

TRAVAIL DE BACHELOR

MÉMOIRE

Conception d'un système temps réel LoRa pour compétitions sportives

Auteur :

Léonard BISE

Conseiller :

Pierre BRESSY

Informatique Embarquée (ISEC)

Mars à Septembre 2018

Travail de Bachelor

Préambule

Résumé

TODO: Import from resume

Table des matières

Table des figures	vii
Liste des tableaux	x
1 Introduction	1
1.1 Énoncé du problème	1
2 Vue d'ensemble du système	3
2.1 Les interactions	4
2.2 La couche radio LoRa	4
3 Développement du projet	5
4 Description du capteur	7
4.0.1 Les contraintes	7
4.1 Le matériel	7
4.1.1 Le module GPS UBloxEVA8M	8
4.1.2 Le module LoRa RN2483	9
4.1.3 Le module rythme cardiaque Adafruit	10
4.1.4 Le module accéléromètre LSM303AGR	10
4.1.5 La batterie	10
4.2 Le système d'exploitation Zephyr	10
4.3 Le logiciel embarqué	12
4.3.1 Architecture logiciel	12
4.3.2 Race Sensor Manager	15

4.3.3 Heart Rate	15
4.3.4 Cadence	15
4.3.5 Debug	16
4.3.6 Race Sensor Shell	16
4.4 Les drivers	18
4.4.1 Driver I^2C ATSAMD21G18	18
4.4.2 Driver UBloxEVA8M	20
4.4.3 Driver LSM303AGR	23
4.4.4 Driver RN2483	25
4.4.5 Driver LEDs	26
4.5 Paquet de donnée	27
4.6 Le débogage	27
4.7 Le boîtier	27
5 Description de la passerelle	29
5.1 Le matériel	29
5.2 Le packet forwarder	30
5.3 Le serveur d'application	32
5.3.1 Architecture logiciel	32
5.3.2 Les librairies externes	34
5.3.3 Les classes	34
6 Description de la base de données	41
7 Description de l'application mobile	43

8 Test phase #1	45
8.1 Scénarios	46
8.2 Résultats	46
8.2.1 Test sur piste	47
8.2.2 Test de distance	47
8.3 Conclusion test phase #1	50
9 Test phase #2	51
10 Test phase #3	53
11 Considérations pour le développement d'un produit	55
12 Conclusion	57
13 Annexes	59
14 Dossier de gestion	61
Bibliographie	63

Table des figures

4.1 Schéma block du capteur SODAQ One	9
4.2 Architecture du système d'exploitation Zephyr - http://www.zephyrproject.org	12
4.3 Processus général du capteur	13
4.4 Architecture statique du logiciel embarqué sur le capteur	14
4.5 Architecture dynamique du logiciel embarqué sur le capteur	15
4.6 Diagramme de classe du Race Sensor Manager	16
4.7 Diagramme de séquence de l'initialisation du Race Sensor Manager	17
4.8 Diagramme de séquence du thread du Race Sensor Manager	17
4.9 Schéma d'un bus I^2C	18
4.10 Les messages I^2C	19
4.11 Diagramme de class du driver I^2C	19
4.12 Diagramme de class du driver UBloxEVA8M	21
4.13 Diagramme de class du driver LSM303AGR	23
4.14 Diagramme de class du driver RN2483	25
4.15 Diagramme de class du driver LED	26
5.1 Schéma block de la passerelle	30
5.2 Architecture statique du serveur d'application	33
5.3 Architecture dynamique du serveur d'application	34
5.4 Diagramme de classe de Race App Server	35
5.5 Diagramme de séquence des opérations du thread de la classe Race App Server	36
5.6 Diagramme de classe de Race Tracker Data	36

5.7	Diagramme de classe de rxpk Handler	36
5.8	Diagramme de classe de Race Mode Handler et Race Mode Record	37
5.9	Diagramme de classe de Test Mode Handler et Test Mode Record	38
5.10	Diagramme de classe de LoRa Packet Forwarder Parser	39
5.11	Diagramme de classe de LoRa Push Data Parser	39
5.12	Diagramme de classe de LoRa Push Data Parser	40
8.1	Format du paquet test1	45
8.2	Positions GPS des tests piste	48
8.3	Positions GPS des tests distance	49

Liste des tableaux

4.1 Caractéristiques de la carte SODAQ One v3	8
4.2 Caractéristiques des threads du capteur	14
5.1 Caractéristiques du Raspberry Pi 3 Model B+	29
5.2 Caractéristiques du Dragino LoRa Hat	30
8.1 Résultats des tests phase 1 - Piste	47
8.2 Résultats des tests phase 1 - Distance	47

1 Introduction

Le développement récent des technologies liées à l'Internet of Things permet la réalisation de systèmes embarqués intelligents, à faible coût et de petite taille. Aussi de plus en plus de carte électronique présentant des systèmes temps réels performant capable de communiquer avec une interface sans-fils existent sur le marché. L'envie de pouvoir développer un système sur cette base m'a amenée à porter une réflexion sur quel type de projet pourrait tirer partie d'une telle application.

