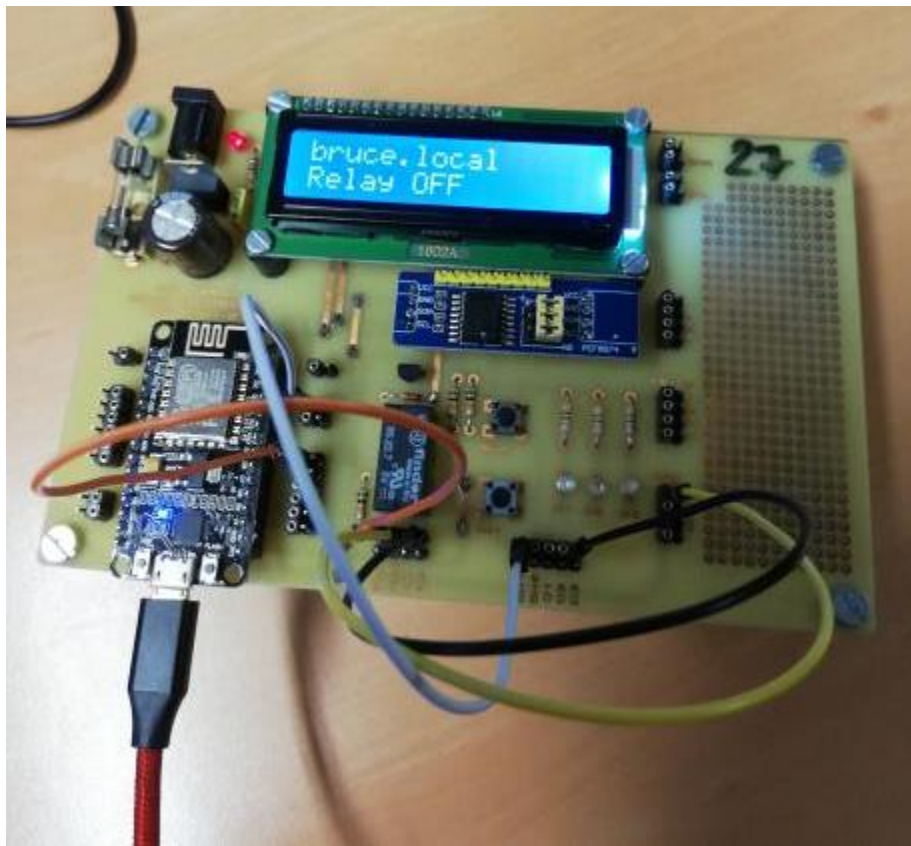


Rapport de TP : Réseaux

L'ESP 8266 comme serveur WEB

Au cours de ces 2 séances de TP, nous avons appris à créer un objet connecté, grâce à un serveur WEB tournant sur une carte ESP8266, permettant, depuis un interface WEB, d'interagir avec un système (dans notre cas un relais).

Nous utiliserons une maquette comprenant une carte ESP8266, un afficheur LCD, un relais et plusieurs LEDs pouvant être câblés au besoin avec des câbles duponts :



Connexion de l'ESP8266 à L'AP Wifi :

Pour commencer, nous allons apprendre à connecter notre carte à un réseau Wifi. Dans notre cas, le point d'accès à pour SSID « tp5&6 » et pour mot de passe « geii2021 ». On déclare donc ces 2 informations, ainsi que toutes les librairies que nous utiliserons au cours de ces TP . On déclare aussi la « ledVie », qui est une led que l'on fait clignoter tant que la carte est active (tant qu'elle est « vivante »). Cette partie du code ne changera pas, car elle est la base requise pour connecter l'ESP8266 à un réseau. Enfin, on inclut la librairie d'un écran à cristaux liquides (afficheur LCD), pour nous donner des informations tout au long des TP :

```

#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

#define ledVie D0

#ifndef STASSID
#define STASSID      "tp5&6"
#define STAPSK       "geii2021"
#endif

const char* ssid = STASSID;
const char* password = STAPSK;

LiquidCrystal_I2C lcd(0x27, 16, 2);

```

Ensuite viens la fonction *void setup()*, dans laquelle on déclare les entrées/sorties, on initialise l'écran LCD et on lance la connexion :

```

pinMode(ledVie, OUTPUT);
lcd.begin();
lcd.backlight();
Serial.begin(115200);
lcd.setCursor(0,0);
lcd.print("Try to connect");
lcd.setCursor(0,1);
lcd.print(STASSID);

//----- WIFI connexion -----
WiFi.mode(WIFI_STA); //station mode : la carte se connecte à un point d'accès
WiFi.begin(ssid, password); //commence la connexion
Serial.println("");
int i=0;
// Wait for connection
while (WiFi.status() != WL_CONNECTED)
{
    delay(500);
    Serial.print(".");
    lcd.setCursor(10,1);

    switch(i%3) { case 0: lcd.print("|"); break;
                  case 1: lcd.print("/"); break;
                  case 2: lcd.print("-"); break;
                }

    i++;
}
Serial.println("");
Serial.print("Connected to ");
Serial.println(ssid);
Serial.print("IP address: ");
Serial.println(WiFi.localIP());

```

Ensuite, on vérifie si le service mDNS est bien lancé :

```
if (MDNS.begin("bruce"))
{
  Serial.println("MDNS responder started");
}
lcd.setCursor(0,0);
lcd.print("Nom : bruce");
lcd.setCursor(0,1);
lcd.print(WiFi.localIP());
```

Le multicast Domain Name System (mDNS) est un service similaire au Domain Name System (DNS), pour des réseaux de petites tailles. Il permet à un appareil d'être identifié grâce à un nom de domaine. Pour ce faire, l'appareil envoie des requêtes mDNS à une adresse multicast, à laquelle les autres appareils sont à l'écoute, ce qui lui permet de s'identifier. Les autres appareils vont noter son IP dans leurs caches DNS, ce qui permettra de directement identifier l'IP de l'appareil en cas de requête (requête WEB par exemple). Ici un exemple de requête mDNS capturée avec Wireshark :

Time	Source	Destination	Protocol	Length	Info
192.168.1.157	224.0.0.251	MDNS	98	Standard query	0x0000 ANY bruce.local, "QU" question A 192.168.1.157
192.168.1.157	224.0.0.251	MDNS	98	Standard query	0x0000 ANY bruce.local, "QU" question A 192.168.1.157
192.168.1.157	224.0.0.251	MDNS	98	Standard query	0x0000 ANY bruce.local, "QU" question A 192.168.1.157
192.168.1.157	224.0.0.251	MDNS	121	Standard query response	0x0000 A, cache flush 192.168.1.157 PTR, cache flush bruce.local
192.168.1.157	224.0.0.251	MDNS	121	Standard query response	0x0000 A, cache flush 192.168.1.157 PTR, cache flush bruce.local
192.168.1.157	224.0.0.251	MDNS	121	Standard query response	0x0000 A, cache flush 192.168.1.157 PTR, cache flush bruce.local

Maintenant que l'appareil est identifié avec le nom « bruce.local », nous pouvons essayer de le ping pour vérifier la connexion :

```
[tihmz@parrotOS]-[~]
$ping bruce.local
PING bruce.local (192.168.43.220) 56(84) bytes of data:
64 bytes from bruce.local (192.168.43.220): icmp_seq=1 ttl=255 time=179 ms
64 bytes from bruce.local (192.168.43.220): icmp_seq=2 ttl=255 time=114 ms
64 bytes from bruce.local (192.168.43.220): icmp_seq=3 ttl=255 time=134 ms
64 bytes from bruce.local (192.168.43.220): icmp_seq=4 ttl=255 time=50.5 ms
64 bytes from bruce.local (192.168.43.220): icmp_seq=5 ttl=255 time=72.9 ms
^C
--- bruce.local ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 50.513/110.005/179.349/45.378 ms
[tihmz@parrotOS]-[~]
```

Cela va nous permettre de ne taper que « bruce.local » dans notre barre de recherche, lors des futurs tests sur le serveur WEB.

