Naif Mehanna
Paul Ribeiro
projet P5

Thomas Vantroys
Alexandre Boé
Xavier Redon

Réseau de capteurs de pollution

Université de Lille 1 SCIENCES ET TECHNOLOGIES

POLYTECH LILLE

Introduction

La pollution est un sujet d'actualité de plus en plus important de nos jours. Les effets de toutes activités humaines sur l'environnement sont scrutés et la plupart des personnes et entreprises cherchent à minimiser leur impact sur l'environnement, que ce soit pour ne pas être exposé à des taux de pollution trop haut ou pour l'image de marque. Nous avons donc pensé que notre sujet de réseau de capteurs de pollution était parfaitement adapté aux problématiques actuelles et avait une réelle utilité pour beaucoup de personnes dans le monde.

Notre projet consiste en la réalisation d'un système permettant d'analyser précisément les taux de pollution en des points donnés. En effet, de nos jours, la pollution est principalement mesurée en un ou deux points d'une ville puis des calculs sont effectués en fonction des conditions météorologiques afin de connaître approximativement les taux de pollution aux autres endroits. Nous voulons donc permettre à chaque personne de pouvoir effectuer ses propres mesures de pollution et de les partager afin que tout le monde ait accès à ces informations sur la pollution actuelle en tout point d'une ville ainsi qu'à l'évolution de la pollution autour de ces points.

Nous avons pour cela mis au point une application mobile, un capteur, un serveur ainsi qu'un site web qui permettent de résoudre ces problématiques en fonctionnant ensemble pour permettre la meilleure expérience utilisateur possible.

Remerciements

Nous remercions l'ensemble de nos encadrants : Mr Redon, Mr Vantroys et Mr Boé pour l'aide qu'ils nous ont apporté durant ce semestre sur notre projet ainsi que pour avoir organisé ces séances de projet qui nous ont permis de progresser énormément sur les technologies web et mobiles et d'appliquer les connaissances que nous avons étudié en cours sur les systèmes communicants.

Nous tenons également à remercier Thierry Flamen qui nous a beaucoup aidé dans la réalisation de certaines soudure délicates notamment pour modifier notre adaptateur de capteur de pollution. Ses conseils quant aux bonnes pratiques de l'électronique et sa disponibilité nous ont été précieux.

Enfin, nous remercions Erwan Dufresne pour son aide dans l'impression de notre dernière pièce 3D à la fin de notre projet. Son aide fut très appréciable au moment où les imprimantes de l'école étaient défaillantes.

Sommaire

I) Réalisation du capteur :

Les principaux objectifs de notre capteur étaient : qu'il soit facilement transportable et donc le plus petit possible; qu'il puisse effectuer des mesures fiables de la pollution environnante et que le capteur puisse être protégé du vent car ce dernier peut affecter la précision du capteur. Enfin, il fallait que les différents composants soient bien fixés à l'intérieur de la coque pour ne pas bouger lors du transport et de l'utilisation.

Pour faire fonctionner notre capteur de pollution, tout en offrant une utilisation facilité et agréable à l'utilisateur, nous avons dû réaliser un PCB, une modélisation 3D de la coque entourant le capteur ainsi qu'un programme à intégrer dans l'ESP32 pour le faire communiquer avec le téléphone mobile.

a) Le programme ESP32 :

Notre choix de microcontrôleur s'est rapidement tourné vers l'ESP32 qui est de plus en plus utilisé de nos jours et dont le principal avantage était d'intégrer le bluetooth. Le programme à téléverser dans l'ESP32 a été réalisé au moyen du logiciel Arduino. En effet, celui-ci nous permettait d'utiliser toutes les fonctionnalités dont nous avions besoin et était assez simple d'utilisation. Le programme nous permet donc de créer l'appareil ainsi que le serveur BLE (Bluetooth Low Energy). Celui-ci signale donc la présence de notre ESP32 comme étant un appareil bluetooth auquel on peut s'appairer. De plus, des fonctions de callback permettent de se connecter et de se déconnecter du téléphone ainsi que de recevoir un caractère depuis le téléphone via bluetooth. Pour faire la communication, nous avons choisi d'envoyer le caractère arbitraire "B" depuis le téléphone pour demander à l'ESP32 d'effectuer une mesure.

Dans notre programme principal, nous attendons qu'un téléphone se connecte à l'ESP32. Puis, lorsque cela se produit, nous utilisons une boucle while pour vérifier toutes les 100ms si notre ESP32 reçoit le caractère "B" dans la variable rxValue. Cela permet de consommer moins d'énergie en mettant au repos notre microcontrôleur. De plus, nous utilisons un signal sur le pin 5 de l'ESP32 permettant de mettre le capteur de pollution en mode sleep. Nous le mettons donc à l'état bas lorsqu'aucune mesure n'est demandée puis nous passons le signal à l'état haut avant de demander une mesure. Cependant, nous avons remarqué que si nous ne laissions pas un delay entre le moment où nous sortons le capteur de pollution du mode sleep et le moment où nous effectuons la mesure, celles-ci étaient fausses. Nous avons donc décidé d'imposer un delay de 5 secondes avant de prendre la mesure. D'après nos différents test, ce temps d'attente semble être le plus adapté car il nous permet d'obtenir des bonnes valeurs à chaque mesure et semble acceptable au vu du gain sur l'autonomie de la batterie (la consommation est presque divisée par 10 entre le mode prise de mesure et le mode repos).

Lorsque le caractère "B" est reçu, on va alors rechercher la mesure prise par le capteur. Les mesures sont envoyées par liaison série suivant le protocole suivant : Baudrate: 9600, Parité: None; Stop Bits: 1; longueur du paquet : 32 octets.

Voici le paquet envoyé par le capteur pour cette communication :

Start Character 1	0x42(fixed bit)
Start Character 2	0x4d(fixed bit)
Frame Length 16-byte	Frame Length = 2*9+2 (data+check bit)
Data 1, 16-byte	concentration of PM1.0, ug/m3
Data 2, 16-byte	concentration of PM2.5, ug/m3
Data 3, 16-byte	concentration of PM10.0, ug/m3
Data 4, 16-byte	Internal test data
Data 5, 16-byte	Internal test data
Data 6, 16-byte	Internal test data
Data 7, 16-byte	the number of particulate of diameter above 0.3um in 0.1 liters of air
Data 8, 16-byte	the number of particulate of diameter above 0.5um in 0.1 liters of air
Data 9, 16-byte	the number of particulate of diameter above 1.0um in 0.1 liters of air
Data 10, 16-byte	the number of particulate of diameter above 2.5um in 0.1 liters of air
Data 11, 16-byte	the number of particulate of diameter above 5.0um in 0.1 liters of air
Data 12, 16-byte	the number of particulate of diameter above 10.0um in 0.1 liters of air
Data 13, 16-byte	Internal test data
Check Bit for Data Sum, 16-byte	Check Bit = Start Character 1 + Start Character 2 + ...all data

Pour vérifier qu'une mesure a été prise, nous allons donc scruter le moment où nous recevons le caractère 0x42 sur la liaison série. Cela signifie alors que les 31 bits qui suivent représentent la trame. Celle-ci est donc enregistrée dans notre variable buf qui est un tableau de char de longueur 31. Nous analysons ensuite celle-ci afin de vérifier que les octets reçus représentent bien le message envoyé par le capteur et qu'aucune erreur n'a été introduite durant la transmission. Nous vérifions donc que le premier octet de la trame est bien l'octet 0x4d puis nous vérifions que le checksum final est bien égal à la somme de tous les octets de la trame. Enfin, après nous être assuré que le paquet est correcte grâce aux vérifications précédentes, nous récupérons les octets 5 et 6 pour le PM1, 7 et 8 pour le PM2.5 et les octets 9 et 10 pour le PM10. Ces valeurs, une fois récupérées doivent être converties en string afin de pouvoir les envoyer au téléphone qui se chargera de les enregistrer et de les afficher à l'utilisateur.

