Communication

Tim Berners-Lee

Les langages du WEB : anatomie d'une page WEB

La place du langage HTML dans le contexte d'une communication bidirectionnelle

Remue-méninges 08

Comment organiser une page HTML lorsque des informations de nature dynamique doivent être présentées ?

Section de Technicien Supérieur
option
Électronique
&
Communication

A. Préambule

L'accès au *WEB* est rendu possible par l'existence des liens hypertextes entre les pages du *WEB* et au protocole de communication *HTTP*. Mais pour être échangées à travers le monde, les pages doivent aussi utiliser des langages communs et de préférence standardisés.

Depuis sa création en 1990, le langage informatique **HTML** (***H**yper**T**ext **M**arkup **L**anguage*) est utilisé pour créer des pages *WEB*. C'est l'une des trois inventions issues du *W3C* avec le protocole *HTTP* et les adresses *WEB*. Son acronyme signifie "*langage de balisage d'hypertexte*" car il permet en effet de réaliser de l'hypertexte à base d'une structure de balisage.

B. Objectifs de ce document

Une page *WEB* est constituée de deux éléments distincts : le code *HTML* pour la structure et la feuille de style *CSS* pour la présentation.
Il existe de nombreuses publications et tutoriels sur le WEB qui permettent d'apprendre en toute autonomie. Il ne s'agit donc pas ici d'apprendre à réaliser un site WEB complet mais uniquement à vous faire découvrir l'écriture d'une page WEB dans un contexte bien particulier puisqu'il est lié à l'utilisation du protocole *WebSockets* pendant une relation de type client-serveur.

Concernant spécifiquement le langage HTML, voici quelques supports très intéressants :

1. Des ressources pour les développeurs de la fondation *Mozilla* (https://developer.mozilla.org/fr/docs/Web) et notamment celles concernant le *HTML* (https://developer.mozilla.org/fr/docs/Web/HTML),
2. Le site https://www.w3schools.com/default.asp pour la bibliothèque des balises *HTML*,
3. Un validateur *W3C* qui permet de vérifier si le code de la page *WEB* est conforme (https://validator.w3.org/),

w3schools.com

C. Situation-problème

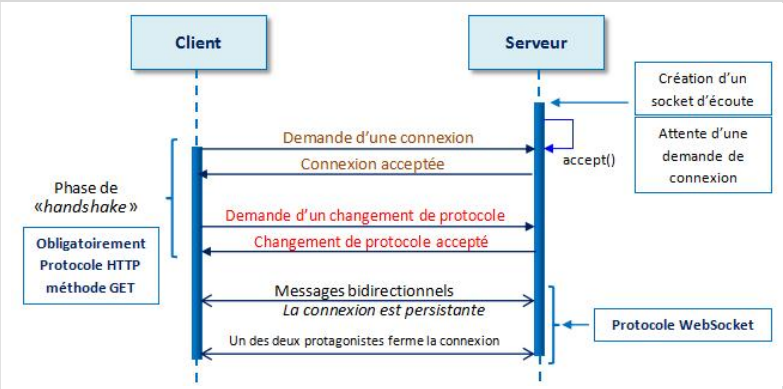


Figure n°1 : Relation Client-Serveur via le protocole *WebSockets*

Nous avons pu établir que les connexions *WebSockets* sont initialisées via un protocole HTTP (Phase de "*handshake*") Ainsi lors de l'établissement d'une relation client-serveur, le client (**qui est toujours à l'initiative de la communication**) envoie d'abord une requête HTTP avec une commande GET (par défaut).
Cette requête prend l'aspect de l'URL suivante :

http://192.168.1.20/led.html

192.168.1.20 est l'adresse IP du serveur et *led.html* représente la commande GET comme le stipule l'en-tête de la requête :

En langage courant, le client (n'importe quel navigateur WEB) demande au serveur WEB (192.168.1.20) de lui envoyer le fichier *led.html*. Cette conversation est conforme au protocole HTTP.

La première difficulté est donc la réalisation d'une page HTML conforme aux préconisations du W3C **et** intégrant les exigences suivantes :

- ✓ **Exigence n°1 :** La demande du changement de protocole soit le passage de HTTP à *WebSockets*. Comme le décrit la figure n°1, **cette demande est à l'initiative du client**.
- ✓ **Exigence n°2 :** La possibilité d'effectuer des actions sur un actionneur placé du côté serveur. Par exemple, deux boutons ON et OFF permettant l'éclairement et l'extinction d'une LED avec si possible l'acquittement de l'action.
- ✓ **Exigence n°3 :** Le retour des informations issues de capteurs connectés au serveur. Par exemple, la température renvoyée par un capteur DS18B20 avec le suivi de son évolution et une prise en compte de l'interface homme machine (IHM).

Nous savons d'autre part que le langage HTML, seul, ne peut prendre en charge toutes ces exigences. Le langage HTML a été conçu pour standardiser le format des pages envoyées par un serveur à un navigateur WEB. C'est donc uniquement un langage de balises qui a seulement un rôle structurant pour la page HTML.

HTML : provenant de l'anglais *Hypertext Markup Language*, Langage de balisage pour hypertexte. *HTML* est un langage de description des pages *WEB* permettant de structurer un texte en lui ajoutant des balises pour préciser comment le navigateur doit interpréter et mettre en forme le contenu.

Énoncé de la situation-problème :
Comment organiser la page HTML demandée par le client pour qu'elle puisse répondre aux besoins de communicabilité énoncées aux exigences 1 à 3 ?