Enfin, pour revenir au code, nous pouvons mettre dans la boucle infinie (void loop()), le code qui permet de faire clignoter la led de vie, ainsi que la mise à jour du service mDNS :

```
void loop(void)
{
  digitalWrite(ledVie,
HIGH);
  delay(50);
  MDNS.update();
  digitalWrite(ledVie,
LOW);
  delay(50);
}
```

Le code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5%266/connectAP.ino>

Un serveur WEB basique :

Nous allons maintenant apprendre à faire tourner un serveur WEB sur notre carte, qui affichera l'état d'un switch (bouton poussoir) sur une page WEB.

Pour commencer, on rajoute dans la partie initialisation (au début du code) les lignes suivante :

```
#define button D5  
ESP8266WebServer server(80);
```

Cela nous permet d'indiquer que l'on va faire tourner un serveur WEB sur le port 80 (port http).

On indique aussi le label/numéro d'entrée du bouton (ici D5).

Il nous faut maintenant ajouter 2 fonctions :

`_handleNotFound()`, qui gère les demandes du client ne menant à aucun répertoire/fichier existant (la fameuse réponse « 404 Not found ») :

```
void handleNotFound()  
{  
    String message = "<html><body><center><h1> 404 : Not found </h1></center></body> </html>";  
    server.send(404, "text/html",message);  
}
```

`_handleRoot()` qui permet l'affichage d'une page dans le répertoire root (le 1^{er} répertoire, indiqué par le '/' après le nom de domaine) :

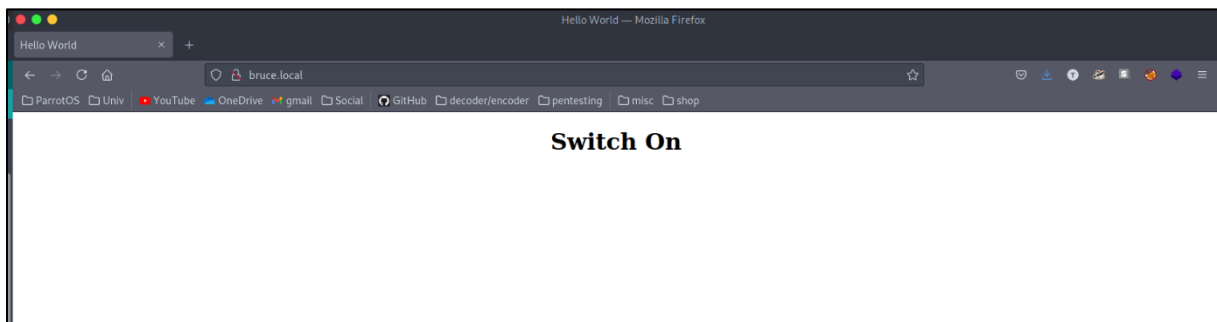
```
void handleRoot()  
{  
    String message = "<html> <head> <title> Hello World </title> </head>";  
    stateButton = digitalRead(button);  
    if(stateButton)  
    {  
        message += "<body> <center> <h1> Switch On </h1> </center> </body> </html> ";  
    }  
    else  
    {  
        message += "<body> <center> <h1> Switch Off </h1> </center> </body> </html> ";  
    }  
    server.send(200, "text/html",message);  
}
```

Dans les 2 cas, on utilise la fonction `server.send()`, qui permet d'afficher une page en html (ici la String « message ») avec un code de réponse (200 = found et 404=not found). Les codes de réponse http permettent de déterminer le résultat d'une requête, avec des codes connus comme 200 found, 404 not found, 403 Forbidden et bien d'autres.

Pour la fonction `handleNotFound()`, on affiche simplement une page avec écrit « 404 : Not Found » :



Et pour la fonction `handleRoot()`, elle affiche une page indiquant l'état du bouton (On ou Off).



Pour vérifier si la carte c'est bien connecté au réseau, on affiche un message de confirmation, l'IP associé à la carte et si le mDNS c'est bien lancé :

```
.....
Connected to WMR
IP address: 192.168.1.157
MDNS responder started
HTTP server started
```

Si l'on s'intéresse maintenant aux trame, on peut observer que chaque échange se fait en 2 trame. Une trame partant de notre machine avec un flag « GET / » (indique que l'on souhaite obtenir le contenu situé dans le répertoire « / »), et une trame partant du serveur vers notre machine, contenant la page à afficher avec le code 200 (found) :

Source	Destination	Protocol	Length	Info
192.168.43.4	192.168.43.220	HTTP	422	GET / HTTP/1.1
192.168.43.220	192.168.43.4	HTTP	209	HTTP/1.1 200 OK (text/html)
192.168.43.4	192.168.43.220	HTTP	422	GET / HTTP/1.1
192.168.43.220	192.168.43.4	HTTP	209	HTTP/1.1 200 OK (text/html)

Si l'on regarde le contenu d'une trame réponse (venant du serveur) on peut retrouver le code html de la page :

e8 6f 38 b4 bd 45 9c 9c 1f 46 61 95 08 00 45 00	..o8..E..
00 c3 00 51 00 00 ff 06 e2 b2 c0 a8 2b dc c0 a8	...Q....
2b 04 00 50 c0 34 00 00 1c 5b 6d 27 42 53 50 18	+..P.4..
06 f0 d7 79 00 00 3c 68 74 6d 6c 3e 20 3c 68 65	...y..<h tml>
61 64 3e 20 3c 74 69 74 6c 65 3e 20 48 65 6c 6c	ad> <tit le>
6f 20 57 6f 72 6c 64 20 3c 2f 74 69 74 6c 65 3e	o World </
20 3c 6d 65 74 61 20 68 74 74 70 2d 65 71 75 69	<meta h ttp-

```

> Internet Protocol Version 4, Src: 192.168.43.220, Dst: 192.168.43.4
> Transmission Control Protocol, Src Port: 80, Dst Port: 49204, Seq: 116, Ack: 369, Len: 155
> [2 Reassembled TCP Segments (270 bytes): #6(115), #8(155)]
> Hypertext Transfer Protocol
> Line-based text data: text/html (1 lines)
  <html> <head> <title> Hello World </title> <meta http-equiv="refresh" content="5" > </head> <body> <center> <h1> Switch Off </h1> </center> </body> </html>

```

Comme le code de la page change en fonction de l'état du bouton, on peut ajouter un élément qui lui permet de se rafraîchir toute seul. L'action de rafraîchir une page WEB consiste à redemander la page au serveur, ce qui peut être fait automatiquement avec la ligne de code suivante :

```
<meta http-equiv="refresh" content="5" >
```

Il suffit d'ajouter cette ligne dans la partie « `<head> </head>` » pour que la page se rafraîchisse toute seule toutes les 5 secondes. On modifie donc la String message de la manière suivante :

```
String message = "<html> <head> <title> Hello World </title>
                  <meta http-equiv=\"refresh\" content=\"5\" > </head>";
```

Le reste du code est le même que le premier. On ajoute cependant les lignes qui permettent de préciser quelles fonctions appeler en fonction de la demande du client :

```
server.on("/", handleRoot);
server.onNotFound(handleNotFound);
server.begin();
Serial.println("HTTP server started");
```

Enfin, dans la boucle `void loop()`, on rajoute la ligne suivante :

```
server.handleClient();
```

Elle permet de prendre en compte les requêtes du client, et de décider quelle fonction utiliser (`handleNotFound()` ou `handleRoot()`).