Le programme inclut également l’allumage et l’extinction des leds permettant d’indiquer à l’utilisateur si le capteur est connecté ou non au téléphone et si celui ci est en train d’effectuer une mesure.

b) Carte éléctronique :

Pour réaliser notre carte électronique, nous avons beaucoup hésité sur le choix du logiciel que nous allions utiliser afin de nous faciliter la tâche. Nous pensions trouver les footprints de nos composants sur au moins l'un des logiciels mais au final, celles-ci n'étaient jamais disponibles. Nous avons donc décidé d'utiliser Altium car nous avions déjà eu un cours sur la réalisation des PCB avec ce logiciel.

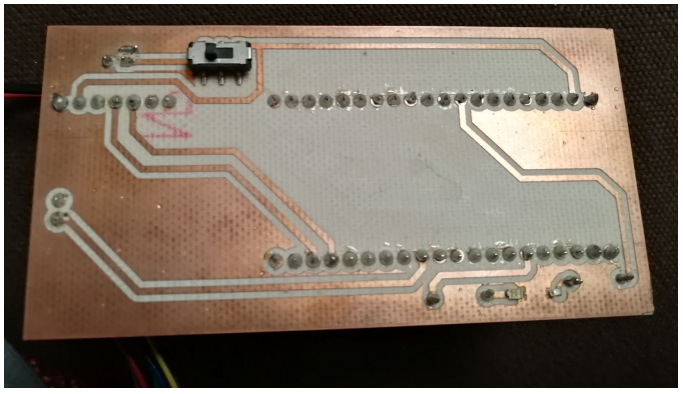
Le premier travail fut donc de réaliser les footprints et schematic de chacun de nos composants. Le plus long étant de prendre en main le logiciel et de comprendre comment réaliser au mieux ces fichiers au moyen de tutoriels vidéo sur internet. L'empreinte de l'ESP32 était la plus simple à réaliser car il suffisait de fixer la taille de la grille à 1,27mm afin de pouvoir poser chaque pastille espacée de 2.54mm. Notre ESP32 étant composé de 20 pastilles de chaque côté. Pour l'adaptateur du capteur de pollution, cela fut un peu plus complexe car nous ne possédions pas de datasheet fiable. La datasheet portait à chaque fois sur le capteur en lui-même et non pas sur l'adaptateur du capteur que nous devions souder sur la carte. Celui-ci est composé de 6 pins espacés de 2.54mm ainsi que de 2 pins représentant le reset et le set qui, eux, furent difficile à positionner. En effet, ceux ci n'étaient pas du tout alignés avec le reste des pins et nous avons donc dû mesurer à l'aide d'un pied à coulisse (pour plus de précision) les espacements entre ces deux pins et le reste des pins afin de réaliser la footprint. De plus, les pins de set et reset étaient positionnés dans le sens opposé au reste des pins et ce positionnement ne nous a donc pas arrangé pour le fixer sur notre PCB. Nous avons donc décidé d'inverser le sens des pins sur l'adaptateur du capteur de pollution pour nous faciliter la tâche.

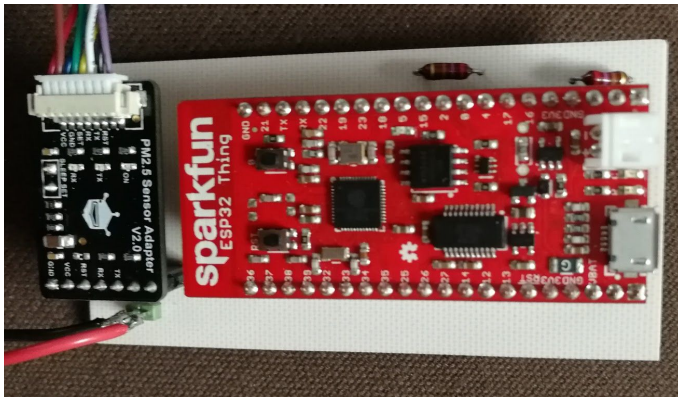
Nous avons également dû faire attention à la largeur des pastilles sur nos footprints. En effet, celles ci étaient plus large que les pastilles habituelles dû au fait que les pins utilisés pour souder nos composants étaient de forme carré de 0.7mm de côté. Cela nous obligeait donc à utiliser des pastilles de 1mm de diamètre. Enfin, pour interrupteur, nous avons dû utiliser des pastilles CMS qui se positionnent de la même façon que les pastilles traversantes mais en modifiant par la suite leur configuration.

Une fois toutes les footprints réalisées, nous avons alors pu passer à la réalisation de notre PCB. Nous avons alors positionné l'ESP32, l'adaptateur du capteur de pollution, l'interrupteur ainsi que 2 leds (une rouge et une jaune) accompagnées de leur résistance et dont les footprints étaient déjà disponibles. Nous avons alors pu reprendre les étapes que nous avions suivies en troisième année d'IMA afin de router notre carte et de finaliser le PCB pour enfin pouvoir l'envoyer à la gravure.

Pour alimenter notre PCB, nous avons décidé de ne pas utiliser la connexion standard offerte par la batterie (prise USB) et de connecter 2 fils directement sur la carte réalisée par le fabricant pour contrôler la batterie. Cette décision a été prise afin de pouvoir miniaturiser notre capteur. En effet, cela nous permet de gagner 4 à 5 cm de longueur sur le capteur car nous n'avons plus besoin de câble USB pour alimenter la carte avec la batterie. Ces deux fils ont été soudés sur la carte au moyen de pins car nous avons utilisé un fil épais pour transporter le courant jusqu'à la carte. Un interrupteur permet par la suite de séparer le reste de la carte de la batterie. En effet, cela nous permet de pouvoir téléverser un nouveau programme dans l'ESP32 facilement sans risquer de poser problème entre l'alimentation par la batterie et le câble usb du PC. Nous avions également positionné des headers sur la carte afin de ne pas souder directement l'ESP32 et l'adaptateur du capteur de pollution sur le PCB. Cependant, les pins utilisés sur l'ESP32 étant assez gros, il n'est donc pas facile de le retirer et cela ne peut donc pas être fait trop fréquemment sans risquer de l'endommager. Notre interrupteur nous permet donc de téléverser nos programmes sans risque d'endommager les pins de l'ESP32 en le retirant.

Lorsque notre capteur est en marche, une lumière bleue apparait au niveau de la batterie tandis que la led rouge s'allumera afin d'indiquer à l'utilisateur que le capteur est connecté en bluetooth. De plus, la led jaune est allumée lorsqu'une mesure est en train d'être prise. Comme ces mesures prennent 5 secondes, nous souhaitions indiquer à l'utilisateur que la demande de mesure a bien été prise en compte et que celle-ci est en cours.





c) La coque :

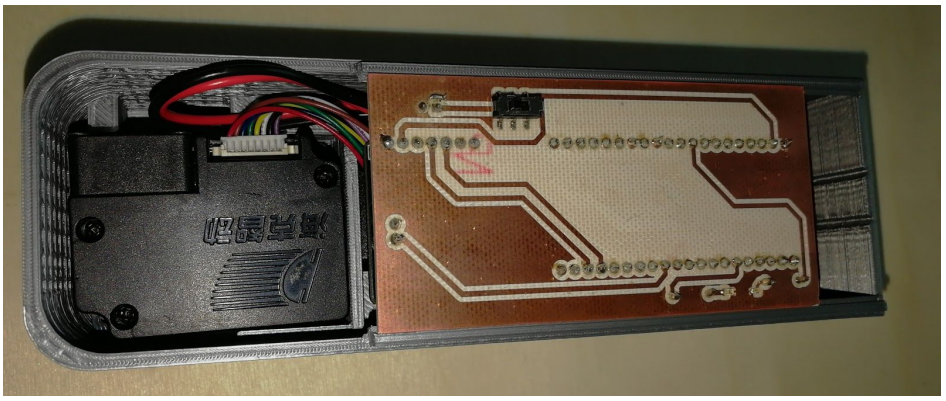
Pour réaliser la coque, nous avons tout d'abord dessiné des schémas à la main représentant le capteur que nous souhaitions avec les dimensions de celui-ci. Nous avons hésité entre un capteur de forme ronde ou rectangulaire mais nous nous sommes finalement décidés pour une forme rectangulaire car le gain de place était plus important. Nous avons donc simplement arrondi les bords afin d'améliorer un peu le design de l'objet. Une chambre d'air a été créée autour du capteur de pollution afin de répondre à notre cahier des charges. Les trous dans les parois qui entourent le capteur permettent de laisser passer l'air à analyser tout en évitant d'avoir des trop grands afflux d'air qui peuvent perturber le capteur. De plus, le capteur est maintenu en position de chaque côté par des petits supports de plastique afin de le centrer et un dernier support permet de le maintenir en hauteur pour que le capteur soit bien au centre de la chambre d'air. Enfin, un trou a été laissé sur le côté afin de laisser passer les câbles du capteur jusqu'au PCB. La batterie est, elle, maintenue des deux côtés afin de ne pas bouger et des trous ont été dessinés afin de pouvoir : appuyer sur le bouton marche/arrêt et de pouvoir brancher le câble de rechargement de la batterie sur le port microUSB de la batterie. Des tranchées ont également été tracées dans la partie interne de la coque afin de pouvoir encastrer les fils reliant la batterie au PCB. Cela permet de pouvoir coller la batterie le plus possible à la paroi et de ne pas endommager les fils en les compressant.