D. Organisation des balises

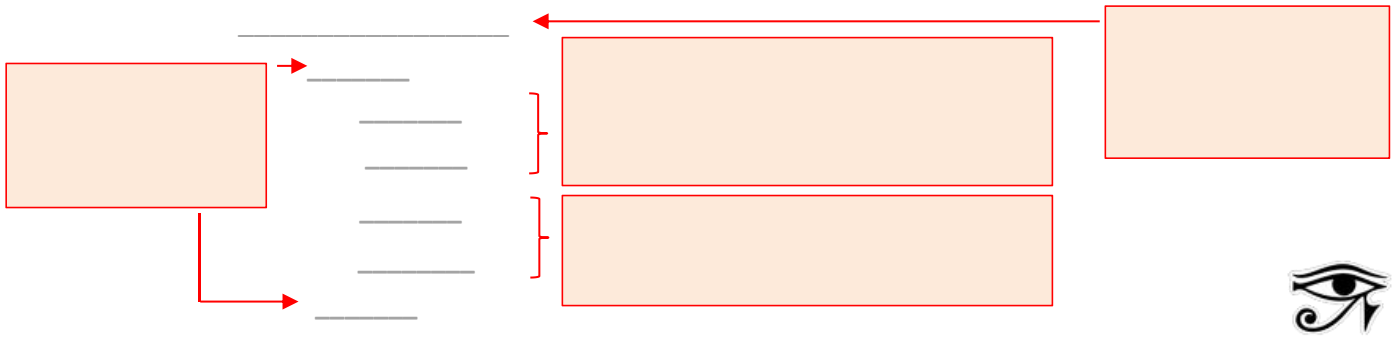
Nous prenons comme support le fichier "*LED.html*" (Voir annexe n°1)

Toutes les pages *HTML* qui vont être construites seront constituées *a minima* de balises dites de premier niveau : **<html> <head> <body>** sans oublier la première ligne **<!DOCTYPE html>** qui permet de définir le type de document afin qu'il soit correctement interprété par le navigateur.
C'est le "code minimal" d'une page *HTML*.

À partir de l'exemple du fichier LED.html, on peut retrouver les balises de premier niveau :

Danse avec les balises !

Replacez les balises telles que l'on doit les retrouver dans toute page HTML.



Et comprendre la structure du fichier "*LED.html*" :

Structure du fichier "*LED.html*"

```
<!DOCTYPE html>
<html>
<head>
```

Métadonnées : <meta charset="utf-8" /> et <meta http-equiv="Pragma" content="no-cache">

titre : <title>ESP32 - MicroWebSocket</title>

style :

```
html {font-family: Helvetica; display:inline-block; margin: 0px auto; text-align: center;}
    h1 {color: #0F3376; padding: 2vh;}
    p {font-size: 1.5rem;}
    .button {display: inline-block; background-color: #e7bd3b;border: none; border-radius: 4px; color: white; padding: 16px 40px;
text-decoration: none; font-size: 30px; margin: 2px;cursor: pointer;}
    .button2 {background-color: #4286f4;}
```

script :

```
var output;
function init() {
output = document.getElementById("output");
EtatLED = document.getElementById("Etat_LED");
```

... / ...

```
</head>
<body>
```

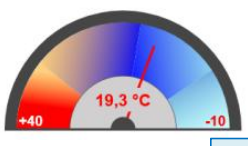
ESP32 avec Websockets2

Etat LED : **OFF**

ON

OFF

```
<h1>ESP32 avec Websockets2</h1>
<p id="Etat_LED">Etat LED : <strong>OFF</strong></p>
<p><button onClick = "LEDon()" class="button">ON</button></p>
<p><button onClick = "LEDoff()" class="button button2">OFF</button></p>
```




```
<svg id="thermometre_disqueSVG" xmlns="http://www.w3.org/2000/svg" version="1.1" width="300"
height="225" viewBox="-100 -100 200 100">
<defs>
<linearGradient id="grad_rouge" x1="60%" y1="20%" x2="50%" y2="70%">
<stop offset="0%" style="stop-color:rgb(255,163,3);stop-opacity:0.5" />
<stop offset="100%" style="stop-color:rgb(255,0,0);stop-opacity:1" />
</linearGradient>
```

... / ...

```
</body>
</html>
```

Conclusion

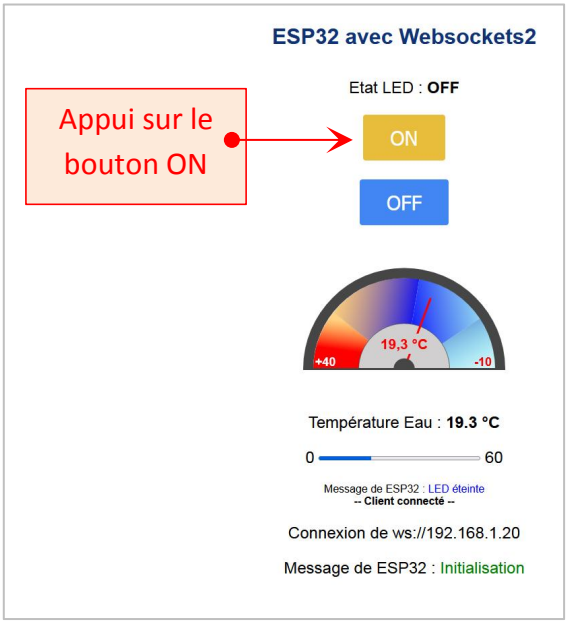
L'analyse du fichier "*LED.html*" a permis d'identifier une structure de balises de premier niveau, condition nécessaire pour que le fichier soit conforme aux préconisations W3C.
Pour obtenir des interactions avec l'utilisateur, côté client, il est nécessaire d'utiliser un langage de programmation, son rôle étant le traitement des commandes de l'utilisateur. Côté client, ce programme de programmation est le *javascript*.

E. Analyse du mécanisme d'interaction avec l'utilisateur

E.1 - Comment gérer l'envoi d'un ordre par le client ?

