optional in production)
time.sleep(2)  # Allow time for message to be received
client.loop_stop()
```

Practical 2 : Simulating Sensor Data Generation and Collection

AIM: To simulate sensor data generation and collection.
Objective: To simulate sensor data generation and collection for IoT applications. This includes:

- Generate random sensor data for parameters like temperature and humidity.
- Collect the generated data in real-time.
- Save the data to a file for further analysis.

```
import random
import time
import json
from datetime import datetime

# Generate sensor data for temperature, humidity, pressure, and light
def generate_sensor_data():
    data = {
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "temperature": round(random.uniform(20.0, 30.0), 2),     # °C
        "humidity": round(random.uniform(40.0, 60.0), 2),        # %
        "pressure": round(random.uniform(980.0, 1030.0), 2),     # hPa
        "light_intensity": round(random.uniform(0.0, 1000.0), 2) # lux
    }
    return data

# Collect sensor data for given duration and interval
def collect_sensor_data(duration, interval, output_file):
    collected_data = []
    start_time = time.time()
    print("Starting sensor data collection...\n")

    while time.time() - start_time < duration:
        data = generate_sensor_data()
        collected_data.append(data)
        print("Collected:", data)
        time.sleep(interval)

    # Save to JSON file
    with open(output_file, 'w') as f:
        json.dump(collected_data, f, indent=4)

    print(f"\nData saved to {output_file}")

# Main execution
if __name__ == "__main__":
```

```
    collect_sensor_data(duration=30, interval=5,
output_file="enhanced_sensor_data.json")
```

Practical 3 : Visualizing Sensor Data in Real-Time using Dash or Plotly

AIM: To visualize sensor data in real-time using Dash or Plotly.
Objective: To build an interactive dashboard for monitoring IoT data, enabling users to visualize real-time trends and analyze sensor readings effectively. The homework section also states the objective as: To build an interactive dashboard for monitoring IoT data, enabling users to visualize real-time trends and analyze sensor readings effectively.

```
!pip install dash

from dash import Dash, dcc, html
from dash.dependencies import Input, Output
import plotly.graph_objs as go
import random
import time
import threading

# Simulated data store
data = {
    'time': [],
    'temperature': [],
    'humidity': [],
    'air_pressure': []
}

# Simulated Sensor Data Generator
def generate_sensor_data():
    while True:
        data['temperature'].append(round(random.uniform(20.0, 30.0), 2))
        data['humidity'].append(round(random.uniform(40.0, 60.0), 2))
        data['air_pressure'].append(round(random.uniform(980.0, 1050.0),
2))
        data['time'].append(time.strftime("%H:%M:%S"))

        if len(data['time']) > 100:
            for key in data:
                data[key] = data[key][-100:]

        time.sleep(1)
```

```
# Start sensor data thread
threading.Thread(target=generate_sensor_data, daemon=True).start()

# Dash App
app = Dash(__name__)
app.layout = html.Div([
    html.H1("Real-Time IoT Sensor Dashboard", style={'textAlign':
'center'}),

    # Digital Displays
    html.Div([
        html.Div(id='temperature-digital', style={'fontSize': '30px',
'margin': '10px'}),
        html.Div(id='humidity-digital', style={'fontSize': '30px',
'margin': '10px'}),
        html.Div(id='air-pressure-digital', style={'fontSize': '30px',
'margin': '10px'})
    ], style={'display': 'flex', 'justifyContent': 'space-around'}),

    # Graphs
    html.Div([
        dcc.Graph(id='temperature-graph'),
        dcc.Graph(id='humidity-graph'),
        dcc.Graph(id='air-pressure-graph')
    ]),

    dcc.Interval(id='interval-component', interval=1000, n_intervals=0)
])

