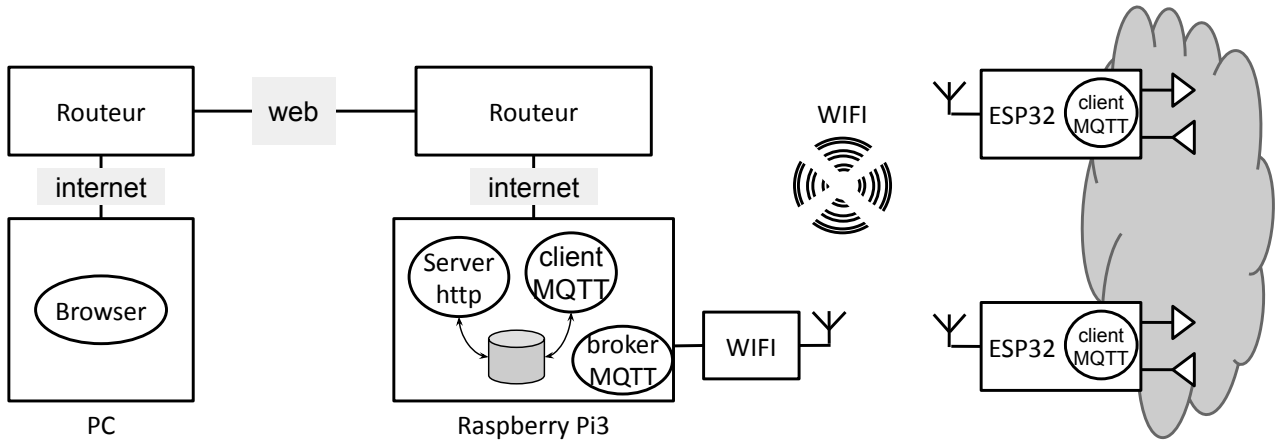
Serveur HTTP et Gateway

Module IOC — MU4IN109

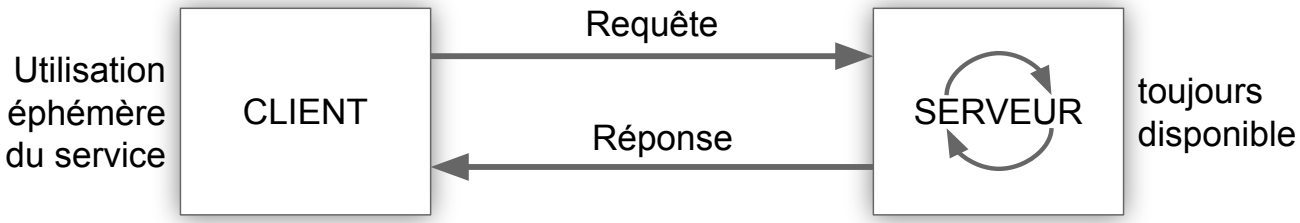
Franck Wajsbürt

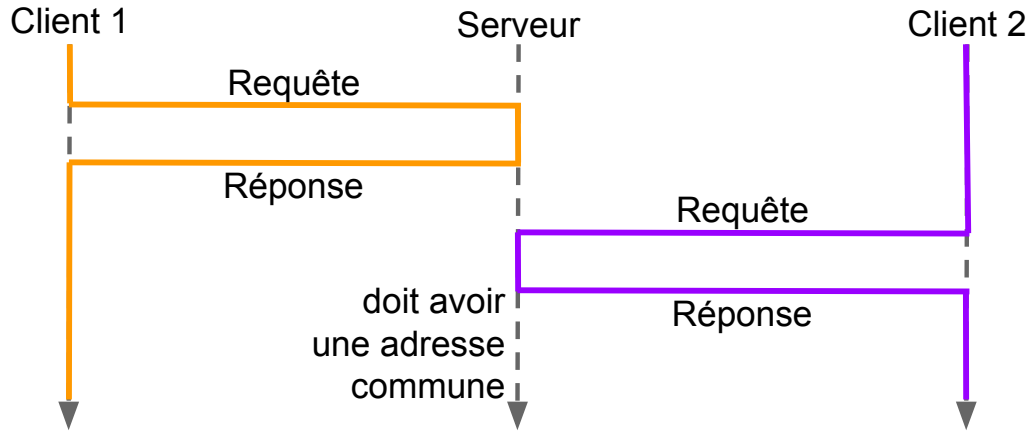
Objectif Final



SERVICES

Un serveur **rend** un service
Un client **demande** un service

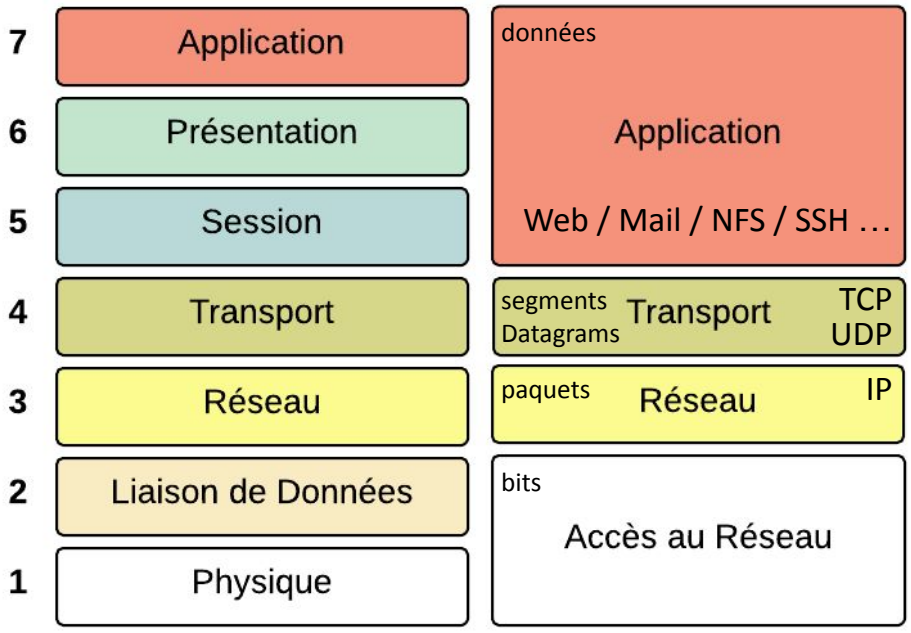




Modèle OSI — Modèle TCP/IP (unix)

OSI
Operating System Interconnection

TCP/IP
TCP Transmission Control Protocol (fiable)
UDP User Datagram Protocol (rapide)



Les couches sont indépendantes et ne communiquent qu'avec ses voisines.

Les paquets créés par une couche sont encapsulés dans des paquets fabriqués par la couche suivante et inversement.

https://juleshuynhvan.business.blog/2017/02/20/modele-osi-modele-tcpip/

Modèle OSI – Modèle TCP/IP (unix)

La couche 7 est la couche application
La couche 6 ou couche présentation :