Étape n°1 : Exemple de l'appui sur le bouton "ON"

Étape n°2 : l'événement **"onClick"** associé au bouton "ON" a lieu et déclenche la fonction associée à l'événement **"LEDon()"**

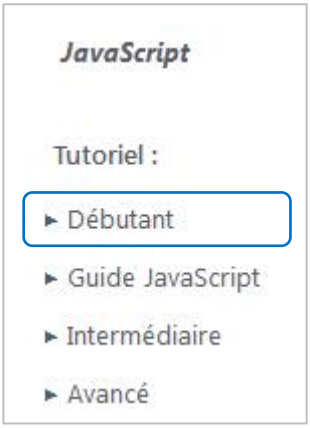


led.html

```
<h1>ESP32 avec Websockets2</h1>
<p id="Etat_LED">Etat LED : <strong>OFF</strong></p>
<p><button onClick = "LEDon()" class="button">ON</button></p>
<p><button onClick = "LEDoff()" class="button button2">OFF</button></p>
```

L'événement **"LEDon()"** est traité par le langage de programmation *javascript*.

La fondation Mozilla propose pour les débutants en *javascript* une zone d'apprentissage sur le site https://developer.mozilla.org/fr/docs/Web/JavaScript .



Étape n°3 : traitement de l'événement avec la fonction *javascript* **"LEDon()"**

```
function LEDon() {
    writeEtatLED("ON");
    //Message vers le serveur ESP32
    websocket.send("LEDon");
    }
```

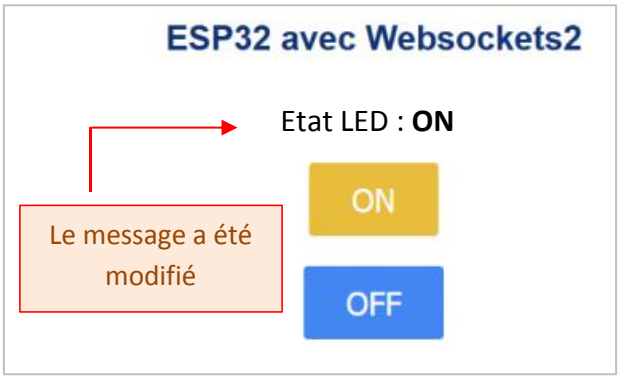
1. _____

led.html

```
function writeEtatLED(msg_LED) {
    EtatLED.innerHTML = "Etat LED : <strong>" + msg_LED + "</strong>";
    }
```

La propriété *innerHTML* permet de remplacer le contenu existant d'un élément (ici l'élément «**EtatLED**») par un nouveau contenu.
Le résultat obtenu est le suivant :

Remarque : aucun communication avec le serveur et aucune réactualisation complète de la page WEB n'ont été nécessaire.



led.html

```
function LEDon() {
    writeEtatLED("ON");
    //Message vers le serveur ESP32
    websocket.send("LEDon");
    }
```

2. _____

L'information «**LEDon**» est envoyée par le client au serveur

Pour pouvoir utiliser le protocole *websocket*, le client a créé un objet *websocket* et a essayé **automatiquement** d'ouvrir une connexion avec le serveur.

led.html

```
var wsUri = "ws://" + window.location.hostname;

websocket         = new WebSocket(wsUri);

websocket.onopen    = function(evt) { onOpen    (evt) };
websocket.onclose   = function(evt) { onClose   (evt) };
websocket.onmessage = function(evt) { onMessage (evt) };
websocket.onerror   = function(evt) { onError   (evt) };
```

3. _____

Le constructeur *WebSocket* accepte un paramètre obligatoire qui représente l'URL à laquelle le client se connecte. Il s'agit par conséquent de **l'adresse IP du serveur**

4. _____

Remarques importantes :

```
var wsUri = "ws://" + window.location.hostname;

websocket           = new WebSocket(wsUri);

websocket.onopen    = function(evt) { onOpen    (evt) };
websocket.onclose   = function(evt) { onClose   (evt) };
websocket.onmessage = function(evt) { onMessage (evt) };
websocket.onerror   = function(evt) { onError   (evt) };
```

1 WebSockets est une API[1] orientée événement , il existe par conséquent des gestionnaires d'événement :

- ✓ Le gestionnaire d'événement «**onopen**» est invoqué lorsque la connexion websocket est établie entre le client et le serveur,
  La fonction ***onOpen*** est associée avec comme paramètre l'événement qui a provoqué la connexion.
- ✓ Le gestionnaire d'événement «**onclose**» est invoqué lorsque la clôture de la connexion a été demandée avec la méthode *close()* (c'est la situation de la fonction *closeconnection()* dans le fichier *led.html*) et terminée.
  La fonction ***onClose*** est associée avec comme paramètre l'événement qui a provoqué la fermeture.
- ✓ Le gestionnaire d'événement «**onmessage**» est invoqué lorsqu'un message est envoyé par le serveur au client.
  La fonction ***onMessage*** est associée avec comme paramètre «**evt**» qui représente l'événement qui a provoqué son déclenchement. Pour accéder au message associé à l'événement, on utilisera la propriété *data* (exemple : msg_donnees = evt.data)
- ✓ Le gestionnaire d'événement «**onerror**» est invoqué lorsque qu'une erreur se produit.
  La fonction ***onError*** est associée avec comme paramètre («**evt**») l'événement qui a provoqué l'erreur. Elle provoque l'écriture de l'erreur («**evt.data**») sur l'écran du client :
  writeToScreen('ERROR : ' + evt.data + '');

2 Avant de réaliser toute action dans la page «*led.html*», il est indispensable d'attendre qu'elle soit chargée complètement par le navigateur.

C'est le rôle de la ligne en *javascript* suivante :

window.addEventListener("load", init, false);

Lorsque l'objet «**window**» est complètement chargé, la fonction «**init**» peut être exécutée.

Étape n°4 : traitement, par le serveur, de l'action demandée par le client

Dès le lancement des programmes côté serveur, **un socket "d'écoute"** est créé dans un fil d'exécution (*thread*). Cette situation de programmation indépendante permet, dans une boucle infinie (voir la fonction "*_serverProcess*" de *microWebSrv.py)*, une scrutation en continu (boucle *while True:*) du socket.

