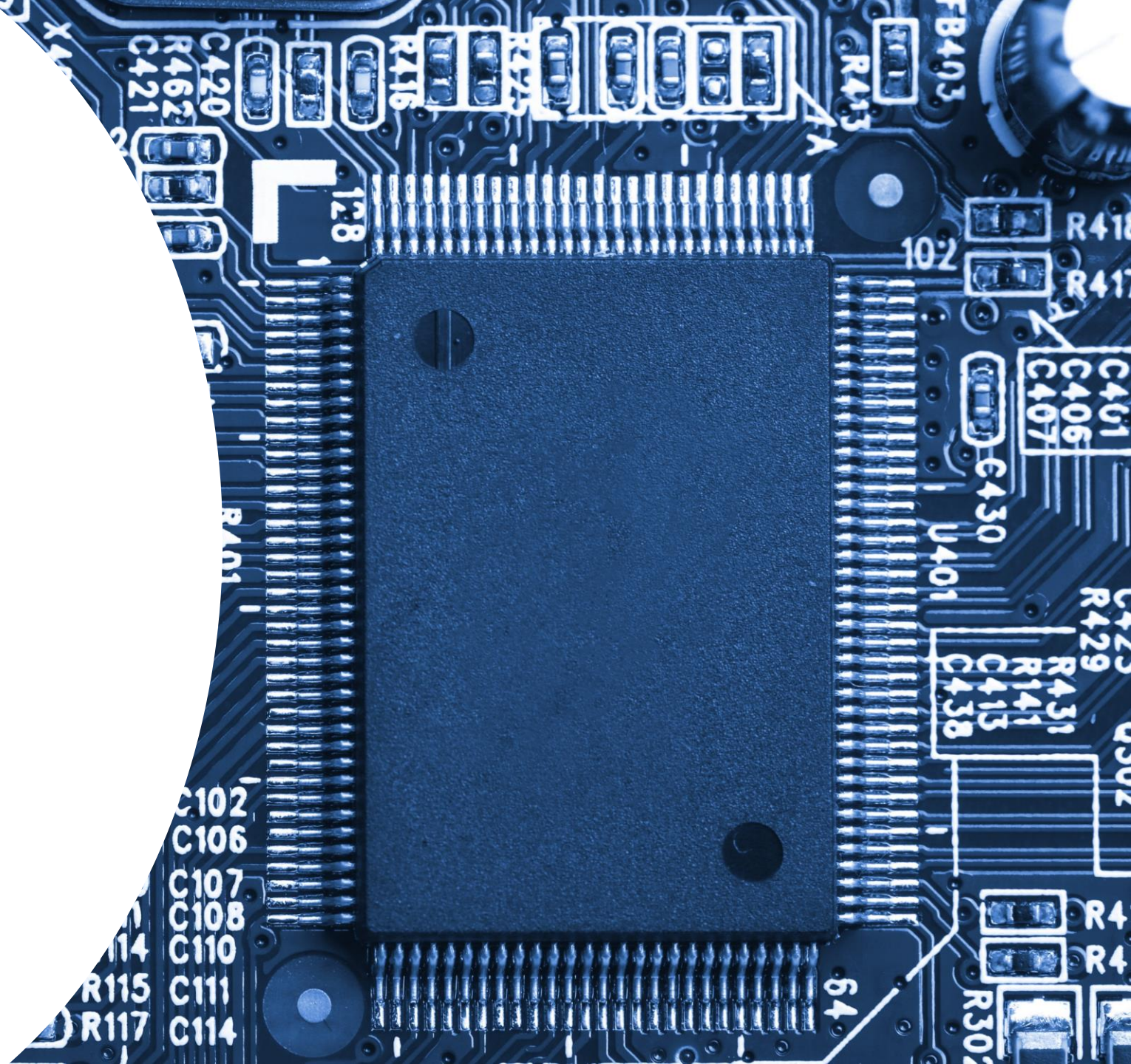


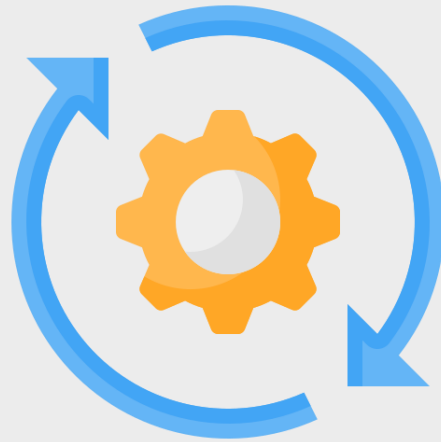
Wifi Zephyr et Python

SNIOT

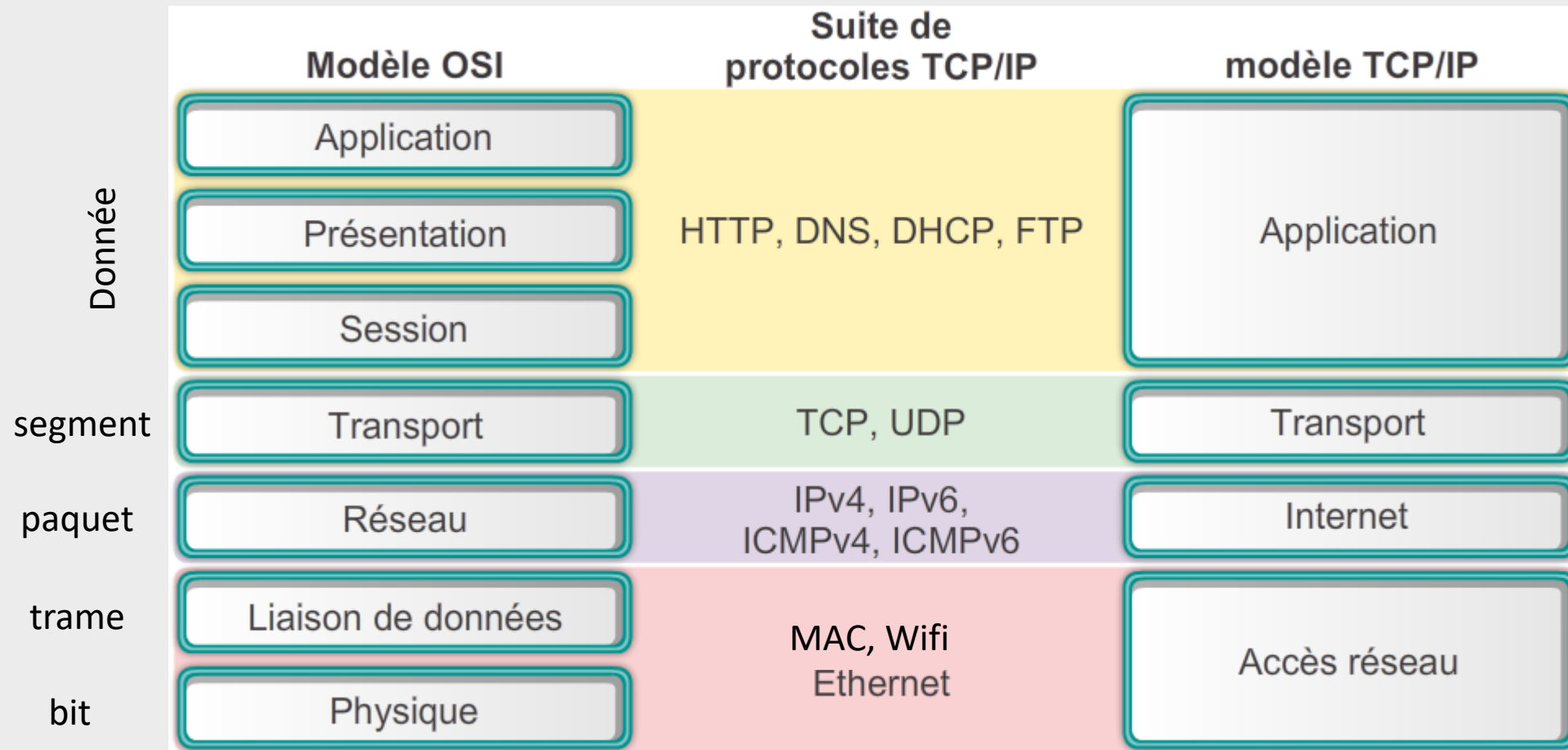


Objectifs

- Comprendre la base d'un réseau TCP/IP
- Créer des requêtes HTTP sous Zephyr et les transmettre en Wifi
