WatchDogeSupercharged

Net and Light

2023 - 2024

Contents

1 Introduction **2**

2 Project Overview **2**

3 System Architecture **2**

4 Features **2**

5 Code Description **2**

5.1 Imports and Configuration 2
5.2 Logging Setup 3
5.3 Flask App Initialization 3
5.4 Job Function 3
5.5 Compteur Bille Function 4
5.6 Get Info Function 4
5.7 Multi Possible Function 5
5.8 Check Plus De 3 Function 6
5.9 Disconnection Function 6

6 Main Execution **6**

7 Conclusion **7**

1 Introduction

This document provides detailed documentation for a Python script that interacts with multiple PLCs (Programmable Logic Controllers) using Flask for the web interface and APScheduler for periodic task scheduling. The script fetches IP addresses, retrieves and processes data, and handles various tasks including logging and sending alerts.

2 Project Overview

The script is designed to monitor and control a network of PLCs by performing periodic checks, retrieving data, and responding to specific conditions. It utilizes several libraries and modules to achieve its functionality:

- `Flask` for creating a web application.
- `APScheduler` for scheduling periodic tasks.
- `requests` for making HTTP requests.
- `logging` for logging messages.
- `numpy` for numerical operations.
- `pygame` for playing sound alerts.

3 System Architecture

The system consists of the following main components:

- **Flask Web App**: Serves as the entry point and provides endpoints for various tasks.
- **Scheduler**: Manages periodic execution of tasks.
- **Logging**: Handles logging of events and errors.
- **HTTP Requests**: Interacts with PLCs through HTTP endpoints.
- **Data Processing**: Processes data received from PLCs.

4 Features

The script offers several key features:

- **Periodic Task Execution**: Uses APScheduler to schedule tasks at regular intervals.
- **Logging**: Logs important events and errors to a file.
- **HTTP Requests**: Sends and receives data from PLCs via HTTP requests.
- **Multi-threading**: Utilizes threading to perform tasks concurrently.
- **Alerts**: Sends alerts and plays sound warnings based on certain conditions.

5 Code Description

5.1 Imports and Configuration

```python
from flask import Flask, jsonify  # Import Flask to create a web app and jsonify to return JSON responses
from apscheduler.schedulers.background import BackgroundScheduler  # Import APScheduler for background task scheduling
from multiprocessing.dummy import Pool as ThreadPool  # Import ThreadPool for parallel task execution
import time
from ip_addresses import fetch_latest_ip_addresses  # Import the fetch_latest_ip_addresses function from ip_addresses.py
import requests  # Imports the requests module to make HTTP requests
import logging  # Imports the logging module for logging messages
import numpy as np  # Import NumPy for numerical operations
import pygame  # Import Pygame for creating video games
import json  # Imports the json module for parsing and generating JSON data

# Load configuration from a JSON file
with open('configDoge.json', 'r') as config_file:
    config = json.load(config_file)
grosse_horloge = time.time()
running_time = time.time()
```

Listing 1: Imports and Configuration

5.2 Logging Setup

```python
def create_logs():
    logging.basicConfig(
        filename=config['logging']['filename'],
        filemode=config['logging']['filemode'],
        format=config['logging']['format'],
        level=getattr(logging, config['logging']['level'])
    )

create_logs()
```

Listing 2: Logging Setup

5.3 Flask App Initialization

```python
# Starts the web app
app = Flask(__name__)
```

Listing 3: Flask App Initialization

5.4 Job Function

```python
def job():
    global running_time
    """
    A scheduled job function to handle a series of tasks periodically.
    It reschedules itself and logs execution time and completion.
    """
    print("+---------------------------------------+")
    logging.info('Job started')
    start_time = time.time()  # Capture the start time

    NombreAPIConnceted = 0

    NombreAPIConnceted = multi_possible(NombreAPIConnceted, fetch_latest_ip_addresses())
    nombremultisql = 2

    checkplusde3(nombremultisql, fetch_latest_ip_addresses())
    get_info(NombreAPIConnceted, fetch_latest_ip_addresses())
    compteur_bille(fetch_latest_ip_addresses())
    deconnection(fetch_latest_ip_addresses())

    logging.info('Job finished')
    print(f"Execution time: {time.time() - start_time}")
```

```
    temps_depuis_le_lancement = time.time() - grosse_horloge
    print(f"Temps depuis le lancement : {temps_depuis_le_lancement//3600} heures {
    temps_depuis_le_lancement%3600//60} minutes {temps_depuis_le_lancement%60} secondes"
    )
    if time.time() - running_time > config['logging']['time_before_flush']:
        # if the program has been running for more than an hour, delete the logs
        open(config['logging']['filename'], 'w').close()
        running_time = time.time()
        create_logs()

    scheduler.add_job(job)  # Reschedule the job for future execution

    print("+--------------------------------------+")
```