Un emplacement a également été prévu pour venir placer le PCB au-dessus de la batterie sans les coller. Le PCB est alors maintenu bien en place par une petite bande en plastique qui ressort de chaque côté. La partie supérieure de la coque est également perforée pour laisser passer l'air et permettre de voir les LEDs s'éclairer pour signaler que le capteur est connecté et si il est en train de prendre une mesure. Cette partie supérieure de la coque vient ensuite se refermer sur le PCB et le bloque totalement afin que celui-ci ne bouge plus. Tous les éléments sont donc bien maintenus dans la coque.

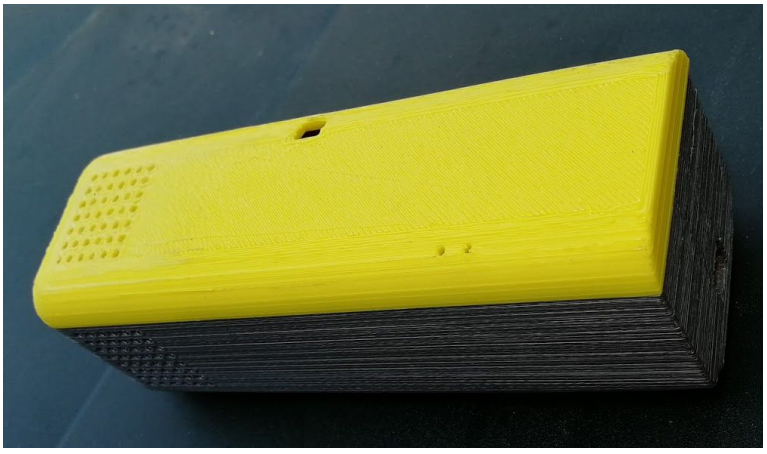
Nous avons réalisé ces pièces sur le logiciel de CAO Freecad. Notre choix s’est porté sur ce logiciel car nous l’avions déjà utilisé l'année dernière et que celui ci est libre et gratuit. Après cela, nous avons pu imprimer les pièces sur les imprimantes 3D disponibles à l'école en utilisant le logiciel Cura et nous avons utilisé du support lors de l'impression afin d'être sûr que toutes les parties soient bien réalisées correctement. En effet, les rebords pour poser le PCB par exemple, se trouvent sans aucune matière plastique en dessous et nécessitent donc un support. Ce projet fut également pour nous l’occasion de valider notre formation imprimante 3D au fabricarium certifiant que nous savons nous servir de ce matériel.

La dernière partie du travail consistait alors à enlever le support des pièces imprimées et faire les dernières modifications nécessaires pour terminer notre coque. Les trous que nous avions percé pour laisser passer l’air dans la coque étaient en effet parfois obstrués dû à l’impression avec support. Il nous a donc fallu utiliser une vrille pour faire ces finitions.

Voici donc la visualisation de notre capteur en fin de projet :



Capteur ouvert



Capteur fermé

d) Vérification de la fiabilité du capteur

Afin de vérifier que notre capteur fonctionnait correctement, nous avons voulu le tester en situation réelle. Nous nous sommes donc promenés en ville avec le capteur dans notre poche ou sur le vélo et avons effectué de multiples relevés afin d'obtenir quelques points de mesures à afficher sur le site et l'application mobile. De plus, nous avons effectué des relevés aux côtés des stations de mesures de l'ATMO qui est un organisme d'observation de la qualité de l'air et qui fournit chaque jour des informations concernant ses mesures.

Nous avons donc effectué nos relevés durant 1h juste à côté de chacune des deux stations et avons effectué une moyenne sur nos relevés afin de pouvoir les comparer :

Station	heure	PM10 (ug/m3)	PM2.5 (ug/m3)	PM1 (ug/m3)
Leeds	10h - 11h	25.46	20.76	11.57
Fives	11h20 - 12h20	28.08	22.57	12.11

valeurs relevées à partir de notre capteur

Plus tard dans l'après midi, nous avons alors pu comparer nos relevés avec les valeurs théoriques disponibles sur le site de l'ATMO qui étaient alors :

Station	heure	PM2.5 (ug/m3)	PM10 (ug/m3)
Fives	10h	20.8	29.9
Fives	11h	24.7	30.1
Fives	12h	23.7	27.4

Station	heure	PM2.5 (ug/m3)
Leeds	10h	18.1
Leeds	11h	21.4

On peut donc voir que pour les polluants PM2.5 et PM10 sur les deux stations, nous semblons être proche des valeurs relevées par l'ATMO. En effet, nous sommes à chaque fois dans l'intervalle d'erreur de +/- 15% donné par la datasheet de notre capteur.

Le résultat que nous avons obtenu pour le capteur convient donc au cahier des charges. En effet, il aurait été difficile de le miniaturiser davantage. Il aurait fallu une batterie très plate et souder nos composants directement sur le PCB sans header. Néanmoins, nous pourrions améliorer ce dernier en réalisant les dernières modifications que nous avons apportées manuellement (blocage de la batterie, augmentation de la hauteur du trou pour le câble de rechargement via microUSB) directement depuis l'imprimante 3D. De plus, d'après les vérifications que nous avons effectuées sur le terrain, notre capteur semble être fiable. Nous avons en effet, pu vérifier que la précision inscrite sur la datasheet de +/- 15% semblait correspondre. Enfin, nous pouvons dire que l'autonomie de notre capteur est satisfaisante également. En effet, nous n'avons rechargé notre batterie qu'une seule fois durant toute la durée de notre projet malgré les nombreux tests effectués. Nous pensons donc que celle ci est largement suffisante pour une utilisation journalière par un utilisateur.

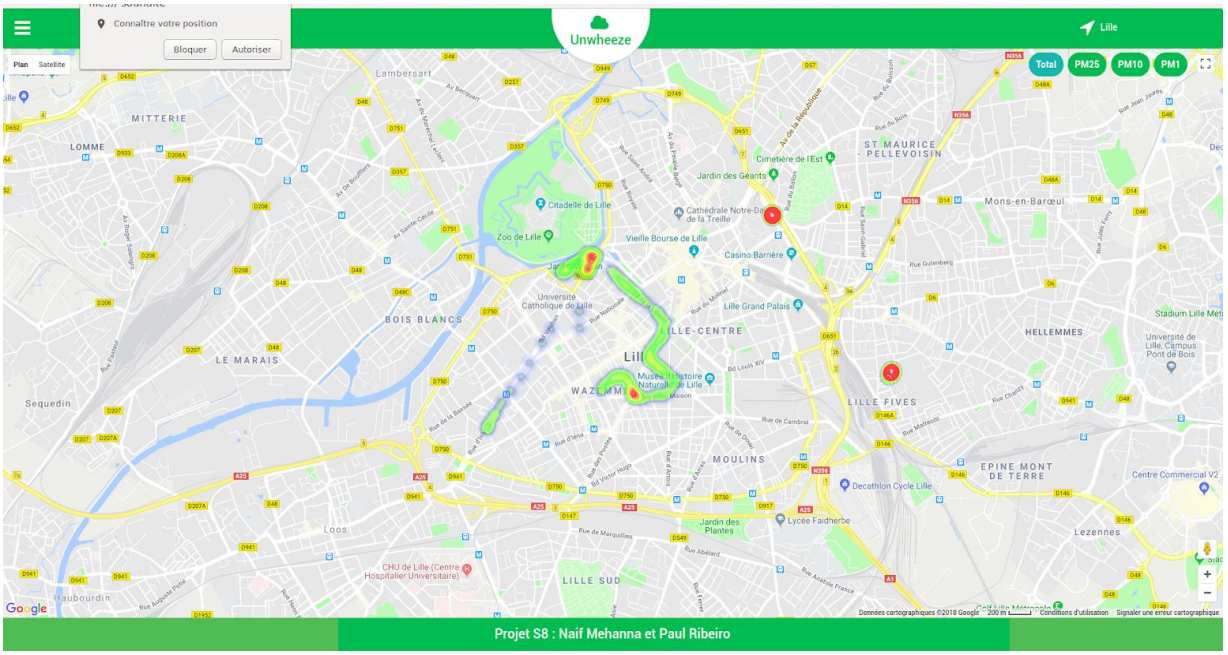
II) Le site web :