- Développer une API REST



Modèle OSI (Open System Interconnexion)



Source: <https://linux-note.com/modele-osi-et-tcpip/>

Protocole TCP/IP

Protocole TCP/IP

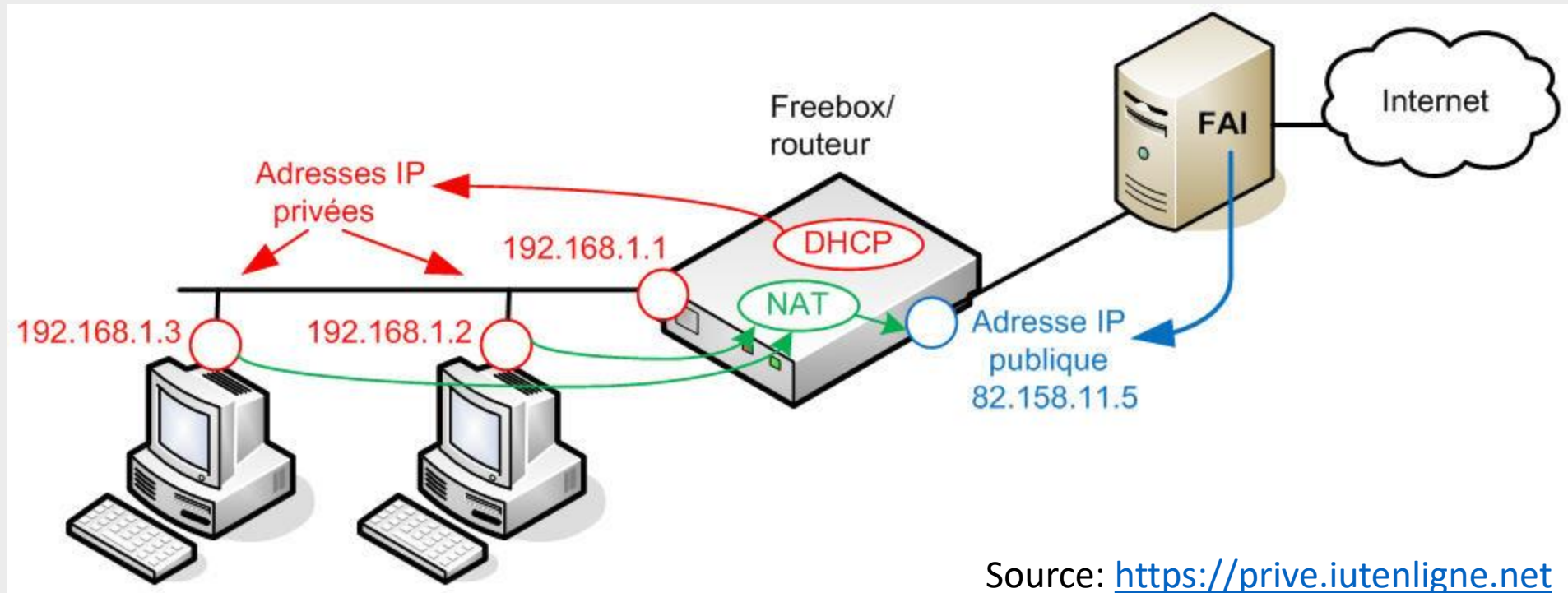
- IP (Internet Protocol): couche 3 OSI (réseau)
 - Adressage: Identification unique de chaque machine sur le réseau
 - Routage: Trouver le chemin pour remettre le paquet
- TCP (Transmission Control Protocol): couche 4 OSI (transport) permet de transmettre les paquets de manière fiable à la destination identifiée dans le protocole IP.

