



Université  
de Toulouse

## Rapport de projet Réseaux Sans Fils

Encadré par Mr. Zoubir Mammeri

Marco Regragui Martins  
Université de Toulouse

Antoine Vallat  
Université de Toulouse

Alexandre Paboeuf  
Université de Toulouse

2 Avril 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Mise en place du réseau Wi-Fi ad hoc</b>	<b>3</b>
2.1	Configuration . . . . .	3
2.2	Changement de canal dynamique . . . . .	4
<b>3</b>	<b>Mise en place du réseau Bluetooth</b>	<b>6</b>
<b>4</b>	<b>Récupération des données GPS</b>	<b>9</b>
A	Connexion à gpsd . . . . .	9
B	Lecture des données . . . . .	9
C	Récupération et stockage des données . . . . .	9
<b>5</b>	<b>Communication entre les machines</b>	<b>10</b>
5.1	Échanges Wi-Fi avec le protocole TCP . . . . .	10
A	Échanges en une session . . . . .	10
B	Échanges en plusieurs sessions . . . . .	11
5.2	Échanges Bluetooth avec le protocole RFCOMM . . . . .	13
5.3	Échange des données GPS . . . . .	14
A	Transmission via socket TCP . . . . .	14
B	Gestion des erreurs et robustesse . . . . .	15
5.4	Représentation sur une carte . . . . .	15
A	Création de la carte interactive . . . . .	15
B	Visualisation . . . . .	15
<b>6</b>	<b>Conclusion</b>	<b>17</b>

# 1 Introduction

Dans le cadre de l'UE Réseaux Sans Fils, il nous a été demandé d'établir un réseau de petite portée de type WPAN entre 3 Raspberry Pi 3. L'objectif de ce projet était de faire communiquer les Raspberry Pi en utilisant les protocoles Wi-Fi en mode ad hoc et Bluetooth afin d'échanger des informations à l'aide du protocole TCP. En plus de cela, il nous a été demandé de communiquer des données GPS à l'aide d'un capteur prévu à cet effet connecté à une des 3 Raspberry Pi et d'interpréter ces coordonnées sur une carte interactive. Ce rapport regroupe le protocole effectué pour mettre en place le réseau, les différentes communications effectuées ainsi que les résultats obtenus.

## 2 Mise en place du réseau Wi-Fi ad hoc

Contrairement au mode infrastructure du protocole Wi-Fi permettant à des machines de communiquer à l'aide d'un point d'accès, le mode ad-hoc permet une communication directe entre les appareils du réseau ce qui en fait un réseau décentralisé. La flexibilité et la mobilité de ce mode de communication sont des atouts particulièrement avantageux dans les environnements dépourvus d'infrastructure de communication pouvant jouer le rôle de point d'accès.

### 2.1 Configuration

Le programme `ad_hoc_startup.sh` rédigé en **shell** regroupe un ensemble d'instructions permettant d'établir le mode ad hoc sur une Raspberry Pi. Il prend en argument le canal du réseau ainsi que son adresse IP.

En premier temps, nous désactivons le service `NetworkManager` dont le rôle est de gérer les protocoles réseaux du système tel que Ethernet ou encore Wi-Fi. Sa désactivation est nécessaire afin de prendre le contrôle total sur la gestion du réseau Wi-Fi.

```
sudo systemctl stop NetworkManager
```

Par la suite, le Wi-Fi de la Raspberry Pi est désactivé temporairement pour que nous puissions modifier sa configuration. Une fois le Wi-Fi désactivé, nous mettons en place le mode ad hoc et définissons le nom de notre réseau. Ces étapes sont effectuées par les commandes suivantes :

```
sudo ip link set wlan0 down
sudo iwconfig wlan0 mode ad-hoc
sudo iwconfig wlan0 essid "ama"
```

Pour finir, nous configurons le canal qui représente la fréquence sur laquelle les données vont être émises par la Raspberry Pi ainsi que l'adresse IP.

```
sudo iwconfig wlan0 channel 6
sudo ifconfig wlan0 192.168.1.1 netmask 255.255.255.0 up
```

Le protocole Wi-Fi utilisé dans ce cadre communique sur la bande 2.4GHz. Cela signifie qu'il permet une communication sur 13 canaux distincts chacun espacé de 5MHz qui peuvent être consultés en utilisant cette commande :

```
iwlist wlan0 channel
```

Pour assurer un réseau local non routable sur Internet, les adresses sont choisies dans la plage 192.168.1.x dans notre cas mais il est aussi possible de choisir parmi les plages d'adresses IP privées **Classe A** (10.0.0.0 à 10.255.255.255), **Classe B** (172.16.0.0 à 172.31.255.255) ou encore **Classe C** (192.168.0.0 à 192.168.255.255).

Un masque de sous-réseau (netmask) permet de séparer une adresse IP en deux parties : la partie réseau, qui identifie le sous-réseau auquel appartient l'appareil, et la partie hôte, qui identifie les appareils individuels au sein de ce réseau. La valeur de chaque octet du masque de sous-réseau indique combien d'adresses IP peuvent être attribuées aux hôtes dans le même réseau. Un octet contenant 255 signifie que cette partie de l'adresse est fixe, tandis qu'une valeur plus faible permet à l'octet correspondant de l'adresse IP de varier.

Par exemple, avec un masque 255.255.255.0, les trois premiers octets définissent le réseau et seul le dernier octet peut changer, permettant d'attribuer des adresses allant de 1 à 254 aux hôtes du réseau.