a) Présentation

L'objectif de ce site web était de pouvoir permettre à l'utilisateur de visualiser les données de pollution relevées en différents points géographiques ainsi que d'en évaluer l'évolution dans le temps au moyen de graphiques.

Le site web a été réalisé en grande partie à l'aide de javascript et de l'API de google maps qui offre énormément de fonctionnalités de localisation et d'affichage géographique très intéressantes pour notre projet. Nous utilisons également Semantic UI qui est un framework de design web au même titre que Bootstrap ainsi que Highchart pour l'affichage des graphiques. Les XHTMLHttpRequest de javascript permettent de communiquer avec l'API REST de notre serveur ainsi que L'API de google maps pour obtenir toutes les informations lors du chargement du site tandis que les websockets nous permettent de mettre à jour en temps réel les nouvelles mesures de pollution sur la carte. En effet, si nous ajoutons une donnée dans la base de données, on peut voir que le point de mesure apparaît sur la carte instantanément sans recharger la page.

Sur notre page web, chaque mesure est symbolisée par un rond de couleur allant du vert au rouge suivant le taux de pollution mesuré par le capteur. Lors de l'arrivée d'un utilisateur sur le site, le navigateur demande si il accepte de divulguer ses informations sur sa position géographique via une fenêtre en haut à gauche de l'écran. Si celui-ci accepte, la carte zoomera alors sur sa position actuelle afin de pouvoir afficher les mesures de pollution effectuées autour de lui. Dans le cas contraire, nous avons décidé de faire démarrer la carte sur la ville de Lille. L'utilisateur pourra alors se déplacer librement sur la carte afin de trouver les informations sur les taux de pollution aux endroits qu'il recherche. La localisation de la personne (ou la localisation par défaut si l'utilisateur n'autorise pas la géolocalisation) est affichée en haut à droit de la page. L'utilisateur peut alors simplement cliquer sur cette information pour centrer la carte autour de sa position. L'utilisateur étant représenté par un marker rouge sur la carte.



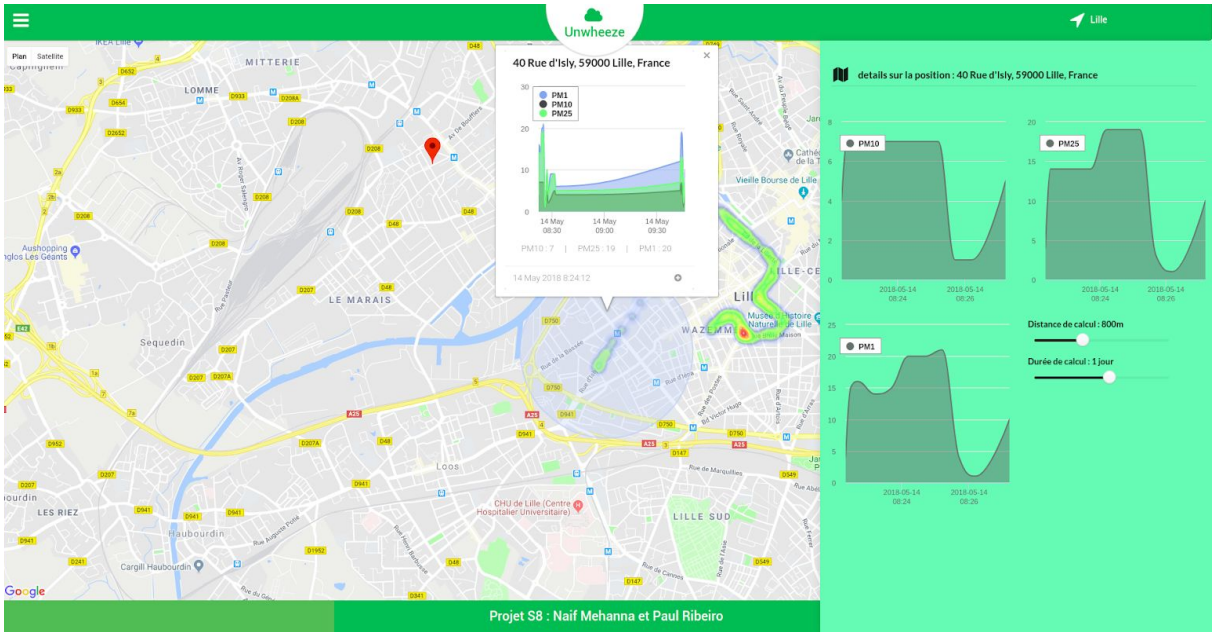
b) Fonctionnalités du site

Les endroits où une mesure a été effectuée sont affichés sur une heatmap qui nous donne sans cliquer une indication de la pollution que l'on trouve aux différents endroits de la ville. Pour cette heatmap, nous aurions pu utiliser la fonctionnalité offerte de base par google maps. Cependant, après avoir implémenté celle-ci, nous nous sommes rendu compte que la heatmap de googlemaps prenait en compte la densité des points. En d'autres termes, si nous réalisions 100 mesures d'un taux très faible de pollution en un même point, celui-ci apparaissait en rouge sur la heatmap dû au fait que les valeurs de pollution mesurée en un point s'additionnent. Ce résultat n'étant absolument pas convenable par rapport à ce que nous souhaitions obtenir. Nous avons donc trouvé heatmap.js sur le web qui est une heatmap fonctionnant sous forme d'interpolation. La densité des points n'est donc plus prise en compte et seul le poids donné à chaque point permet de modifier la couleur d'apparence sur la carte. Heatmap.js étant compatible avec la carte offerte par maps. Nous avons également ajouté 4 boutons en haut à droite de la page pour permettre à l'utilisateur de choisir le type de polluant avec lequel il souhaite que la heatmap soit mise à jour. Le bouton le plus à gauche représentant la moyenne des 3 polluants (Il représente la heatmap affichée par défaut lors de l'arrivée de l'utilisateur sur le site). Les 3 autres boutons permettent de mettre à jour la heatmap en fonction d'un polluant en particulier. Le type de heatmap active est alors indiqué par la couleur d'un des 4 boutons qui devient alors plus foncé que les autres.

On peut également obtenir une visualisation graphique des mesures de polluant. Celle-ci est disponible en cliquant sur un des marker de la heatmap. Une fenêtre s'ouvre alors au-dessus du point pour indiquer l'adresse du point cliqué (celle-ci est récupérée à l'aide d'une requête à l'API de google qui converti le point exprimé en latitude/longitude, en adresse réelle. Le graphique contient les évolutions au cours du temps des 3 polluants. L'abscisse représentant la date à laquelle la mesure a été effectuée et l'ordonnée représentant le taux de PM10, PM2.5 ou PM1 mesuré à cet instant en µg/m3. Une légende est également présente en haut à gauche du graphique afin d'expliquer à l'utilisateur quelle courbe correspond à quel polluant. Enfin, en dessous du graphique, on retrouve les informations relatives à la mesure effectuée en ce point précis de la carte. On peut alors trouver le taux de pollution sur chaque polluant ainsi que l'heure exacte de la mesure. De plus, pour faciliter l'expérience utilisateur, nous avons également implémenté une fonctionnalité qui n'oblige pas l'utilisateur à cliquer précisément sur un marker. En effet, si l'utilisateur clic simplement sur un point de la carte, la fenêtre correspondant au point le plus proche s'ouvrira. Le problème de cette fonctionnalité était de permettre à l'utilisateur de se déplacer sur la carte (en déplaçant la souris en restant appuyer) sans déclencher l'ouverture de la fenêtre du point le plus proche (simple click). Un événement click représentant un événement mouseDown, suivi d'un mouseUp sans mouseMove entre les deux.