Le code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5%266/BasicWebServer.ino>

Un serveur WEB utilisant les paramètres de L'URL:

Comme vu précédemment, lors d'une requête WEB, on utilise la méthode « GET » pour obtenir les informations contenu dans un répertoire. Ce répertoire est indiqué dans l'URL (Uniform Resource Locator) de la page que l'on demande. Exemple : <http://stockage.fr/dossier/photo.jpg>

On demande ici l'image photo.jpg, dans le répertoire « dossier » du site à l'adresse stockage.fr

Cependant, il est aussi possible de transmettre des informations au serveur à travers l'URL, en modifiant les paramètres de ce dernier. Dans notre cas, cela ressemblera à :

<http://bruce.local/?relay=on>

L'indicateur « ? » indique le début des paramètres. « relay » correspond à l'argument et « on » correspond à sa valeur. On peut ajouter autant de paramètres que l'on veut, en les séparant avec l'indicateur « & ». On peut ainsi imaginer commander un moteur, et indiquer sa vitesse:

<http://bruce.local/?moteur=on&vitesse=30>

Nous allons donc ajouter à notre code un test pour vérifier si l'URL récupéré contient un argument et si oui, le traiter.

On ajoute donc dans notre fonction `handleRoot` le code suivant :

```
if(server.argName(0) == "relay")
{
  //relay=[on|off|tog]
  if(server.arg(0) == "on")
  {
    stateRelay = 1;
    digitalWrite(cmdeRelay,stateRelay);
    lcd.setCursor(0,1);
    lcd.print("Relay ON ");
  }
  else if (server.arg(0) == "off")
  {
    stateRelay = 0;
    digitalWrite(cmdeRelay,stateRelay);
    lcd.setCursor(0,1);
    lcd.print("Relay OFF");
  }
  else if (server.arg(0) == "tog")
  {
    stateRelay = 1;
    digitalWrite(cmdeRelay,stateRelay);
    lcd.setCursor(0,1);
    lcd.print("Relay TOG");
    stateTog = 1;
    timeTog = millis();
  }
}

stateRelay = digitalRead(cmdeRelay);
if(stateRelay) message += "OFF "; else message += "ON";
message += "</H1> <BR> <BR>If you want to change <BR> Use URL:
           http://name.local/?relay=[on|off|tog]</center></body> </html>";
server.send(200, "text/html",message);
```

Le premier test avec la fonction `server.argName()` vérifie si l'URL contient un argument nommé « relay ». Si c'est le cas, il va alors vérifier si sa valeur est « on », « off » ou « tog » avec la fonction `server.arg()`, qui contient la valeur de l'argument.

La valeur « on » active le relay qui allume une LED, la valeur « off » désactive le relay qui éteint la LED, et la valeur « tog » active le relay pour un temps indiqué avant de l'éteindre. Pour cela, on allume le relay, on stocke dans la variable `timeTog` le nombre de millisecondes écoulées depuis le lancement de la carte (lancement du programme), et on passe une variable booléenne `stateTog` à 1.

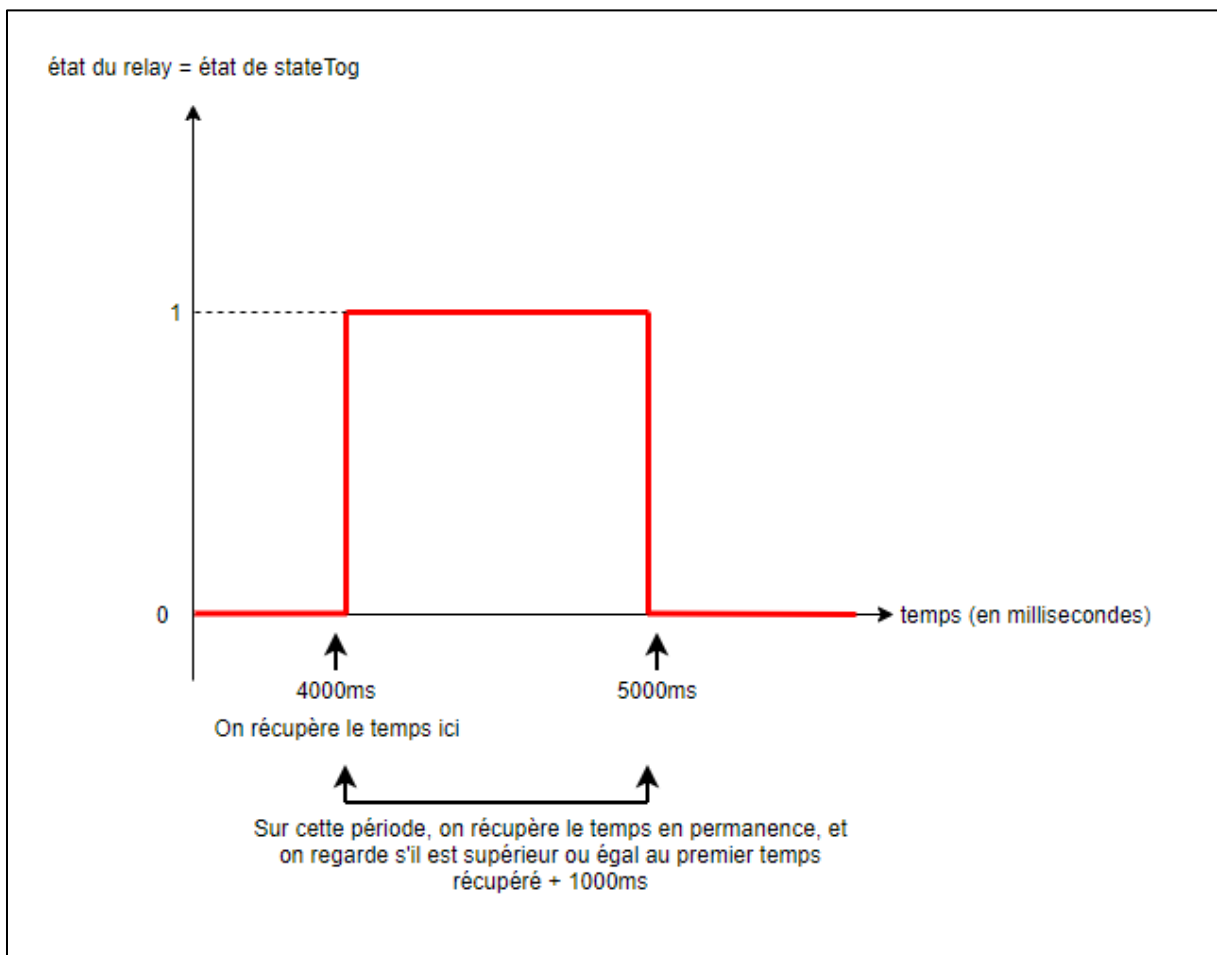
Ainsi, dans la boucle infinie `void loop()`, on ajoute une fonction qui test si la variable `stateTog` est à 1, et si c'est le cas, regarde le temps écoulé depuis l'activation du relay grâce à la variable `timeTog` et l'éteint au bout d'un certain temps :


```

if(stateTog == 1)
{
    if(millis() > timeTog + intervall )
    {
        stateTog = 0;
        stateRelay = 1; //0
        //digitalWrite(cmdeRelay,stateRelay);
        digitalWrite(LED_BUILTIN,stateRelay);
        lcd.setCursor(0,1);
        lcd.print("Relay OFF");
    }
}