Une fois le mode ad hoc configuré, il est possible de consulter l'état du réseau Wi-Fi via la commande `sudo iwconfig wlan0` ce qui affiche le contenu illustré sur la Figure 16.

```
paul@RPiPAUL:~/ad_hoc_raspberry $ ./ad-hoc-startup.sh 11 192.168.1.1
lo      no wireless extensions.

eth0    no wireless extensions.

wlan0   IEEE 802.11  ESSID:"ama"
        Mode:Ad-Hoc  Frequency:2.462 GHz  Cell: Not-Associated
        Tx-Power=20 dBm
        Retry short limit:7  RTS thr:off  Fragment thr:off
        Encryption key:off
        Power Management:on

Connection ad-hoc au channel 11 à l'adresse ip 192.168.1.1 effectuée
```

Figure 1: Affichage après avoir exécuté le script `ad_hoc_startup.sh`

## 2.2 Changement de canal dynamique

Tandis que la perte de données lors d'une communication réseaux filaire provient généralement d'un réseau trop congestionné, une trame perdue lors d'une communication sans fil entre deux appareils est probablement due aux interférences ambiantes. Pour pallier à cela, il existe un certain nombre de mesures visant à empêcher le bruit de perturber un signal. Dans le cadre de notre projet, nous avons opté pour un système de changement de canal dynamique.

Parmi les 13 canaux disponibles pour communiquer des données à travers le protocole Wi-Fi 2.4 GHz, la plupart se chevauchent. Des données circulant au sein de ces canaux sont donc susceptibles de subir des interférences provenant d'autres canaux Wi-Fi en plus de celles provenant du bruit. De ce fait, nous avons fait le choix d'uniquement connecter nos machines sur les canaux 1, 6 et 11 car ce sont les seuls à ne pas se chevaucher mutuellement.

En premier temps, nous avons rédigé un script en **shell** appelé `envoyerCanalOpti.sh` qui prend en argument l'adresse IP de la machine ainsi que le chemin vers le fichier **Python** détaillé plus bas. La principale fonction du programme **shell** est de trouver le canal le moins utilisé parmi ceux cités précédemment.

Pour ce faire, le programme fait appel aux commandes suivantes :

```
sortedChannels=$(sudo iwlist wlan0 scan | grep Channel: | grep -Eo '([1,6,11])' | sort | uniq -c | sort -n | awk '{print $2}')

bestChannel=$(echo "$sortedChannels" | head -1)

for ch in 1 6 11; do
    if ! echo "$sortedChannels" | grep -q "$ch"; then
        bestChannel=$ch
    fi
done
```

La commande `sudo iwlist wlan0 scan` examine les réseaux Wi-Fi à portée de l'appareil et en donne les caractéristiques. Notre programme récupère ensuite ces caractéristiques et les filtre de manière à obtenir la liste des canaux utilisés parmi 1, 6 et 11 par ordre d'apparition. Si un des canaux n'apparaît pas dans la liste car il n'est pas utilisé par les réseaux Wi-Fi aux alentours, celui-ci est alors sélectionné.

Finalement, nous vérifions si aucune communication est en cours en récupérant les données renvoyées par la commande `sudo lsof -i -P -n | grep -q <adresse_IP_de_la_machine>` qui analyse l'ensemble des fichiers ouverts sur la machine en affichant uniquement les connexions réseau ainsi que les numéros de port tout en désactivant les renommages DNS afin d'exposer les adresses IP complètes.

Dans le cas où aucune Raspberry Pi n'est impliquée dans une communication, le programme lance le programme Python `envoiChangementCanal.py` qui a pour but d'envoyer le meilleur canal aux autres machines avant d'appliquer le changement. Le programme Python s'assure que le changement n'ait pas lieu si une des deux machines n'a pas reçu le message. Si une Raspberry Pi est en pleine communication Wi-Fi à ce moment-là, le changement ne se fait pas.

La périodicité du changement de canal au sein du réseau est apportée par le logiciel **Cron** qui est chargé de lancer une tâche sur le système à un moment précis défini par l'utilisateur. Afin de configurer une action périodique à l'aide du logiciel, il suffit d'exécuter la commande `crontab -e`. Cette commande lance un éditeur de texte dans lequel il faut rajouter une instruction sous la forme :

```
* * * * * commande
```

où chaque astérisque représente un paramètre de temps selon la structure suivante :

```
min heure jour mois jour_semaine commande
```

Après avoir installé **Cron** sur chacune des Raspberry Pi, nous avons configuré l'envoi du meilleur canal sur l'une d'entre elles de manière à ce que le script `envoyerCanalOpti.sh` s'exécute toutes les 2 minutes.

```
2 * * * * /usr/bin/bash /chemin_vers/envoyerCanalOpti.sh
/chemin_vers_script_Python <adresse_IP_de_la_machine>
```

Les deux autres machines ont été configurées de sorte à ce qu'elles puissent récupérer le meilleur canal en lançant un programme **shell** appelé `recevoirChangementCanal.sh` qui prend les mêmes arguments que le script d'envoi et dont le but est de lancer le script Python `receptionChangementCanal.py` responsable d'écouter sur le port où la valeur du meilleur canal sera envoyée et d'appliquer le changement après avoir reçu la confirmation de l'émetteur que toutes les machines avaient bien reçu le canal.

```
2 * * * * /usr/bin/bash /chemin_vers/recevoirChangementCanal.sh
/chemin_vers_script_Python <adresse_IP_de_la_machine>
```

Étant donné que **Cron** ne permet pas d'afficher la sortie des programmes directement sur le terminal, les deux scripts **shell** que nous avons programmés redirigent la sortie du processus de changement de canal dans un fichier log placé dans le répertoire personnel de la Raspberry Pi. De plus, les programmes sont configurés de sorte à ce que les récepteurs aient le temps de mettre en place leur écoute sur le port avant que l'émetteur ne puisse envoyer quoi que ce soit. Notons aussi qu'il est important que les 3 appareils soient configurés sur le même fuseau horaire afin que leurs programmes se lancent au même moment.