- Conversion de code caractère, compression.

La couche 5 ou couche session :
- Synchroniser les communications, gestion des « transactions »
- Corriger les erreurs de traitements par restauration d'un états antérieur connu

La couche 4 ou couche transport :
- gérer les connexions applicatives.
- garantir la connexion..

La couche 3 ou couche réseau :
- interconnecter les réseaux entre eux.
- fragmenter les paquets.
- routeur.

La couche 2 ou couche liaison :
- connecter les machines entre elles sur un *réseau local*.
- détecter les erreurs de transmission.
- *switch*

La couche 1 ou couche physique :
- support de transmission pour la communication
- *hub*

La couche 4 : Application
- Services

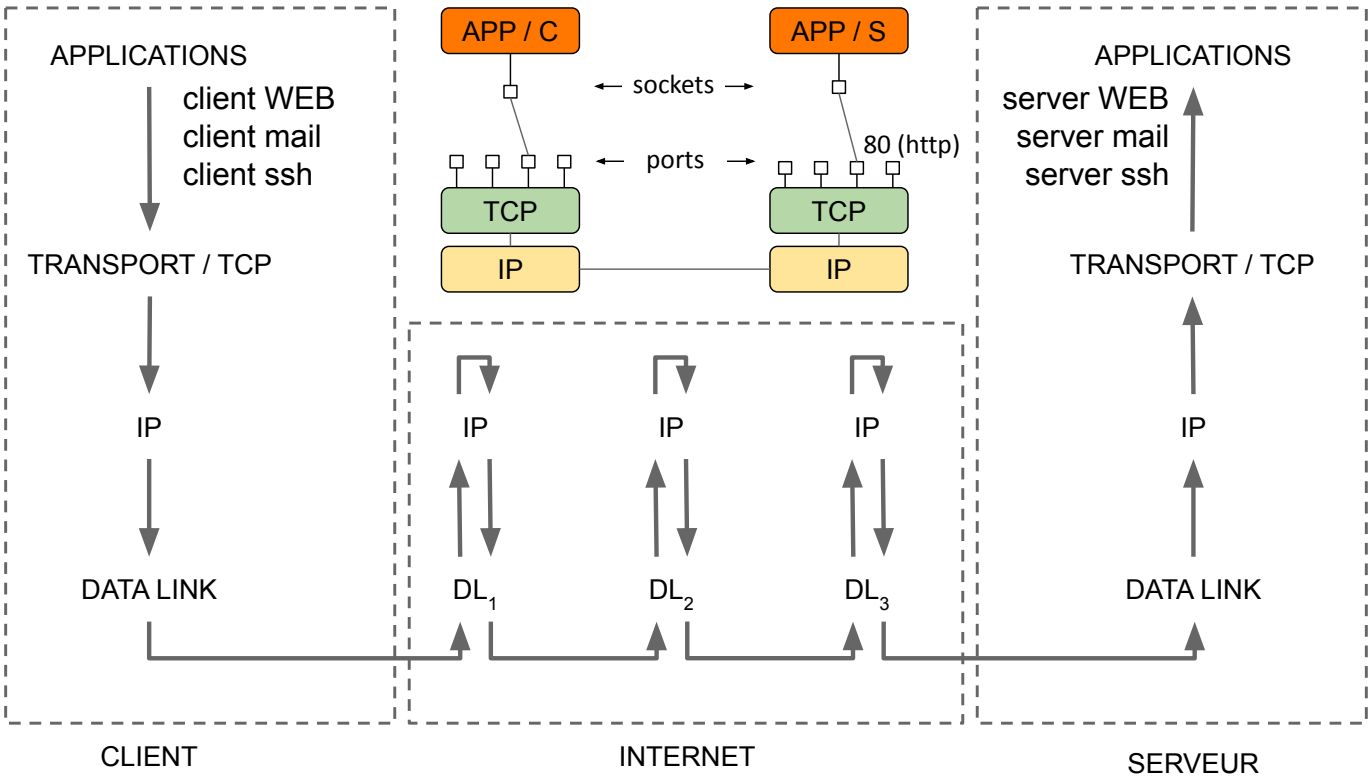
La couche 3 : Transport (→ TCP / UDP)
- Gestion des connexions applicatives et de garantir la connexion