---

[1] API : Application Programming Interface ou Interface de programmation d'application est un ensemble normalisé de classes, de méthodes, de fonctions et de constantes qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

```
print("Préparation du serveur")
srv = MicroWebSrv(webPath='www/')
srv.MaxWebSocketRecvLen     = 256
srv.WebSocketThreaded       = True
srv.AcceptWebSocketCallback = _acceptWebSocketCallback
print("Serveur démarré")
srv.Start(threaded = True)
```

main.py

Fonctionne avec le principe des fils d'exécution (*thread*)

```
def _acceptWebSocketCallback(webSocket, httpClient) :
    print("Protocole WS accepté")
    webSocket.RecvTextCallback   = _recvTextCallback
    ## webSocket.RecvBinaryCallback = _recvBinaryCallback
    webSocket.ClosedCallback     = _closedCallback
```

```
def Start(self, threaded=False) :
    if not self._started :
        self._server = socket.socket()
        self._server.setsockopt( socket.SOL_SOCKET,
                                 socket.SO_REUSEADDR,
                                 1 )
        self._server.bind(self._srvAddr)
        self._server.listen(16)
        if threaded :
            MicroWebSrv._startThread(self._serverProcess)
        else :
            self._serverProcess()
```

microWebSrv.py

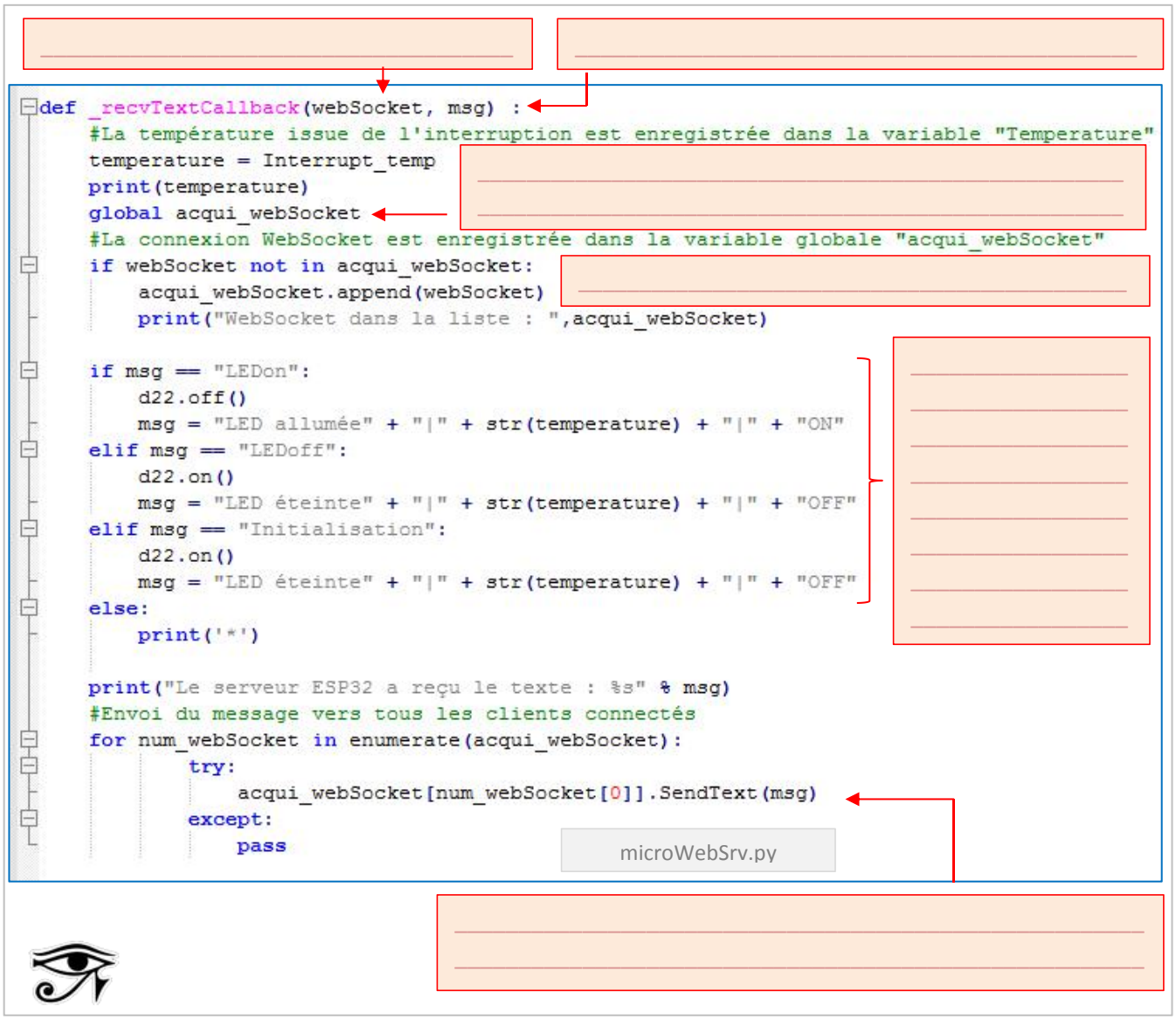
```
@staticmethod
def _startThread(func, args=()) :
    try :
        start_new_thread(func, args)
    except :
        global _mwsrv_thread_id
        try :
            _mwsrv_thread_id += 1
        except :
            _mwsrv_thread_id = 0
        try :
            start_new_thread('MWSRV_THREAD_%s' % _mwsrv_thread_id, func, args)
        except :
            return False
    return True
```

```
def _serverProcess(self) :
    self._started = True
    while True :
        try :
            client, cliAddr = self._server.accept()
        except Exception as ex :
            if ex.args and ex.args[0] == 113 :
                break
            continue
        self._client(self, client, cliAddr)
    self._started = False
```

Dès la création d'un objet *websocket* par le client et sa connexion automatique au serveur, un fil d'exécution (*thread*) **spécifique au client** est créé sur le serveur. Cette situation se répète pour les éventuels autres clients qui se connecteraient au serveur.