### 3 Mise en place du réseau Bluetooth

Le Bluetooth est une technologie de communication sans fil à courte portée, utilisée pour connecter des appareils électroniques sans nécessiter de câbles. Elle fonctionne à une fréquence de 2,4 GHz et permet des connexions sécurisées et rapides avec un faible coût et une faible consommation d'énergie. Grâce à sa capacité à établir des réseaux entre plusieurs appareils, le Bluetooth facilite des échanges de données instantanés, tout en étant conçu pour minimiser les interférences et assurer une communication fiable.

Afin de configurer la connexion Bluetooth entre nos 3 appareils, nous avons utilisé l'outil en ligne de commande **bluetoothctl**, qui est le service principal permettant à une distribution Linux de gérer le matériel Bluetooth utilisé par la machine. Le processus décrit ci-dessous concerne la connexion entre 2 des 3 machines. Ces étapes ont ensuite été répliquées jusqu'à ce que toutes nos machines aient été proprement connectées entre elles.

Après avoir lancé l'outil depuis le terminal avec la commande du même nom, la première étape est d'activer l'agent Bluetooth chargé d'interagir avec les périphériques lors du jumelage de deux appareils. Suite à cela, nous garantissons que l'agent activé demeure le même pour toutes les opérations suivantes en le définissant par défaut. L'agent étant configuré, nous rendons l'un des deux appareils visible et appairable afin qu'il puisse être détecté par l'autre Raspberry Pi.

```
iban1@RPIIban:~/ad_hoc_raspberry $ bluetoothctl
Agent registered
[bluetooth]# agent on
Agent is already registered
[bluetooth]# default-agent
Default agent request successful
[bluetooth]# discoverable on
Changing discoverable on succeeded
[bluetooth]# pairable on
Changing pairable on succeeded
```

Figure 2: Processus de configuration du Bluetooth via l'outil **bluetoothctl**

La Raspberry Pi qui initie le jumelage doit ensuite scanner les alentours dans le but de retrouver l'adresse MAC de l'appareil avec lequel on cherche à se connecter avec la commande **scan**. Une adresse MAC est une adresse statique propre à un périphérique lui servant d'identifiant. C'est à l'aide de cette adresse que la connexion ainsi que la communication Bluetooth entre deux appareils sont établies. L'affichage de son adresse MAC peut directement être effectué depuis **bluetoothctl** en utilisant la commande **show**.

```
[bluetooth]# scan on
Discovery started
[CHG] Controller B8:27:EB:4E:AB:CE Discovering: yes
[NEW] Device 48:33:93:37:4D:33 48-33-93-37-4D-33
[NEW] Device 6C:4B:77:D9:F0:EA 6C-4B-77-D9-F0-EA
[NEW] Device 43:02:B3:BE:55:21 43-02-B3-BE-55-21
[NEW] Device 5A:0F:5B:59:C7:FC 5A-0F-5B-59-C7-FC
```

Figure 3: Initiation du scan des différents appareil par la Raspberry Pi

```
[bluetooth]# show
Controller B8:27:EB:4E:AB:CE (public)
  Name: RPIIban
  Alias: RPIIban
  Class: 0x002c0000
  Powered: yes
  Discoverable: yes
  DiscoverableTimeout: 0x000000b4
  Pairable: yes
  UUID: A/V Remote Control      (0000110e-0000-1000-8000-00805f9b34fb)
  UUID: Audio Source            (0000110a-0000-1000-8000-00805f9b34fb)
  UUID: PnP Information         (00001200-0000-1000-8000-00805f9b34fb)
  UUID: Audio Sink              (0000110b-0000-1000-8000-00805f9b34fb)
  UUID: Headset                  (00001108-0000-1000-8000-00805f9b34fb)
  UUID: A/V Remote Control Target (0000110c-0000-1000-8000-00805f9b34fb)
  UUID: Generic Access Profile   (00001800-0000-1000-8000-00805f9b34fb)
  UUID: Generic Attribute Profile (00001801-0000-1000-8000-00805f9b34fb)
  UUID: Device Information       (0000180a-0000-1000-8000-00805f9b34fb)
  UUID: Headset AG               (00001112-0000-1000-8000-00805f9b34fb)
  Modalias: usb:v1D68p0246d0537
  Discovering: no
  Roles: central
  Roles: peripheral
Advertising Features:
  ActiveInstances: 0x00 (0)
  SupportedInstances: 0x05 (5)
  SupportedIncludes: tx-power
  SupportedIncludes: appearance
  SupportedIncludes: local-name
```

Figure 4: Affichage des caractéristiques de l'adaptateur Bluetooth de l'appareil avec la commande `show`

Dès lors que l'adresse MAC recherchée apparaît dans la liste de scans de la Raspberry Pi, nous devons exécuter la commande

```
pair XX.XX.XX.XX.XX.XX
```

en remplaçant les X par l'adresse MAC de la Raspberry Pi avec laquelle on cherche à se connecter. Un message de confirmation d'appairage devra ensuite être validé sur les deux appareils. Après l'appairage, il est possible de définir un périphérique comme étant "de confiance". Cette configuration permet à l'appareil de s'y connecter automatiquement lors des interactions futures, sans nécessiter de nouvelle approbation ou authentification à chaque connexion.