La couche 2 : Réseau (→ IP)
- Interconnection des réseaux entre eux et de fragmentation les paquets

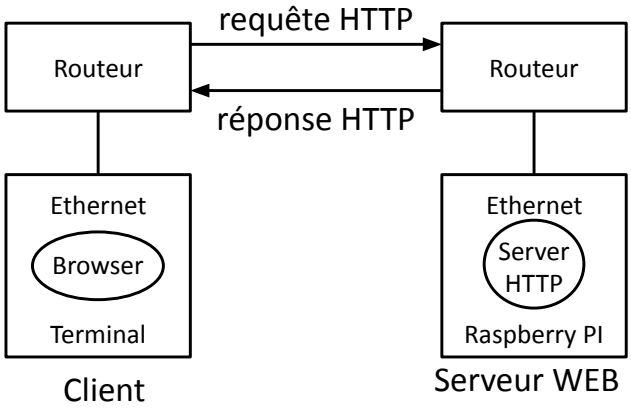
La couche 1 : Accès réseau
- couche 1 et 2 du modèle OSI
- Support de transmission pour la communication
- Connection des machines entre elles sur un réseau local
- Détection des erreurs de transmission

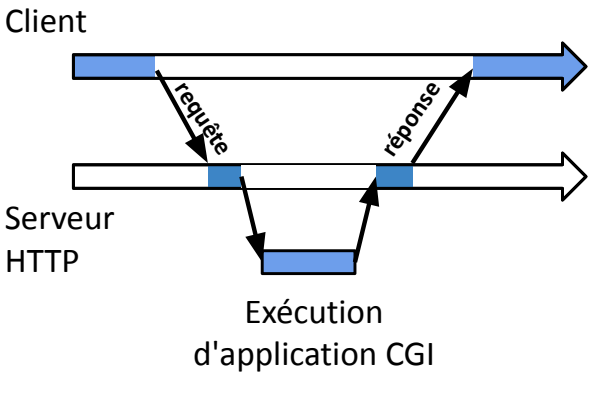
https://juleshuynhvan.business.blog/2017/02/20/modele-osi-modele-tcpip/

Echange Client — Serveur



Serveur WEB

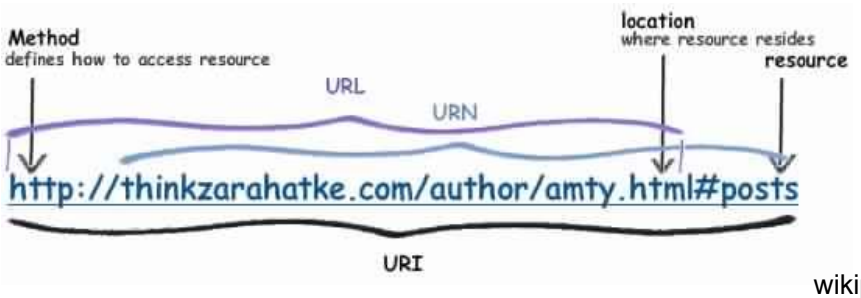




- Un **serveur HTTP** est un **serveur WEB** spécialisé dans l'interprétation des **requêtes HTTP** venant de **clients** (ex: navigateurs web) et répond en envoyant des ressources web (**HTML**, JSON, etc.) par **réponse HTTP**
- Le protocole **HTTP** définit des méthodes comme [GET **URI** HTTP/ver] ou [POST URI HTTP/ver data].
- Une **URI** (Uniform Resource Identifier) est une chaîne de caractères permettant d'identifier une ressource sur un serveur. Une URL est un type spécifique d'URI qui inclut le *schéma* qui désigne le protocole (ex : https://).
- **HTML** (HyperText Markup Language) structure les pages web et permet d'inclure des liens vers d'autres ressources, ainsi que du contenu multimédia comme des images, sons et vidéos.
- Le serveur peut renvoyer de simples pages au format **HTML** statique ou exécuter des applications **CGI** (Common Gateway Interface) pour générer des pages dynamiques. Les applications sont écrites en n'importe quel langage mais sont souvent interprétées (**Python**, PHP, Javascript, etc.)