Étant moi même amateur de course à pied, je me suis posé la question de savoir si une application à base de capteur pourrait apporter une évolution dans ce sport ce qui me permettrait de combiner deux sujets qui m'intéressent particulièrement. Ceci m'a amené à l'idée poursuivie par mon travail de Bachelor, réaliser un système de suivi temps réel à base de la technologie LoRa afin d'être utilisé pendant des compétitions sportives. L'idée de base du projet est tirée du fait que pendant des compétitions, de course à pied mais cela est également applicable à d'autres sports comme le cyclisme par exemple, il n'est pas facile aux spectateurs d'avoir une vue d'ensemble de la situation de la course ou un classement précis ce qui peut rendre l'événement parfois ennuyeux ou difficile à suivre.

Afin d'essayer de rendre de tels événements plus vivant j'ai donc décidé de réaliser un système qui propose aux spectateurs l'utilisation d'une application mobile, avec laquelle il lui serait possible à tout moment de connaître la position actuelle des concurrents ainsi que d'autres informations intéressantes comme son temps de course, la distance parcouru ou son rythme cardiaque par exemple.

Pour pouvoir proposer ces fonctionnalités, le système définit les éléments suivants. Un capteur porté par les sportifs qui est en charge de l'acquisition des différents paramètres et de leur transmission, une passerelle qui s'occupe de récupérer les données transmises par les capteurs, de les traiter et de les stocker dans une base de données et enfin de l'application mobile elle même qui permettra aux spectateurs de visionner une carte avec la position actuelle des compétiteurs ainsi que tous les paramètres acquis durant la course.

TODO: Add datasheet url to biblio

1.1 Énoncé du problème

TODO:

2 Vue d'ensemble du système

Le système est composé de plusieurs acteurs différents qui ont chacun une tâche bien précise, ce chapitre propose une vue d'ensemble de ses éléments et explique également les interactions qu'ils existent entre eux.

L'objectif du système est de permettre la récupération de toutes les données récoltées par le ou les capteurs et de centraliser ces informations afin que l'application mobile puisse les exploiter et les afficher aux utilisateurs au travers de son interface graphique.

Afin de pouvoir réaliser cet objectif, les éléments suivants sont développés.

- Un capteur
- Une passerelle
- Une base de données
- Une application mobile

Le capteur est porté par un sportif, il est en charge de l'acquisition des données et de leur transmissions à une passerelle en utilisant la couche radio LoRa. Il est défini en détails dans le chapitre 4.

La passerelle se charge de récupérer les données transmises par le ou les capteurs, les traite et puis les centralise dans une base de données. Elle est décrite dans le chapitre 5.

La base de données permet le stockage de toutes les informations collectées durant la course mais également d'autres informations qui sont saisies avant, comme le nom et prénom, le numéro de dossard ou le pays d'origine des athlètes. Chaque compétition ainsi que leurs informations associées sont également enregistrées dedans. Le chapitre 6 explique son fonctionnement.

L'application mobile est la fenêtre sur le système, elle permet aux utilisateurs de visualiser en temps réel l'évolution de la course. Sa description se trouve dans le chapitre 7.

Afin de pouvoir être utilisable pendant des compétitions sportives, le système doit être capable de gérer plusieurs capteurs en parallèle, il est donc développé dans cette optique. Cependant dans la mesure où dans le cadre du travail de Bachelor le projet est une preuve de concept, un seul capteur sera assemblé et testé.

En ce qui concerne les passerelles, idéalement plusieurs d'entre elles doivent pouvoir être utilisées durant une course, cela permet de diminuer les zones d'ombres le long du parcours et également de minimiser le nombre de paquets perdus. Cependant cela complique passablement le système, car cela implique que la base de données doit être hébergée sur un serveur connecté à internet et donc que la passerelle doit également pouvoir s'y connecter. Afin de simplifier le développement du projet, il est décidé de ne développer qu'une seule passerelle et d'y héberger localement la base de données.