La dernière étape est d'établir la connexion avec la commande `connect XX.XX.XX.XX.XX.XX` avant de valider un message de confirmation de connexion.

```
[bluetooth]# pair B8:27:EB:24:E8:A5
Attempting to pair with B8:27:EB:24:E8:A5
[CHG] Device B8:27:EB:24:E8:A5 Connected: yes
Request confirmation
[agent] Confirm passkey 180193 (yes/no): [CHG]
[agent] Confirm passkey 180193 (yes/no): yes
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 00001108-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 0000110a-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 0000110b-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 0000110c-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 0000110e-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 00001112-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 00001200-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 00001800-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 00001801-
[CHG] Device B8:27:EB:24:E8:A5 UUIDs: 0000180a-
[CHG] Device B8:27:EB:24:E8:A5 ServicesResolved
[CHG] Device B8:27:EB:24:E8:A5 Paired: yes
Pairing successful
```

Figure 5: Appairage des appareils avec la commande pair

```
[bluetooth]# trust B8:27:EB:24:E8:A5
[CHG] Device B8:27:EB:24:E8:A5 Trusted: yes
Changing B8:27:EB:24:E8:A5 trust succeeded
[CHG] Device 48:33:93:37:4D:33 RSSI: -52
[CHG] Device 58:50:E6:48:76:D1 RSSI: -71
[CHG] Device 67:76:34:CF:D7:C8 RSSI: -70
[DEL] Device 6E:17:EB:48:49:38 6E-17-EB-48-4
[bluetooth]# connect B8:27:EB:24:E8:A5
Attempting to connect to B8:27:EB:24:E8:A5
[DEL] Device 4B:6A:55:26:A5:1C 4B-6A-55-26-A
[CHG] Device B8:27:EB:24:E8:A5 Connected: yes
[CHG] Device 4D:2E:E2:49:19:FF RSSI: -55
[CHG] Device 4D:2E:E2:49:19:FF ManufacturerID:
[CHG] Device 4D:2E:E2:49:19:FF ManufacturerID:
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80
00
[CHG] Device 48:33:93:37:4D:33 RSSI: -69
[CHG] Device 68:0A:E2:F3:95:E1 RSSI: -90
Connection successful
```

Figure 6: Mise en confiance et connexion des deux appareils

Désormais, les deux appareils sont connectés et prêts à échanger des données en utilisant le protocole Bluetooth.

## 4 Récupération des données GPS

### A Connexion à gpsd

Afin d'exploiter les données GPS, nous avons utilisé le capteur **G MOUSE VK-162**. Son branchement USB nous permet de le connecter directement sur la Raspberry Pi, il nous fournit la latitude et la longitude en temps réel. À l'aide de la bibliothèque **gpsd-py3**, le script se connecte au démon **gpsd**, qui doit être lancé en amont.



Figure 7: Modèle du capteur GPS utilisé

### B Lecture des données

Le script `gpsEnvodata.py` exploite la fonction `gpsd.getcurrent()` pour obtenir le dernier paquet de données GPS. Ces informations sont ensuite extraites (par exemple, à l'aide de `packet.position()`) et associées à un timestamp formaté via `time.strftime()`. Les données sont capturées toutes les deux secondes (temps entre la conversion en JSON et l'envoi des données plus l'affichage), bien que cette fréquence puisse être paramétrée selon les besoins de précision ou de réactivité en temps réel. Il est à noter que, dans certains cas (par exemple, lorsque le capteur se trouve à l'intérieur d'un bâtiment), aucune donnée GPS ne sera transmise et le script affichera alors le message « aucune donnée GPS reçue ». Pour lancer ce script, il est nécessaire de spécifier le PORT (ex. : 5000) ainsi que l'adresse IP de la machine destinataire (ex. : 192.168.1.2) via la commande : `python3 gpsEnvodata.py 5000 192.168.1.2`

```
data envoyée : 2025-03-25 18:51:36, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:38, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:40, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:42, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:44, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:46, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:48, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:50, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:52, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:55, 43.5620862, 1.4703563
data envoyée : 2025-03-25 18:51:57, 43.5620862, 1.4703563
```

Figure 8: Envoi data GPS

### C Récupération et stockage des données

Sur l'appareil récepteur, le script `gps_csvData.py` se met en attente d'une connexion entrante sur le port spécifié (par exemple, 5000). Grâce à une connexion TCP, il reçoit les trames JSON envoyées par l'émetteur. Chaque trame, une fois décodée via `json.loads()`, fournit un ensemble de trois valeurs : un horodatage, la latitude et la longitude.

```

server en attente sur le port 5000.
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:31,43.5620862,1.4703563
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:33,43.5620862,1.4703563
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:35,43.5620862,1.4703563
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:37,43.5620862,1.4703563
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:39,43.5620862,1.4703563
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:41,43.5620862,1.4703563
connexionrecue de ('127.0.0.1', 35780)
data enregistrees : 2025-03-25 17:44:43,43.5620862,1.4703563

```

Figure 9: Réception data GPS

Ces données sont ensuite enregistrées dans un fichier CSV nommé `gpsdata.csv`. Ce fichier est initialisé avec un en-tête (`timestamp`, `latitude`, `longitude`) et, pour chaque trame reçue, une nouvelle ligne est ajoutée en mode « append ». Ainsi, le fichier constitue un historique des positions GPS, permettant ultérieurement de représenter graphiquement le trajet parcouru sur une carte.

1	<b>timestamp</b>	<b>latitude</b>	<b>longitude</b>
2	2025-03-25 18:51:36	43.5620862	1.4703563
3	2025-03-25 18:51:38	43.5620862	1.4703563
4	2025-03-25 18:51:40	43.5620862	1.4703563
5	2025-03-25 18:51:42	43.5620862	1.4703563

Figure 10: données csv

Le fichier CSV indique que les données sont capturées toutes les deux secondes. Dans cet exemple, les coordonnées restent identiques car le capteur est immobile. Ce fichier sert à la fois de journal de bord et de source de données pour la représentation graphique sur une carte.