L'utilisateur peut également obtenir davantage d'informations autour de ce point de mesure. En effet, il lui suffit de cliquer sur le petit "+" en bas à droite de la fenêtre du marker pour ouvrir un volet sur le côté droit de la page. Celui-ci est composé de 3 graphiques représentant chaque polluant ainsi que de 2 curseurs permettant à l'utilisateur de personnaliser son analyse de l'évolution des polluants au cours du temps. Les 3 graphiques permettent d'obtenir une meilleure précision sur les mesures car l'échelle des ordonnées sera alors adaptée à chaque polluant.

Les curseurs permettent à l'utilisateur de modifier l'intervalle de temps au cours duquel les données de pollution vont être prisent en compte dans l'affichage du graphique ainsi que l'intervalle de distance jouant sur cette affichage. L'utilisateur peut alors choisir le nombre d'heures durant lesquelles il souhaite que l'analyse soit réalisée (par pas de 1h allant de 1h à 24h puis par pas de 1 jour allant de 1 à 3 jours). L'utilisateur peut également sélectionner dans quel périmètre il souhaite prendre en compte les données relevées autour du point sélectionné. Ce périmètre est représenté par un cercle bleu qui s'affiche lorsque l'on maintient enfoncé le curseur de distance afin de mieux visualiser la modification en cours. Le périmètre autour du point sélectionné peut être modifié par pas de 100m de rayon allant de 0 à 3km. On peut alors voir sur cette capture d'écran que le graphique est modifié en temps réel lorsque l'on actionne l'un des deux curseurs.



c) Résultats et améliorations possibles

Nous sommes donc satisfaits du résultat obtenu sur le site web car toutes les fonctionnalités recherchées initialement ont été atteintes. Cependant, certains points auraient pu être améliorés avec un peu de temps supplémentaire. En effet, nous aurions pu gérer le problème de localisation inaccessible sur le site web lorsque celui-ci est dans sa version en ligne car celui-ci ne peut pas communiquer en HTTPS. Les fonctionnalités de géolocalisation sont donc seulement accessibles lorsque le site est dans sa version locale.

De plus, nous aurions pu implémenter un système permettant de trouver le meilleur chemin pour se rendre d'un point A à un point B en évitant au maximum la pollution. En effet, nous avons préparé l'interface permettant à l'utilisateur de trouver le meilleur itinéraire et nous avons tenté d'afficher 2 chemins (le premier étant le plus court indiqué par google maps et le deuxième étant celui que nous calculons pour éviter au maximum la pollution). Cette interface est disponible en cliquant sur le bouton en haut à gauche mais nous n'avons implémenté que le plus court trajet. Le trajet le moins pollué étant un peu plus compliqué à mettre en place car nous ne pouvons pas spécifier à l'API google maps les points que l'on veut éviter. Nous devons donc spécifier les points par lesquels nous souhaitons passer ou récupérer tous les chemins possibles et sélectionner le chemin sur lequel nous sommes assez éloignés des forts taux de pollution. Nous avons tenté d'implémenter la première solution (fichier : findBestPath.js) en recherchant le meilleur trajet au moyen d'une interpolation sur une matrice représentant l'espace entre notre point de départ et notre point d'arrivée. Mais lorsque notre chemin est défini, les requêtes demandées à google semblent être trop longues car les points spécifiés peuvent se trouver en plein milieu d'un champ par exemple. Cette fonctionnalité n'est donc pas fonctionnelle.

Enfin, pour une utilisation plus grande échelle du projet, nous aurions également pu implémenter un système de connexion pour chaque utilisateur afin de personnaliser l'interface et les fonctionnalités en fonction du client. Mais cela ne nous semblait pas nécessaire à cette échelle pour notre projet.

II) Réalisation du backend :

Nous avons choisi d'utiliser JEE afin de réaliser notre back-end, en raison notamment du fait qu'il s'agit du langage le plus utilisé actuellement dans le milieu professionnel pour la réalisation des services côté serveur. Notre choix s'est également porté au sein du framework Jersey, basé sur JAX-RS, en raison de sa grande facilité d'utilisation, mais aussi de la flexibilité et la puissance de ses composants.

Avant de passer à l'écriture du code, nous avons dû mettre en place plusieurs types d'architectures, peser les pour et les contres de chacune d'entre elles, et ainsi sélectionner la plus viable pour notre projet.
Ainsi, nous étions tout d'abord partis sur une API REST couplée à un serveur websocket et aux clients par le biais d'un message broker. Nous avions proposés cette solution car nous pensions qu'il aurait été utile de pouvoir cacher les données et de les délivrer dès que possible en utilisant un message broker. Cette solution n'est cependant pas mauvaise et aurait pu fonctionner de la même manière que notre architecture actuelle, cependant, il s'agit d'une solution plus adaptée à des projets de bien plus grosse envergure, qui ne tolèrent aucune perte d'information.
Nous avons donc choisis une solution qui est parfaitement adaptée à notre projet, et qui est réalisable dans la durée impartie, décrite par la suite.

a) Choix de la base de donnée

Nous avions le choix entre différents types de bases de données, de type SQL classiques, ou NoSQL. Ayant de l'expérience dans l'utilisation de ces deux types de bases de données, nous avions décidés qu'une base de données NoSQL est plus adaptée à nos besoins car offrant bien plus de flexibilité et de rapidité d'accès, et offrant tout à fait les fonctionnalités que nous désirons. Les bases de données NoSQL travaillent très souvent avec des données sous format JSON, ce qui simplifie fortement le traitement de ces données.
Notre choix s'est donc porté un peu plus loin que des bases de données NoSQL majeures, telles que MongoDb, pour se poser sur RethinkDb, qui à ainsi l'avantage d'être open-source, de disposer d'une très large communauté, et d'offrir toutes les fonctionnalités qu'offre MongoDb, et une particularité en plus : la présence de la fonctionnalité de changefeed, qui permet à un client de recevoir les changements sur une table ou un document précis de la table en temps-réel, où encore, comme la documentation le précise, les changements qui sont apportés sur les résultats d'une requête spécifique.

Cette fonctionnalité est très importante pour notre projet car nous tenions fortement à implémenter un aspect temps-réel, ce qui est rendu possible bien plus facilement par la fonctionnalité de changefeeds. En effet, sans cette fonctionnalité, nous aurions dû faire recours à d'autre méthodes permettant d'approcher le temps réel. Des recherches nous ont permis de comparer ces différentes méthodes : le long polling par exemple, qui détient

cependant le désavantage d'être très gourmand en ressource et peut parfois paraître assez long, empêchant cet aspect de temps-réel que nous désirons implémenter.

Les changefeeds permettent, par de simples requêtes, de communiquer avec les clients en ouvrant une liaison websocket : ainsi, chaque changement dans la base de donnée apparaîtra et sera traité chez chaque client connecté.

b) Réalisation de l'API REST :

La réalisation d'une API REST (ou API CRUD dans notre cas) était nécessaire dans le cadre de notre projet. En effet, nous sommes dans l'obligation de stocker les valeurs dans une base de donnée et de faire en sorte que ces données puissent être lues de tous les clients.

