```python
from threading import Thread

import time
from time import time as unixtime
import requests
requests.packages.urllib3.disable_warnings()
import xml.etree.ElementTree as ET

from collections import OrderedDict

from requests.adapters import HTTPAdapter, Retry
from typing import Optional

from device_name_mapping import HOSTNAME_TO_IP

def parse_allnet_json(j_decoded):
    # j_decoded = json.loads(j, encoding='ISO-8859-1')
    d = OrderedDict({'Wechselspannung': None,
                     'Wechselstrom': None,
                     'Leistung': None,
                     'Leistungsfaktor': None,
                     'Frequenz': None,
                     'Kontakt Eingang': None,
                     'Intern': None,
                     'Schaltrelais': None,
                     'Geräte LED': None,
                     'Geräte LED 3': None
                     })
    # for sub_dict in j_decoded:
    #    key = sub_dict['name']
    #    messwert = float(sub_dict['value'])
    #    d[key] = messwert
    sensors = ET.fromstring(j_decoded)
    for sensor in sensors:
        measurement_id = sensor[1].text
        measurement = sensor[2].text
        mapped_id = mapSensorIDToDict(measurement_id)
        if measurement == 'error' and mapped_id is not None:
            d = None
            return d
        else:
            d[mapped_id] = measurement
    return d


def mapSensorIDToDict(measurement_id: str) -> Optional[str]:
    """
    Maps ordered xml sensor objects to the dict names defined in OrderedDict
    @rtype: str
    """
    sensor_map_dict = {'AC Voltage': 'Wechselspannung',
                       'AC Current': 'Wechselstrom',
                       'Power': 'Leistung',
                       'Power factor': 'Leistungsfaktor',
                       'Frequency': 'Frequenz',
                       'Contact input': 'Kontakt Eingang',
                       'Internal': 'Intern'}
    if measurement_id in sensor_map_dict:
        return sensor_map_dict[measurement_id]
    else:
        return None


class AllnetPoll(Thread):
    TIMEOUT = 20  # max response-time with one powerplug recorded = 11.14s

    def __init__(self, name, output_queue, auth=None):
        super(AllnetPoll, self).__init__()
        self.name = str(name)
        self.daemon = True
        self.ip = HOSTNAME_TO_IP[name]
        if auth is None:
            self.url = "https://%s/xml/?mode=sensor"
```

```
 % self.ip
        else:
            self.url = f'https://{auth["username"]}:{auth["password"]}@{self.ip}/xml/?mode=sensor'
        self.output_queue = output_queue
        retry_counter = 0
        while retry_counter < 2:
            try:
                session = requests.Session()
                retry = Retry(connect=3, backoff_factor=0.5)
                adapter = HTTPAdapter(max_retries=retry)
                session.mount('http://', adapter)
                session.mount('https://', adapter)
                response = session.get(self.url, timeout=AllnetPoll.TIMEOUT, verify=False)
                assert response.status_code == 200, ('HTTP-Statuscode', response.status_code, response.content)
                print(f'{self.url} is ok!')