Enfin le chapitre 11 rassemble les éléments qu'il faudrait retravailler afin de faire de la preuve de concept un produit à part entière.

2.1 Les interactions

Le système exploite deux types de communication différentes afin de stocker et d'extraire des données de la base. D'une part la couche radio LoRa est utilisée pour la communication entre les capteurs et les passerelles, et d'autre part le WiFi pour les requêtes entre la base de données et l'application mobile.

Dans le cadre de la preuve de concept et pour des raisons de simplifications, la couche MAC LoRaWAN n'est pas employée, elle prendrait cependant tout son sens dans une optique de développement d'un produit.

La figure ?? montre les interactions existantes entre les acteurs du système.

FIGURE

2.2 La couche radio LoRa

3 Développement du projet

TODO: Etape de la démarche suivie, identification de l'env et des contraintes. capteur->passerelle->db->app simple

4 Description du capteur

Le travail du capteur et d'acquérir les données nécessaires, puis de les transmettre à intervalles réguliers par la couche radio LoRa à la passerelle. Le cœur du capteur est le micro-contrôleur, celui-ci permet l'exécution du firmware qui est en charge de la gestion des opérations. C'est cette application qui va effectuer aux moments voulus les acquisitions nécessaire et ensuite créer un paquet de données pour être envoyé.

Pour se faire le capteur est muni de plusieurs modules permettant l'acquisition des différents paramètres. Il sont présentés dans la liste suivante.

- GPS : Il permettra de connaître la position (latitude/longitude) du capteur et également d'avoir une référence de temps. Voir la section 4.1.1 pour plus de détails.
- Accéléromètre : Ce module sera utilisé pour connaître le nombre de pas effectués par le sportif ce qui permettra de calculer sa cadence de course. Voir la section 4.1.4 pour plus de détails.
- Rythme cardiaque : Au moyen d'une sangle pectorale portée par l'athlète, ce module déclenchera une impulsion à chaque fois qu'un battement du cœur sera détecté. Voir la section 4.1.3 pour plus de détails.
- Radio LoRa : C'est au moyen de cet élément que le capteur transmettra les paquets de données à la passerelle. Voir la section 4.1.2 pour plus de détails.

Dans le cadre du travail de Bachelor, un seul exemplaire de capteur sera assemblé et utilisé durant les tests.

4.0.1 Les contraintes

Le capteur est également soumis à des contraintes liées à son utilisation dans les conditions d'une compétition sportive.

Afin de gêner au minimum le sportif pendant la course et afin que le capteur soit utilisable pour l'entièreté d'une compétition, il est soumis aux contraintes définies durant la pré-étude et listées ci-dessous.

- Veiller à ce que sa taille soit minimale
- Son poids ne doit pas dépasser 200 g
- Il doit disposer d'une autonomie d'au moins 10 heures
- Il doit être capable de transmettre les paquets de données à une passerelle située à une distance de 5 km en espace libre
- Il sera placé dans un boîtier étanche

4.1 Le matériel

Lors de la pré-étude du projet, trois différentes cartes avaient été étudiées chacune avec leurs avantages et inconvénients. Pour la réalisation du projet, j'ai décidé d'utiliser la carte qui dispose de base du plus de module, c'est à dire la carte SODAQ One. En effet elle a l'immense avantage d'embarquer de base un module LoRa, un module GPS ainsi qu'un accéléromètre ce qui me permet de me focaliser sur le développement du logiciel embarqué. Il reste seulement

à connecter sur une entrée du micro-contrôleur le module qui permettra de compter les battements du cœur en détectant les impulsions produite. Enfin afin de faciliter le debug de l'application embarquée, un UART et une sonde de debug seront également connectés pendant la phase de debug ce qui permettra d'afficher des messages et de permettre le debug du firmware. Un autre avantage de taille est que son micro-contrôleur est déjà disponible dans le système d'exploitation Zephyr ce qui facilitera le travail de portage.

Pour rappel, les caractéristiques du SODAQ One sont décrit dans la table 4.1.