Ainsi chaque client connecté a un identifiant "*websocket*" dûment actualisé par le serveur dans une liste (appelée *acqui_webSocket*)

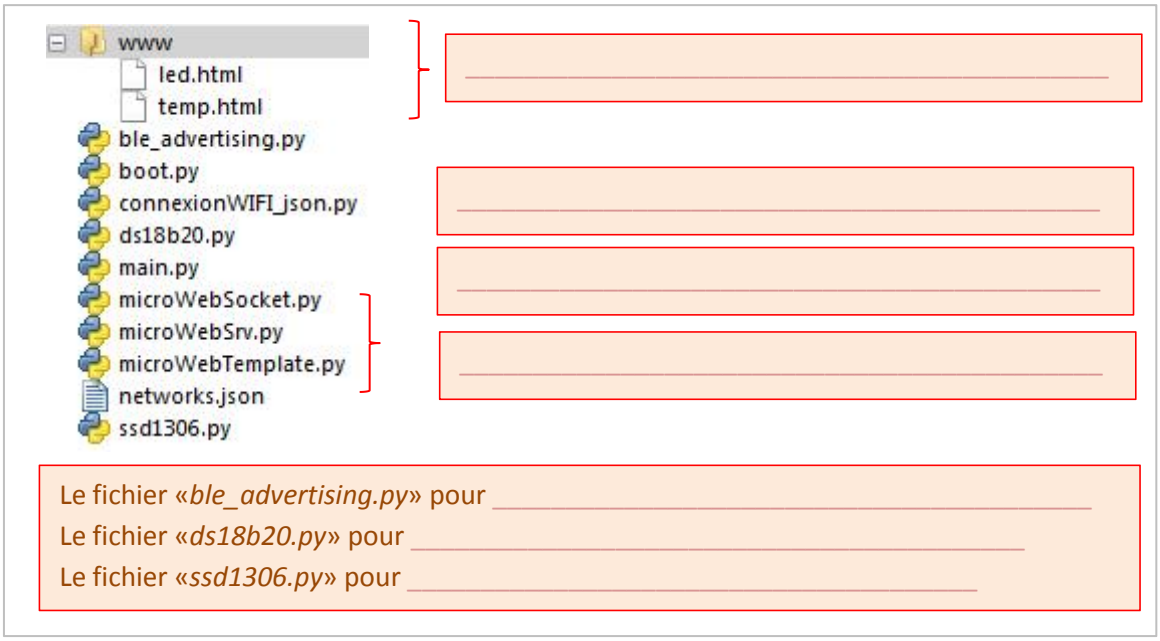
Dans le fil d'exécution attribué au client, une boucle infinie scrute le *socket* de la connexion :



Tout message envoyé par un client vers le serveur déclenche la fonction "***_recvTextCallback***"

Conclusion

Les mécanismes permettant l'envoi par le client d'un ordre au serveur selon le protocole *websocket* nécessite la présence et l'organisation des fichiers suivants :



Le fichier «*ble_advertising.py*» pour _____
Le fichier «*ds18b20.py*» pour _____
Le fichier «*ssd1306.py*» pour _____

Nous avons montré que les exigences n°1 et n°2 (demande de changement de protocole, demande d'actions avec acquittement) étaient satisfaites. Pour cela :

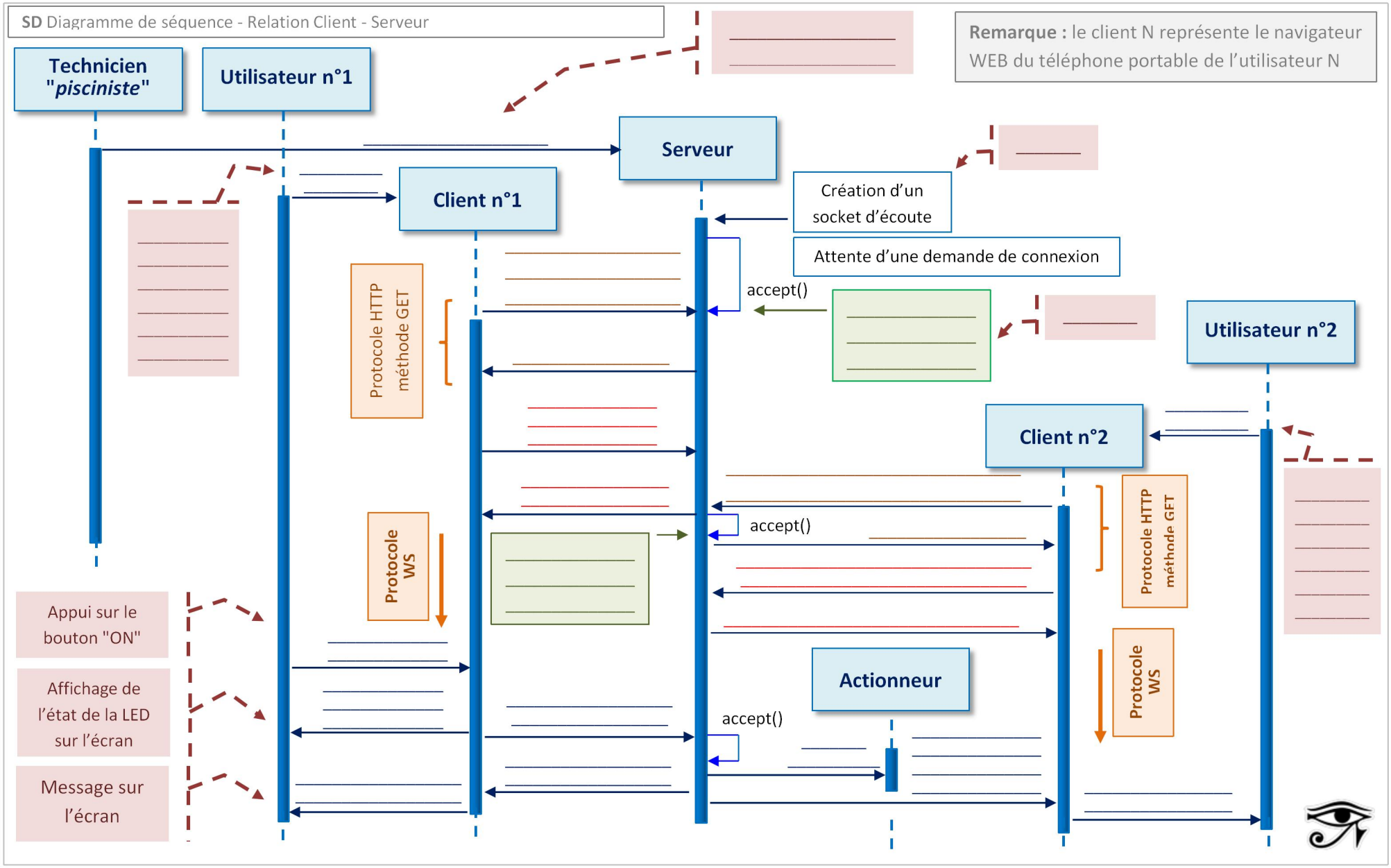
- ✓ **L'envoi d'un message par le client vers le serveur s'effectue simplement par : *websocket.send(*" «message à envoyer» "*)* dans la partie script du fichier .html**
- ✓ **L'acquittement par le serveur à destination des clients connectés s'effectue simplement par :**

```
for num_webSocket in enumerate(acqui_webSocket):
    try:
        acqui_webSocket[num_webSocket[0]].SendText(msg)
```