@app.callback(
    [Output('temperature-graph', 'figure'),
     Output('humidity-graph', 'figure'),
     Output('air-pressure-graph', 'figure'),
     Output('temperature-digital', 'children'),
     Output('humidity-digital', 'children'),
     Output('air-pressure-digital', 'children')],
    [Input('interval-component', 'n_intervals')]
)
def update_dashboard(n):
    temp_fig = go.Figure([go.Scatter(x=data['time'],
y=data['temperature'], mode='lines+markers')])
    temp_fig.update_layout(title="Temperature Over Time",
xaxis_title="Time", yaxis_title="°C")

    hum_fig = go.Figure([go.Scatter(x=data['time'], y=data['humidity'],
mode='lines+markers')])
    hum_fig.update_layout(title="Humidity Over Time", xaxis_title="Time",
yaxis_title="%")

    press_fig = go.Figure([go.Scatter(x=data['time'],
```

```
y=data['air_pressure'], mode='lines+markers')])
    press_fig.update_layout(title="Air Pressure Over Time",
xaxis_title="Time", yaxis_title="hPa")

    temp_val = f"Temperature: {data['temperature'][-1]} °C" if
data['temperature'] else "N/A"
    hum_val = f"Humidity: {data['humidity'][-1]} %" if data['humidity']
else "N/A"
    press_val = f"Air Pressure: {data['air_pressure'][-1]} hPa" if
data['air_pressure'] else "N/A"

    return temp_fig, hum_fig, press_fig, temp_val, hum_val, press_val

if __name__ == '__main__':
    app.run(debug=True)
```

Practical 4 : Sending and Visualizing Simulated IoT Data using ThingSpeak

AIM: To send and visualize simulated IoT data using ThingSpeak.
Objective: Interact with ThingSpeak's API to upload and display IoT data.

```
import requests
import random
import time
import matplotlib.pyplot as plt
from datetime import datetime

# ThingSpeak API Keys and Channel ID
WRITE_API_KEY = 'YOUR_WRITE_API_KEY'
READ_API_KEY = 'YOUR_READ_API_KEY'
CHANNEL_ID = 'YOUR_CHANNEL_ID'

# Function to send data to ThingSpeak
def send_data_to_thingspeak(temperature, humidity, pressure):
    url = f"https://api.thingspeak.com/update?api_key=
{WRITE_API_KEY}&field1={temperature}&field2={humidity}&field3={pressure}"
    response = requests.get(url)
    if response.status_code == 200:
        print(f"Data Sent: Temp={temperature}°C, Humidity={humidity}%,
Pressure={pressure} hPa")
    else:
        print("Failed to send data")

# Simulate and upload sensor data
for _ in range(10):  # Send 10 data points
```

```
    temperature = round(random.uniform(20.0, 30.0), 2)
    humidity = round(random.uniform(40.0, 60.0), 2)
    pressure = round(random.uniform(900, 1100), 2)
    send_data_to_thingspeak(temperature, humidity, pressure)
    time.sleep(15)  # ThingSpeak allows updates every 15 seconds

# Function to retrieve data from ThingSpeak
def retrieve_data_from_thingspeak():
    url = f"https://api.thingspeak.com/channels/{CHANNEL_ID}/feeds.json?
api_key={READ_API_KEY}&results=10"
    response = requests.get(url).json()
    feeds = response['feeds']
    timestamps = [datetime.strptime(feed['created_at'], '%Y-%m-
%dT%H:%M:%SZ') for feed in feeds]
    temperatures = [float(feed['field1']) for feed in feeds]
    humidities = [float(feed['field2']) for feed in feeds]
    pressures = [float(feed['field3']) for feed in feeds]
    return timestamps, temperatures, humidities, pressures

# Retrieve and plot data
timestamps, temperatures, humidities, pressures =
retrieve_data_from_thingspeak()
plt.figure(figsize=(12, 6))
plt.plot(timestamps, temperatures, label='Temperature (°C)', marker='o')
plt.plot(timestamps, humidities, label='Humidity (%)', marker='s')
plt.plot(timestamps, pressures, label='Pressure (hPa)', marker='^')
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('IoT Sensor Data from ThingSpeak')
plt.legend()
plt.grid(True)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

Practical 5 : Simulating Edge Data Processing and Filtering

AIM: To simulate edge data processing and filtering.
Objective: Process simulated IoT sensor data using basic filtering techniques. The homework section implicitly aims to extend this by adding more filtering techniques.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
from filterpy.kalman import KalmanFilter
from filterpy.common import Q_discrete_white_noise

# Simulated Sensor Data Generation
def generate_sensor_data(n=100):
    np.random.seed(42)
    temperature = np.random.normal(25, 2, n)  # Mean 25C, Std Dev 2
    humidity = np.random.normal(50, 5, n)  # Mean 50%, Std Dev 5
    return pd.DataFrame({'Temperature': temperature, 'Humidity':
humidity})