## 5 Communication entre les machines

### 5.1 Échanges Wi-Fi avec le protocole TCP

#### A Échanges en une session

Le script `envoiTCP.py` permet d'envoyer un fichier à une machine cible sur un réseau en utilisant un socket TCP. Il commence par définir l'adresse IP de la machine cible et le port d'envoi.

Les ports TCP sont utilisés pour identifier des services spécifiques sur un réseau. Ils vont de 0 à 65535, et sont répartis en trois catégories principales : les ports bien connus (de 0 à 1023), les ports enregistrés (de 1024 à 49151) et les ports dynamiques ou privés (de 49152 à 65535). Il est essentiel d'éviter les ports réservés ou déjà utilisés, afin d'éviter les conflits avec d'autres services.

Ensuite, le programme crée un socket en IPv4 (AF\_INET) et en mode TCP (SOCK\_STREAM) avant d'établir une connexion avec la machine cible. Une fois connecté, le script ouvre le fichier en mode binaire, lit son contenu et l'envoie via le socket. Après l'envoi, il ferme le socket en écriture et attend un message de confirmation indiquant la bonne réception des données.

Le script `receptionTCP.py` quant à lui permet de mettre un socket TCP en écoute sur un port défini pour recevoir un fichier. Il commence par créer un socket en IPv4 (AF\_INET) et en mode TCP (SOCK\_STREAM), puis l'associe au port 6000 sur toutes les interfaces du réseau. Le serveur

est ensuite mis en écoute pour accepter une connexion entrante. Lorsqu'un client se connecte, il récupère le socket correspondant et ouvre un fichier en mode binaire pour y écrire les données reçues. Une boucle permet de lire et d'écrire les données en continu jusqu'à ce que la transmission soit terminée. Une fois le fichier complètement reçu, le serveur envoie un message de confirmation au client, affiche un message indiquant la réception du fichier, puis ferme les connexions.

```
iban1@RPIIBan:~/ad_hoc_raspberry/TCP $ python3 envoiTCP.py 192.168.1.1 programme.py
Reception ok
iban1@RPIIBan:~/ad_hoc_raspberry/TCP $ cat programme.py
:
import sys
import socket
import json

with socket.socket(socket.AF_INET , socket.SOCK_STREAM ) as serversocket :
    serversocket.setsockopt(socket.SOL_SOCKET , socket.SO_REUSEADDR , 1 )
    serversocket.bind(('',5000))
    serversocket.listen(1)
    print("test1")
    (clientsocket,adress) = serversocket.accept()
    print("test2")
    while True:
        data = clientsocket.recv(1024)
        if not data :
            break
        print(json.loads(data.decode()))
```

Figure 11: Exécution du programme `envoiTCP.py` et affichage du fichier envoyé

```
paul@RPIPAUL:~/ad_hoc_raspberry/TCP $ python3 receptionTCP.py programme.py
Fichier reçu
paul@RPIPAUL:~/ad_hoc_raspberry/TCP $ cat programme.py
:
import sys
import socket
import json

with socket.socket(socket.AF_INET , socket.SOCK_STREAM ) as serversocket :
    serversocket.setsockopt(socket.SOL_SOCKET , socket.SO_REUSEADDR , 1 )
    serversocket.bind(('',5000))
    serversocket.listen(1)
    print("test1")
    (clientsocket,adress) = serversocket.accept()
    print("test2")
    while True:
        data = clientsocket.recv(1024)
        if not data :
            break
        print(json.loads(data.decode())))
```

Figure 12: Exécution du programme `receptionTCP.py` et affichage du fichier reçu

## B Échanges en plusieurs sessions

Le script `EnvoieParSegment.py` permet l'envoi d'un fichier en plusieurs segments tout en assurant la reprise en cas d'interruption du récepteur ou l'émetteur . Il utilise un fichier d'index pour suivre la progression de l'envoi, dans le cas où un arrêt soudain (perte de connexion à cause de la distance des Raspberry Pi ou coupure du programme) ne nécessite pas forcément de recommencer depuis le début. Le fichier est d'abord découpé en segments de taille fixe définie par `taille_seg`. Ensuite, un fichier d'index (`index.txt`) est utilisé pour sauvegarder la progression de l'envoi. Si le programme s'arrête avant la fin du transfert, il peut reprendre là où il s'était arrêté au lieu de recommencer depuis le début.

```

ACK reçu pour segment 92
ACK reçu pour segment 93
ACK reçu pour segment 94
ACK reçu pour segment 95
ACK reçu pour segment 96
ACK reçu pour segment 97
Fichier envoyé
ibanl@RPIIban:~/ad_hoc_raspberry/TCP $ cat ../GPS/gps_csvData.py
import socket
import json
import time
import sys
HOST = "0.0.0.0" #écoute partout
PORT = int(sys.argv[1])
FILENAME = "gps_data.csv"

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #on établit la connection en TCP
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((HOST, PORT))
server.listen(5)
print(f"server en attente sur le port {PORT}.")

with open(FILENAME, "w") as file:
    file.write("timestamp,latitude,longitude\n")
conn, addr = server.accept()

while(True):

