INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

4A STI

Réseaux non filaires

ESP32/IoT

ESP32, µ-Python, Bottle, Chart.j, Asynchrone, REST, SSE

Réalisation d'une interface pour l'affichage de graphe dynamique basé sur chart.js avec un lien permanent SSE avec le serveur et une interface REST pour le composant IoT

Le but est de :

- lancer un serveur en Python capable de supporter des liaisons permanentes avec le navigateur basé sur SSE, « *Server Sent Event* » en asynchrone ;
- ce serveur mettra
  - à disposition une interface REST pour la récupération de valeurs depuis un IoT ;
  - transferra un SSE vers le naviateur client contenant les dernières valeurs relevées par l'Iot ;
- le navigateur client :
  - se connecte au serveur et établit une connexion persistante et asynchrone avec le serveur ;
  - affiche à l'aide de chart.js le graphe des mesures fournies par le serveur ;

Le fonctionnement du serveur est asynchrone : à chaque requête d'un client IoT, il met à jour les clients Web connectés.

Configuration ESP32 et Raspberry Pi

Vous devez installer :

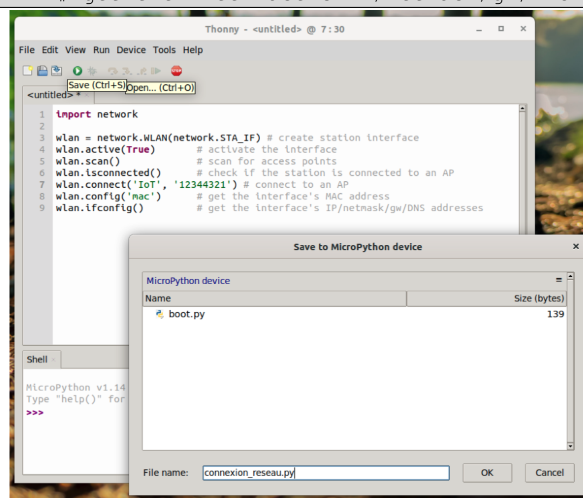
- le Raspberry Pi pour offrir un point d'accès WiFi suivant la fiche associée.
- µPython sur l'ESP32 suivant la fiche associée.

Configuration de l'ESP32 et teste de la connexion WiFi

Pour tester la connexion WiFi depuis l'ESP32 :

```
import network

wlan = network.WLAN(network.STA_IF) # create station interface
wlan.active(True)       # activate the interface
wlan.scan()             # scan for access points
wlan.isconnected()      # check if the station is connected to an AP
wlan.connect('essid', 'password') # connect to an AP
wlan.config('mac')      # get the interface's MAC address
wlan.ifconfig()         # get the interface's IP/netmask/gw/DNS addresses
```



Ce qui produit sur le Raspberry Pi si vous l'avez configuré :

```
xterm
wlan0: STA 24:0a:c4:83:30:60 IEEE 802.11: associated
wlan0: AP-STA-CONNECTED 24:0a:c4:83:30:60
wlan0: STA 24:0a:c4:83:30:60 RADIUS: starting accounting session B5CAD8028304EFCB
wlan0: STA 24:0a:c4:83:30:60 WPA: pairwise key handshake completed (RSN)
dnsmasq-dhcp: DHCPDISCOVER(wlan0) 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPOFFER(wlan0) 10.33.33.140 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPDISCOVER(wlan0) 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPOFFER(wlan0) 10.33.33.140 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPDISCOVER(wlan0) 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPOFFER(wlan0) 10.33.33.140 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPREQUEST(wlan0) 10.33.33.140 24:0a:c4:83:30:60
dnsmasq-dhcp: DHCPACK(wlan0) 10.33.33.140 24:0a:c4:83:30:60 espressif
```

Installation du serveur HTTP sur le Raspberry Pi et du client HTTP sur l'ESP32

Vous récupérerez le code du serveur sur le Raspberry Pi et pour l'ESP32 :

```
xterm
$ git clone https://git.unilim.fr/pierre-francois.bonnefoi/IOT_bottle.git
```