Analogie:

IP = Adresse postale ou l'on dépose la lettre.

TCP = Méthode d'envoi de la lettre

Protocole TCP/IP



DHCP: Permet de fournir des adresses IP automatiquement sur le réseau privé

NAT/PAT: Permet d'attribuer une adresse publique aux machines du réseau privé

Protocole HTTP

Définition

- Protocole de la couche application dans le modèle OSI
- Permet d'effectuer des requêtes avec des commandes:
 - **GET** => Permet de demander une ressource au serveur
Exemple: Afficher une page internet
 - **POST** => Permet d'envoyer des données
Exemple: Formulaire, image
 - Autres: DELETE / HEAD / PUT ...

Exemple de requête

GET

```
▼ Hypertext Transfer Protocol
  > GET / HTTP/1.1\r\n
    User-Agent: PostmanRuntime/7.29.2\r\n
    Accept: */*\r\n
    Postman-Token: e64400e9-6017-4f15-a4ad-e913212fb998\r\n
    Host: www.google.com\r\n
    Accept-Encoding: gzip, deflate, br\r\n
    Connection: keep-alive\r\n
    \r\n
```

POST

```
▼ Hypertext Transfer Protocol
  > POST / HTTP/1.1\r\n
    Content-Type: text/plain\r\n
    User-Agent: PostmanRuntime/7.29.2\r\n
    Accept: */*\r\n
    Postman-Token: 90533282-3767-4f89-b645-7c3c15d7d1e8\r\n
    Host: www.google.com\r\n
    Accept-Encoding: gzip, deflate, br\r\n
    Connection: keep-alive\r\n
  > Content-Length: 53\r\n
    \r\n
    \[Full request URI: http://www.google.com/\]
    [HTTP request 4/4]
    \[Prev request in frame: 15689\]
    \[Response in frame: 16006\]
    File Data: 53 bytes
  ▼ Line-based text data: text/plain (4 lines)
    {
      "temperature" = "26"\r\n
      "humidity" = "58"\r\n
    }
```

API REST

Avec Python et Flask

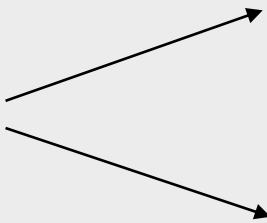
Définition

- Permet la communication entre des applications indépendamment de la technologie utilisée via le protocole HTTP.
- Communication de type client / serveur
- Communication stateless => chaque requête contient l'ensemble des informations permettant de la traiter.
- Les commandes sont:
 - GET: accès à une ressource
 - POST: création d'une ressource
 - DELETE: suppression d'une ressource
 - PATCH: mise à jour d'une ressource

Exemple

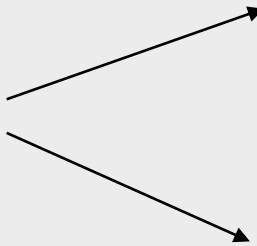
- API pour gérer les élèves d'une école

url: www.seatech.com/students



- GET: Renvoi la liste des élèves
- POST: Ajoute un élève à la liste des élèves

url: www.seatech.com/students/dupont



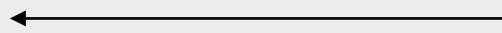
- GET: Renvoi les informations sur l'étudiant Dupont
- PUT: Modifie les informations sur l'étudiant Dupont

API avec Python et Flask

```
from flask import Flask
```

```
app = Flask(__name__)
```

```
@app.route("/", methods=['GET'])  
def hello_world():  
    return "Hello world !"
```



