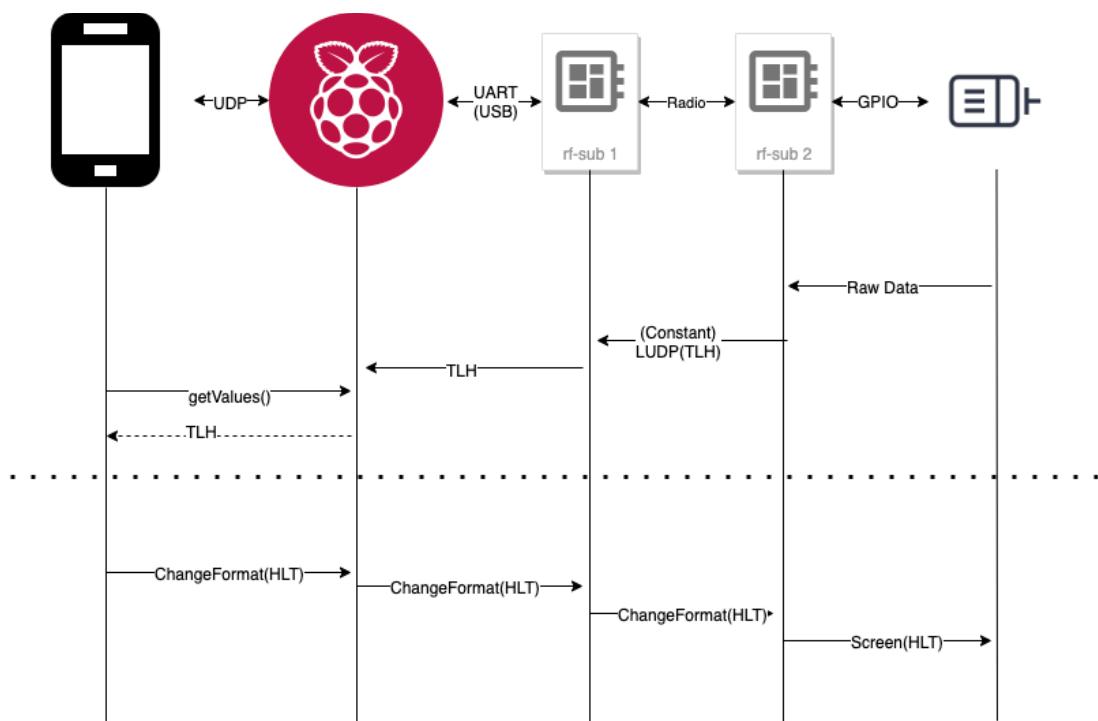


sensorwatch

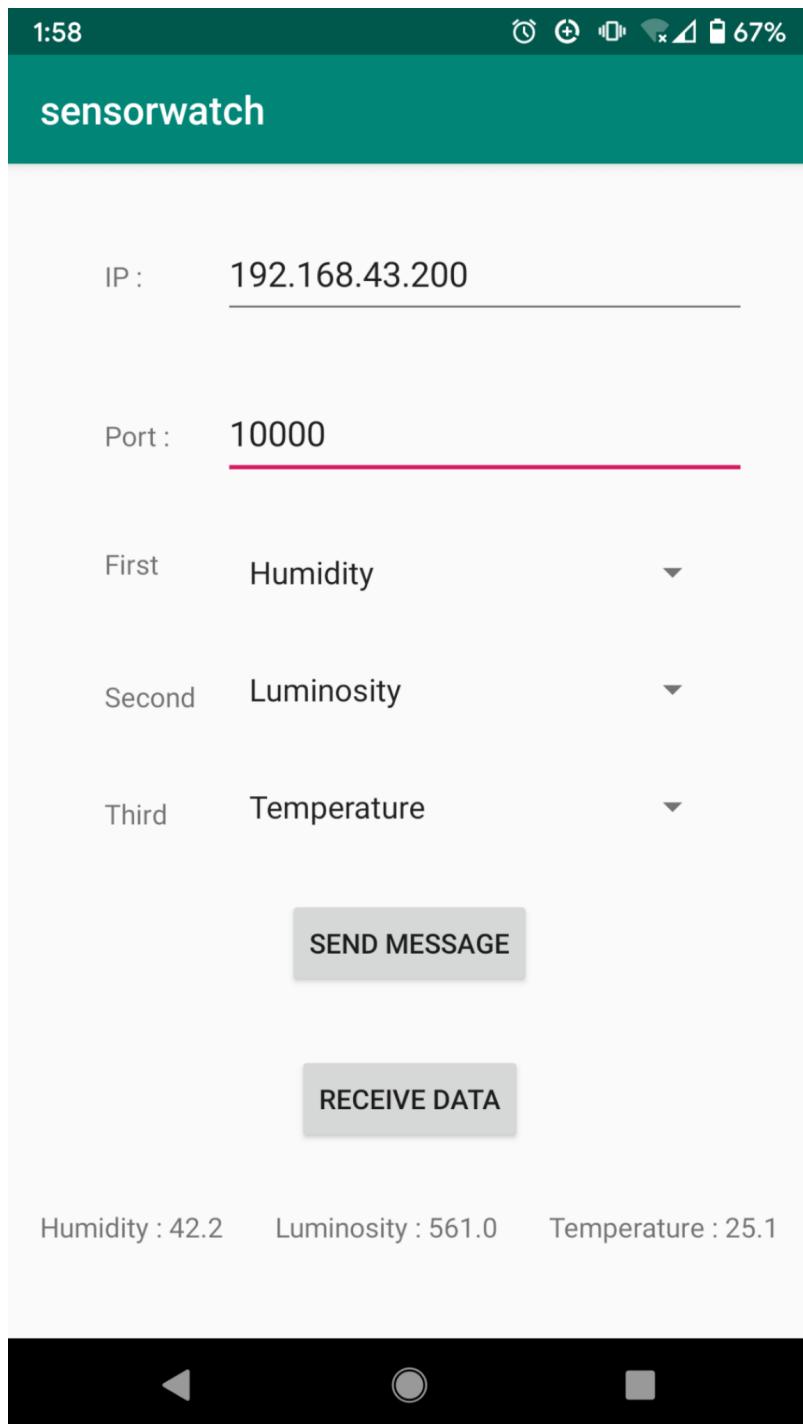
Projet IoT pour CPE Lyon

Antoine Gamain, Antoine Bouquin, Lois Motel, Lucas Philippe



Application Android

Objectifs : contrôle et affichage des données sur le smartphone



Réalisation technique

Création d'un interface graphique simple et fonctionnelle pour saisir l'adresse de destination et le port cible, ainsi que trois listes pour sélectionner l'ordre d'affichage des données sur l'écran de l'objet connecté. Un code a été mis en place pour empêcher un doublon dans

les possibilités d'affichage : TLH, LTH, HLT.

Le bouton "envoyer" récupère toutes les données saisis dans l'interface pour les envoyer à l'adresse de destination via les objets `UDPSocket` et `DatagramPacket`. Le bouton "recevoir" envoie l'instruction clé "getValues()" via le même mécanisme. Il attend ensuite une réponse de la part du serveur contenant les valeurs à afficher au sein de l'interface.

Problèmes rencontrés

Impossible de manipuler l'interface graphique depuis un thread. Solution de contournement : on attend la fin du thread pour actualiser les variables. Ce n'est clairement pas optimum mais fonctionnel, il faudrait avec plus de temps, rendre les variable observable et déclencher une méthode sur l'événement onchange.

Travail sur la réalisation du protocole :

Objectifs : Mettre en place un protocole répondant au besoin et qui est évolutif

Réalisation technique

Nous avons réalisé un protocole nommé **LUDP** pour *Light UDP*. Nos paquets sont encapsulé avec un entête de 4 octets :