TABLE 4.1 – Caractéristiques de la carte SODAQ One v3

Dimensions	45mm x 25mm
Microcontrôleur	ATSAMD21G18 – ARM Cortex M0
Oscillateur	48 Mhz
Flash	256 kB
RAM	32 kB
LoRa	Microchip RN2483
GPS	uBlox EVA 8M
Accéléromètre	STMicroelectronics LSM303AGR
Prix	114 CHF

Aux modules de base, comme expliqué précédemment, il faudra rajouter un module qui permettra de compter le nombre de battement du cœur. Il est développé par la société Adafruit sous le nom de "Adafruit Heart Rate Start Pack".

TODO: PHOTO SODAQ

Le module LoRa RN2483 est connecté par un lien série UART et utilise une interface de type AT commands, c'est à dire qu'il est piloté avec l'envoie de chaînes de caractère représentant des commandes, dans la même idée les réponses reçues sont de type text. Le module GPS ainsi que l'accéléromètre sont quant à eux connecté sur le bus I^2C . Le module rythme cardiaque sera lui connecté simplement sur un General Purpose I/O.

Le schéma block 4.1 présente les différents modules et leurs connections avec le micro-contrôleur.

TODO: Ajouter debug SEGGER

4.1.1 Le module GPS UBloxEVA8M

TODO: figure

Le EVA8M de la compagnie UBlox est un module GPS à haute précision et qui propose 8 moteurs de positionnement avec des performances très intéressantes. Il est capable de gérer les signaux GPS, GLONASS, QZSS et SBAS et dispose d'une sensibilité très haute de -164 dBm. Son temps d'acquisition de la position est minime et il dispose de mécanisme d'optimisation de la consommation d'énergie.

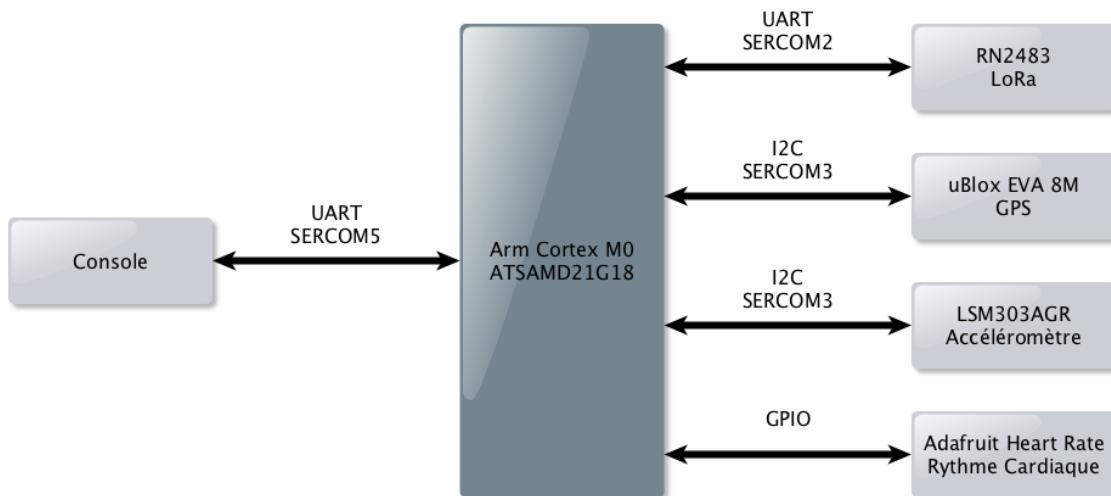


FIGURE 4.1 – Schéma block du capteur SODAQ One

Ce module est très simple d'utilisation, dispose de son propre oscillateur et pour la plupart des applications il ne requiert qu'une antenne GNSS externe. De plus il propose plusieurs interfaces différentes, SPI, USB, I^2C et UART.

Il assure une précision de la position GPS à 2.5 m et de 4 m en GLONASS, Il est capable de générer des messages jusqu'à une fréquence de 18 Hz et il propose 3 protocoles différents, NMEA, UBX et RTCM.

Une fois le module configuré par l'utilisateur, il va générer les messages voulues à une certaine fréquence. En fonction des message différentes information seront disponible, le choix des messages sélectionnés dépend du type d'application que l'on souhaite réaliser. Lorsque la configuration est effectuée, il suffit à l'utilisateur de venir lire la queue de message du module périodiquement afin de pouvoir y récupérer les messages et d'y extraire les informations.

Le module est détaillé dans le datasheet [1] ainsi que dans le document d'explication de son protocole [2].