Quelques précisions

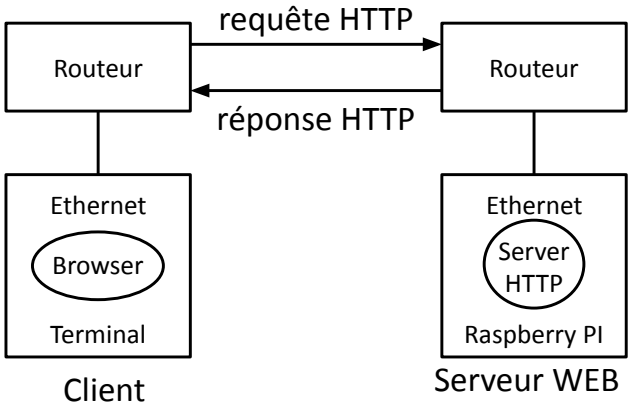
- Les commandes possibles sont : GET, HEAD, POST, OPTIONS, CONNECT, TRACE, PUT, PATCH et DELETE
- Pour désigner la ressource, on utilise l'URI

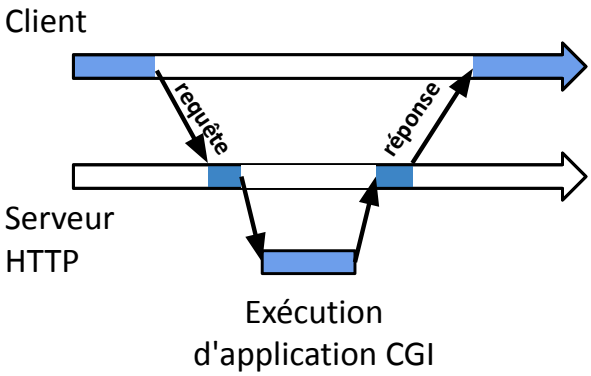


wikipédia

- GET
  Méthode la plus courante pour demander une ressource.
  Une requête GET est sans effet sur la ressource, il doit être possible de répéter la requête sans effet.
  Les arguments sont transmis en clair dans l'URL
- POST
  Méthode utilisée pour transmettre des données en vue d'un traitement à une ressource (le plus souvent depuis un formulaire HTML). L'URI fourni est l'URI d'une ressource à laquelle s'appliqueront les données envoyées. Le résultat peut être la création de nouvelles ressources ou la modification de ressources existantes. Les données sont transmises dans les corps du paquet HTTP et non pas dans l'URL.

Serveur WEB





- Les pages envoyées au client sont interprétées par un navigateur (browser)
  - HTML pour décrire la structure (titre, liste, tables, etc.)
  - CSS pour décrire le style (forme, couleur, position, etc.)
  - Javascript pour décrire des comportements sur le poste du client
- Le serveur HTTP peut être écrit en C (Apache, IIS) ou dans un autre langage comme Python (Zope)
- Les pages sont préparées par le serveur
  - Statiques : juste du HTML
  - Dynamiques : du HTLM (+ javascript)
    - fabriquées par des scripts écrits en PHP, Python, Perl, Java, etc…
    - en consultant des bases de données : MySQL, SQLite3, PostgreSQL, OracleBD, etc..
    - en communiquant avec des applications

Serveur WEB minimaliste

```
server
├── server.py                serveur http
└── www                      répertoire du serveur
    ├── index.html           page d'accueil
    └── cgi                  répertoire des scripts CGI
        ├── command.py       scripts CGI en python
        ├── charts.py
        └── stat.py
```

```
$ chmod u+x server.py
$ ls -F
$ img/ www/ server.py*
$ cd www
$ ../server.py
```

Python

Langage de programmation **objet**, **multi-paradigme** et **multiplateformes** [...] Il est doté d'un **typage dynamique** fort, d'une **gestion** automatique de la **mémoire** par **ramasse-miettes** et d'un système de **gestion d'exceptions**.

http://westmarch.sjsoft.com/2012/11/zen-of-python-poster/

- http://openclassrooms.com/courses/apprenez-a-programmer-en-python
- http://python.developpez.com/tutoriels/cours-python-uni-paris7/
- http://en.wikibooks.org/wiki/A_Beginner%27s_Python_Tutorial

Le serveur est programmé en python pour sa simplicité, nous allons expliquer le code mais vous allez devoir vous former grâce aux tutoriaux si vous voulez aller plus loin…

Serveur HTTP/CGI en Python

server.py

```
#!/usr/bin/env python

import BaseHTTPServer
import CGIHTTPServer
import cgitb;
cgitb.enable()

server = BaseHTTPServer.HTTPServer
handler = CGIHTTPServer.CGIHTTPRequestHandler

server_address = ("", 8000)
handler.cgi_directories = ["/cgi"]

httpd = server(server_address, handler)
httpd.serve_forever()
```

#! : Shebang
Placer au début du fichier, cela permet au shell de savoir quel programme appeler pour ce fichier s'il exécutable.