```

Pour voir si le temps est écoulé, on reprend le nombre de millisecondes écoulées depuis le démarrage du programme, et on regarde s'il est supérieur au temps pris au moment de l'activation du relay + l'intervalle que l'on a fixé :



Le reste du code est le même, excepté l'ajout des variables *stateTog*, *timeTog* et *intervall* :

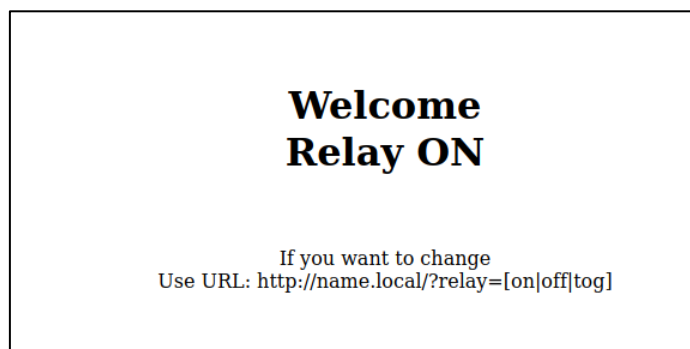
```

bool stateTog = false, stateRelay;
unsigned int timeTog = 0, intervall = 1000;

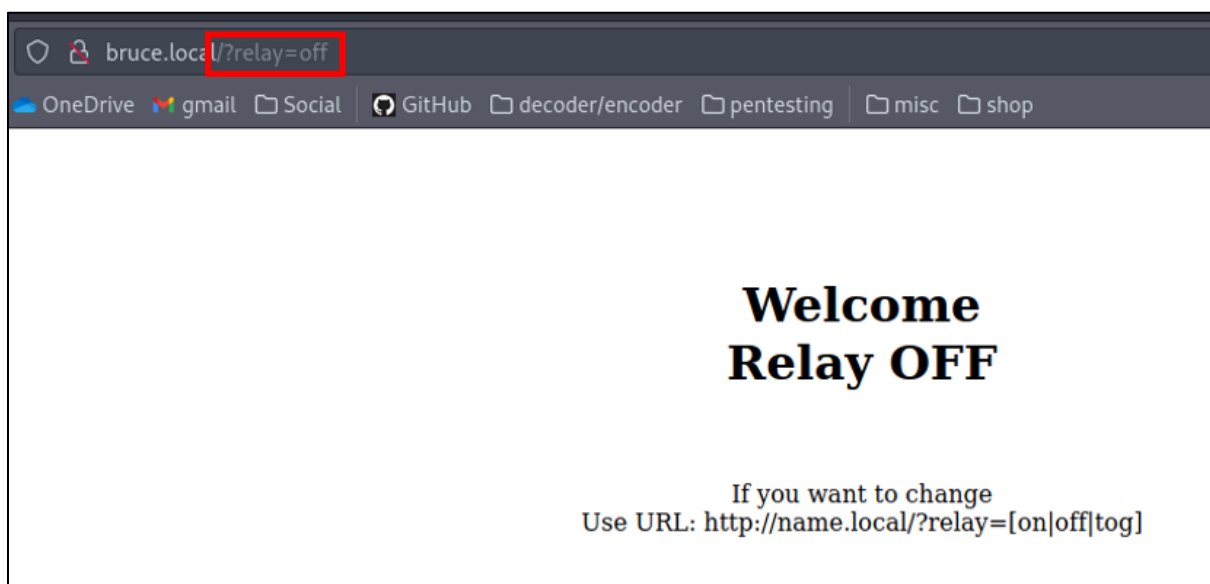
```


Le code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5%266/WebServerGETMethode.ino>

Ainsi, la page affichée est la suivante :



On peut ainsi changer l'état du relay en ajoutant le paramètres voulu dans l'URL :



Et si l'on regarde les trames, on peut observer les arguments dans la requete WEB :

Source	Destination	Protocol	Length	Info
192.168.43.4	192.168.43.220	HTTP	405	GET /?relay=on HTTP/1.1
192.168.43.220	192.168.43.4	HTTP	227	HTTP/1.1 200 OK (text/html)
192.168.43.4	192.168.43.220	HTTP	406	GET /?relay=off HTTP/1.1
192.168.43.220	192.168.43.4	HTTP	229	HTTP/1.1 200 OK (text/html)

Pour le reste, à chaque action, le serveur nous renvoie la page modifiée, indiquant l'état du relay.

Si on utilise un proxy pour intercepter les requêtes (un peu comme avec wireshark), il est facile de visualiser les paramètres URL de la méthode GET. Il permet aussi de voir le contenu de la requête, comme le code de réponse, ou des informations sur l'encodage et la taille des données transmises :

Request

1/ On demande le relay

```

1 GET /?relay=on HTTP/1.1
2 Host: 192.168.1.157
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:89.0) Gecko/20100101 Firefox/89.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 DNT: 1
8 Connection: close
9 Upgrade-Insecure-Requests: 1
10 Sec-GPC: 1
11
12

```

Response

```

1 HTTP/1.1 200 OK
2 Content-Type: text/html
3 Content-Length: 173
4 Connection: close
5
6 <html>
  <body>
    <center>
      <h1>
        <BR>
        <BR>
        Welcome <BR>
        Relay ON
      </h1>
    </center>
    <BR>
    <BR>
    If you want to change <BR>
    Use URL: http://name.local/?relay=[on|off|tog]
  </body>
</html>

```

2/ Le serveur nous renvoie une page avec état = ON →

Un serveur WEB utilisant la méthode POST :

Une autre méthode pour envoyer des données au serveur est la méthode POST. Contrairement à la méthode GET, on envoie des données non pas dans l'URL mais directement dans le corps de la requête.

