Faculty of Computers, Informatics and Microelectronics
Technical University of Moldova

Multimedia System
Laboratory work #2-3

HTTP Client with concurrency superpowers

Authors:

Sezer AKSOY

Supervisor:

Alexandr Gavrisco

2018
Chişinau

Laboratory work #2-3

1. **General purpose**

Study OSI model, HTTP and implement a client app which can do multiple HTTP requests
concurrently.

2. **Prerequisites**

- Read about OSI model (definition, basic info)
- Read about (de)serialization
- Concurrency (definition, primitives etc)

3. **Task:**

Metrics Aggregator

Your new client is a super secret organization which needs to collect a bunch of different metrics from an object and aggregate them. An object is a real location that has some devices which collects data from sensors. Each device has an id, type or sensor_type and a value which describes device's state. So, metrics is just a fancy word for data which describes state of your object from a perspective (e.g. temperature, air pressure etc).

Device/Sensor Types:

- Temperature sensor - 0
- Humidity sensor - 1
- Motion sensor - 2
- Alien Presence detector - 3
- Dark Matter detector - 4

However, there are some problems about those devices:

- Each device has its own URL and can be accessed only using a secret key
- Devices can return values in different formats (JSON, XML, CSV)
- Depending on format, a device can return multiple values (e.g. a box which has multiple sensors)
- Sometimes a device can respond you in up to 29 sec.
- And the most important constraint - your secret key is valid only 30 sec. (because the organization is super secret and don't want to leak access keys).

Here's what functionality your app must offer:

1. Request your secret key at https://desolate-ravine-43301.herokuapp.com/, in response you'll receive a list of URLs (for each device)
2. Using your secret key, request data from all devices concurrently
3. If you get an error related to your access key, go back to **step 1** and retry
4. Parse data from all devices
5. Aggregate all responses ordering by sensor type (example of output below)

```
Temperature:
Device 123 - 100°C

Humidity:
Device 312 - 0.5

Alien Presence:
Device 132 - 0 (No aliens detected)

Dark Matter
Device 132 - 50 (is CERN' particle accelerator turned off?)
```

4. Task Realization:

1) Requesting the key from the given url is done by establishing a http connection first. We are using the "http.client (httplib) " python native library. It is used like:

```
import httplib as httpclient (library from python)
  conn = httpclient.HTTPConnection(URL_RAW)
```

The request is actually done easily by using the "requests" library. In order to do the requests, we need some information in the header. The whole process is simple:

```
main_response = requests.post(URL)
urls_body = main_response.json()
urls_header = main_response.headers
secret_key = urls_header['Session']
```

This way we make a POST request to the url (accessing the url with a browser and checking the source code of the page, a secret message was found "POST MEMBERS ONLY").

Then the response is stored in a variable, following up to the processing of data and parting it to other appropriately named variables for later use.

So the main response has a body, which is the required info - the list of urls we need to access. So to store it and use later, we put it into a json using the json() method from the request. Next, the header is stored using the .headers attribute.

2) Now, to use the secret key, it is enough to simply put it in the header of the request. So for the next pack of parallel http requests, the following is set inside the "requests.get" function's second parameter, "headers=...".

Note, that the request is "GET" - because the body with urls mentioned the "get" method as well.
Here's the function used in parallel for every url:

```
def fetch_url(url):
     print"HTTP request sent!"

try:
    result = requests.get(url, headers=secret_key_header)

     process_result(result)
```

It manages to actually receive response for all the requests, because some respond in up to 29s while the session length is 30s. In practice, with the help of the "threading" library it was taken care of. And the code taking care of it looks like:

```
     def parallel_t_requests(urls):
    threads = [t.Thread(target=fetch_url, args=(url,)) for url in
urls]
    for thread in threads:
        thread.start()
    for thread in threads:
        thread.join()
```

Where threads is a temporary list of threads that have the target function set to "fetch url"

(the method that sends a http request and stores the response), with the arguments being a diferent url each, from the urls list taken from the post request in the previous task portion.

The start() method starts a new thread running the given function, and after all the threads are started, they are joined, which allows one thread to wait for the completion of another, one by one.