Importation des bibliothèques python, respectivement :
BaseHTTPServer serveur sur un port défini à la création
CGIHTTPServer gestionnaire pour l'exécution des cgi
cgitb pour avoir les erreurs des scripts cgi

Création du serveur HTTP
Création du Gestionnaire de CGI

Spécification de l'adresse ethernet et du port d'écoute
Spécification du répertoire contenant les scripts CGI

création et
Lancement du serveur

index.html

Documentation : aide-mémoire / doc html / éditeur de test

- http://www.html.su/
- http://www.w3schools.com/html/default.asp
- http://www.w3schools.com/html/tryit.asp?filename=tryhtml_basic_document

index.html

```
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph</p>
</body>
</html>
```


Appel de script CGI

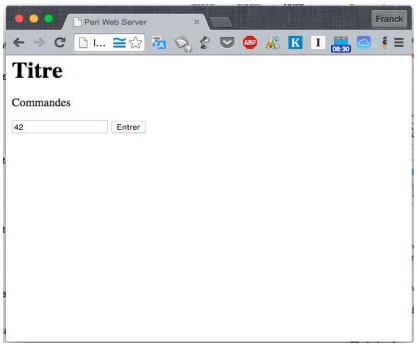
Pour exécuter un script python avec des valeurs recueillies par un formulaire

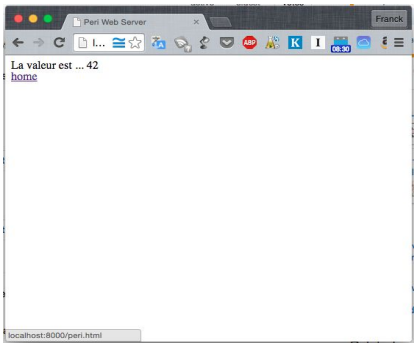
cgi/command.py

```
#!/usr/bin/env python
import cgi

form = cgi.FieldStorage()
val1 = form.getvalue('val1')

print """
<html>
<body>
La valeur entrée est ... %s
<a href="/peri.html">home</a>
</body>
</html>
""" % (val1,)
```

index.html

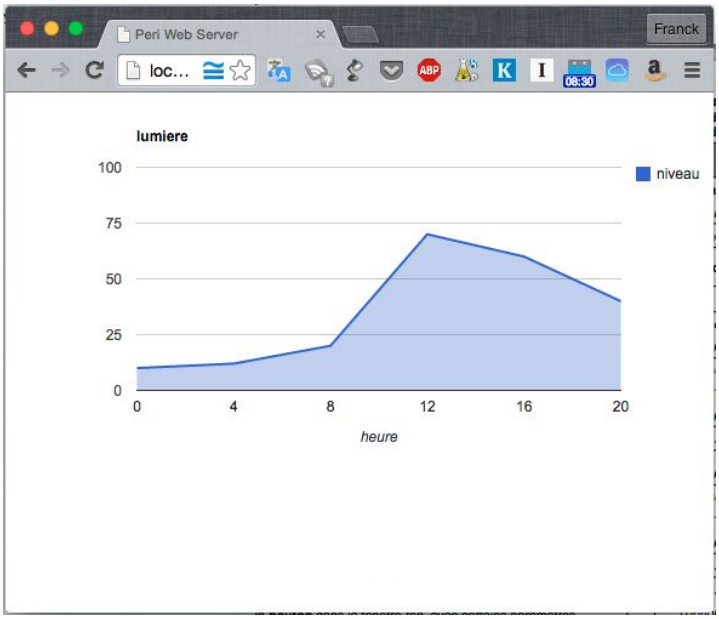
```
<html>
<head><title>IOC Web Server</title></head>
<body>
<h1>Titre</h1>
<p>Commandes</p>
<form method="post(ou get)" action="cgi/command.py">
   <input name="val1" cols="20"></input>
   <input type="submit" value="Entrer">
</form>
</body>
</html>
```