Pour cela, nous devons modifier la page pour qu'elle affiche 2 boutons (ON et OFF) permettant de contrôler le relay. Comme nous commençons à avoir une page de taille importante, on crée la fonction `getPage()` qui nous retourne une String contenant la page :

```

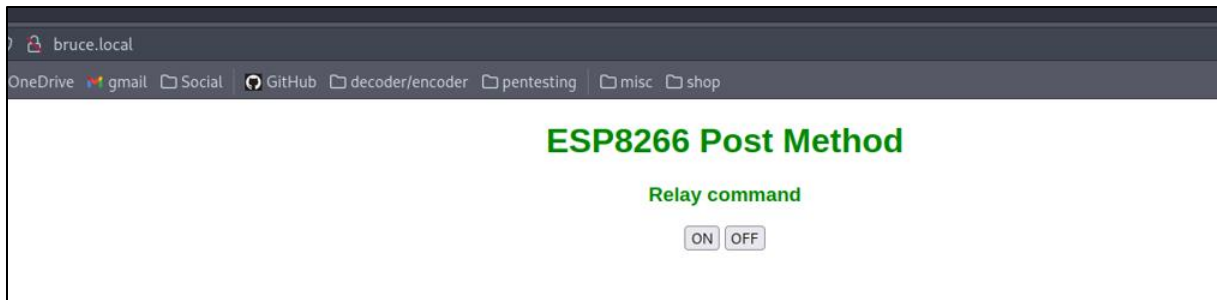
String getPage ()
{
    String page = "<html>";
    page += "<head><title>ESP8266 Post Method Q1</title>";
    page += "<style> body { background-color: #ffffff; font-family: Arial, Helvetica, Sans-Serif; Color: #008800; }</style>";
    page += "</head><body><center><h1>ESP8266 Post Method</h1>";
    page += "<h3>Relay command</h3>";
    page += "<form action='/' method='POST'>";
    page += "<button type='button submit' name='RLY' value='1'>ON</button>";
    page += "<button type='button submit' name='RLY' value='0'>OFF</button>";
    page += "</form></center></body></html>";
    return page;
}

```

On déclare donc 2 boutons utilisant la méthode POST avec la ligne :

```
<form action='/' method='POST'> ... </form>
```

On obtient donc la page suivante :



Ainsi, si on clique sur le bouton ON, on envoie une requête POST avec l'argument « RLY=1 »

On peut l'observer sur wireshark :

Source	Destination	Protocol	Length	Info
192.168.43.4	192.168.43.220	HTTP	528	POST / HTTP/1.1 (application/x-www-form-urlencoded)
192.168.43.220	192.168.43.4	HTTP	472	HTTP/1.1 200 OK (text/html)

▶	Frame 17: 528 bytes on wire (4224 bits), 528 bytes captured (4224 bits) on interface wlan0, id 0
▶	Ethernet II, Src: Chongqin_b4:bd:45 (e8:6f:38:b4:bd:45), Dst: Espressi_46:61:95 (9c:9c:1f:46:61:95)
▶	Internet Protocol Version 4, Src: 192.168.43.4, Dst: 192.168.43.220
▶	Transmission Control Protocol, Src Port: 49958, Dst Port: 80, Seq: 1, Ack: 1, Len: 474
▶	Hypertext Transfer Protocol
▼	HTML Form URL Encoded: application/x-www-form-urlencoded
▶	Form item: "RLY" = "1"

Cependant, le code reste pratiquement le même, car même si on utilise la méthode POST au lieu de GET, le serveur reçoit les mêmes arguments, on utilise donc les mêmes fonctions :

```
void handleRoot()
{
    String postValue;
    if ( server.hasArg("RLY") )
    {
        postValue = server.arg("RLY");
        if ( postValue == "1" )
        {
            stateRelay = HIGH;
            //digitalWrite(cmdeRelay, stateRelay);
            digitalWrite(LED_BUILTIN, stateRelay);
            lcd.setCursor(0,1);
            lcd.clear();
            lcd.print("Relay ON ");
            server.send ( 200, "text/html", getPage() );
        }
        else if ( postValue == "0" )
        {
            stateRelay = LOW;
            //digitalWrite(cmdeRelay, stateRelay);
            digitalWrite(LED_BUILTIN, stateRelay);
            lcd.setCursor(0,1);
            lcd.print("Relay OFF");
            server.send ( 200, "text/html", getPage() );
        }
    }
}
```

Si l'on n'a pas d'arguments, on affiche quand même la page sans autres actions, pour que le client voit la page quand il se connecte pour la première fois :

```
else
{
    String page = getPage();
    server.send ( 200, "text/html", page );
}
```

Si on observe la requête a travers un proxy, on peut noter l'apparition de l'argument « rly » :

The screenshot shows the Request and Response tabs of a web browser's developer tools. The Request tab is selected, showing a POST request to / HTTP/1.1. The headers include Host: 192.168.1.157, User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:89.0) Gecko/20100101 Firefox/89.0, Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8, Accept-Language: en-US,en;q=0.5, Accept-Encoding: gzip, deflate, Content-Type: application/x-www-form-urlencoded, Content-Length: 5, Origin: http://192.168.1.157, DNT: 1, Connection: close, Referer: http://192.168.1.157/?relay=on, Upgrade-Insecure-Requests: 1, Sec-GPC: 1. The body of the request is 'RLY=1', which is highlighted with a red box. The Response tab shows a 200 OK response with Content-Type: text/html, Content-Length: 616, and Connection: close. The response body is HTML content, including a title 'ESP8266 Post Method Q2' and a style block with background-color: #ffff; font-family: Arial, Helvetica, Sans-Serif; Color: #008800;.

On voit ainsi que l'argument RLY est du même niveau que les autres arguments par défaut tel que l'encodage, la taille, ...

Le code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5&6/webServerPostMethodQ1.ino>

Maintenant que nous savons comment la méthode POST fonctionne, nous pouvons améliorer notre interface WEB. Par exemple, nous pouvons ajouter un bouton pour rafraichir la page, ou encore afficher l'état du relay. On modifie donc le code de la page :

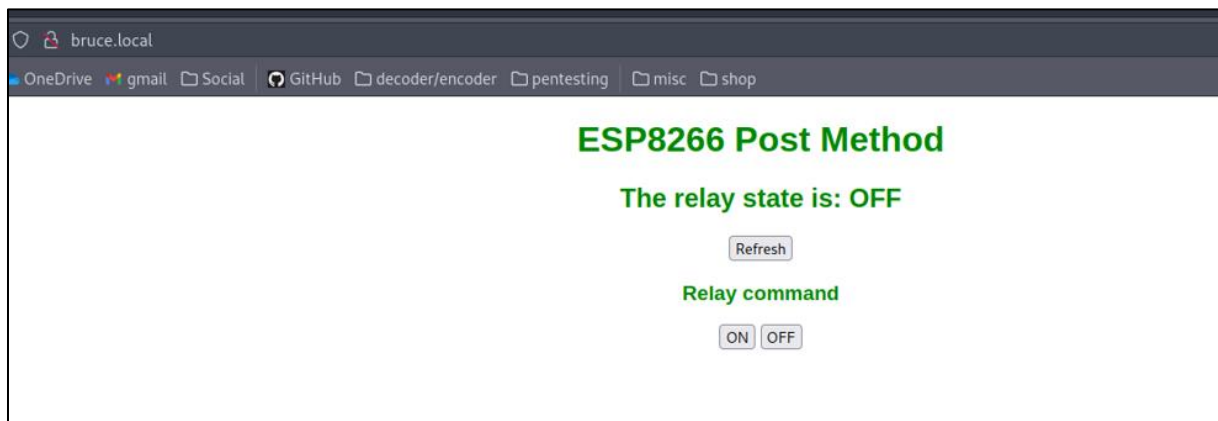
```
String getPage() {
    String page = "<html>";
    page += "<title>ESP8266 Post Method Q2</title>";
    page += "<style> body { background-color: #ffff; font-family: Arial, Helvetica, Sans-Serif; Color: #008800; }</style>";
    page += "</head><body><center><h1>ESP8266 Post Method</h1>";
    page += "<h2>The relay state is: ";
    page += stateRelay ? "ON" : "OFF"; //état du relay
    page += "</h2>";
    page += "<form action='/' method='POST'>";
    page += "<button type='button submit' name='refresh' value='1' >Refresh</button>";
    page += "</form>";
    page += "<h3>Relay command</h3>";
    page += "<form action='/' method='POST'>";
    page += "<button type='button submit' name='RLY' value='1'>ON</button> ";
    page += "<button type='button submit' name='RLY' value='0'>OFF</button>";
    page += "</form></center></body></html>";
    return page;
}
```