# Moving Average Filter
def moving_average_filter(data, window_size=5):
    return data.rolling(window=window_size, center=True).mean()

# Median Filter
def median_filter(data, window_size=5):
    return data.rolling(window=window_size, center=True).median()

# Threshold-Based Filtering
def threshold_filter(data, temp_range=(20, 30), hum_range=(40, 60)):
    filtered_data = data[(data['Temperature'].between(*temp_range)) &
                         (data['Humidity'].between(*hum_range))]
    return filtered_data

# Kalman Filter Implementation
def setup_kalman_filter():
    kf = KalmanFilter(dim_x=2, dim_z=1)  # State: [position, velocity],
Measurement: position
    dt = 1.0  # time step
    kf.F = np.array([[1., dt], [0., 1.]])  # State transition matrix
    kf.H = np.array([[1., 0.]])  # Measurement matrix
    kf.R = np.array([[0.5]])  # Measurement noise variance
    kf.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.13)  # Process noise
    kf.x = np.array([[0.], [0.]])  # Initial state
    kf.P = np.array([[1., 0.], [0., 1.]])  # Initial state covariance
    return kf

def kalman_filter(data):
    kf = setup_kalman_filter()
    filtered_data = np.zeros(len(data))
    for i, measurement in enumerate(data):
        kf.predict()
        kf.update(measurement)
        filtered_data[i] = kf.x[0]
    return filtered_data

# Generate and process data
data = generate_sensor_data()
smoothed_data = moving_average_filter(data)
```

```
median_filtered_data = median_filter(data)
kalman_filtered_temp = kalman_filter(data['Temperature'].values)

# Visualization
plt.figure(figsize=(12, 8))

# Plot 1: Temperature
plt.subplot(2, 1, 1)
plt.plot(data['Temperature'], 'gray', alpha=0.5, label='Raw Temperature',
linestyle='dotted')
plt.plot(smoothed_data['Temperature'], 'b', label='Moving Average',
alpha=0.7)
plt.plot(median_filtered_data['Temperature'], 'g', label='Median Filter',
alpha=0.7)
plt.plot(kalman_filtered_temp, 'r', label='Kalman Filter', alpha=0.7)
plt.ylabel('Temperature (°C)')
plt.title('Comparison of Different Filtering Techniques')
plt.legend()
plt.grid(True, alpha=0.3)

# Plot 2: Error Analysis
plt.subplot(2, 1, 2)
mae_ma = np.abs(data['Temperature'] - smoothed_data['Temperature']).mean()
mae_median = np.abs(data['Temperature'] -
median_filtered_data['Temperature']).mean()
mae_kalman = np.abs(data['Temperature'] - kalman_filtered_temp).mean()
errors = [mae_ma, mae_median, mae_kalman]
plt.bar(['Moving Average', 'Median Filter', 'Kalman Filter'], errors,
color=['blue', 'green', 'red'], alpha=0.6)
plt.ylabel('Mean Absolute Error')
plt.title('Error Analysis of Different Filters')
plt.tight_layout()
plt.show()

# Print error metrics
print("\nError Analysis:")
print(f"Moving Average MAE: {mae_ma:.3f}°C")
print(f"Median Filter MAE: {mae_median:.3f}°C")
print(f"Kalman Filter MAE: {mae_kalman:.3f}°C")
```

Practical 6 : Simulating Basic IoT Security Measures

AIM: To simulate basic IoT security measures. The homework section states the AIM as: To simulate basic IoT security measures using HMAC.
Objective: Test IoT security vulnerabilities using Python tools. The homework section

states the Objective as: Test IoT security vulnerabilities using Python tools with HMAC for message authentication.

Part 1: Basic hashing with SHA-256

```
import hashlib
import random
import time

def hash_data(data):
    """Hashes input data using SHA-256 algorithm."""
    return hashlib.sha256(data.encode()).hexdigest()

def verify_hash(original_data, hashed_data):
    """Verifies whether the hash of the given data matches the stored
hash."""
    return hash_data(original_data) == hashed_data

try:
    while True:
        # Generate random IoT sensor data
        temperature = round(random.uniform(20.0, 30.0), 1)
        humidity = random.randint(40, 80)
        sensor_data = f"Temperature: {temperature}C, Humidity:
{humidity}%"