Le framework Jersey facilite très fortement la tâche en termine de réalisation d'API REST, car fonctionnement par annotations Java. Il est cependant possible de procéder de l'ancienne manière en héritant et implémentant différentes classes, mais les directives du framework recommandent d'utiliser les annotations, qui offrent les même fonctionnalités pour un code plus lisible et maintenable.
Ainsi, les annotations concernent quasiment toutes les fonctionnalités offertes par Jersey, allant des providers, désignés par l'annotation @Provider, qui nécessitant néanmoins d'implémenter la classe désirée. Un provider constitue simplement un moyen d'hériter des fonctionnalités d'exécution de JAX-RS et ainsi d'en modifier le comportement.
Dans notre cas, nous utilisons un unique provider, représenté par la classe CORSFilter, qui permet d'autoriser le cross-origin en ajoutant des headers personnalisés à toutes les requêtes sortantes. La méthode d'implémentation est très simple, comme représenté sur la capture d'écran suivante :

```
@Provider
public class CORSFilter implements ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext containerRequestContext,
                       ContainerResponseContext response) throws IOException {
        response.getHeaders().add( k: "Access-Control-Allow-Origin", v: "*");
        response.getHeaders().add( k: "Access-Control-Allow-Headers",
                v: "origin, content-type, accept, authorization, x-api-key");
        response.getHeaders().add( k: "Access-Control-Allow-Credentials", v: "true");
        response.getHeaders().add( k: "Access-Control-Allow-Methods"
                , v: "GET, POST, PUT, DELETE, OPTIONS, HEAD");
    }
}
```

Cette classe illustre la simplicité d'utilisation des annotations Jersey.

En plus des annotations provider, les annotations qui nous sont réellement utiles

correspondent aux requêtes HTTP. Ainsi, Jersey dispose d'annotation définissant les requêtes GET, POST, DELETE, PUT, OPTION et HEAD.
Chaque annotation est placée au dessus de la méthode qui représente l'action que celle-ci doit effectuer lors de son appel. Nous caractérisons également cette méthode par une annotation Path qui permet de spécifier le chemin d'accès.
Les annotations Consume et Produce permettent de spécifier les types MIME de ressources que la méthode accepte et renvoie.
Ainsi, un exemple d'utilisation de ces requêtes est proposé dans la capture d'écran suivante :

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Path("/putAirData")
public Response insertData(String json, @Context HttpHeaders headers) {
```

Il est à noter l'utilisation de l'annotation Context, qui permet d'obtenir différents éléments du contexte de la requête. Dans ce cas précis, nous récupérons les en-têtes de la requête, représentés par la classe HttpHeaders. Une autre annotation très utilisée dans notre API, est l'annotation @PathParam , qui permet de récupérer les paramètres d'URI.

Nous utilisons le format JSON pour toutes nos requêtes, incluant les résultats, les erreurs, où encore les corps des requêtes POST consommées.

Notre API est divisée en trois ressources principales, dont deux sont indispensables au fonctionnement du projet.
La ressource, représentée par la classe UnwheezeResources, définie par le chemin d'accès /auth, permet de retourner les données d'identification, tels qu'une clé API, nécessaire à l'utilisation des ressources liés aux données de l'air. Aucune identification n'est nécessaire pour obtenir une clé API.
Une simple requête GET sur le lien de la méthode retournera une donnée au format JSON contenant la clé, sous le champ "key".

La ressource représentée par la classe AirResource constitue le coeur du projet car permet d'effectuer l'ensemble des requêtes liés aux données de l'air.
Ainsi, chaque requête contenue dans cette ressource effectue une vérification de la présence d'un en-tête X-Api-Key, dont la valeur correspond à une clé API contenue dans la table réservée à cet effet.
Dans le cas où cette clé n'est pas valide, une réponse 403 (FORBIDDEN) est renvoyée, accompagnée d'un corps spécifiant la raison de l'erreur.

Différentes méthodes sont fournies par cette ressource, permettant d'ajouter une donnée de l'air, de récupérer une donnée spécifique, identifiée par son ID, de récupérer l'ensemble des données dans un tableau JSON, où encore, de récupérer un tableau des éléments les plus proches de la localisation du point passé en paramètre.
Cette dernière fonctionnalité utilise le support de GeoJSON offert par RethinkDb, qui permet de manipuler des points géographiques, et de calculer ainsi, dans notre cas, des distances

entre plusieurs points.
Cette fonctionnalité nécessite l'implémentation d'un index sur un champ de format GeoJSON, correspondant au champ geolocation dans notre cas.

Le fonctionnement de notre API à été vérifié tout le long de la conception par la réalisation d'unités de test, qui sont facilement implémentés sous Java. Un exemple de test est représenté ci-dessous :

```
@Test
public void testGetDataFromCollection() {
    UnwheezeDbAirData db = new UnwheezeDbAirData();
    String id = "debac791-5b86-4151-a2f9-b32974974f62";
    String json="";
    try{
        json = db.getDataFromCollection(id);
    } catch(IllegalAccessException e) {
        e.printStackTrace();
        fail();
    }
    String expected = "{" +
            "\"datetime\":  \"1518017373390\" ," +
            "\"id\":  \"debac791-5b86-4151-a2f9-b32974974f62\" ," +
            "\"location\":  \"50.633333,3.066667\" ," +
            "\"no2\": 0 ," +
            "\"pm10\": 14.5 ," +
            "\"pm25\": 20.5 ," +
            "\"userID\":  \"54fd5fe5f5d\"" +
            "}";
    Gson gson = new Gson();
    AirData airdataJson = gson.fromJson(json,AirData.class);
    AirData airdataExp = gson.fromJson(expected,AirData.class);

    assertTrue(EqualsBuilder.reflectionEquals(airdataExp,airdataJson));