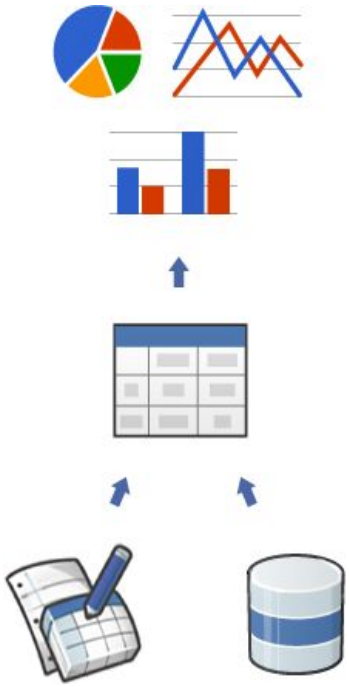




charts.py

Pour l'affichage formaté des résultats, google propose des programmes javascript : https://developers.google.com/chart/interactive/docs/index





charts.html

```
<html>
  <head>
    <script type="text/javascript" src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data = google.visualization.arrayToDataTable([
          ['heure', 'niveau']
         ,['0', 10]
         ,['4', 12]
         ,['8', 20]
         ,['12', 70]
         ,['16', 60]
         ,['20', 40]
        ]);

        var options = {
          title: 'lumiere',
          hAxis: {title: 'heure',  titleTextStyle: {color: '#333'}},
          vAxis: {minValue: 0, maxValue: 100}
        };
        var chart = new google.visualization.AreaChart(document.getElementById('chart_div'));
        chart.draw(data, options);
      }
    </script>
  </head>
  <body>
    <div id="chart_div" style="width: 600; height: 300px;"></div>
  </body>
</html>
```

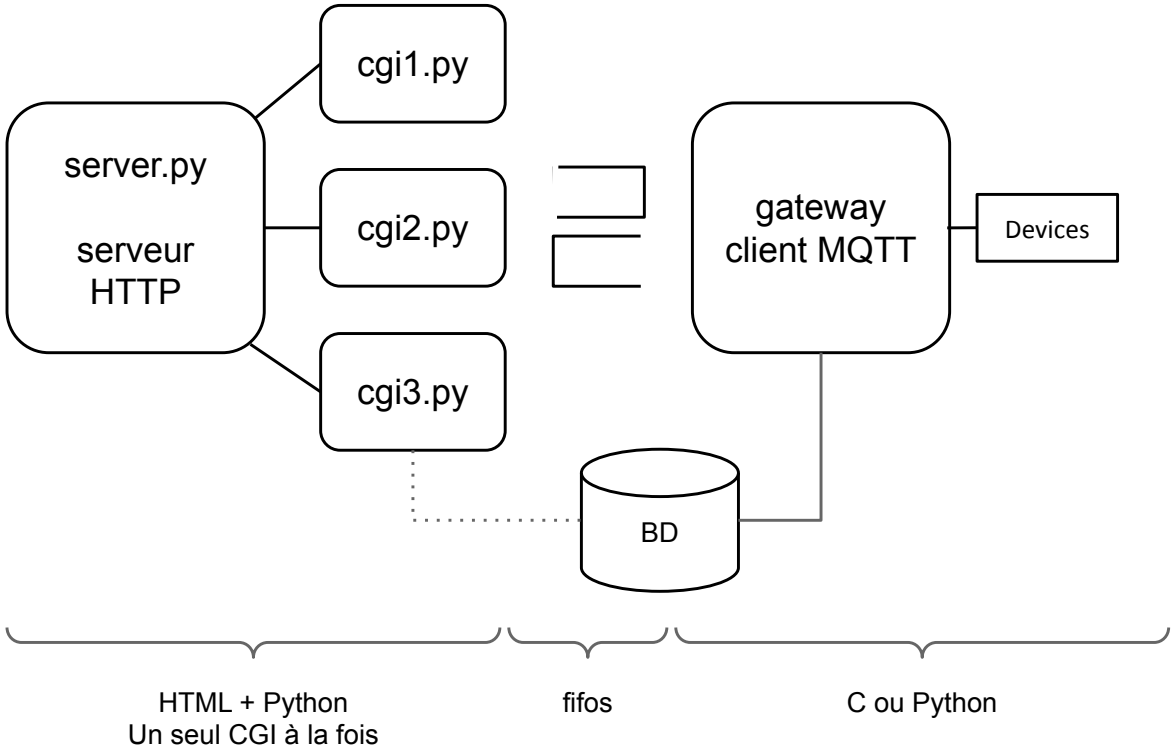
charts.py

```
#!/usr/bin/env python
import cgi
form = cgi.FieldStorage()
val1 = (int)(form.getvalue('val1'))
tab = [10,12,20,70,60,40]
print """
<html>
  <head>
    <script type="text/javascript" src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
      google.load("visualization", "1", {packages:["corechart"]});
      google.setOnLoadCallback(drawChart);
      function drawChart() {
        var data = google.visualization.arrayToDataTable([
          ['heure', 'niveau']"""
i=0
while i <= val1:
  print "         ,['%d', %d]" % (i,tab[i/4])
  i += 4
print """\
       ]);
        var options = {
          title: 'lumiere',
          hAxis: {title: 'heure',  titleTextStyle: {color: '#333'}},
          vAxis: {minValue: 0, maxValue: 100}
        };
        var chart = new google.visualization.AreaChart(document.getElementById('chart_div'));
        chart.draw(data, options);
      }
    </script>
  </head>
  <body>
    <div id="chart_div" style="width: 600; height: 300px;"></div>
  </body>
</html>
"""
```