                retry_counter = 2
            except requests.exceptions.Timeout as e:
                print("%s %s is unreachable" % (self.name, self.ip))
                raise e
            except requests.exceptions.ConnectionError:
                print(f'waiting for {self.ip} to reconnect')
                time.sleep(10)
                retry_counter += 1

    def run(self):
        with requests.Session() as session:
            while True:
                try:
                    t_request = unixtime()
                    response = session.get(self.url, timeout=AllnetPoll.TIMEOUT, verify=False)
                    assert response.status_code ==
```

```python
200, ('HTTP-Statuscode', response.status_code, response.content)
                    t_reply = unixtime()

                    allnet_dict = parse_allnet_json(response.content.decode('utf-8'))
                    if allnet_dict:  # allnet_dict is None if the parser encounters an error
                        allnet_dict['Unixtime Request'] = t_request
                        allnet_dict['Unixtime Reply'] = t_reply
                        allnet_dict['DeviceName'] = self.name

                        self.output_queue.put(allnet_dict)
                except requests.exceptions.Timeout as e:
                    print("%s %s Timeout " % (self.name, self.ip), e)
                except requests.exceptions.ConnectionError as e:
                    # ConnectionError occurs if plug is unreachable, eg during and after a powerloss
                    print("%s %s No Connection to Host " % (self.name, self.ip), e)
                    time.sleep(3)
                except requests.exceptions.RequestException as e:
                    print("------ Unknown Exception ------ ", e)
                    time.sleep(3)

```

```python
# Object containing the mapping of sensor names and ip addresses
HOSTNAME_TO_IP = {
    'BLADL_00_001' : '132.231.12.151:100',  #Sensor 1
    'BLADL_00_002' : '132.231.12.151:101',  #Sensor 2
    'BLADL_00_003' : '132.231.12.151:102',  #Sensor 3
    'BLADL_00_004' : '132.231.12.151:103',  #Sensor 4
    'BLADL_00_005' : '132.231.12.151:104',  #Sensor 5
    'BLADL_00_006' : '132.231.12.151:105',  #Sensor 6
    'BLADL_00_007' : '132.231.12.151:106',  #Sensor 7
    'BLADL_00_008' : '132.231.12.151:107',  #Sensor 8
    'BLADL_00_009' : '132.231.12.151:108',  #Sensor 9
    'BLADL_00_010' : '132.231.12.151:109',  #Sensor 10

    'BLADL_00_011' : '132.231.12.151:110', #Sensor 11
    'BLADL_00_012' : '132.231.12.151:111', #Sensor 12
    'BLADL_00_013' : '132.231.12.151:112', #Sensor 13
    #'BLADL_01_004' : '192.168.42.113', Prof
    'BLADL_00_014' : '132.231.12.151:114', #Sensor 15
    'BLADL_00_015' : '132.231.12.151:115', #Sensor 16
    'BLADL_00_016' : '132.231.12.151:116', #Sensor 17
    'BLADL_00_017' : '132.231.12.151:117', #Sensor 18
    'BLADL_00_018' : '132.231.12.151:118', #Sensor 19
    'BLADL_00_019' : '132.231.12.151:119', #Sensor 20

    'BLADL_00_020' : '132.231.12.151:120', #Sensor 21
    'BLADL_00_021' : '132.231.12.151:121', #Sensor 22