}
```

La classe contenant cette méthode hérite de la classe TestCase, ce qui permet d'hériter également de méthodes telles que assert(), ou assertTrue().

c) Réalisation du serveur websocket

Jersey fournit un support natif des websockets à travers son endpoint @ServerEndpoint, dans lequel est spécifié le lien d'accès, ainsi que les encoder et décodeurs utilisés.
Dans notre cas, à l'ouverture de la connexion avec un client, un nouveau thread est créé et initie un abonnement aux changefeeds sur la table relative aux données de l'air.

Les encodeurs sont nécessaires dans notre cas, car les données reçues par le changefeed sont d'abord traités avant d'être renvoyés, et il est plus aisé de traiter des valeurs mappés dans un objet java bean.
Le décodeur quant à lui, suit la même idéologie. Un paquet représentant une chaîne de caractères est reçue puis mappée à l'objet java bean correspondant afin d'offrir un traitement plus aisé.

IV) Réalisation de l'application mobile

Notre choix pour la réalisation de l'application mobile s'est porté sur la plateforme Android, en raison du très grand nombre de smartphones basés sur cet OS. Le langage de programmation utilisé y joue également sa part, les application Android étants codés en Java ou Kotlin.

a. L'application mobile

Les buts principaux de notre application mobile étaient de pouvoir se connecter au capteur à travers une liaison Bluetooth, de pouvoir communiquer avec celui-ci et ainsi de faire une demande de mesure et de récupérer les valeurs, de pouvoir envoyer ces valeurs vers notre base de donnée, mais également d'implémenter le plus de fonctionnalités que nous retrouvons sur le site web. Ces buts ont tous été atteints, ce qui nous a permis de nous concentrer sur davantage de fonctionnalités.

L'application se connecte au capteurs à travers une liaison Bluetooth Low Energy, introduite dans les versions les moins anciennes d'Android (à partir de 4.3) . Notre choix s'est porté sur BLE, en lieu et place du Bluetooth classique en raison du fait que celui-ci permet de réaliser des économies d'énergie considérable sur le smartphone, mais également en raison du fait que le capteur utilise également BLE.
BLE, au contraire d'une liaison Bluetooth classique, utilise un transfert périodique de petites quantités de données pour récupérer des informations légères, tandis que le Bluetooth classique transfert et échange des données en continu, ce qui n'est pas utile pour notre cas.

Le BLE utilise le protocole GATT, dont Android dispose d'un support natif. Le protocole GATT (Generic Attribute Profile), défini la manière dont les données sont échangées à travers de concepts nommés Services et Caractéristiques.
Ainsi, à la connexion, nous récupérons la liste des services et des caractéristiques dont dispose notre capteur. Les caractéristiques encapsulent une unique donnée. Notre capteur dispose de deux caractéristiques, l'une associée à l'écriture d'un caractère, lançant l'acquisition, et l'autre à la lecture des valeurs du capteurs, qui sont transmises.
Notre application Android s'abonne ainsi à ces caractéristiques afin d'être notifié de tout changement sur celles-ci. Lorsqu'une nouvelle mesure est effectuée, les valeurs sont exploités.
Ainsi, nous suivons un pattern de Publish-Subscribe afin de récupérer les données.

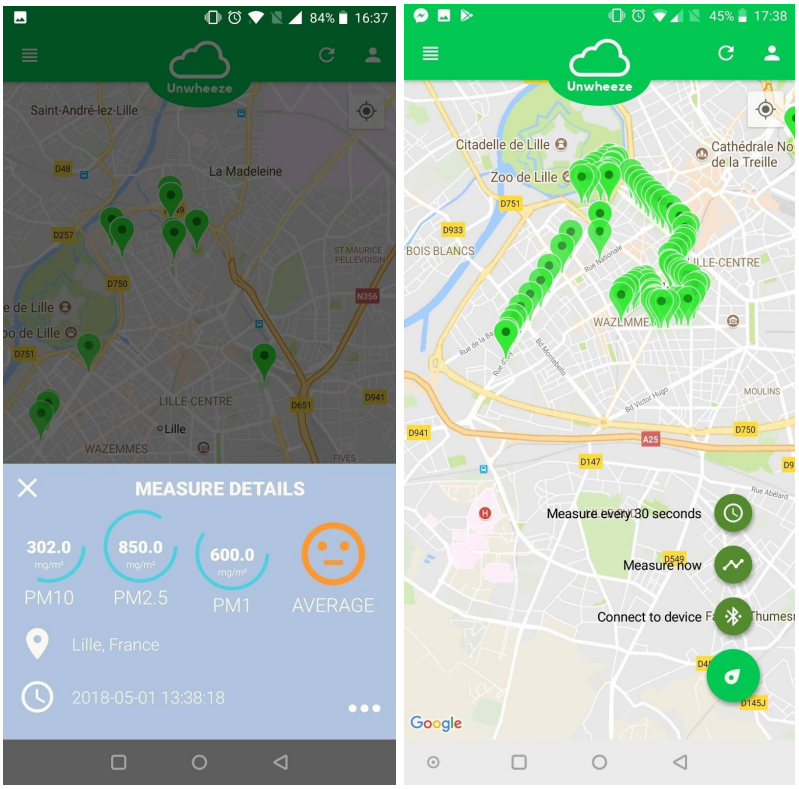
La demande de mesure se fait en envoyant le caractère arbitraire ‘B’ au capteur. Nous écrivons ainsi dans la caractéristique associée.
Les UUID des caractéristiques sont stockés par défaut dans le fichier strings.xml, dans le cas où la détection automatique échoue.
Nous avons également la possibilité de réaliser des mesures en continu, espacés de trentes secondes. Cette fonctionnalité n’est active que lorsque l’écran est allumé. Pour cela, nous un ScheduledExecutorService, qui exécute un thread parallèle périodiquement.

Les valeurs reçues sont ensuite affichés sous forme de progress bar, accompagnés d’un smiley représentant le niveau global de la mesure.
Un bouton intitulé “Measure Again” permet de prendre une nouvelle mesure.

Une fois les mesures reçues, ces dernières sont placés dans la queue de requêtes HTTP et sont ainsi envoyés vers le serveur.
Les requêtes HTTP sont effectués en utilisant la librairie Volley, recommandée par Google, et dont l’un des avantages majeurs est d’autoriser le caching des données reçue et la gestion des requêtes dans le cas où l’utilisateur ne dispose pas de connectivité, ce qui simplifie la conception de l’application.

Un service android est également lancé au lancement de l’application. Ce service permet d’établir une liaison avec le serveur websocket lors de la durée de vie de l’application. Un service android tourne en fond et communique en même temps avec le thread principal à traver un BroadcastReceiver. Ainsi, notre service, à la réception d’un paquet websocket, notifie notre BroadcastReceiver, implémenté sur la classe MainActivity, représentant l’activité principale, qui traite les données reçues.
Le processus de traitement est le suivant : nous vérifions dans un premier temps le bon format des données. Ces données sont ensuite enregistrés dans la base de donnée SQLite locale, pour ensuite être affichées sur la carte.

La page de chargement est inspirée des lignes de conduites google, qui recommandent d'effectuer le chargement principal, souvent lourd et imposant une latence à l'ouverture de l'application, dans un thread secondaire, tandis que le thread principal affiche une interface plus plaisante à regarder qu'une interface blanche et vide.
Le thread de chargement contacte l'API REST par une requête GET, afin d'obtenir une clé API et la stocker dans un fichier de données privé et propre à l'application. Nous vérifions également la présence du bluetooth ainsi que des services de localisation. La présence de ce dernier est primordiale, car nous chargeons la localisation dans le même fichier de données que précédemment afin d'afficher la carte centrée sur la localisation de l'utilisateur, mais également pour télécharger uniquement les données relative à la ville actuelle.



L'activité principale charge quant à elle les données disponibles dans la base de donnée, les stock dans la base de donnée locale, et les affiches sur la carte sous forme de marqueurs. Toutes les actions susceptibles d'induire une latence sont effectués dans un thread parallèle.
Un clic sur un marqueur permet d'afficher un fragment disposant des différentes valeurs associés, tels que la date de mesure, la localisation exacte, ainsi que les valeurs accompagnés d'un smiley représentant l'état global lors de la mesure.

```
public void getAllAirDataElements(NetworkActionListener listener) {
    mBuilder.appendPath(RequestsScheme.AIRDATA_GETALL);
    Uri uri = mBuilder.build();

    JsonArray jsonArray = new JsonArray();
    Gson gson = new Gson();
    JsonArrayRequest jsonArrayRequest = new JsonArrayRequest(Request.Method.GET,uri.toString(), jsonRequest: null,(response)-> {
        jsonArray.addAll(gson.fromJson(response.toString(), JsonArray.class));
        listener.onResponseResult(jsonArray);
        Log.d(TAG,jsonArray.toString());
    },(error) -> {
        //TODO: Handle error
    }){
        @Override
        public Map<String, String> getHeaders() throws AuthFailureError {
            HashMap<String,String> map = new HashMap<>();
            map.put("X-Api-Key",getApiKeyFromPrefs());