où msg représente le message à envoyer.

Diagramme de séquence

Le diagramme de séquence permet de décrire la chronologie des événements et des messages passés entre le client et le serveur, chaque élément possède sa propre ligne de vie.
La progression temporelle est verticale et les éléments (client et serveur) sont représentés horizontalement.


SD Diagramme de séquence - Relation Client - Serveur
Remarque : le client N représente le navigateur WEB du téléphone portable de l'utilisateur N
Technicien "pisciniste"
Utilisateur n°1
Serveur
Client n°1
Création d'un socket d'écoute
Attente d'une demande de connexion
accept()
Protocole HTTP méthode GET
Utilisateur n°2
Client n°2
Protocole WS
accept()
Protocole HTTP méthode GFT
Actionneur
Protocole WS
accept()
Appui sur le bouton "ON"
Affichage de l'état de la LED sur l'écran
Message sur l'écran

E.2 - Comment gérer l'envoi de messages par le serveur à tous les clients ?

Lorsqu'un client se connecte auprès du serveur, celui-ci lui attribue un identifiant unique lorsque le protocole *websocket* est accepté.

Exemple d'identifiants :

Lorsqu'un message doit être envoyé par le serveur à un client, il doit nécessairement actualiser les informations des autres clients.

Il existe deux situations possibles :

1

- ✓ Un client demande une action au serveur. Après avoir effectuée cette action, le serveur informe les autres clients connectés.
  Exemple : un client demande au serveur d'allumer la LED. Le serveur allume la LED et informe les autres clients que la LED est allumée. Ainsi les autres clients ne peuvent plus demander l'allumage de la LED mais seulement son extinction.

2

- ✓ Une information arrive au serveur et doit être diffusée à tous les clients connectés.
  Exemple : Une interruption périodique déclenche la mesure de la température (capteur DS18B20 et bus de terrain 1-wire connectés à un diffuseur BLE). La température est réceptionnée par l'observateur BLE qui joue également le rôle de serveur WIFI. Le serveur transmet ensuite la valeur de la température aux clients connectés.

Voyons quelles sont les étapes pour obtenir ces échanges d'informations :

Étape n°1 : Initialisation d'une variable contenant la liste des clients connectés.
Cette variable porte le nom de : acqui_webSocket = []. Elle possède le type «liste», c'est donc une variable qui contiendra d'autres variables.

```
#Initialisation de la variable qui permet de savoir si un WebSocket existe (un client s'est connecté)
acqui_webSocket = []
```

Étape n°2 : À chaque fois que le serveur reçoit une requête de la part d'un client **[Situation 1]** (activation de la fonction "*_recvTextCallback*"), par définition connecté, on procède à la scrutation du contenu de "*acqui_webSocket*" pour vérifier si ce client est répertorié.

```
def _recvTextCallback(webSocket, msg) :
    #La température issue de l'interruption est enregistrée dans la variable "Temperature"
    temperature = Interrupt_temp
    print(temperature)
    global acqui_webSocket
    #La connexion WebSocket est enregistrée dans la variable globale "acqui_webSocket"
    if webSocket not in acqui_webSocket:
        acqui_webSocket.append(webSocket)
        print("WebSocket dans la liste : ",acqui_webSocket)
```

main.py

Étape n°2bis : La diffusion d'une information issue d'un programme d'interruption (température par exemple). Ce programme d'interruption est hébergé sur le serveur.

```
import machine

time0 = machine.Timer(0)

#Mise en place de l'interruption périodique pour le captage et la transmission de la température
time0.init(period=15000, mode=machine.Timer.PERIODIC, callback=handleInterrupt)
```

main.py

```
def handleInterrupt(timer):
    #Fonction d'interruption permettant l'acquisition périodique de la température
    import ds18b20
    global Interrupt_temp
    global acqui_webSocket
    #Acquisition de la température du capteur
    scanner.scan(callback=on_scan)
    #Interrupt_temp = ds18b20.mesure_temp()
    print(Interrupt_temp)
    print("Tableau : ",acqui_webSocket)

    if len(acqui_webSocket) != 0:

        #Sans WebSocket existant, pas de transmission de température
        msg = "*" + "|" + str(Interrupt_temp) + "|" + "*"
        #Transmission de la température avec le WebSocket actif
        for num_webSocket in enumerate(acqui_webSocket):
            try:
                acqui_webSocket[num_webSocket[0]].SendText(msg)
            except:
                pass
```