    'BLADL_00_022' : '132.231.12.151:122', #Sensor 23
    'BLADL_00_023' : '132.231.12.151:123', #Sensor 24
    'BLADL_00_024' : '132.231.12.151:124', #Sensor 25
    'BLADL_00_025' : '132.231.12.151:125', #Sensor 26
    'BLADL_00_026' : '132.231.12.151:126', #Sensor 27
    'BLADL_00_027' : '132.231.12.151:127', #Sensor 28
    'BLADL_00_028' : '132.231.12.151:128', #Sensor 29
    'BLADL_00_029' : '132.231.12.151:129', #Sensor 30

    'BLADL_00_030' : '132.231.12.151:130', #switched to coffee Sensor 31
    'BLADL_00_031' : '132.231.12.151:131', #Sensor 32
    'BLADL_00_032' : '132.231.12.151:132', #switched
to Alex Sensor 33
    'BLADL_00_033' : '132.231.12.151:133', #Sensor 34
    'BLADL_00_034' : '132.231.12.151:134', #Sensor 35
    'BLADL_00_035' : '132.231.12.151:135', #Sensor 36
    #'BLADL_00_036' : '132.231.12.151:136', Sensor 37 error values
    'BLADL_00_036' : '132.231.12.151:137', #Sensor 38
    'BLADL_00_037' : '132.231.12.151:138', #Sensor 39
    #'BLADL_03_010' : '192.168.42.170', unused
}
```

```
RABBITMQ_SERVICE_NAME=server-rabbit
RABBITMQ_CONTAINER_NAME=servermq-container
RABBITMQ_HOSTNAME=servermq-container
RABBITMQ_DEFAULT_USER=rabbitmq
RABBITMQ_DEFAULT_PASSWORD=rabbitmq
# use docker port for service, ext port for external mq
RABBITMQ_PORT_CON_DOCKER="5672"
RABBITMQ_PORT_CON_EXT="5673"
RABBITMQ_PORT_WEB_DOCKER="15672"
RABBITMQ_PORT_WEB_EXT="8181"
RABBIT_IN_DOCKER=yes #comment line if rabbitmq is run externally
ALLNETCRAWLER_NAME=servercrawler-container
NETWORK_NAME=server-mttq-net
```

```yaml
version: '3.8'
services:
  server-rabbit:
    image: "rabbitmq:3.7-management"
    container_name: ${RABBITMQ_CONTAINER_NAME}
    hostname: "${RABBITMQ_HOSTNAME}"
    restart: always
    environment:
      - RABBITMQ_DEFAULT_USER=${RABBITMQ_DEFAULT_USER}
      - RABBITMQ_DEFAULT_PASS=${RABBITMQ_DEFAULT_USER}
    ports:
      - "${RABBITMQ_PORT_CON_EXT}:${RABBITMQ_PORT_CON_DOCKER}" # Message Queue Main Port
      - "${RABBITMQ_PORT_WEB_EXT}:${RABBITMQ_PORT_WEB_DOCKER}"
    volumes:
            - server-rabbit-volume:/docker/projects/volumes

  servercrawler:
    build:
      context: .
    container_name: ${ALLNETCRAWLER_NAME}
    depends_on:
      - ${RABBITMQ_SERVICE_NAME}
    restart: always
    environment:
      - RABBIT_HOST=${RABBITMQ_HOSTNAME}
      - RABBITMQ_PORT_CON_EXT=${RABBITMQ_PORT_CON_EXT}
      - RABBITMQ_PORT_CON_DOCKER=${RABBITMQ_PORT_CON_DOCKER}
      - RABBIT_IN_DOCKER=${RABBIT_IN_DOCKER} #remove if rabbitmq runs externally
      - RABBITMQ_DEFAULT_USER=${RABBITMQ_DEFAULT_USER}
      - RABBITMQ_DEFAULT_PASSWORD=${RABBITMQ_DEFAULT_USER}

```

```yaml
volumes:
  server-rabbit-volume:
    driver: local

networks:
  default:
    name: ${NETWORK_NAME}

```

```
FROM python:3.7-stretch

COPY requirements.txt /tmp/

RUN pip install --no-cache-dir -r /tmp/requirements.txt

RUN useradd --create-home appuser
WORKDIR /home/appuser
USER appuser

COPY device_name_mapping.py .
COPY AllnetPoll.py .
COPY pooling.py .
COPY pika_producer.py .
COPY local_config.py .
COPY .k.ey .
COPY .credentials.auth .
#COPY wait-for-it.sh .