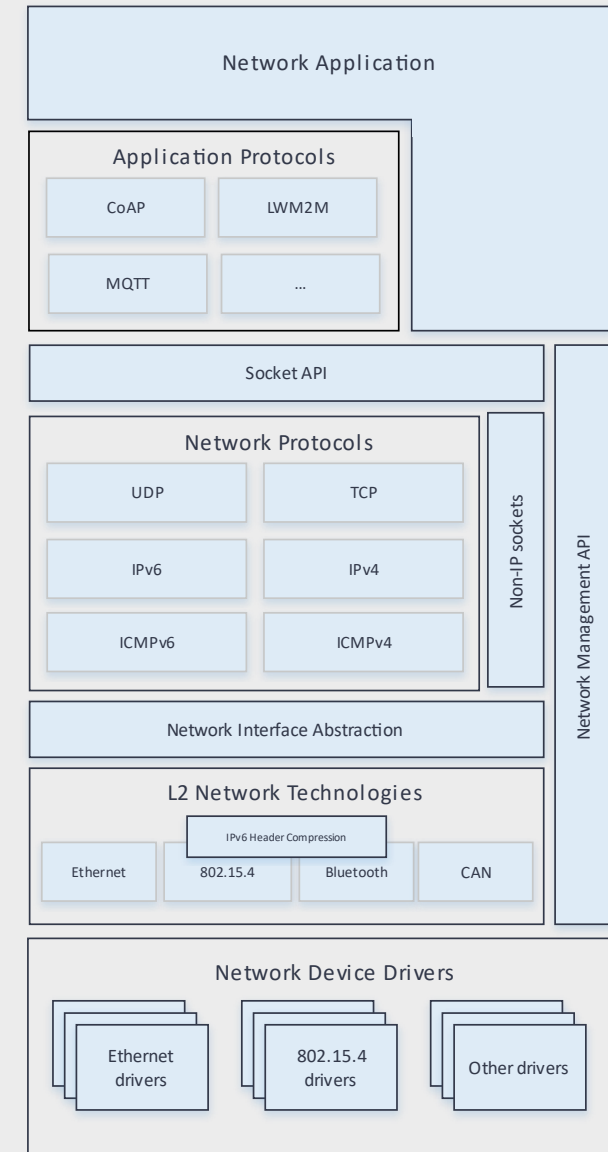
Endpoint sur « / » qui accepte
le type de commande GET