1. La longueur
2. L'adresse de destination
3. L'adresse source
4. Un checksum

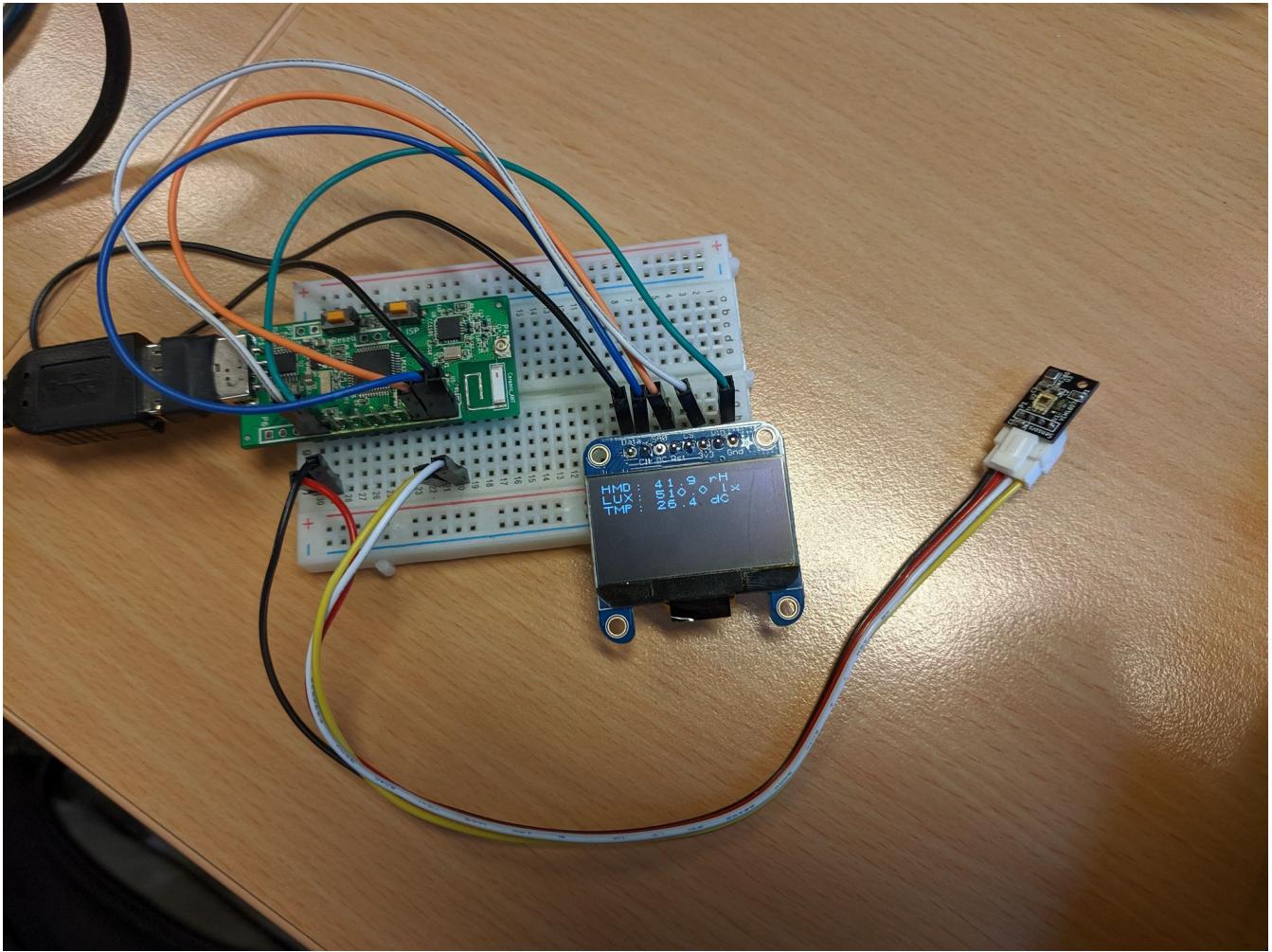
Les deux premiers octets étant définis dans le code donné, le reste de l'entête se décompose comme ceci :

n° octet	0								1							
n° bit dans l'octet	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
représentation	adresse source				type de message		divisibilité température		divisibilité lumi		divisibilité humid					
									00 : valeurs	00 : divisible par 2						
									01 : changement format	01 : divisible par 3						
									10 : demande de clé	10 : divisible par 5						
									11 : échange de clé	11 : pas divisible par aucun						

Problèmes rencontrés

Au début, 7 bits étaient prévu pour stocker la longueur du paquet. Cependant, apprenant que nous ne pouvions pas nous affranchir du code donné nous avons dû par la suite composer une entête en gardant les 2 premier octet dont l'un possédait déjà la longueur du message.

Affichage des données des capteurs



Objectifs : Récupérer et afficher la luminosité, la température et la luminosité sur l'écran.

Réalisation technique

Afin de récupérer la luminosité, on utilise le capteur **TSL256x** qui était proposé. Pour la température et la luminosité, c'est le capteur **BME280** qui a été utilisé. Quant à l'affichage de ces données, on utilise l'écran **Adafruit OLED**.

En pratique, on configure ces capteurs et l'écran via leurs fonctions de setup respectives, puis dans le `main()` on a une boucle qui va rafraîchir l'affichage en allant récupérer les valeurs des capteurs avant de les afficher sur l'écran. Cette boucle étant aussi celle qui sert à envoyer les données à la passerelle (nous y reviendrons plus tard), afin de ne pas envoyer en permanence et saturer la fréquence, nous envoyons un message toutes les secondes : il y a donc un `msleep(1000);` qui sert à "ralentir" la récupération et l'envoi de données.

Les données des capteurs n'étant pas envoyés via l'USB (l'USB étant uniquement présent pour l'alimentation du module), il fallait un moyen de prévenir l'utilisateur final si quelque chose ne fonctionnait pas. Ainsi, nous avons décidé d'utiliser l'écran. Des codes d'erreur sont définis comme constantes au début du code, et sont affichés sur l'écran ainsi que les valeurs de retour des fonctions retournant des erreurs. De plus, la LED rouge clignote afin de prévenir l'utilisateur.