```

Figure 13: Envoi par segment

Lors de l'envoi, le programme établit une connexion TCP avec le récepteur et transmet les segments un par un. Après chaque envoi, il attend un accusé de réception (ACK). Si l'ACK reçu correspond au segment envoyé, l'index est mis à jour et l'envoi continue. En revanche, si l'ACK est incorrect ou absent, cela signifie que l'émetteur et le récepteur ne sont plus synchronisés. Dans ce cas, l'envoi est réinitialisé et le fichier est retransmis depuis le début. Une fois tous les segments envoyés et confirmés, un message de fin (finish) est transmis au récepteur. L'index est alors remis à zéro, et la connexion est fermée proprement. Ce système permet de garantir une transmission fiable en cas de coupure ou de perte de paquets. Grâce à l'enregistrement de l'index, l'envoi peut reprendre là où il s'était arrêté, et la vérification des ACK assure que chaque segment est bien reçu avant de continuer. Enfin, en cas de désynchronisation, le reset complet permet d'éviter toute erreur dans le fichier transmis.

```

File "/home/ibanl/ad_hoc_raspberry/TCP/EnvoiParSegment.py", line 55, in envoie
    time.sleep(0.1)
KeyboardInterrupt

ibanl@RPIIban:~/ad_hoc_raspberry/TCP $ python3 EnvoiParSegment.py 192.168.1.1 ../GPS/gps_csvData.py
ACK reçu pour segment 36
ACK reçu pour segment 37
ACK reçu pour segment 38
ACK reçu pour segment 39
ACK reçu pour segment 40
ACK reçu pour segment 41
ACK reçu pour segment 42

```

Figure 14: Interruption puis reprise avec l'index

```

File "/home/ibanl/ad_hoc_raspberry/TCP/EnvoiParSegment.py", line 45, in envoie
    ack = client.recv(1024) # On attend le ack de reçus
KeyboardInterrupt

ibanl@RPIIban:~/ad_hoc_raspberry/TCP $ python3 EnvoiParSegment.py 192.168.1.1 ../GPS/gps_csvData.py
ACK incorrect, Reset
ACK reçu pour segment 0
ACK reçu pour segment 1
ACK reçu pour segment 2
ACK reçu pour segment 3

```

Figure 15: Reset du programme en cas de désynchronisation

Le script `ReceptionSegment.py` assure la réception d'un fichier envoyé en plusieurs segments, tout en permettant la reprise en cas d'interruption. Il utilise un fichier d'index (`index_reception.txt`) pour suivre la progression de l'envoi, en cas d'arrêt soudain (perte de connexion à cause de la distance des Raspberry Pi ou coupure du programme) ne nécessite pas forcément de recommencer la réception depuis le début. Le serveur écoute sur le port 5000 et accepte une connexion entrante.

Une fois la connexion établie, il attend la réception des segments du fichier. Le fichier reçu est ouvert en mode binaire (ab - ajout binaire) pour écrire chaque segment au fur et à mesure. L'index actuel est récupéré depuis index\_reception.txt afin de savoir combien de segments ont déjà été reçus et d'envoyer le bon accusé de réception (ACK) à l'émetteur. Pour chaque segment reçu, le programme l'écrit dans le fichier et envoie un ACK correspondant à l'émetteur, avant de mettre à jour l'index. Si une désynchronisation est détectée (signalée au récepteur par l'émetteur avec un message reset ), la réception est entièrement réinitialisée : le fichier est effacé, l'index est remis à zéro et la transmission reprend depuis le début. Lorsque le message de fin (finish) est reçu, la boucle d'attente s'arrête, l'index est réinitialisé et la connexion est fermée proprement. Ce système permet de garantir une transmission fiable en cas de coupure ou de perte de paquets. Grâce à l'enregistrement de l'index, le récepteur peut reprendre l'envoi de ses ACK au bon endroit pour s'assurer de la bonne synchronisation avec l'émetteur même après un problème technique.

```

Reçu segment 90
Reçu segment 91
Reçu segment 92
Reçu segment 93
Reçu segment 94
Reçu segment 95
Reçu segment 96
Reçu segment 97
Fichier reçu
paul@RPIPAUL:~/ad_hoc_raspberry/TCP_S cat gps_data.py
import socket
import json
import time
import sys
HOST = "0.0.0.0" #écoute partout
PORT = int(sys.argv[1])
FILENAME = "gps_data.csv"

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) #on établit la connection en IPv4 (INET) et
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind((HOST, PORT))
server.listen(5)
print(f"server en attente sur le port {PORT}.")

with open(FILENAME, "w" ) as file:
    file.write("timestamp,latitude,longitude\n")
conn, addr = server.accept()

while(True):
    print(f"connexion reçue de {addr}")

```

Figure 16: Réception et affichage d'un programme par segment

## 5.2 Échanges Bluetooth avec le protocole RFCOMM

Nos scripts de communication Bluetooth sont centrés autour de sockets configurés à partir de 3 paramètres spécifiques au protocole. En effet, les paramètres AF\_BLUETOOTH, SOCK\_STREAM et RFCOMM permettent d'établir une connexion Bluetooth fiable entre les appareils.

AF\_BLUETOOTH spécifie que le socket doit utiliser la famille d'adresses Bluetooth lors de la communication des données. Tout comme lors de la transmission Wi-FI, SOCK\_STREAM définit que le type de socket utilisé est orienté flux de connexion, ce qui permet une communication bidirectionnelle fiable, autrement dit, il s'agit d'une connexion TCP traditionnelle. Enfin, le protocole RFCOMM est utilisé au niveau de la couche liaison de données pour transmettre des données de manière simple et structurée sur un lien Bluetooth. Ensemble, ces éléments permettent une communication Bluetooth tout en garantissant la fiabilité et la stabilité de la transmission des données.

L'émetteur ouvre donc un socket défini avec les paramètres décrits précédemment et se connecte à son destinataire à l'aide de son adresse MAC (contrairement à l'adresse IP qui a été utilisée lors des échanges Wi-FI). En plus de spécifier l'adresse MAC, nous devons aussi indiquer au socket le port sur lequel envoyer les données. Les ports RFCOMM, sont utilisés pour les connexions Bluetooth et leur plage est limitée de 1 à 30. Contrairement aux ports TCP, les canaux RFCOMM ne sont pas aussi largement utilisés et sont généralement attribués dynamiquement lors de l'établissement d'une connexion Bluetooth. L'utilisation de canaux déjà occupés peut provoquer des conflits et des

erreurs dans la communication. À condition que le récepteur soit en écoute sur le port, l'émetteur peut finalement procéder à l'envoi de données dans un format pouvant être encodé.