Vous déploierez le code du fichier `requete_iot_lumiere_esp32.py` sur l'ESP32 :

```
import network
import urequests
import time
from machine import Pin, Timer
from esp32 import hall_sensor

wlan = network.WLAN(network.STA_IF) # create station interface
wlan.active(True)       # activate the interface
# wlan.scan()           # scan for access points
wlan.isconnected()      # check if the station is connected to an AP
time.sleep_ms(500)
if not wlan.isconnected():
    print('connecting to network...')
    wlan.connect('IoT', '12344321') # connect to an AP
    time.sleep_ms(500)
    while not wlan.isconnected():
        pass
    print('network config:', wlan.ifconfig())

url="http://serveur.iot.com:8080/luminosity/%s"

def envoyer_valeur(t):
    v = 4*hall_sensor()
    print(v)
    rep = urequests.get(url%str(v))
    rep.close()

t = Timer(-1)
t.init(period=3000,mode=Timer.PERIODIC, callback=envoyer_valeur)
```

Sur le Raspberry Pi, vous exécuterez l'application Web :

```
xterm
$ python3 serveur_chart.py
Bottle v0.12.19 server starting up (using GeventServer())...
Listening on http://:8080/
Hit Ctrl-C to quit.
```

Annexe : détails de l'application et explications

Le code du serveur Python tournant sur le serveur embarquant le point d'accès WiFi :

```
from json import dumps
from datetime import datetime
from gevent import monkey; monkey.patch_all()
from bottle import route, run, static_file, redirect, response, Response
from gevent import queue
import gevent

valeurs_date = []
valeurs = []
valeurs_json ="{}"

souscripteur = []

@route('/')
def page_accueil():
    redirect("/contenu/index.html")

@route('/contenu/<filepath:path>')
def contenu(filepath):
    return static_file(filepath,root="contenu")

@route('/luminosity/<lum:int>')
def luminosity(lum):
    global valeurs, valeurs_date, valeurs_json
    date = datetime.now()
    valeurs_date.append(date.strftime("%H:%M:%S"))
    valeurs.append(lum)
    valeurs_date = valeurs_date[-10:]
    valeurs = valeurs[-10:]
    valeurs_json = dumps({ 'labels' : valeurs_date, 'data' : valeurs})
    for s in souscripteur:
       s.put('data: '+valeurs_json+'\n\n')

@route('/connexion')
def connexion():
    response.content_type = "text/event-stream"
    body = gevent.queue.Queue()
    souscripteur.append(body)
    return body

run(host='', port=8080, debug = True,server="gevent")
```

Le code du client web envoyé vers le navigateur pour information :

```
<!DOCTYPE html>
<html>
<head>
  <script src=
  "https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.5.0/Chart.min.js">
  </script>
  <script type="text/javascript" src=
  "https://code.jquery.com/jquery-1.8.0.min.js">
  </script>
  <script type="text/javascript">
        $(document).ready(
             function() {
               sse = new EventSource('/connexion');
               sse.onmessage = function(message) {
                   console.log('Arrive message ! ');
                   valeurs_graphe = JSON.parse(message.data);
                var ctx = $('#mon_graphe');
                var mon_graphe = new Chart(ctx, {
                        options: { scales: { yAxes: [{ ticks: { min:0,max:900 }}] } ,
animation: false},
                        type: 'line',
                        label: 'luminosité',
                        data: { labels: valeurs_graphe['labels'], datasets: [ { la
bel: 'luminosité', borderColor: "#9510cd", data: valeurs_graphe['data']} ] }
                        });
        }});
  </script>
  <title></title>
</head>
<body>
  <h2>Demo IoT: mesures de luminosité</h2>
  <canvas id="mon_graphe"></canvas>
</body>
</html>
```

Présentation des différentes technologie du Web « temps réél »

Le Web « temps réél », une évolution du Web traditionnel

Par défaut, un serveur Web gère **une requête http à la fois** dans un contexte de **travail temporaire** : utilisation de **fork** pour se dupliquer et gérer la requête au travers d'un processus enfant.