Voir ci-dessous

main.py

Explication ligne «scanner.scan(callback=on_scan)»

```
# Instanciation du BLE
ble = ubluetooth.BLE()
scanner = BLE_Scan_Env(ble)

# Procède au scan
def scan(self, callback = None):
    # Initialise (vide) le set qui va contenir les messages
    self._message = set()
    # Assigne le callback qui sera appelé en fin de scan
    self._scan_callback = callback
    # Scanne pendant _SCAN_DURATION_MS, pendant des durées de _SCAN_WINDOWS_US espacées de _SCAN_INTERVAL_US
    self._ble.gap_scan(_SCAN_DURATION_MS, _SCAN_INTERVAL_US, _SCAN_WINDOW_US)
```

La fonction qui devra être appelée à l'issue de l'observation est «on_scan»

La fonction qui sera appelée à l'issue de l'observation est «on_scan»

```
# Fonction "callback" appelée à la fin du scan
def on_scan(message):
    # Pour chaque message d'advertising enregistré
    for payload in message:
        # On sépare les mesures grâce à l'instruction split
        global Interrupt_temp
        objet, temp = payload.split("|")
        Interrupt_temp = temp
        print("Message de " + objet + " : la température est de ", end="")
        print(temp + " °C", end="        \r")
        oled.fill(0)
        oled.text("Tair = "+ temp + " C", 0, 0)
        oled.show()
```

Seule la charge utile est considérée

Dissociation avec le caractère «|»

Écritures sur l'afficheur OLED

La température est placée dans la variable globale «*Interrupt_temp*»

main.py

Étape n°3 : L'envoi de tout message s'effectue sur l'ensemble des clients connectés.

```
#Envoi du message vers tous les clients connectés
for num_webSocket in enumerate(acqui_webSocket):
    try:
        acqui_webSocket[num_webSocket[0]].SendText(msg)
    except:
        pass
```

L'utilisation de la fonction *enumerate()* permet de ne pas à avoir à gérer une variable itérative

Envoi du message à chaque client de la liste

Variable de scrutation "num_webSocket"