Il suffit ensuite d'ajouter à la fonction *handleRoot()* un test pour l'argument refresh=1 :

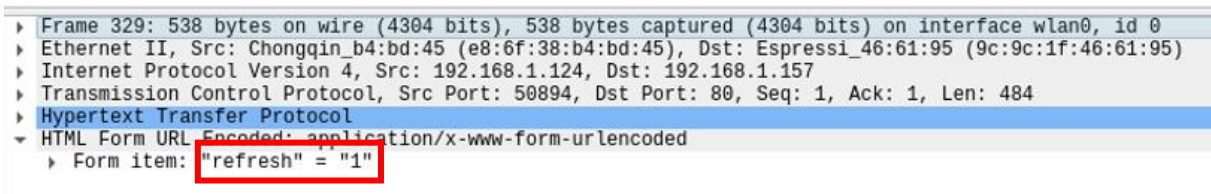
```
else if ( server.hasArg("refresh"))
{
    postValue = server.hasArg("refresh");
    if (postValue = "1")
    {
        server.send ( 200, "text/html", getPage() );
    }
}
```

Cette argument renvoie juste la page au client (ce qui correspond à l'action de rafraichir).

On obtient ainsi la page suivante :



Si on clique sur le bouton rafraichir, on peut observer sur wireshark le même échange de trame, mais avec un argument différent :



Le code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5&6/webServerPostMethodQ2.ino>

Pour finir, nous allons rajouter la fonction « toggle », comme vu avec la méthode GET. Pour cela rien de plus simple. Il suffit d'ajouter un bouton « toggle » dans l'argument relay :

```
page += "<button type='button submit' name='RLY' value='1'>ON</button>";
page += "<button type='button submit' name='RLY' value='0'>OFF</button>";
page += "<button type='button submit' name='RLY' value='tog'>TOG</button>";
```

Ensuite, il suffit d'ajouter une test pour « RLY=tog », de la même manière qu'avec la méthode GET :

```

else if (postValue == "tog")
{
    stateRelay = 1;
    digitalWrite(cmdeRelay,stateRelay);
    lcd.setCursor(0,1);
    lcd.print("Relay TOG");
    stateTog = 1;
    timeTog = millis();
    server.send ( 200, "text/html",
getPage() );
}

```

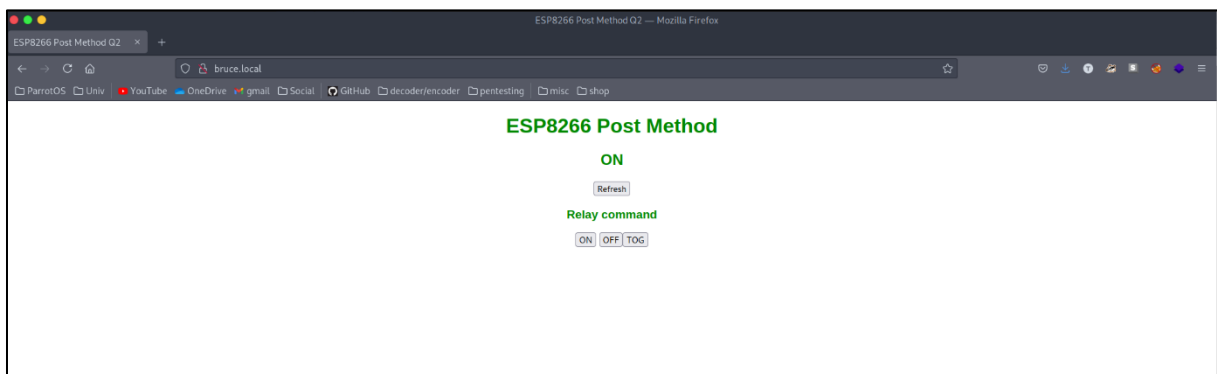
Et enfin, on rajoute le test dans la boucle infinie *void loop()* si *stateTog=1* :

```

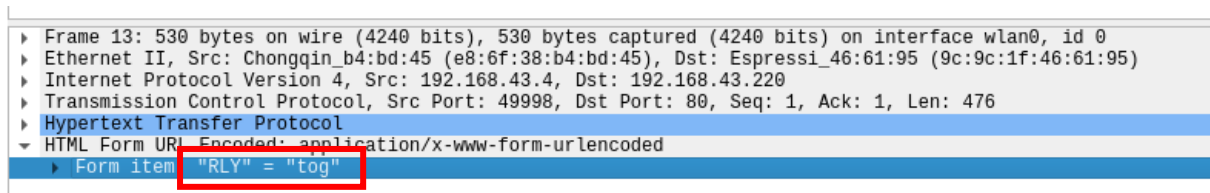
if(stateTog == 1)
{
    if(millis() > timeTog + intervall )
    {
        stateTog = 0;
        stateRelay = 0;
        digitalWrite(cmdeRelay,stateRelay);
        lcd.setCursor(0,1);
        lcd.print("Relay OFF");
    }
}

```

On obtient ainsi la page WEB suivante :



Et si on observe les trames, on retrouve la valeur « tog » pour l'argument « RLY » :



Le code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5%266/webServerPostMethodQ3.ino>

Bonus : Un serveur TCP Basique :

Pour cette partie du TP, on souhaite héberger un simple serveur TCP sur notre carte, qui permet à un client de se connecter, d'envoyer une commande au relay (ON ou OFF), puis se fait déconnecter.

Pour ce faire, nous allons garder de l'ancien code que la partie « connexion au réseau wifi ». On enlève aussi toute ligne de code se référant au serveur web. On rajoute également les variables globales suivantes :

```
char recept[18]="";
int i=0;
WiFiClient client;
char c;
bool verif=false;
unsigned long connexion_time;
```

Le tableau de caractères *recept[]* est le buffer qui contiendra les commandes du client (maximum 18 caractères). L'*int i* permettra de faire fonctionner une boucle *for()*, le *char c* contiendra le caractères tapé par le client, la variable booléenne *verif* permettra de vérifier que le client est bien en train de taper une commande et le *long connexion_time* permettra de mettre en place un timeout (expulsion du client au bout d'un certain temps). Enfin, on déclare *client*, type *WifiClient* (librairie *WifiClient.h*), qui permet de créer un client qui pourra se connecter sur notre serveur.