Listing 4: Job Function

5.5 Compteur Bille Function

```
def compteur_bille(ip_addresses):
    """ Computes the number of marbles in the circuit """
    A = np.zeros(3, dtype=int)  # Initialize an array to store counts of 'compteur_bille
    ' with three zeros
    i = 0  # Counter for tracking successful API responses

    # Function to fetch the number of marbles from each PLC
    def fetch_data(ip_address):
        nonlocal A, i  # Allow the function to modify A and i defined in the outer scope
        try:
            response = requests.get(f"http://{ip_address['RASP_catch']}:8000/
    compteur_bille")
            A = np.add(A, response.json())  # Add the received data to the existing
    matrix A
            print(f"Matrice de l'API : {ip_address['API']} : {response.json()}")
            i += 1
        except Exception as e:
            logging.error(e)
            print("Error occurred during API request.")

    pool = ThreadPool(len(ip_addresses))  # Create a thread pool with a size equal to
    the number of IP addresses
    pool.map(fetch_data, ip_addresses.values())  # Map the fetch_data function to each
    IP address, running them concurrently
    pool.close()  # Close the pool
    pool.join()   # and wait for all tasks to complete

    print(f"Matrice totale : {A}")
    if i != len(ip_addresses):  # Check if all PLC responses were successful
        print("Erreur de compteur")
    else:
        # Function to send the total counts back to each PLC
        def send_data(ip_address):
            try:
                for i in range(3):
                    requests.get(f"http://{ip_address['RASP_catch']}:8000/compteur_bille
    /{i}/{A[i]}")
            except Exception as e:
                logging.error(e)

        pool = ThreadPool(len(ip_addresses))  # Idem
        pool.map(send_data, ip_addresses.values())
        pool.close()
        pool.join()
```

Listing 5: Compteur Bille Function

5.6 Get Info Function

```
valid_request = config['requests']

def get_info(NombreAPIConnceted, ip_addresses):
```

```
    """
    Retrieves and processes information for each connected PLC based on predefined
    requests.
    It sends requests to each IP address for each type of configured request, processes
    the responses,
    and may trigger warnings or further actions depending on the response content.
    """
    if valid_request:
        logging.info('Request get_info started')
        for i in range(len(valid_request)):
            valid_request[i]["values"] = np.zeros(len(ip_addresses))
        # Function to send requests and gather information
        def fetch_data(ip):
            for i in range(len(valid_request)):
                try:
                    response = requests.get(f"http://{ip['RASP_catch']}:8000/{
    valid_request[i]['name']}")
                    valid_request[i]["values"][list(ip_addresses.keys()).index(ip['API'
    ])] = response.json()["value"]
                except Exception as e:
                    logging.error(e)
        pool = ThreadPool(len(ip_addresses))  # Create a thread pool
        pool.map(fetch_data, ip_addresses.values())  # Map the fetch_data function to
    each IP address
        pool.close()  # Close the pool
        pool.join()   # and wait for all tasks to complete

        print("Data fetched successfully:")
        for i in range(len(valid_request)):
            print(valid_request[i]["values"])

        if NombreAPIConnceted == len(ip_addresses):
            for i in range(len(valid_request)):
                if valid_request[i]["sum"] and np.sum(valid_request[i]["values"]) >
    valid_request[i]["threshold"]:
                    print(f"Warning: Sum of {valid_request[i]['name']} exceeded
    threshold")
                    pygame.mixer.init()
                    pygame.mixer.music.load(valid_request[i]['sound'])
                    pygame.mixer.music.play()
        logging.info('Request get_info finished')
```

Listing 6: Get Info Function

5.7 Multi Possible Function

```
def multi_possible(NombreAPIConnceted, ip_addresses):
    """ Checks if multiple PLCs are connected simultaneously """
    A = np.zeros(3, dtype=int)
    i = 0

    def fetch_data(ip):
        nonlocal A, i, NombreAPIConnceted
        try:
            response = requests.get(f"http://{ip['RASP_catch']}:8000/multi_possible")
            A = np.add(A, response.json())
            i += 1
        except Exception as e:
            logging.error(e)

    pool = ThreadPool(len(ip_addresses))
    pool.map(fetch_data, ip_addresses.values())
    pool.close()
    pool.join()

    if i != len(ip_addresses):
        logging.error("Connection error in multi_possible.")
    else:
        print(f"Matrice multi_possible : {A}")
        NombreAPIConnceted = i
```

```
    return NombreAPIConnceted
```

Listing 7: Multi Possible Function

5.8 Check Plus De 3 Function

```
def checkplusde3(nombremultisql, ip_addresses):
    """
    Checks and logs if there are more than three connections to the PLC network.
    """
    A = np.zeros(3, dtype=int)
    i = 0

    def fetch_data(ip):
        nonlocal A, i, nombremultisql
        try:
            response = requests.get(f"http://{ip['RASP_catch']}:8000/checkplusde3")
            A = np.add(A, response.json())
            i += 1
        except Exception as e:
            logging.error(e)

    pool = ThreadPool(len(ip_addresses))
    pool.map(fetch_data, ip_addresses.values())
    pool.close()
    pool.join()

    if i != len(ip_addresses):
        logging.error("Connection error in checkplusde3.")
    else:
        print(f"Matrice checkplusde3 : {A}")
        nombremultisql = i
```

Listing 8: Check Plus De 3 Function

5.9 Disconnection Function

```
def deconnection(ip_addresses):
    """
    Handles the disconnection of PLCs by sending a disconnection request to each PLC.
    """
    def fetch_data(ip):
        try:
            response = requests.get(f"http://{ip['RASP_catch']}:8000/deconnection")
            print(f"Disconnection status for {ip['API']} : {response.json()}")
        except Exception as e:
            logging.error(e)

    pool = ThreadPool(len(ip_addresses))
    pool.map(fetch_data, ip_addresses.values())
    pool.close()
    pool.join()
```

Listing 9: Disconnection Function

6 Main Execution

```
if __name__ == '__main__':
    scheduler = BackgroundScheduler()
    scheduler.add_job(job, 'interval', minutes=config['job']['interval_minutes'])
    scheduler.start()

    try:
        app.run(host=config['flask']['host'], port=config['flask']['port'])
    except (KeyboardInterrupt, SystemExit):
        pass

```

```
11      scheduler.shutdown()
```

Listing 10: Main Execution

7 Conclusion

This document provides an overview and detailed description of a Python script for monitoring and controlling PLCs using Flask and APScheduler. The script performs periodic checks, retrieves data, processes it, and handles various tasks including logging and sending alerts.