3) In order to avoid incomplete data or handle unexpected server errors, the following exception was implemented inside the url fetching method inside which, the requests are sent:

```
except urllib.error.HTTPError as e:
        print("\n!!! SERVER KEY TIMED OUT!!!")
        print("Or perhaps:",e)
        retrying = True
```

and the Main(); it looks like;

```
retrying = True # only once
while(retrying):


    urls_body,secret_key_header,urls_count = main_request()
    urls = refactor_urls(urls_count,urls_body,URL)
    retrying = False #job considered complete...
    parallel_t_requests(urls)
    if(retrying): #...unless an error requests a retry
        print"\n!!! RETRYING !!!"
```

This way, if something happens(i.e. a http related error is caught during an url fetch, thus a retry is requested), it's going to retry, otherwise, it will provide the results.

4) Parsing data is a tricky task. The data comes in multiple formats:

- CSV
- JSON
- XML

Which means, the approach of parsing is di_erent depending on the response.
So we have the function:

```
def process_result(result):

    value_format = result.headers['Content-Type']
    device_body = deserialize(result,value_format)
    save_data(device_body, value_format)

    print"\nHTTP Response status:"
    print result.status_code
```

Value format is provided in the headers under the “content-type” argument.

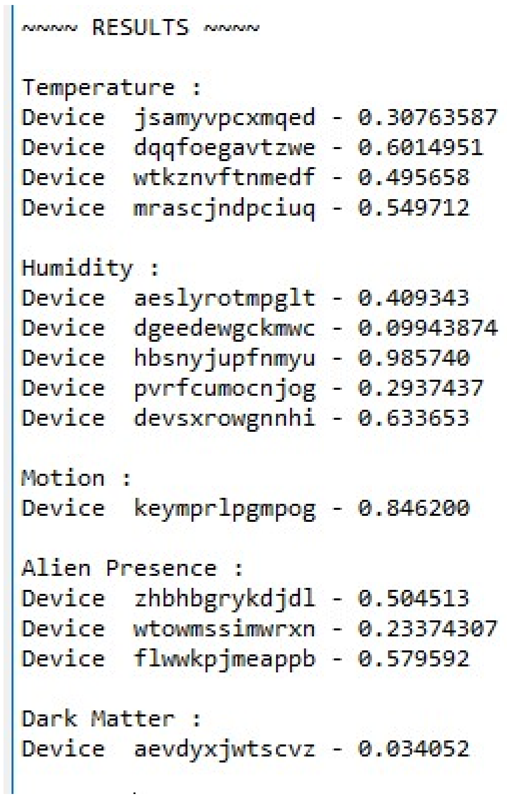
```
def deserialize(response, type):
    if (type == "Application/json"):
        return response.json()

    elif (type == "Application/xml"):
        return xmltodict.parse(response.text)["device"]

    elif (type == "text/csv"):
        d = {'device_id':[],
             'sensor_type':[],
             'value':[]}
        reader = csv.DictReader(response.text.splitlines())
        for row in reader:
            for key in row:
                d[key].append(row[key])
        return d
```

Other types not supported yet.

```
return "not relevant"
```

5) Ordering by sensor type was accomplished in a way that's rather simple but not so intuitive in code. So to put it in a sentence:

We iterate through each type of device we have, and check if the collected data is of that type, if yes, it is displayed. And so, we get this as a final resut:



```
~~~~ RESULTS ~~~~

Temperature :
Device  jsamyvpcxmqed - 0.30763587
Device  dqqfoegavtzwe - 0.6014951
Device  wtkznvftnmedf - 0.495658
Device  mrascjndpciuq - 0.549712

Humidity :
Device  aeslyrotmpglt - 0.409343
Device  dgeedewgckmwc - 0.09943874
Device  hbsnyjupfnmyu - 0.985740
Device  pvrfcumocnjog - 0.2937437
Device  devsxrowgnnhi - 0.633653

Motion :
Device  keymprlpgmpog - 0.846200

Alien Presence :
Device  zhbhbgrykdjdl - 0.504513
Device  wtowmssimwrxn - 0.23374307
Device  flwwkpjmeappb - 0.579592

Dark Matter :
Device  aevdyxjwtscvz - 0.034052
```

Conclusion

This laboratory work is useful because we learnt about the basics of network programming.

Specially, http requests - are great for fetching data.

Also, there's the "threading" library - an incredibly useful tool that in the current lab, allows multiple http requests without returning an error "request already sent", shortening the total responses time considerably because they are done in parallel rather than in a manner of a queue which involves unnecessary waiting.

Bibliography

https://medium.com/@bfortuner/python-multithreading-vs-multiprocessing-73072ce5600b
https://docs.python.org/2/library/threading.html
https://stackoverflow.com/questions/16181121/a-very-simple-multithreading-parallel-url-fetching-without-queue