On rajoute aussi la ligne suivante :

```
WiFiServer server(1026);
```

Qui permet de déclarer que l'on souhaite faire tourner notre serveur sur le port 1026.

Dans le *void setup()*, rien ne change, nous avons toujours la déclaration de la led Vie et du relay, ainsi que la procédure de connexion au WIFI.

Pour le serveur TCP, toute la prise en compte des demandes de connexion et des commandes du client se fera dans la *void loop()*, la boucle infinie.

Pour commencer, on commence par tester si un client s'est connecté. Si oui, on récupère le temps de connexion et on affiche les instructions au client :

```
if (!client.connected() )//client is connected?
{
    client = server.available(); //test if one client is available
    if (client) //connexion established
    {
        client.write("\nEnter \"rly on\" or \"rly off\":\n");
        connexion_time=millis();
    }
}
```

Si on essaye de se connecter au serveur avec le client telnet (client permettant de se connecter à un serveur via TCP, avec la commande *telnet ip_adress port*), on obtient bien les instructions en cas de connexion :


```

[tihmz@parrotOS]-[~]
$telnet bruce.local 1026
Trying 192.168.43.220...
Connected to bruce.local.
Escape character is '^]'.

Enter "rly on" or "rly off":

```

Ensuite, on définit le timeout :

```

else //client is connected
{
    if(millis() >= connexion_time + 10000)
    {
        client.stop();
    }
    else
    {
        //handle client
    }
}

```

On utilise le même principe que pour le toggle de tout à l'heure : comme nous sommes dans la boucle infinie, les test se font en permanence, et il suffit de regarder si le temps présent est supérieur ou égal au temps de connexion + le temps de timeout que l'on a configuré (ici 10 secondes). Ainsi un client se connectant sera déconnecter au bout de 10 secondes.

On peut donc tenter de se connecter, et de ne taper aucune commande pour vérifier que l'on se fait bien timeout :

```

[tihmz@parrotOS]-[~]
$telnet bruce.local 1026
Trying 192.168.43.220...
Connected to bruce.local.
Escape character is '^]'.

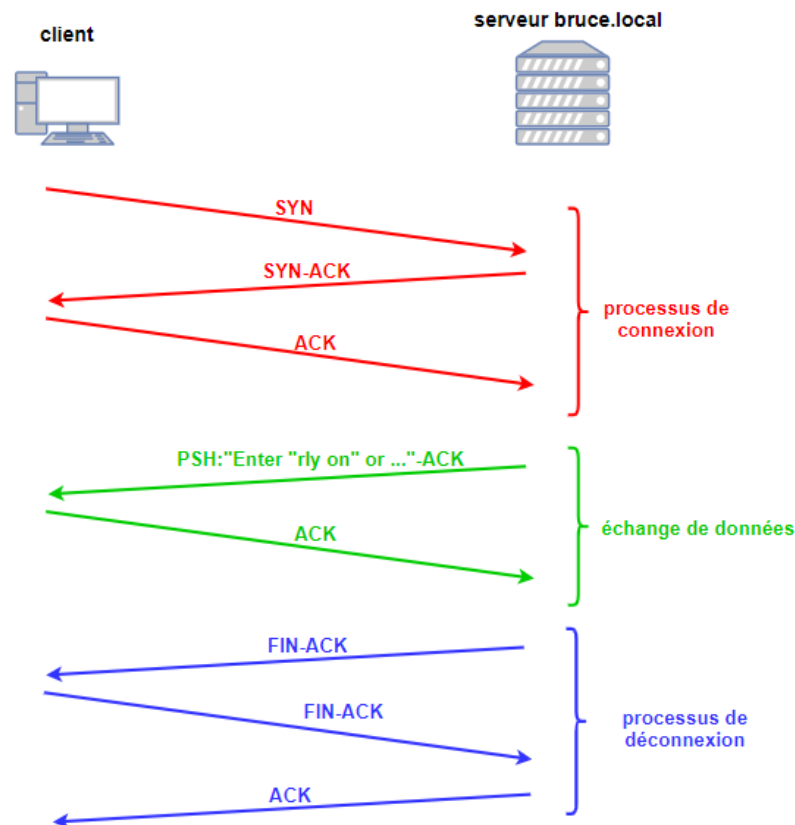
Enter "rly on" or "rly off":
Connection closed by foreign host.
[x]-[tihmz@parrotOS]-[~]

```

Si on s'intéresse aux échanges de trames, on peut retrouver les « flags » vus lors du TP n°2 :

Source	Destination	Protocol	Length	Info
192.168.43.4	192.168.43.220	TCP	74	38642 → 1026 [SYN] Seq= Win=64240 Len=0 MSS=1460 SACK_PERM=1 T
192.168.43.220	192.168.43.4	TCP	62	1026 → 38642 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0 MSS=536 SACK
192.168.43.4	192.168.43.220	TCP	54	38642 → 1026 [ACK] Seq= Ack=1 Win=64240 Len=0
192.168.43.220	192.168.43.4	TCP	84	1026 → 38642 [PSH, ACK] Seq=1 Ack=1 Win=2144 Len=30
192.168.43.4	192.168.43.220	TCP	54	38642 → 1026 [ACK] Seq= Ack=31 Win=64210 Len=0
192.168.43.220	192.168.43.4	TCP	54	1026 → 38642 [FIN, ACK] Seq=31 Ack=1 Win=2144 Len=0
192.168.43.4	192.168.43.220	TCP	54	38642 → 1026 [FIN, ACK] Seq=1 Ack=32 Win=64209 Len=0
192.168.43.220	192.168.43.4	TCP	54	1026 → 38642 [ACK] Seq=2 Ack=2 Win=2143 Len=0

L'échange sui le modèle suivant :



Avec SYN pour la connexion, ACK pour la confirmation de réception du dernier message, PSH pour le transfert de données et FIN pour la demande de déconnexion. Cet échange est logique, puisque on se connecte au serveur (partie rouge), le serveur nous envoie des instructions (partie verte), et comme nous n'envoyons aucune commande, on se fait expulser par le timeout au bout de 10 secondes (partie bleue).

Maintenant que nous arrivons à nous connecter au serveur et que le timeout fonctionne, nous allons coder la partie permettant de prendre en compte les commandes du client.

Pour commencer, quand le client se connecte, il faut récupérer 's'il y en a une) sa commande :

```
c = client.read(); //read char one by one
if(c=='r') verif=true; //check if the client is writing a command
if((c!='\n') && (c!='\r') && (i<17)) //c is not CR not LF and buffer is not overflow
{
    if(verif)
    {
        recept[i++] = c; //put c in the string recept
    }
}
```

Grace à la variable booléenne *verif*, on peut vérifier si le 1^{er} caractères est un 'r' (puisque les 2 commandes par le 'r' de « rly »). Cela permet d'éviter au programme de lire un caractères vides, et donc de remplir le buffer avec des caractères vides, et de déconnecter le client avant qu'il ait pu taper quoi que ce soit (si le buffer est plein, on éjecte le client). Avec le 2^{ème} test, on vérifie si le caractère n'est pas un '\n' (saut de ligne) ou un '\r' (retour à la ligne), car ils signifieraient que le client à appuyé sur la touche « Enter », et souhaite donc que sa commande soit traité. On vérifie aussi que le buffer n'est pas plein (buffer overflow). Si toutes les conditions sont bonnes, on vérifie que la *verif* vaut bien 1, et on remplit le buffer *recept[]* avec les caractères, un par un.