        # Apply Hashing
        hashed_data = hash_data(sensor_data)
        verification = verify_hash(sensor_data, hashed_data)

        # Print results
        print("\nOriginal Data:", sensor_data)
        print("Hashed Data:", hashed_data)
        print("Verification Successful:", verification)

        # Pause before generating new data
        time.sleep(2)
except KeyboardInterrupt:
    print("\nProcess stopped by user.")
```

Part 2: HMAC for Message Authentication

```
import hmac

def generate_hmac(data, key):
    """Generates an HMAC authentication code using MD5."""
    return hmac.new(key.encode(), data.encode(), hashlib.md5).hexdigest()
```

```
def verify_hmac(data, key, received_hmac):
    """Verifies if the received HMAC matches the computed one."""
    computed_hmac = generate_hmac(data, key)
    return computed_hmac == received_hmac

def generate_sensor_data():
    """Generates random sensor data for Temperature, Humidity, and
Pressure."""
    temperature = round(random.uniform(20.0, 30.0), 1)
    humidity = round(random.uniform(40.0, 70.0), 1)
    pressure = round(random.uniform(900.0, 1100.0), 1)
    return f"Temperature: {temperature}C, Humidity: {humidity}%, Pressure:
{pressure} hPa"

secret_key = "IoT_Secret_Key"

try:
    while True:
        sensor_data = generate_sensor_data()
        hashed_data = hash_data(sensor_data)
        hmac_data = generate_hmac(sensor_data, secret_key)

        print("\nOriginal Data:", sensor_data)
        print("Hashed Data:", hashed_data)
        print("Generated HMAC:", hmac_data)
        print("Verification Successful:", verify_hmac(sensor_data,
secret_key, hmac_data))

        time.sleep(2)  # Wait for 2 seconds before generating new values
except KeyboardInterrupt:
    print("\nProcess stopped by user.")
```

Practical 7 : Simple IoT-based Device Control System using Flask and ngrok

Aim: To develop a simple IoT-based device control system using Flask as a web framework and ngrok for exposing the local server to the internet, allowing remote control of an LED-like device. The homework section states the Aim as: Extend the program by adding another virtual IoT device (e.g., a fan) and allow users to control its state.

Objectives:

- To implement a web-based interface for controlling an IoT device (LED). The homework section updates this to: To implement a web-based interface for controlling an IoT device (LED & FAN).

- To use Flask to create a lightweight API for device control.
- To expose the local Flask server to the internet using ngrok.
- To enable remote control of the device via API calls.

Part 1: Basic LED Control

```
from flask import Flask, request, jsonify

app = Flask(__name__)
device_state = {"LED": "OFF"}  # Initial state of the virtual IoT device

@app.route('/')
def home():
    return "<h1>IoT Device Control</h1><p>Use /control?device=LED&state=ON
to control the device.</p>"

@app.route('/control', methods=['GET'])
def control_device():
    device = request.args.get('device')
    state = request.args.get('state')
    if device in device_state and state in ["ON", "OFF"]:
        device_state[device] = state
        return jsonify({"message": f"{device} turned {state}"})
    else:
        return jsonify({"error": "Invalid device or state"})