Le récepteur de la communication Bluetooth est très similaire à celui de la communication Wi-Fi avec pour unique différence le fait que son socket doive s'attacher et écouter sur son adresse MAC et le bon port RFCOMM. Contrairement aux sockets Wi-Fi qui peuvent écouter sur toutes les interfaces du réseau, les sockets Bluetooth doivent obligatoirement spécifier l'adresse MAC de l'appareil.

Dans le cadre de notre projet, la communication Bluetooth a été utilisée pour transmettre des messages d'instructions que les appareils récepteurs doivent exécuter. Les Figures 17, 18, 19 et 20 illustrent l'échange entre une Raspberry Pi et les deux autres. La Raspberry Pi émettrice transmet à l'une des autres Raspberry Pi le port sur lequel elle doit écouter, et à l'autre Raspberry Pi, le même port ainsi que l'adresse IP à laquelle elle devra envoyer des données pour une future communication Wi-Fi entre les deux machines.

```
ibanl@RPIIban:~/ad_hoc_raspberry/Bluetooth $ python3 envoiBluetooth.py 8000 192.168.1.3
Received message from sender: Bien recu
```

Figure 17: Envoi du port et de l'adresse IP au futur émetteur Wi-Fi par Bluetooth

```
paul@RPIPAUL:~/ad_hoc_raspberry/Bluetooth $ python3 recoisBluetooth.py B8:27:EB:24:E8:A5
Received message : Send coordinates to: 192.168.1.3 at port: 8000
```

Figure 18: Réception du port et de l'adresse IP par le futur émetteur

```
ibanl@RPIIban:~/ad_hoc_raspberry/Bluetooth $ python3 envoiBluetooth.py 8000 192.168.1.3
Received message from receiver: Bien recu
```

Figure 19: Envoi du port au futur récepteur Wi-Fi par Bluetooth

```
paul@RPIPAUL:~/ad_hoc_raspberry/Bluetooth $ python3 recoisBluetooth.py B8:27:EB:24:E8:A5
Received message : Listen for coordinates from port: 8000
```

Figure 20: Réception du port et de l'adresse IP par le futur récepteur

### 5.3 Échange des données GPS

Une fois les données GPS récupérées, nous cherchons à échanger ces informations avec d'autres Raspberry Pi et ainsi représenter de manière visuelle les données GPS pour faciliter l'analyse. Ici, notre solution développée s'articule autour de deux volets complémentaires : l'échange des données par socket et la représentation cartographique.

#### A Transmission via socket TCP

Les données GPS enregistrées (au format JSON pour chaque trame contenant le timestamp, la latitude et la longitude) sont envoyées d'un Raspberry Pi vers un autre via une connexion socket. Le script client (exécuté sur le Raspberry Pi émetteur) crée un socket TCP/IP, se connecte au serveur (dont l'adresse IP et le port sont passés en arguments), et envoie périodiquement les données GPS. Du côté serveur, un script dédié attend les connexions entrantes, reçoit les trames, les décode (à l'aide de la bibliothèque json), puis les enregistre dans un fichier CSV (gpsdata.csv). Ce mécanisme permet une communication fiable et en temps réel, avec une gestion de la segmentation des messages pour garantir que chaque trame est correctement reconstruite.

## B Gestion des erreurs et robustesse

Pour assurer la fiabilité de l'échange de données, le code intègre plusieurs mécanismes de gestion des erreurs. Du côté client, des blocs `try/except` permettent de capturer et de gérer les exceptions liées à la récupération des données GPS ou à l'envoi via le socket, évitant ainsi des interruptions brutales de l'application. Du côté serveur, la vérification de la taille des trames reçues et le contrôle de la validité du format JSON garantissent que seules des données cohérentes sont enregistrées. Par ailleurs, l'utilisation de l'option `SO_REUSEADDR` sur les sockets permet d'éviter les conflits d'adresse lors des redémarrages du serveur. Ces mesures assurent une robustesse accrue du système dans des environnements réels.

## 5.4 Représentation sur une carte

La visualisation des données GPS est une étape cruciale pour analyser le trajet et la répartition géographique des positions enregistrées. Nous utilisons pour cela la bibliothèque `Folium`, qui permet de créer des cartes interactives en s'appuyant sur `Leaflet.js`, ainsi que `pandas` pour manipuler facilement les données CSV.

## A Création de la carte interactive

Le script `generatemap.py` se charge de lire le fichier `gpsdata.csv` et de générer une carte interactive. Tout d'abord, les données sont chargées dans un DataFrame grâce à `pandas`. La dernière position enregistrée est utilisée pour centrer la carte (`location=mapcenter`), assurant ainsi que la zone d'intérêt est bien visible lors de l'affichage. On a également la possibilité d'avoir une vue éloignée ou rapprochée quand la carte est générée (`zoomstart=15`). Ensuite, pour chaque ligne du DataFrame, un marqueur (généralement un `CircleMarker`) est ajouté sur la carte aux coordonnées correspondantes. Chaque marqueur est configuré pour afficher un popup contenant l'horodatage, offrant ainsi une information temporelle sur la position. Ce procédé permet non seulement de visualiser chaque point de position, mais également de fournir des indices sur le moment auquel ces positions ont été enregistrées. Commande pour lancer la génération de la carte à partir des données GPS reçues : `"python3 generatemap.py"`

```
m1info20@raspberrypi:~/Desktop/projet/ad_hoc_raspberry/GPS $ python3 generate_map.py
carte générée OK
```

Figure 21: Création de la carte

## B Visualisation

Une fois la carte générée, elle est sauvegardée dans un fichier HTML (`gpsmap.html`). Ce fichier peut ensuite être ouvert dans n'importe quel navigateur web, offrant une interface interactive où l'utilisateur peut zoomer et se déplacer sur la carte. Cette visualisation facilite l'analyse d'un trajet et l'identification des périodes d'inactivité ou de mouvement en temps réel.