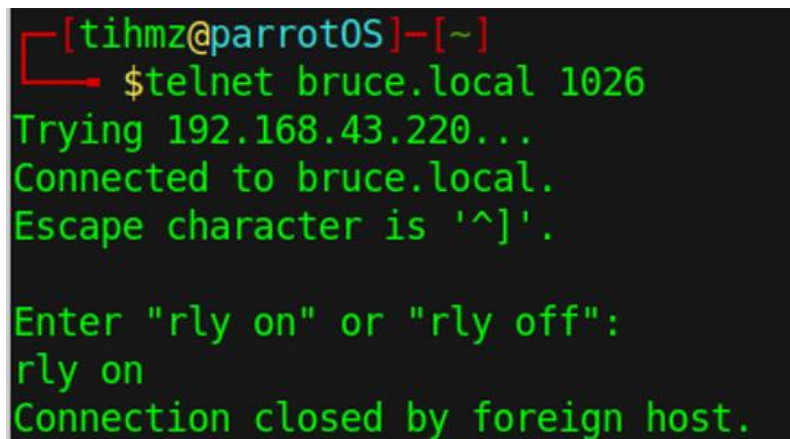
Une fois que le buffer est plein, ou que le client appuie sur la touche « Enter », il faut traiter la commande :

```
else
{
    recept[i]='\0';
    i=0;
    verif=false;
    client.stop();//disconnect the client
    lcd.clear();
    lcd.setCursor(0,0);
    String cmd = (String)recept;
    lcd.print("Last command:");
    lcd.setCursor(0,1);
    lcd.print(cmd);
    //Serial.println(cmd, HEX);
    if(cmd == "rly on") digitalWrite(RLY, HIGH);//command equals "rly on"?
    else if (cmd == "rly off") digitalWrite(RLY, LOW);//command equals "rly off"?
}
```

On commence par placer le caractères '\0' après le dernier caractères reçus. On réinitialise les variables *i* et *verif* pour les futurs connexion, et on éjecte le client. On transforme ensuite le tableau de caractères en String (chaines de caractères), et on traite la commande en conséquence (relay ON ou OFF).

Code complet : <https://github.com/IUT-Theophile-Wemaere/TP-reseau/blob/main/tp5%266/BasicTCPServer.ino>

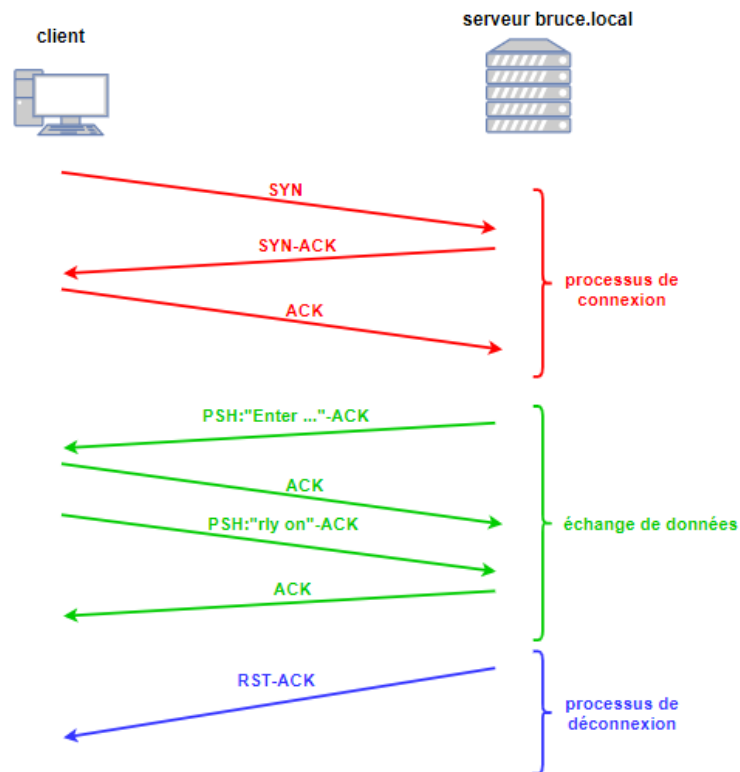
On peut maintenant essayer de connecter au serveur et de lui envoyer une commande :



```
[tihmz@parrotOS]-[~]
$telnet bruce.local 1026
Trying 192.168.43.220...
Connected to bruce.local.
Escape character is '^]'.

Enter "rly on" or "rly off":
rly on
Connection closed by foreign host.
```

Source	Destination	Protocol	Length	Info
192.168.43.4	192.168.43.220	TCP	74	38638 → 1026 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SAC
192.168.43.220	192.168.43.4	TCP	62	1026 → 38638 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0 MS
192.168.43.4	192.168.43.220	TCP	54	38638 → 1026 [ACK] Seq=1 Ack=1 Win=64240 Len=0
192.168.43.220	192.168.43.4	TCP	84	1026 → 38638 [PSH, ACK] Seq=1 Ack=1 Win=2144 Len=30
192.168.43.4	192.168.43.220	TCP	54	38638 → 1026 [ACK] Seq=1 Ack=31 Win=64210 Len=0
192.168.43.4	192.168.43.220	TCP	62	38638 → 1026 [PSH, ACK] Seq=1 Ack=31 Win=64210 Len=8
192.168.43.220	192.168.43.4	TCP	54	1026 → 38638 [ACK] Seq=31 Ack=9 Win=2136 Len=0
192.168.43.220	192.168.43.4	TCP	54	1026 → 38638 [RST, ACK] Seq=31 Ack=9 Win=24584 Len=0



Nous avons un échange similaire au premier, mais cette fois, on envoie aussi des données au serveur (partie verte). De plus, la connexion est coupée brutalement par le serveur, comme l'atteste le flag « RST ». La connexion n'étant pas cryptée, il est possible de récupérer les données échangées :

e8 6f 38 b4 bd 45 9c 9c 1f 46 61 95 08 00 45 00	08 00 E ..
00 46 00 3f 00 00 ff 06 e3 41 c0 a8 2b dc c0 a8	F ? ..
2b 04 04 02 96 ee 00 00 1b ed d0 3e e4 d4 50 18	+ ..
08 60 5d 0e 00 00 0a 45 6e 74 65 72 20 22 72 6c] ... E nter
79 20 6f 6e 22 20 6f 72 20 22 72 6c 79 20 6f 66	y on" or "rly
66 22 3a 0a	f":

9c 9c 1f 46 61 95 e8 6f 38 b4 bd 45 08 00 45 10	Fa ..
00 30 a0 11 40 00 40 06 c2 75 c0 a8 2b 04 c0 a8	@ @ ..
2b dc 96 ee 04 02 d0 3e e4 d4 00 00 1c 0b 50 18	+ ..
fa d2 08 ac 00 00 72 6c 79 20 6f 6e 0d 0a	rl y on ..

La blague de la semaine :