Dans le cas où un **état** doit être maintenu entre les différentes requêtes, on utilise :
- une **base de données** pour gérer cet état ;
- des **notions de sessions** pour faciliter la continuation d'une communication avec le même client au travers de des différentes requêtes http (sessions gérées par cookies, et/ou des passages de paramètres) ;
- la **mise à jour de données partagées** entre différents utilisateurs et utilisées pour construire des contenus « *à la volée* » au travers de langages gérés au sein du serveur Web (PHP, Java Servlet, .net) ;

Ce fonctionnement est **coûteux** (trop lent et consommateur de ressource), et un nouveau modèle est proposé pour gérer plus efficacement ces états, et leur partages, avec la **programmation asynchrone** (dont vous trouverez sur le site Web, une présentation) et de nouvelles technologies :
- ▷ du côté client, dans le navigateur : Javascript et les échanges asynchrones proposées par AJAX, « Asynchronous JavaScript and XML » ;
- ▷ du côté serveur :
  - ◇ des **notions de sessions** pour faciliter la continuation d'une communication avec le même client au travers de des différentes requêtes http (sessions gérées par cookies, et/ou des passages de paramètres) ;
  - ◇ la **mise à jour de données partagées** entre différents utilisateurs et utilisées pour construire des contenus « *à la volée* » au travers de langages gérés au sein du serveur Web (PHP, Java Servlet, .net) ;

  Ce fonctionnement est **coûteux** (trop lent et consommateur de ressource), et un nouveau modèle est proposé pour gérer plus efficacement ces états, et leur partages, avec la **programmation asynchrone** (dont vous trouverez sur le site Web, une présentation) et de nouvelles technologies :
  - ▷ du côté client, dans le navigateur : Javascript et les échanges asynchrones proposées par AJAX, « Asynchronous JavaScript and XML » ;
  - ▷ du côté serveur :
    - ⋆ des base de données NoSQL : Memcache, Redis, Apache accumulo, Hypertable, Bigdata, MongoDB, CouchDB, *etc*
    - ⋆ une application en relation directe avec les multiples clients simultanément : Node.js, micro frameworks en Python *etc* ;
    - ⋆ des modèles de communication **persistantes** et mono ou bidirectionnels : SSE, WebSockets.
    - ⋆ des protocoles de streaming : WebRTC pour le son et la vidéo.

  Du côté client, si l'utilisation de Javascript pour manipuler le DOM, « *Document Object Model* », pour obtenir des pages HTML interactives est relativement « facile » grâce à des frameworks comme AngularJS, il n'en va pas de même avec la nouvelle possibilité de faire des échanges asynchrones, c-à-d sans rechargement de la page HTML courante, grâce à la fonction `XMLHttpRequest`.

  En effet, l'utilisation de cette fonction présente des difficultés :
  - ▷ elle dépend de la nature du navigateur ;
  - ▷ elle est complexe dans sa mise en œuvre : sa réalisation asynchrone dans le navigateur impose de lier son déroulement à des fonctions Javascript, en rapport à des évenements difficiles à maîtriser : chargement complet de la page, réalisation et succès de la requête *etc.*
  - ▷ elle utilise XML pour son format d'échange.

  Heureusement, l'utilisation de la bibliothèque à tout faire **JQuery** permet :
  - une portabilité à toute épreuve : Jquery prend soin de s'adapter aux différents navigateurs qu'il rencontre ;
  - une facilité d'utilisation : grâce à sa syntaxe simplifiée, ses *sélecteurs* CSS ou DOM ou XPath pour localiser les éléments du DOM, sa facilité de manipulation de données *etc.*
  - une gestion transparente d'AJAX : une syntaxe simplifiée, faisant abstraction du navigateur utilisé ;

  Enfin, l'utilisation du format d'échange JSON, « *JavaScript Object Notation* », permet le codage et l'échange de données simplifiés sans avoir recours à XML et autres XSLT.