Problèmes rencontrés

Le principal problème, à cette étape-là du projet restait la prise en main des différents composants et outils de développement (`lpcprog`, `minicom`, ...). Le code fourni en exemple étant simple et clair, il était très facile de le prendre en main.

Communication entre les micro-contrôleurs

Objectifs : Mettre en place l'émission et la réception en respectant le protocole sur les micro-contrôleurs.

Réalisation technique

Le composant qui transmet les données est une puce radio-fréquence dont le modèle est CC1101. Encore une fois, du code exemple était fourni, mais il était difficile à comprendre.

L'idée de base était de tout transmettre comme chaîne de caractères d'une carte à une autre. Cependant, cette approche posait trop de problèmes, nous avons décidé de faire autrement : nous forgeons un paquet via deux types de struct différents : un qui contient l'adresse source (celle du micro-contrôleur, définie au début du code), le checksum des valeurs ainsi que les valeurs dans le cas de l'envoi du micro-contrôleur des capteurs à celui du récepteur, et un contenant la source et les caractères correspondant à l'ordre souhaité dans le cas d'une requête de changement d'ordre. Les requêtes de changement d'ordre n'ont pas de checksum pour plusieurs raisons :

- La première est une raison technique : chaque élément étant un char, soit un octet, cela fait que nous avons un paquet de 6 octets à chaque fois, ce qui permet d'alléger la charge de travail au micro-contrôleur des capteurs qui doit déjà gérer ses propres envois, l'affichage, la récupération des données et le rafraîchissement de celle-ci ;
- La deuxième est plus temporelle : lors du calcul du checksum, alors qu'il fonctionnait parfaitement dans le cas de l'envoi capteurs → récepteurs, pour une raison encore inconnue à ce jour, il n'était pas correctement calculé dans l'autre sens, ce qui créait énormément d'erreurs. Après des heures passées sur ce problème sans succès, il a été jugé plus utile d'abandonner le checksum.

Une fois les données que l'on souhaite envoyer encapsulées dans un struct, avec la source et éventuellement le checksum, on copie ce struct dans un buffer que l'on envoie sur la radio. Pour ce faire, on set un flag à 1, et lorsque ce flag est set, le buffer sera encapsulé dans un header une couche en-dessous qui rajoutera la longueur du message ainsi que son adresse de destination, et sera envoyé.

Une fois envoyées, il faut recevoir et traiter ces données de l'autre côté. Une fonction, nommée handle_rf_rx_data(), sera appelée lors de la réception d'un message, via la vérification d'un flag qui sera set par un appel système déclenché à la réception.

Une vérification de l'adresse de destination sera effectuée (on ne traite pas les messages qui ne nous sont pas destinés). Ensuite, on copie les données reçues dans un struct similaire à celui utilisé de l'autre côté pour stocker les données.

Du côté des capteurs (qui recevra donc une requête de changement d'ordre d'affichage), on regarde dans notre struct les données directement, et en fonction de quelle lettre est stockée dans quel élément de notre struct, on change les variables hmdpos, tmppos et luxpos, qui codent à quelle ligne sont affichées les données sur l'écran Adafruit.

Du côté du récepteur, lorsqu'on reçoit des données, on utilise uprintf() pour les afficher sur l'UART0, ce qui aura pour effet de les envoyer en USB. Les données arriveront sur la Raspberry Pi, et lorsqu'une requête getValues() lui sera envoyée, elle renverra les dernières données stockées. En ce qui concerne le format des données stockées, il s'agit de données au format CSV très simple et léger, séparé uniquement par des points-virgules afin de pouvoir split les données pour l'affichage plus simplement. Le récepteur recevra aussi la demande de changement de format via l'UART0, où il l'encapsulera comme spécifié précédemment avant de l'envoyer.

Problèmes rencontrés :

Cette partie était particulièrement compliquée. En effet, comprendre le fonctionnement du code exemple était complexe, et des incompréhensions nous ont créé des problèmes qui nous ont beaucoup ralenti dans notre travail. Par exemple, nous étions persuadés que pour envoyer des données, il fallait rediriger l'UART0 vers une fonction qui va prendre ce qui a été envoyé dessus, les mettre dans un buffer avant de les envoyer. En réalité, cela ne marche qu'en entrée, et pour la sortie il faut passer les données directement à la fonction d'envoi.

De plus, comme spécifié précédemment, beaucoup de problèmes de transmissions de données du récepteur jusqu'aux capteurs qui n'envoyaient pas exactement ce qu'on y envoyait. Minicom ne pouvant être utilisé pour envoyer des données, il était donc difficile de

débuguer du côté du récepteur, il a donc fallu faire des concessions et trouver des méthodes différentes.