#CMD ["./wait-for-it.sh", "dbrabbit-container:5672", "--", "python3", "-u", "./pika_producer.py"]
#CMD ["./wait-for-it.sh", "--strict", "dbrabbit-container:5672", "--", "echo", "SATAN"]
CMD ["python3", "-u", "./pika_producer.py"]

```

```python
import os.path

from cryptography import fernet
import json

SETUP_NR = 0
USE_CREDENTIALS = os.path.exists('.credentials.auth')

if USE_CREDENTIALS:
    with open('.credentials.auth', 'rb') as cred_file:
        with open('.k.ey', 'rb') as key_file:
            key = key_file.read()
        f = fernet.Fernet(key)
        credentials_enc = cred_file.read()
        CREDENTIALS = json.loads(f.decrypt(credentials_enc))
else:
    CREDENTIALS = None

devicelist = list(range(1,2))
#devicelist.remove(4)
#devicelist.remove(14)
#devicelist.remove(30)
DEVICE_LIST = [
    #[1,2,3,5,6,7,8,9,10],
    #[1,2,3,4,5,6,7,8,9,10],
    devicelist,
    #[1,2,3,4,5,6,7,8,9,10],
]

def configure_authentication() -> dict:
    username = input('Username:')
    password = input('Password: ')

    return {'username': username, 'password': password}

def read_authentication(f) -> dict:
    with open('.credentials.auth', 'rb') as cred_file:
        encrypted_dump = cred_file.read()
        dump = f.decrypt(encrypted_dump)
        print(dump)
        return json.load(dump)

# use this to create key and insert credentials
if __name__ == '__main__':

    if os.path.exists('.k.ey'):
        with open('.k.ey', 'rb') as key_file:
            key = key_file.read()
    else:
        key = fernet.Fernet.generate_key()
        with open('.k.ey', 'wb') as key_file:
            key_file.write(key)

    f = fernet.Fernet(key)
    if not os.path.exists('.credentials.auth'):
        credentials_dec = configure_authentication()
    else:
        auth_dict = read_authentication(f)

    with open('.credentials.auth', 'wb') as cred_file:
        cred_file.write(f.encrypt(json.dumps(credentials_dec).encode('utf-8')))
        cred_file.close()
```

```python
from os import environ

if __name__ == "__main__":
    if 'RABBIT_HOST' in environ and 'RABBITMQ_PORT_CON_EXT' in environ:
        rabbit_host = str(environ['RABBIT_HOST'])  # e.g 10.10.10.2
        if 'RABBIT_IN_DOCKER' in environ:
            rabbit_port = str(environ['RABBITMQ_PORT_CON_DOCKER'])
        else:
            rabbit_port = str(environ['RABBITMQ_PORT_CON_EXT'])
    else:
        rabbit_host = 'localhost'
        rabbit_port = 5672
    if ('RABBITMQ_DEFAULT_USER' in environ) and ('RABBITMQ_DEFAULT_PASSWORD' in environ):
        rabbit_user = str(environ['RABBITMQ_DEFAULT_USER'])
        rabbit_password = str(environ['RABBITMQ_DEFAULT_PASSWORD'])
    else:
        print('missing pika credentials in environment')
        exit(10)
    import json
    import zlib
    import time
    from queue import SimpleQueue

    import pika

    from AllnetPoll import AllnetPoll
    from local_config import SETUP_NR, DEVICE_LIST, CREDENTIALS
    from pooling import blocking_delay_generator, round_robin_pooling


    print("Connecting to Allnet-Plugs ... ", )
    devices = ["strommessung_%d" % i for i in DEVICE_LIST[SETUP_NR]]
    assert len(devices)
    q_list = [SimpleQueue() for device in devices]
    thread_list = [AllnetPoll(dev, q, auth=CREDENTIALS) for dev, q in zip(devices, q_list)]
    [thread.start() for thread in thread_list]
    print("Sucessfully started all Allnet-Crawlers")

    # noinspection PyUnboundLocalVariable
    credentials = pika.PlainCredentials(rabbit_user, rabbit_password)
    for i in range(10):
        try:
            connection = pika.BlockingConnection(pika.ConnectionParameters(host=rabbit_host, port=rabbit_port, credentials=credentials))
            break
        except pika.exceptions.AMQPConnectionError:
            print("Couldn't connect to RabbiMQ # ", i)
            time.sleep(2)
    try:
        channel = connection.channel()
        print("Established Connection to RabbitMQ Server")
    except NameError:
        print("unable to connect to RabbitMQ, check parameters:")
        for element,value in environ.items():
            print(f'{element}: {value}')
        print(f"used docker port: {rabbit_port}")
        print("exiting with code 20")
        exit(20)


    channel.confirm_delivery()

    channel.queue_declare(queue='task_queue', durable=True)

    prop = pika.BasicProperties(content_type='application/json',
                                content_encoding='zlib',
                                delivery_mode=2,) # Non-persistent (1) or persistent (2).