WIFI / HTTP sous Zephyr

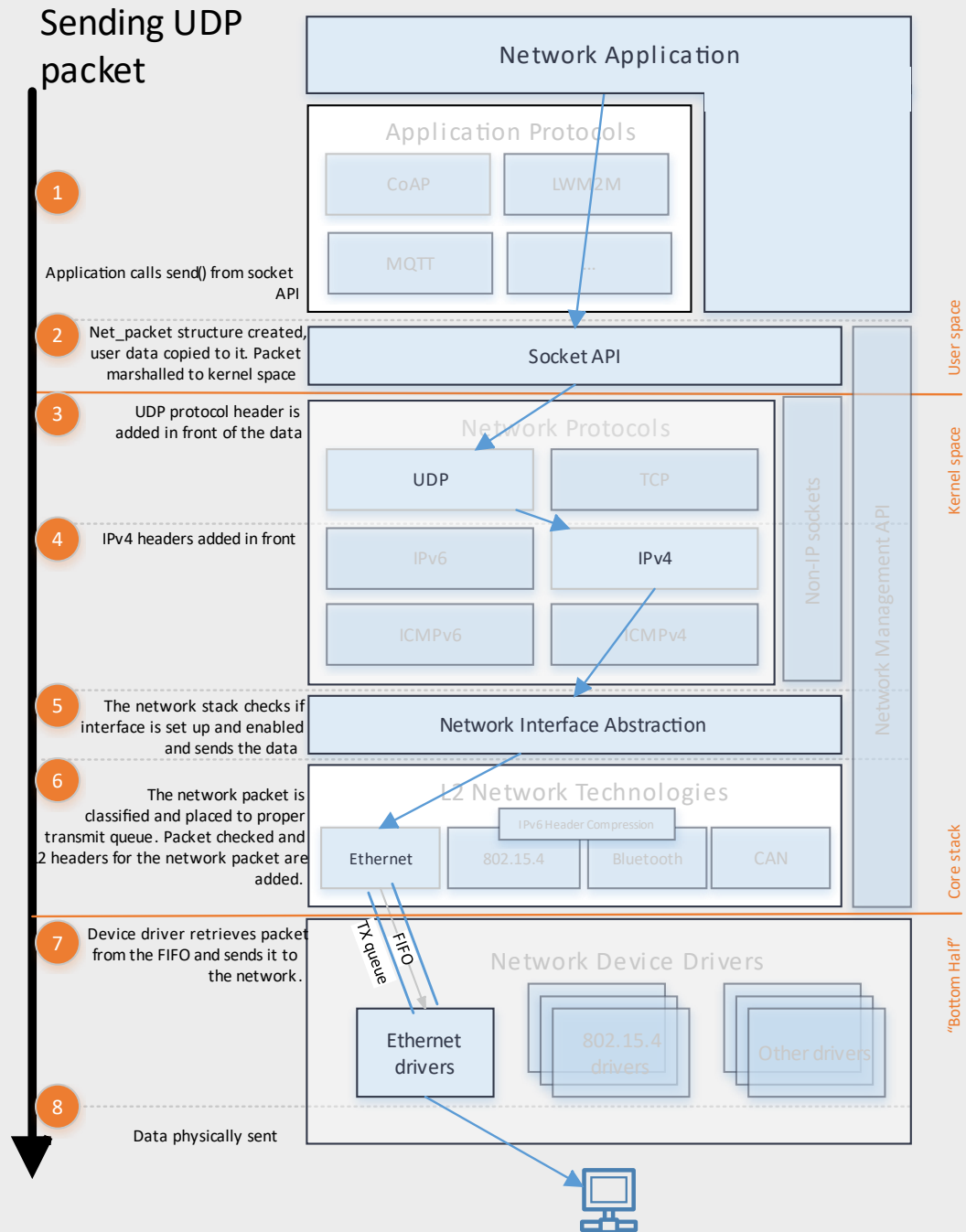
Wifi et protocole HTTP

Networking IP Stack

- IPv6
- IPv4
- UDP
- TCP
- MQTT
- DNS
- Wifi
- VLAN
- Et bien d'autres


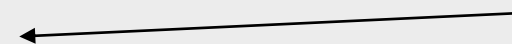


Example:

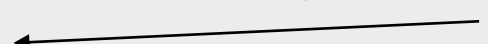


Network Management

Enregistrer et initialiser une callback pour les événements WIFI

```
net_mgmt_init_event_callback(  
    &wifi_cb,  static struct net_mgmt_event_callback wifi_cb;  
    wifi_mgmt_event_handler,  Fonction à implémenter  
    NET_EVENT_WIFI_CONNECT_RESULT | NET_EVENT_WIFI_DISCONNECT_RESULT  
);
```

Enregistrer et initialiser une callback pour les événements IP

```
net_mgmt_init_event_callback(  
    &ipv4_cb,  static struct net_mgmt_event_callback ipv4_cb;  
    wifi_mgmt_event_handler,  
    NET_EVENT_IPV4_ADDR_ADD  
);
```

Network Management

```
static void wifi_mgmt_event_handler(struct net_mgmt_event_callback *cb, uint32_t mgmt_event, struct net_if *iface)
{
    switch (mgmt_event)
    {
        case NET_EVENT_WIFI_CONNECT_RESULT:
            handle_wifi_connect_result(cb); ← Fonction à implémenter
            break;

        case NET_EVENT_WIFI_DISCONNECT_RESULT:
            handle_wifi_disconnect_result(cb); ← Fonction à implémenter
            break;

        case NET_EVENT_IPV4_ADDR_ADD:
            handle_ipv4_result(iface); ← Fonction à implémenter
            break;

        default:
            break;
    }
}
```