main.py

Annexe n°1 : Fichier "*LED.html*"

```
<!DOCTYPE html>
<html>
<head>
	<meta charset="utf-8" />
	<title>ESP32 - MicroWebSocket</title>
	<meta http-equiv="Pragma" content="no-cache">

	<style>
	 html {font-family: Helvetica; display:inline-block; margin: 0px auto; text-align: center;}
	 h1 {color: #0F3376; padding: 2vh;}
	 p {font-size: 1.5rem;}
	 .button  {display: inline-block; background-color: #e7bd3b;
		   border: none; border-radius: 4px; color: white; padding: 16px 40px; text-decoration: none; font-size: 30px; margin: 2px;
		   cursor: pointer;}
	 .button2 {background-color: #4286f4;}
	</style>

	<script>
	var output;
	function init() {
		 output = document.getElementById("output");
		EtatLED = document.getElementById("Etat_LED");
		Msg_ESP32 = document.getElementById("MsgESP32");
		Msg_Valeur_Temp = document.getElementById("Valeur_Temp");
		Msg_ProgValeur_Temp = document.getElementById("ProgValeur_Temp");
		Msg_ESP32_Connexion = document.getElementById("MsgESP32Connexion");

		var wsUri = "ws://" + window.location.hostname;
		 writeToScreen("Connexion de " + wsUri)
		 websocket         = new WebSocket(wsUri);
		 websocket.onopen    = function(evt) { onOpen    (evt) };
		 websocket.onclose   = function(evt) { onClose   (evt) };
		 websocket.onmessage = function(evt) { onMessage (evt) };
		websocket.onerror   = function(evt) { onError   (evt) };
	 }
	function onOpen(evt) {
		Msg_ESP32_Connexion.innerHTML = "<strong>-- Client connecté --</strong>";
		//writeToScreen("<strong>-- CONNECTED --</strong>");
		SendMsg("Initialisation");
	 }
	function onClose(evt) {
		Msg_ESP32_Connexion.innerHTML = "<strong>-- Client déconnecté --</strong>";
		//writeToScreen("<strong>-- DISCONNECTED --</strong>");
	 }
	 function onMessage(evt) {
		//msg_donnees est déclarée comme une variable globale
		msg_donnees = evt.data.split("|");
		//MsgESP32('Message de ESP32 : <span style="color: blue;">' + evt.data + '</span>');
		if (msg_donnees.length == 3) {
			if (msg_donnees[0] != "*") {
			//Exclusion du message de l'état de la LED lorsque seule la température est transmise
				MsgESP32('Message de ESP32 : <span style="color: blue;">' + msg_donnees[0] + '</span>');
				writeEtatLED(msg_donnees[2]);
			}
```

```
                        MsgValeur_Temp("Température Eau : <strong>" + msg_donnees[1] + " °C</strong>");
                        MsgProgValeur_Temp('0 <progress max="60" value="' + msg_donnees[1] + '"></progress> 60');
                        var str_temperature_mesure = msg_donnees[1].toString();
                        champ_valeur_temp.innerHTML = str_temperature_mesure.replace(".",",") + " °C";
                        var angle = -msg_donnees[1]*18/5;
                        temps.angle_temp_1_svg.setAttribute("transform","rotate\("+angle+"\)");
                        }
    }
    function onError(evt) {
                writeToScreen('ERROR : <span style="color: red;">' + evt.data + '</span>');
    }
     function SendMsg(msg) {
                writeToScreen('Message de ESP32 : <span style="color: green;">' + msg + '</span>');
                websocket.send(msg);
    }
    function MsgProgValeur_Temp(s) {
                Msg_ProgValeur_Temp.innerHTML = s;
                }
    function MsgValeur_Temp(s) {
                Msg_Valeur_Temp.innerHTML = s;
    }
    function MsgESP32(s) {
                Msg_ESP32.innerHTML = s;
    }
     function writeToScreen(s) {
                 var pre = document.createElement("p");
                pre.style.wordWrap = "break-word";
                pre.innerHTML = s;
                output.appendChild(pre);
    }
    function writeEtatLED(msg_LED) {
                EtatLED.innerHTML = "Etat LED : <strong>" + msg_LED + "</strong>";
                }

//Lorsque l'objet "windows" est complètement chargé, la fonction "init" est exécutée
    window.addEventListener("load", init, false);

    function sayHello() {
                writeToScreen("HELLO");
                websocket.send("HELLO from button");
      }
    function LEDon() {
                writeEtatLED("ON");
      //Message vers le serveur ESP32
                 websocket.send("LEDon");
      }
     function LEDoff() {
                writeEtatLED("OFF");
      //Message vers le serveur ESP32
                websocket.send("LEDoff");
      }
function closeconnection() {
      writeToScreen("Closing ...");
      websocket.close();
      }
</script>
```

```
</head>

<body>

	<h1>ESP32 avec Websockets2</h1>
	<p id="Etat_LED">Etat LED : <strong>OFF</strong></p>
	<p><button onClick = "LEDon()" class="button">ON</button></p>
	<p><button onClick = "LEDoff()" class="button button2">OFF</button></p>

	<div>

<svg id="thermometre_disqueSVG" xmlns="http://www.w3.org/2000/svg" version="1.1" width="300" height="225" viewBox="-100 -100 200 100">
	<defs>
		<linearGradient id="grad_rouge" x1="60%" y1="20%" x2="50%" y2="70%">
		<stop offset="0%" style="stop-color:rgb(255,163,3);stop-opacity:0.5" />
		<stop offset="100%" style="stop-color:rgb(255,0,0);stop-opacity:1" />
		</linearGradient>
		<linearGradient id="grad_orange" x1="0%" y1="20%" x2="100%" y2="40%">
		<stop offset="0%" style="stop-color:rgb(255,163,3);stop-opacity:0.5" />
		<stop offset="100%" style="stop-color:rgb(3,7,255);stop-opacity:1" />
		</linearGradient>
		<linearGradient id="grad_bleu_fonce" x1="0%" y1="50%" x2="100%" y2="30%">
		<stop offset="0%" style="stop-color:rgb(3,7,255);stop-opacity:1" />
		<stop offset="100%" style="stop-color:rgb(108,200,235);stop-opacity:0.75" />
		</linearGradient>
		<linearGradient id="grad_bleu_clair" x1="0%" y1="0%" x2="100%" y2="100%">
		<stop offset="0%" style="stop-color:rgb(56,58,185);stop-opacity:1" />
		<stop offset="100%" style="stop-color:rgb(21,229,239);stop-opacity:0.15" />
		</linearGradient>
	</defs>

	<path d="M -90,0 a 90,90 0 0 1 20,-56 L 0,0 z" stroke="none" fill="url(#grad_rouge)" />
	<path d="M -70,-56 a 90,90 0 0 1 85,-33 L 0,0 z" stroke="none" fill="url(#grad_orange)" />
	<path d="M 15,-89 a 90,90 0 0 1 60,39 L 0,0 z" stroke="none" fill="url(#grad_bleu_fonce)" />
	<path d="M 75,-50 a 90,90 0 0 1 15,50 L 0,0 z" stroke="none" fill="url(#grad_bleu_clair)" />
	<path d="M -95,0 a 90,90 0 0 1 190,0" stroke="#464646" stroke-width="10" fill="none" />
	<path d="M -45,0 a 45,45 0 0 1 90,0 L 0,0 z" stroke="#464646" stroke-width="0.5" fill="#d1cfcf" />
	<line x1="-100" y1="0" x2="100" y2="0" stroke="#464646" stroke-width="2" />

	<text x="78" y="-3" text-anchor="middle" font-size="12px" font-weight="bold" font-family="Arial" fill="red">-10</text>
	<text x="-78" y="-3" text-anchor="middle" font-size="12px" font-weight="bold" font-family="Arial" fill="white">+40</text>

	<line id="angle_temp_1" x1="0" y1="0" x2="75" y2="0" fill="red" stroke="red" stroke-width="2" transform="rotate(-105)" />
	<path d="M -10,0 a 10,10 0 0 1 20,0" stroke="#464646" fill="#464646" />
	<rect x="-22.5" y="-35" width="45" height="20" stroke="none" fill="#d1cfcf" />
	<text id="valeur_temp" x="0" y="-20" text-anchor="middle" font-size="14px" font-weight="bold" font-family="Arial" fill="red">25,5 °C</text>
</svg>

	</div>

  <p id="Valeur_Temp"></p>
  <p id="ProgValeur_Temp"></p>
  <div id="MsgESP32"></div>
  <div id="MsgESP32Connexion"></div>
  <div id="output"></div>
```

```
        <script>
                var temperature_mesure = 20.5;
                var str_temperature_mesure = temperature_mesure.toString();
                var image_svg = document.querySelector("#thermometre_disqueSVG");
                var temps = {
                        angle_temp_1_svg: document.querySelector("#angle_temp_1"),
                }
                var champ_valeur_temp = document.getElementById("valeur_temp");
        </script>

    </body>
</html>
```