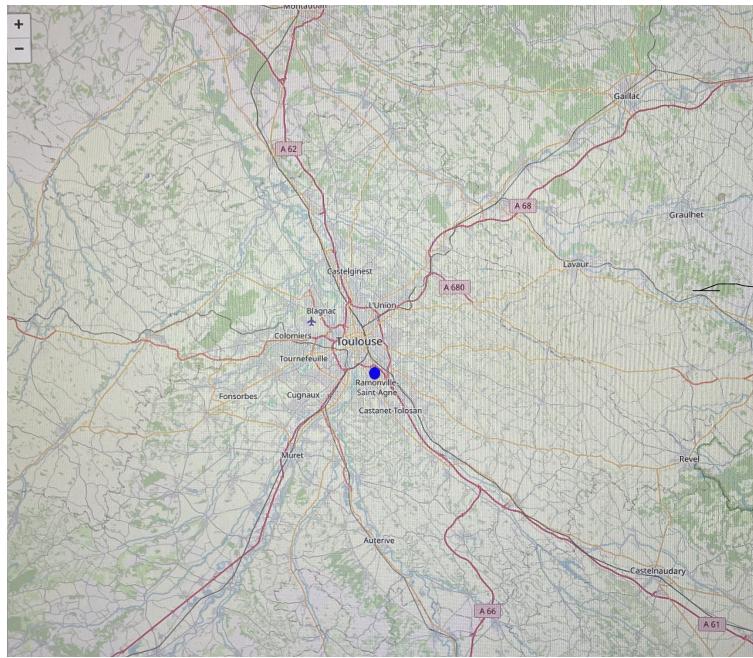


Figure 22: Affichage carte - Capteur Fixe

Malheureusement, nous ne disposons pas du matériel nécessaire pour tester notre solution en extérieur. C'est pourquoi nous proposons une visualisation basée sur des valeurs simulées, avec de fausses coordonnées GPS générant un tracé linéaire à espacements réguliers, afin d'obtenir une autre représentation visuelle.

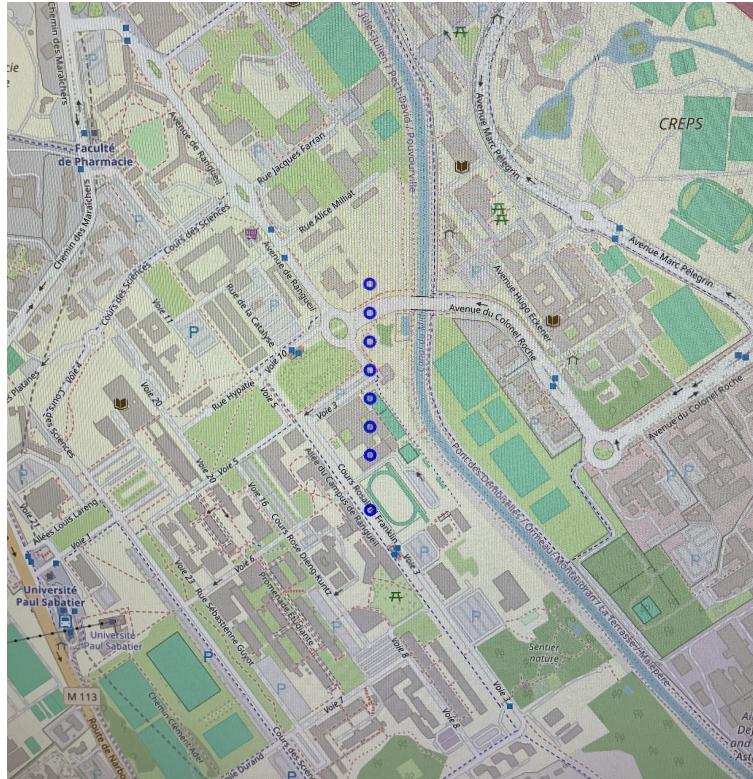


Figure 23: Affichage carte - Capteur en mouvement

## 6 Conclusion

Ce projet nous a permis d'explorer les possibilités offertes par les Raspberry Pi 3. Nous avons appris à identifier et comprendre les informations de notre réseau local, ainsi qu'à configurer différents types de communication : le Wi-Fi en mode Ad Hoc et l'Ethernet avec Bluetooth. L'utilisation du mode Ad Hoc nous a permis de mieux comprendre la configuration d'un réseau Wi-Fi et d'identifier les problèmes potentiels pouvant survenir lors de l'envoi et la réception de données entre deux appareils, tels que les interférences.

Nous nous sommes également familiarisés avec la communication entre un capteur et un Raspberry Pi grâce à l'ajout du GPS. Cette expérience nous a montré qu'il est possible de concevoir et de mettre en place des protocoles complexes au sein d'un réseau local, même en utilisant de petits appareils tels que des Raspberry Pi. En outre, les réseaux Ad Hoc et Bluetooth offrent des possibilités intéressantes, notamment dans des situations où les infrastructures traditionnelles sont limitées ou inexistantes, permettant ainsi de créer des réseaux de communication flexibles et autonomes.