            return map;
        }
    };
    mQueue.add(jsonArrayRequest);
}
```

Un bouton refresh est disponible et permet de recharger les données affichés sur la carte en cas d'erreur de chargement.
Les erreurs rencontrés et les actions interdites sont notifiés à l'utilisateur à travers des boîtes de dialogue dont nous avons modifiés l'apparence.

L'activité secondaire permet d'observer les graphiques représentants les données disponibles. Ces graphiques peuvent être configurés sur le temps ou sur l'espace, de la même façon que sur le site web. En effet, nous désirions reproduire fidèlement les fonctionnalités du site web sur l'application mobile, sachant que certains utilisateurs préfèrent l'un ou l'autre.
Nous avons choisi la librairie MPAndroidChart afin de tracer nos graphiques, car libre et open-source, et réalisant le travail assez fidèlement.
La base de donnée utilisée est la base de donnée intégrée par la librairie de base Android, SQLite.
Ce service est cependant moins développé que sur le site web.

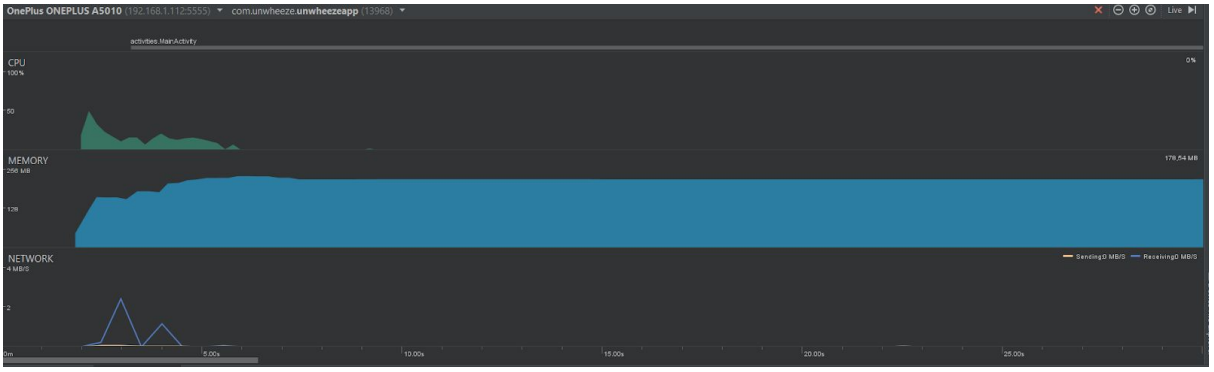
Nous utilisons intensivement le multi-threading afin de permettre à l'utilisateur de ne pas être impacté par les divers chargements. Ce multi-threading s'observe et se montre très utile majoritairement dans la manière dont nous gérons les requêtes HTTP. En effet, la documentation Android recommande très fortement d'utiliser la librairie Volley, maintenue par Google, afin de réaliser toute action relative aux requêtes. L'un des avantages majeurs de cette solution est qu'elle permet un caching des données reçues : certes, la mémoire utilisée sera plus élevée. Cependant, dans le cas de plusieurs requêtes d'affilés, sans changement entre celles-ci, nous n'aurons pas besoin de solliciter la connectivité de l'utilisateur, et stockerons moins de valeurs en mémoire. Nous retrouvons également un très grand avantage dans l'utilisation de l'observer pattern dans la gestion des réponses en cas de succès où d'erreurs: les requêtes sont ainsi asynchrones et n'impactent pas l'utilisateur, car indépendants de la latence du serveur ou de la vitesse de connexion.

Nous avons décidé de porter une attention particulière à la réalisation de l'application mobile: afin de refléter au mieux l'importance du projet, il était nécessaire de nous conformer également aux codes de design des applications actuellement disponibles réalisant plus ou moins le même but. Ainsi, nous avons passé un temps conséquent sur la réalisation du design de l'application, qui, bien que sobre, reflète les normes dictées par Google afin de se conformer à son Material Design, tout en se permettant quelques libertés particulières.
Les couleurs choisies correspondent à la couleur verte, représentant l'environnement, notre projet étant très lié à l'environnement et son respect, tandis que la couleur secondaire est le bleu, correspondant au ciel et à l'air, dont notre projet traite principalement.
Nous tentons également d'éviter de surcharger les écrans en proposant uniquement les fonctionnalités désirés, accompagnés d'icônes ou de texte décrivant brièvement mais efficacement leur fonction.

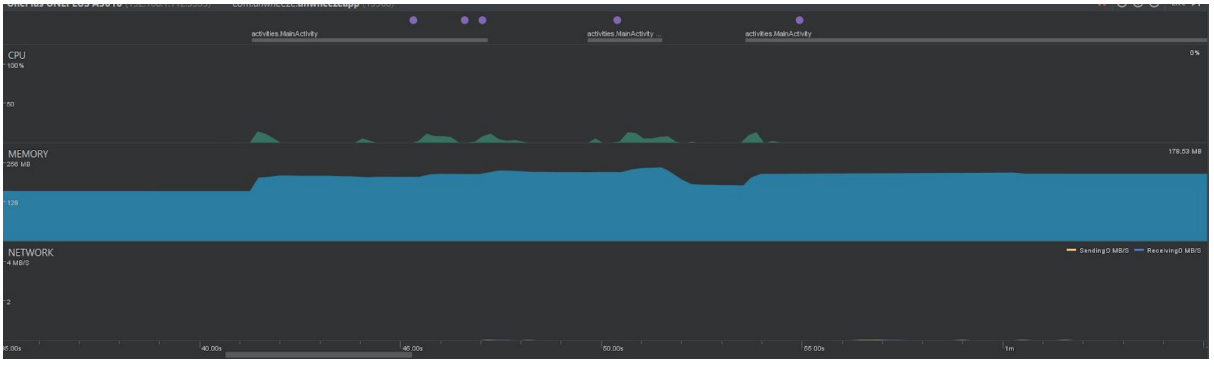
b. Analyse de l'utilisation en ressources de l'application

Android Studio fournit un utilitaire d'analyse des performances en temps-réel, que nous avons trouvés pas très intelligent de ne pas nous en servir. En effet, l'un des majeurs casse-têtes qui intervient dans le développement d'une solution logicielle sur un objet connecté (bien qu'un smartphone actuel ne soit pas très concerné), est d'utiliser le moins de ressources possibles et d'éviter les taches de fond lorsqu'elles ne sont pas absolument nécessaires.

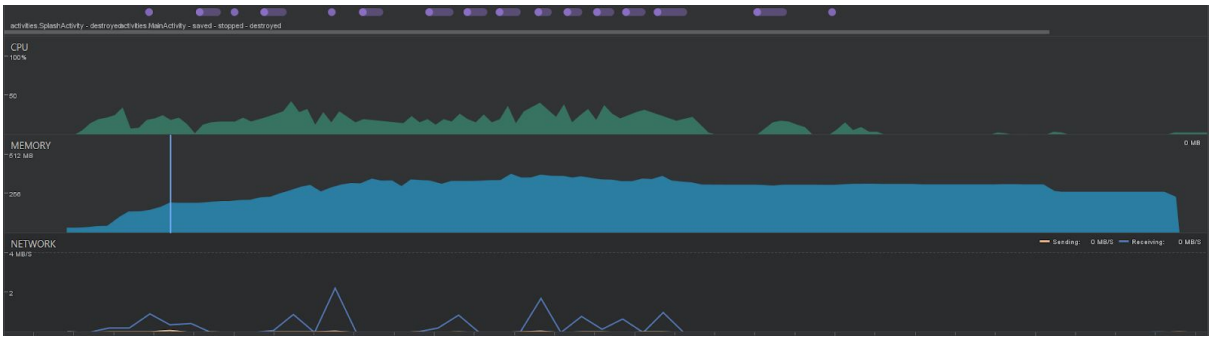
Ainsi, le lancement de l'application montre une utilisation relativement élevée du CPU et des fonctions réseau, ce qui s'explique par le fait que nous réalisons une requête chargeant l'ensemble des données contenues sur notre serveur, dans une base de données interne au smartphone SQLite, dans la perspective de diminuer le nombre de requêtes par la suite et le temps de chargement. L'utilisation élevée du CPU est normale et est dû au fait que nous créons l'interface de l'application en sollicitant une grande partie des ressources libres du smartphone.



Nous réalisons une nouvelle acquisition, cette fois-ci durant la connexion du smartphone au capteur par bluetooth LE: les avantages de l'utilisation du bluetooth faible énergie se confirment car l'utilisation du CPU est très faible.



Enfin, une dernière acquisition est réalisée, cette fois-ci, lors de l'envoi d'une demande de mesure au capteur, et la réception de celle-ci, donc l'envoi au serveur et la nouvelle réception d'un paquet websocket contenant ces nouvelles valeurs. Nous observons clairement une activité sur le CPU et réseau, mais rien qui se trouve anormalement élevé.


activities.SplashActivity - destroyedactivities.MainActivity - saved - stopped - destroyed
CPU
100%
50
MEMORY
512 MB
256
0 MB
NETWORK
4 MB/S
2
Sending: 0 MB/S
Receiving: 0 MB/S

Conclusion

En conclusion, nous pouvons affirmer que ce projet nous a permis de nous placer dans une situation réelle de projet, avec la nécessité d'une communication entre binôme, un apprentissage tout le long du projet et nous a amené à faire face à plusieurs problèmes, qui, en les résolvant, nous ont appris énormément.
Nous avons également beaucoup pris soin de faire en sorte que notre projet soit facilement maintenable par la suite, à travers l'écriture d'un code modulaire et propre, tout en étant concis dans notre travail.
Enfin, le caractère assez "actuel" du projet, à savoir, portant sur la problématique de la pollution, très d'actualité, nous à permis de nous sensibiliser encore plus à ce problème et nous a incité à réaliser un système qui soit fiable.

Dans une perspective d'avenir, le projet peut être amélioré, en ajoutant par exemple des algorithmes de calculs de chemin les plus propres, où encore, en implémentant une interface utilisateur complète, partie déjà bien aboutie.