    print("Unixtime                 Power Measurements")
    for timestamp in blocking_delay_generator(10):
        messages = round_robin_pooling(q_list)
        messages = list(messages)
        if len(messages) > 0:
            body = json.dumps(messages, ensure_ascii=False)
            body = body.encode('utf-8')
            body = zlib.compress(body)
            channel.basic_publish(exchange='', routing_key='task_queue',
                                  body=body, properties=prop, mandatory=True)

        print(timestamp, '\t', len(messages))

    assert False

```

```python
import time
from collections import deque
from math import ceil

# If we perform a RabbitMQ Transaction for *every* sample, we overload the Disk I/O.
# samples_per_second = 7 Hz * nr_of_plugs
# We merge the individual samples into a list, and generate the list in timely regularly spaced repetitions


# We compress the json, therefore limiting the size of the messages is not really needed anymore
# Periodically run this function
def round_robin_pooling_size_limited(q_list):
    pool_deq = deque()  # deque instead of list for O(1) append

    #  if all allnet-queues are sufficently filled,
    #  we still want to limit the rabbitMQ-message size to 100kB
    #  100kB is a good message size for RabbitMQ. 1 Message = 324 Bytes
    #  https://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/
    n_iter = ceil(100*1024/324 / len(q_list))

    for i in range(n_iter):
        if all((q.empty() for q in q_list)):
            print("All Allnet Queues are empty")
            break
        else:
            samples = [q.get() for q in q_list if not q.empty()]
            [pool_deq.append(sample) for sample in samples]
        if i == n_iter-1:
            print("Warning Allnet-Queues are not yet emptied and round_robin_pooling_stopped")

    return pool_deq


def round_robin_pooling(q_list):
    batch = []
    while any((not q.empty() for q in q_list)):
        samples = [q.get() for q in q_list if not q.empty()]
        batch.extend(samples)
    return batch


def blocking_delay_generator(T=1.0):
    """This generator tries to proceed at a regular time intervall T.
    If the generator's consumer is slower than T, the generator immediately proceeds.
    The generator yields the timepoints in unixtime at which the generator proceeds.
    The blocking iteration ensures that round_robin_pooling() is executed sequentially
    instead of async-concurrently"""
    next_call = time.time()
    while True:
        yield time.time()
        next_call = next_call + T
        sleep_length = next_call - time.time()
        if sleep_length > 0:
            time.sleep(sleep_length)


# unused
#def periodically_run(f=lambda : print(time.time()), T=1.0, *args, **kwargs):
#    next_call = time.time()
#    while True:
#        f(*args, **kwargs)
#        next_call = next_call + T
#        time.sleep(next_call - time.time())
#
#timerThread = threading.Thread(target=
 periodically_run,
 #                                kwargs={'f': round_robin_pooling, 'T':1.0, 'q_list':[]},
 #                                daemon=True)
 #timerThread.start()


 if __name__ == "__main__":
     from AllnetPoll import AllnetPoll
     from local_config import SETUP_NR, DEVICE_LIST
     from queue import SimpleQueue

     devices = ["BLADL_0%d_0%02d" % (SETUP_NR, i) for i in DEVICE_LIST[SETUP_NR]]
     assert len(devices)
     q_list = [SimpleQueue() for device in devices]
     thread_list = [AllnetPoll(dev, q) for dev, q in zip(devices, q_list)]
     [thread.start() for thread in thread_list]

     for timestamp in blocking_delay_generator(10):
         x = round_robin_pooling(q_list)
         print(timestamp, '\t', len(x))

```