Gérer une connexion

Cette fonction est appelée en cas d'événement de connexion par `wifi_mgmt_event_handler`

```
static void handle_wifi_connect_result(struct net_mgmt_event_callback *cb)
{
    const struct wifi_status *status = (const struct wifi_status *)cb->info;

    if (status->status)
    {
        printk("Connection request failed (%d)\n", status->status);
    }
    else
    {
        printk("Connected\n");
        k_sem_give(&wifi_connected);
    }
}
```

← Sémaphore pour signaler
la fin de la procédure

Gérer une déconnexion

Cette fonction est appelée en cas d'événement de déconnexion par `wifi_mgmt_event_handler`

```
static void handle_wifi_disconnect_result(struct net_mgmt_event_callback *cb)
{
    const struct wifi_status *status = (const struct wifi_status *)cb->info;

    if (status->status)
    {
        printk("Disconnection request (%d)\n", status->status);
    }
    else
    {
        printk("Disconnected\n");
        k_sem_take(&wifi_connected, K_NO_WAIT);
    }
}
```


Gérer l'attribution IPv4

Cette fonction permet d'afficher les résultats de l'obtention d'adresse IP par DHCP depuis `wifi_mgmt_event_handler`

```
static void handle_ipv4_result(struct net_if *iface)
{
    int i = 0;
    for (i = 0; i < NET_IF_MAX_IPV4_ADDR; i++) {
        char buf[NET_IPV4_ADDR_LEN];
        if (iface->config.ip.ipv4->unicast[i].addr_type != NET_ADDR_DHCP) {
            continue;
        }

        printk("IPv4 address: %s\n",
               net_addr_ntop(AF_INET,
                             &iface->config.ip.ipv4->unicast[i].address.in_addr,
                             buf, sizeof(buf)));

        printk("Subnet: %s\n", net_addr_ntop(AF_INET,
                                              &iface->config.ip.ipv4->netmask,
                                              buf, sizeof(buf)));

        printk("Router: %s\n",
               net_addr_ntop(AF_INET,
                             &iface->config.ip.ipv4->gw,
                             buf, sizeof(buf)));
    }

    k_sem_give(&ipv4_address_obtained);
}
```

La `struct net_if` est définie dans `zephyr/net_if.h`
=> Network Interface

`net_addr_ntop`
Convert IP address to string form.

Sémaphore pour signaler
la fin de la procédure

Network interface (net_if)

- Permet de faire l'interface entre le « Network device driver » et le « Network Stack »

```
struct net_if {  
    struct net_if_dev *if_dev;  
    struct net_if_config config;  
    struct k_mutex lock;  
};
```

Connexion au réseau wifi

```
void wifi_connect(void)
{
    struct net_if *iface = net_if_get_default();

    struct wifi_connect_req_params wifi_params = {0};

    wifi_params.ssid = SSID;
    wifi_params.psk = PSK;
    wifi_params.ssid_length = strlen(SSID);
    wifi_params.psk_length = strlen(PSK);
    wifi_params.channel = WIFI_CHANNEL_ANY;
    wifi_params.security = WIFI_SECURITY_TYPE_PSK;
    wifi_params.band = WIFI_FREQ_BAND_2_4_GHZ;
    wifi_params.mfp = WIFI_MFP_OPTIONAL;

    printk("Connecting to SSID: %s\n", wifi_params.ssid);

    if (net_mgmt(NET_REQUEST_WIFI_CONNECT, iface, &wifi_params, sizeof(struct wifi_connect_req_params)))
    {
        printk("WiFi Connection Request Failed\n");
    }
}
```

Get the default network interface.

Structure permettant de définir les paramètres de connexion

A définir !