@app.route('/status', methods=['GET'])
def device_status():
    return jsonify(device_state)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Part 2: Extended to Include a Fan

```
from flask import Flask, request, jsonify

app = Flask(__name__)
device_state = {"LED": "OFF", "Fan": "OFF"}  # Initial state of LED and
Fan

@app.route('/')
def home():
    return "<h1>IoT Device Control</h1><p>Use /control?device=FAN&state=ON
to control the device.</p>"

@app.route('/control', methods=['GET'])
```

```python
def control_device():
    device = request.args.get('device')
    state = request.args.get('state')
    if device in device_state and state in ["ON", "OFF"]:
        device_state[device] = state
        return jsonify({"message": f"{device} turned {state}"})
    else:
        return jsonify({"error": "Invalid device or state"})

@app.route('/status', methods=['GET'])
def device_status():
    return jsonify(device_state)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Practical 8 : Predicting Equipment Failures using IoT Sensor Data (Python)

AIM: To predict equipment failures using IoT sensor data.
Objective: Apply machine learning for predictive maintenance using Python's scikit-learn library. The homework section also includes implementing a deep learning model.

Part 1: Data Generation and Pre-processing

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Set random seed for reproducibility
np.random.seed(42)

# Generate dataset with 1000 samples
num_samples = 1000

# Simulate sensor readings
temperature = np.random.randint(30, 100, num_samples)  # Temperature in
Celsius
vibration = np.round(np.random.uniform(0.1, 3.0, num_samples), 2)  #
Vibration level
```

```
pressure = np.random.randint(50, 400, num_samples)  # Pressure in kPa
humidity = np.random.randint(20, 80, num_samples)  # Humidity in %
machine_age = np.random.randint(1, 20, num_samples)  # Machine age in
years

# Generate failure labels (1 = Failure, 0 = No Failure) based on
conditions
failure = np.where(
    (temperature > 80) & (vibration > 2.0) & (pressure > 300) & (humidity
> 60),
    1,  # High failure risk
    np.random.choice([0, 1], size=num_samples, p=[0.85, 0.15])  # 15%
random failures
)

# Create DataFrame
df = pd.DataFrame({
    "Temperature": temperature,
    "Vibration": vibration,
    "Pressure": pressure,
    "Humidity": humidity,
    "Machine_Age": machine_age,
    "Failure": failure
})

# Save dataset as CSV file
csv_filename = "iot_sensor_data.csv"
df.to_csv(csv_filename, index=False)
print(f"Dataset generated and saved as '{csv_filename}' successfully!")

# Display first 5 rows of the dataset
print("\nFirst 5 rows of the dataset:")
print(df.head())

# Load dataset (if running separately)
df = pd.read_csv("iot_sensor_data.csv")

# Split dataset into features and labels
X = df.drop(columns=['Failure'])  # Features
y = df['Failure']  # Target variable

# Split into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Part 2: Random Forest Model

```
# Train a Random Forest model
model_rf = RandomForestClassifier(n_estimators=100, random_state=42)
model_rf.fit(X_train, y_train)

# Predictions and Evaluation for Random Forest
y_pred_rf = model_rf.predict(X_test)
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print("\nRandom Forest Model Accuracy:", accuracy_rf)
print("\nRandom Forest Classification Report:\n",
classification_report(y_test, y_pred_rf))

# Confusion Matrix Visualization for Random Forest
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, fmt='d',
cmap='Blues',
            xticklabels=['No Failure', 'Failure'], yticklabels=['No
Failure', 'Failure'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Random Forest Confusion Matrix")
plt.show()

# Feature Importance Analysis for Random Forest
feature_importance = pd.Series(model_rf.feature_importances_,
index=X.columns).sort_values(ascending=False)
plt.figure(figsize=(8, 5))
sns.barplot(x=feature_importance, y=feature_importance.index)
plt.xlabel("Feature Importance Score")
plt.ylabel("Features")
plt.title("Feature Importance in Random Forest Model")
plt.show()
```

Part 3: Neural Network Model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Build a Neural Network model
model_nn = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),  #
Input layer
    Dense(32, activation='relu'),  # Hidden layer
    Dense(1, activation='sigmoid')  # Output layer for binary
classification
])

# Compile the model
```

```
model_nn.compile(optimizer='adam', loss='binary_crossentropy', metrics=
['accuracy'])

# Train the neural network
model_nn.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)  #
Adjust epochs and batch size as needed

# Evaluate the neural network
loss, accuracy_nn = model_nn.evaluate(X_test, y_test, verbose=0)
print(f"\nNeural Network Accuracy: {accuracy_nn}")

# Predictions using Neural Network
y_pred_nn = (model_nn.predict(X_test) > 0.5).astype("int32")  # Predict
using the neural network

# Classification Report for Neural Network
print("\nNeural Network Classification Report:\n",
classification_report(y_test, y_pred_nn))

# Confusion Matrix Visualization for Neural Network
plt.figure(figsize=(5, 4))
sns.heatmap(confusion_matrix(y_test, y_pred_nn), annot=True, fmt='d',
cmap='Blues',
            xticklabels=['No Failure', 'Failure'], yticklabels=['No
Failure', 'Failure'])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Neural Network Confusion Matrix")
plt.show()
```