Python et la programmation asynchrone

En Python, les opérations liées à la programmation socket sont **bloquantes** et doivent être gérées avec un `select` quand on veut pouvoir en gérer plusieurs séquentiellement.
Pour pouvoir faire de la programmation asynchrone, il faut utiliser un module spécial, `gevent` permettant de disposer de « *greenlet* » : une « **coroutine** », c-à-d une fonction qui fournit un résultat, *yield*, et se susprendre. Elle peut être réactivée, appelée de nouveau, et fournir un nouveau résultat dépendant :

- ⋆ de son état au moment de sa suspension ;
- ⋆ éventuellement de nouveaux paramètres passés lors de sa réactivation ;

À tout moment, **une seule greenlet** est exécutée, ce qui est conforme au principe de la programmation asynchrone, à la différence des *threads* dont le comportement est proche : autonomie, synchronisation lors de la fourniture d'un résultat.

Ainsi, pour tirer partie de possibilités **asynchrone** dans Python, pour les opérations bloquantes comme la programmation socket, il est nécessaire de procéder à du « *monkey patching* », c-à-d de la substitution automatique de l'appel de fonctions bloquantes par des fonctions non bloquantes fournies et gérées par le module :

```
#!/usr/bin/python2
# encoding=utf-8

from gevent import monkey; monkey.patch_all() # on import et de suite on patche !
```

AJAX & La protection des données de l'utilisateur

L'utilisation de l'API « XHR », « *XMLHttpRequest* », dans le navigateur Web s'accompagne d'une gestion automatique de cache, de redirection, de négociation de contenu, d'authentification *etc.*
Pour protéger les données de l'utilisateur, une requête XHR que réalise le navigateur contient en entête une option « Origin » qui indique depuis quel site le code qui la contient a été téléchargé.
La valeur de cette entête **ne peut être modifiée** : le navigateur en garantie le contenu.
Le serveur qui reçoit la requête peut savoir de quelle origine elle provient : adresse serveur, service (http ou https) et numéro de port (80 par défaut). Il applique la règle « *same-origin* » : la requête d'une ressource ne peut être faite que depuis le même site que la ressource demandée : on empêche qu'un script téléchargé depuis un site A ne puisse demander une ressource d'un site B, en lui transmettant des informations (cookies, données privées, éléments d'authentification).

La technique du « CORS », « *Cross-Origin Resource Sharing* » permet de contourner cette limitation en autorisant la récupération de données (sans transmission de la part du navigateur de cookies) vers un serveur différent (en ajoutant une autorisation dans l'entête de réponse du serveur HTTP).
Nous n'utiliserons pas le CORS, et nous devrons tenir compte de cette contrainte en gérant les communications persistantes de chaque client par une « greenlet » dans notre application utilisant le micro-framework bottle. **Les « SSE », « *Server-Sent Events* »**

Les SSE permettent d'envoyer un flux de données, « *stream* » du serveur Web vers le client d'événements, « *events* » : des données au format texte associées à des notifications ou des mise à jour en temps réél du serveur.

Cette technologie :

- □ introduit des composants :
  - ⋆ une interface `EventSource` dans l'API du navigateur, qui permet au client de récupérer des notifications du serveur sous forme d'événements du DOM ;
  - ⋆ le format « `event stream` » qui permet au navigateur de transmettre des mise à jour par morceau.
- □ permet :
  - ◦ l'utilisation d'une connexion TCP longue durée, ce qui permet de limiter la latence due aux connexions successives ;
  - ◦ l'analyse efficace par le navigateur de messages reçus depuis le serveur ;
  - ◦ le suivi automatique du dernier message reçu et l'auto reconnexion ;
  - ◦ la notification du navigateur de la réception d'un message par un événement du DOM.