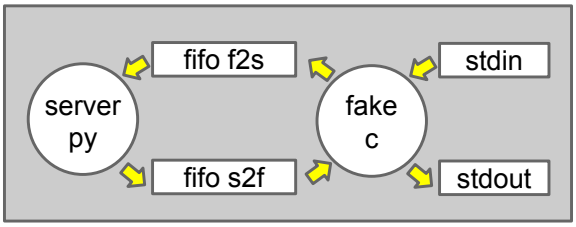
Il y a 3 programmes dans 3 langages dans ce fichier !
python, html, javascript

Communication inter-processus par FIFO

Architecture possible pour l'accès aux capteurs



communication par FIFO



fake lit une valeur sur stdin, ajoute un numéro id et écrit le message sur f2s, le « server » lit le message, calcule la parité et renvoie l'id et la parité sur s2f, fake lit le message et affiche l'id, la valeur et la parité.

- Communication par fifo en C et en python
- Attente de plusieurs flux en C

Liste des fonctions à utiliser

La liste n'est pas exhaustive,
il y a seulement les fonctions spécifiques

en C		en python
`open()`	⟷	`open()`
`read()`	⟷	`readline()`
`write()`	⟷	`write()`
`close()`	⟷	`close()`
`mkfifo()`	⟷	`mkfifo()`
`select()`		

Communication par FIFO anonyme (pipe)

Objectif communication dans un sens fifo entre deux processus.

- Une fifo à deux extrémités
- Chaque extrémité est référencée par un descripteur de fichiers (entier)

Les pipes peuvent être anonymes

- création : `int pipe(int fd[2]);`
  - `pipe()` renvoie des numéro de descripteurs de fichiers dans `fd[]`
  - On écrit dans `fd[1]`, on lit depuis `fd[0] : fd[1]` → `pipe` → `fd[0]`
  - la taille d'un pipe est limitée à 64ko (depuis la 2.6.11)
- l'écriture : `ssize_t write(int fd, const void *buf, size_t count);`
  - écrit `count` octets de `buf` dans `fd`
  - rend le nombre d'octet écrits
- la lecture : `ssize_t read(int fd, void *buf, size_t count);`
  - lit `count` octets dans `fd` et les écrit dans `buf`
  - rend -1 si echec (+ `errno`) sinon le nombre d'octets lus, 0 si `fd` vide
- la fermeture : `int close(int fd);`
  - il faut fermer les deux extrémités

exemple un seul processus

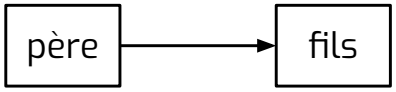
```
#include <stdio.h>
#include <memory.h>
#include <unistd.h>

int main( int argc, char ** argv )
{
  char buffer[BUFSIZ+1];

  /* create the pipe */
  int fd[2]; // in fd[0] outfd[1]
  pipe(fd);

  /* write into the pipe */
  write(fd[1], "Hello World\n", strlen("Hello World\n"));

  /* read the pipe and print the read value */
  read(fd[0], buffer, BUFSIZ);
  printf("%s", buffer);
}
```


http://www.zeitoun.net/articles/communication-par-tuyau/start

Communication entre un père et son fils

- Le pipe est créé dans un processus
- Le processus est dupliqué par **`fork()`**
  - **`fork()`** rend **`0`** pour le fils et le **`pid`** créé chez le père
  - les deux nouveaux processus ont tous les deux accès au pipe
  - chacun va fermer l'une des extrémités avec **`close()`**
- Si on veut une communication dans les deux sens, il faut deux pipes

Communication entre un père et son fils




```
int main( int argc, char ** argv )
{
    int  pfd[2];                                  // file descriptors of
pipe
    pipe(pfd)                                     // create anomymous pipe
    if (fork() == 0)                              // fork returns 0 to son
    {
        char buffer[BUFSIZ];
        close(pfd[1]);                            // close write side
CHILD   while (read(pfd[0], buffer, BUFSIZ) != 0) // read and ...
            printf("child reads %s", buffer);     // ... print result on
screen
        close(pfd[0]);                            // close the pipe
    }
    else {
        char buffer[BUFSIZ];
        close(pfd[0]);                            // close read side
PARENT  strcpy(buffer, "HelloWorld\n");
        write(pfd[1], buffer, strlen(buffer)+1);
        close(pfd[1]);                            // close the pipe
    }
    return 0;
}
```


Communication par FIFO nommée

Si on veut faire communiquer deux processus qui n'ont pas de liens de parenté direct, on utilise des fifos nommés qui seront placés sur le disque, nommé fifo.