Afficher l'état de la connexion

```
void wifi_status(void)
{
    struct net_if *iface = net_if_get_default();

    struct wifi_iface_status status = {0};

    if (net_mgmt(NET_REQUEST_WIFI_IFACE_STATUS, iface, &status, sizeof(struct wifi_iface_status)))
    {
        printk("WiFi Status Request Failed\n");
    }

    printk("\n");

    if (status.state >= WIFI_STATE_ASSOCIATED)
    {
        printk("SSID: %-32s\n", status.ssid);
        printk("Channel: %d\n", status.channel);
        printk("RSSI: %d\n", status.rssi);
    }
}
```

Configuration

```
CONFIG_WIFI=y
CONFIG_INIT_STACKS=y
# Enable Wifi Management Interface
CONFIG_NET_L2_WIFI_MGMT=y
# Wifi requires large heap size
CONFIG_HEAP_MEM_POOL_SIZE=98304
# Add support for ethernet layer (used in wifi)
CONFIG_NET_L2_ETHERNET=y
CONFIG_ESP32_WIFI_STA_AUTO_DHCPV4=y

CONFIG_NETWORKING=y
CONFIG_NET_IPV4=y
CONFIG_NET_IPV6=n
CONFIG_NET_UDP=y
CONFIG_NET_TCP=y
CONFIG_DNS_RESOLVER=y
CONFIG_DNS_RESOLVER_AI_MAX_ENTRIES=10
CONFIG_NET_SOCKETS_POSIX_NAMES=y
CONFIG_NET_SOCKETS=y
CONFIG_HTTP_CLIENT=y
```

```
# Use DHCP for IPv4
CONFIG_NET_DHCPV4=y
CONFIG_NET_CONFIG_SETTINGS=y

CONFIG_NET_TX_STACK_SIZE=2048
CONFIG_NET_RX_STACK_SIZE=2048

CONFIG_NET_PKT_TX_COUNT=10
CONFIG_NET_PKT_RX_COUNT=10

CONFIG_NET_BUF_RX_COUNT=20
CONFIG_NET_BUF_TX_COUNT=20

CONFIG_NET_MAX_CONTEXTS=10
```

Création du socket

Socket: permet d'établir une communication entre deux processus

```
static struct addrinfo hints;  
struct addrinfo *res;  
int st, sock, ret;
```

criteria for selecting the socket address structures

ai => « address information »

AF_INET => IPv4

SOCK_STREAM => IPv4

```
hints.ai_family = AF_INET;  
hints.ai_socktype = SOCK_STREAM;  
st = getaddrinfo(HTTP_HOST, HTTP_PORT, &hints, &res);  
printf("getaddrinfo status: %d\n", st);  
if (st != 0) {  
    printf("Unable to resolve address, quitting\n");  
    return 0;  
}
```

Hostname to address resolution (DNS)

```
dump_addrinfo(res);  
sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);  
CHECK(zsock_connect(sock, res->ai_addr, res->ai_addrlen));
```

Create the socket

Connect the socket (TCP Handshake)

Requête HTTP

- Voir l'api Zephyr HTTP Request:
<https://docs.zephyrproject.org/latest/connectivity/networking/api/http.html>

Requête HTTP

```
req.method = HTTP_GET; ← Méthode HTTP_GET ou HTTP_POST
req.url = "/";
req.host = HTTP_HOST; ← Host à définir
req.protocol = "HTTP/1.1";
req.response = response_cb; ← Fonction de callback à définir
req.recv_buf = recv_buf; ← static uint8_t recv_buf[512];
req.recv_buf_len = sizeof(recv_buf);

ret = http_client_req(sock, &req, 5000, NULL); ← Envoi de la requête
zsock_close(sock);
```

Projet

1. Ajouter la fonctionnalité Wifi sur la maison connectée en Zephyr
2. Créer une API avec Python et Flask permettant de récupérer un message d'accueil et de sauvegarder les valeurs de température et d'humidité (sans base de données pour l'instant) et la déployer sur une carte Raspberry PI.
3. Récupérer le message d'accueil sur la maison connectée et l'afficher sur l'écran LCD
4. Envoyer les données de température et d'humidité à l'API

Bonus 1: Sauvegarder les données de température et d'humidité dans une base de données (MongoDB / Postgresql par exemple)

Bonus 2: Créer une interface de visualisation sur la technologie de votre choix pour visualiser les données.