- La création d'une fifo :

  `int mkfifo(const char *pathname, mode_t mode);`

  la fifo est un fichier de type fifo créer avec le mode mode
  (p. ex: 0x666 pour un droit rw pour l'utilisateur, le groupe et les autres)

  rend 0 en cas de succès

- La destruction d'une fifo

  `int unlink(const char * pathname);`

  efface le fichier mais reste accessible par les process qui l'ont encore ouvert.

Ouverture d'une fifo

Une fois créée une fifo doit être ouverte pour avec `open`.

```
int open(const char *pathname, int flags);
```

`flags` :

- `O_RDONLY`
- `O_WRONLY`
- `O_RDWR`

Les deux extrémités doivent être ouvertes pour utiliser la fifo.

- L'ouverture en lecture en bloquante tant qu'il n'y a pas d'ouverture en écriture afin de synchroniser les deux processus.
- On peut ajouter le mode `O_NONBLOCK` pour ne pas se bloquer à la condition que le lecteur ET l'écrivain fassent de même, sinon c'est une erreur.

lecture d'une fifo

```
nb_lu = read(fd, buffer, TAILLE_READ);
```

Si le tube n'est pas vide et contient `taille` caractères :

Lecture de `nb_lu = min (taille, TAILLE READ)` caractères.

Si le tube est vide

Si le nombre d'écrivains est nul

Alors c'est la fin de fichier et `nb_lu` est nul.

Si le nombre d'écrivains est non nul

Si lecture bloquante alors blocage en attente de caractères

Si lecture non bloquante (open avec flag `O_NONBLOCK`)

`nb lu= -1 et errno=EAGAIN.`

écriture d'une fifo

```
nb_ecrit = write(fd, buf, n);
```

L'écriture est atomique si le nombre de caractères à écrire est inférieur à `PIPE_BUF` (linux 4096), la taille du tube sur le système. (cf `<limits.h>`).

Si le nombre de lecteurs est nul

Envoi du signal `SIGPIPE` à l'écrivain et `errno = EPIPE.`

Sinon

Si l'écriture est bloquante, il n'y a retour que quand

Les n caractères ont été écrits dans le tube.

Si écriture non bloquante

Si `n > PIPE_BUF`, retour avec un nombre inférieur à n

Si `n <= PIPE BUF`

Et si n emplacements libres, écriture nb_ecrit = n

Sinon retour -1 ou 0.

écoute de plusieurs canaux

SI la fonction `read()` est bloquante, il ne faut la lire que si l'on est sûr qu'il y a des données, une fonction permet de se mettre en attente sur plusieurs flux.

```
int  select(   int n, fd_set *readfds, fd_set *writefds,
               fd_set *exceptfds, struct timeval *timeout);
```

- `n` numéro du plus grand descripteur + 1
- `readfds` masque des descripteurs de fichiers attendus en lecture
- `writefds` masque des descripteurs de fichiers attendus en écriture
- `exceptfds` masque des descripteurs de fichiers attendus pour des conditions spéciales…. (je n'ai pas compris ce que c'est)

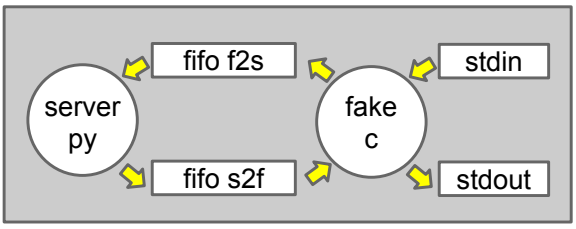
La fabrication des masques se fait avec les macros suivantes:

- `FD_ZERO(  fd_set *set);` -> efface tout l'ensemble
- `FD_SET( int fd, fd_set *set);` -> met à `1` un descripteur
- `FD_CLR( int fd, fd_set *set);` -> met à `0` descripteur
- `FD_ISSET(  int fd, fd_set *set);` -> rend l'état du descripteur `fd`

Exemple d'usage de select pour un timeout

```
#include <stdio.h>
#include <sys/time.h>

int main(void) {
     fd_set rfds;
     struct timeval tv;
     int retval;

     FD_ZERO(&rfds);    /* Surveiller stdin (fd 0) en attente d'entrées */
     FD_SET(0, &rfds);

     tv.tv_sec = 5;    /* Pendant 5 secondes maxi */
     tv.tv_usec = 0;

     retval = select(1, &rfds, NULL, NULL, &tv);

     if (retval) {
          char buffer[80];
          printf("Données disponibles\n"); /* FD_ISSET(0, &rfds) est vrai */
          fgets(buffer, sizeof(buffer),stdin);
          printf("got: %s\n", buffer);
     } else
          printf("Pas de données depuis 5 secondes\n");
     return 0;
}
```


TME



- Ecrire un site web sur une RaspberryPi 1 qui permet d'interagir avec les LEDs et le bouton poussoir
  (l'application ne devrait plus s'appeler fake, mais gateway…)