4.1.2 Le module LoRa RN2483

TODO: figure

Le RN2483 est un module de gestion de la couche radio LoRa et également de la couche protocolaire LoRaWAN. Il utilise un simple protocole de commande/réponse sur une interface de type UART. Bien qu'il ne soit pas utilisé dans ce projet, le module est capable d'assurer la gestion de la couche LoRaWAN de classe A automatiquement, c'est à dire de la gestion du mécanisme d'authentification avec gestion du chiffrement des messages LoRa et du reste des mécanismes propres au LoRaWAN.

Chose intéressante, il est possible de désactiver entièrement la couche protocolaire LoRaWAN si l'on désire uniquement utiliser la couche radio LoRa, ce qui est le cas de ce projet. Dans ce cas de figure tous les éléments de gestion du protocole sont désactivés, reste la possibilité de configurer la couche radio LoRa avec la modification des différents paramètres, comme le facteur d'étalement ou la puissance de sortie du signal par exemple, et l'envoie ou la réception de

données.

En plus le composant est capable d'envoyer les signaux en utilisant diverses modulations, FSK, GFSK ou LoRa.

L'utilisation ainsi que toutes les commandes sont décrite dans la datasheet du composant [3].

4.1.3 Le module rythme cardiaque Adafruit

Le module Adafruit pour le rythme cardiaque est composé de deux éléments principaux, d'une part la sangle pectorale qui sera portée par le sportif et le récepteur qui permettra au micro-contrôleur de détecter les battements du cœur. La sangle et le récepteur utilise le protocole sans-fil WeakLink+ de la compagnie Polar ce qui permet au récepteur, lorsqu'un battement est transmis par la sangle, de générer une impulsion sur une entrée du micro-contrôleur.

TODO: figure

4.1.4 Le module accéléromètre LSM303AGR

TODO: figure

Le LSM303AGR est un accéléromètre trois axes couplé à un magnétomètre. Seul l'accéléromètre est utilisé dans le cadre du projet, il est utilisé afin de détecter lorsque le sportif effectue un pas. Le composant est capable de proposer des échelles d'accélération linéaire de +/-2 à +/-16g, il peut communiquer soit sur le bus I^2C , ce qui est le cas sur la carte SODAQ One mais il peut aussi être connecté à un bus SPI. Il propose toute une série de registres qui permettent de configurer les différents paramètres de l'accéléromètre ainsi que du magnétomètre et de récupérer les valeurs d'accélération sur chaque axes. Il est aussi possible de configurer l'utilisation d'une FIFO, auquel cas plusieurs valeurs d'accélérations peuvent être stockées à la suite et lues lorsque nécessaire.

Enfin le module peut être configuré pour générer une interruption lorsqu'un certain seuil d'accélération est dépassé sur une combinaison des 3 axes afin de pouvoir détecter un certain mouvement ou déplacement par exemple.

4.1.5 La batterie

TODO:

4.2 Le système d'exploitation Zephyr

Zephyr project ou Zephyr est un système d'exploitation temps réel (RTOS) open source réalisé par la Linux Foundation. Il a été développé pour être utilisé sur des petits systèmes embarqués avec de grosses contraintes au niveau des ressources à disposition. A sa base il reprend un "micro" noyau développé par la société Wind River pour son système d'exploitation commercial VxWorks qui est employé dans beaucoup de projet dans les domaines aérospatial, militaire et automobile. Plusieurs architectures de micro-contrôleur sont pris en charge comme ARM, RISC ou x86 par exemple et plusieurs dizaines de configuration pour différentes cartes du marché

existent.

Ce RTOS est également très facile à configurer pour ses propres besoins au moyen de fichiers de configuration ou l'on peut sélectionner les éléments que l'on veut utiliser dans son application. De plus il propose toutes les fonctionnalités que l'on peut attendre de ce genre de système : scheduler, thread, semaphore, message queue, ring buffer, gestion de l'allocation de la mémoire dynamiquement... En plus de ces fonctions de base il dispose également de drivers pour piloter différents types de composants comme des UART, SPI, ADC ou GPIO par exemple. Enfin des couches réseaux tel que Ethernet, IPv6 ou Bluetooth sont disponibles ainsi que la gestion de système de fichier. [4]

Autour de cet RTOS il existe une importante communauté qui travail activement sur son développement ce qui permet de pouvoir avoir des réponses à ses questions rapidement et efficacement.

Ce système d'exploitation a été choisi car en plus d'être moderne il dispose déjà de la configuration d'une carte très similaire au SODAQ One, le Adafruit Feather M0 Basic Proto qui embarque le même micro-contrôleur, ceci facilitera passablement le travail de modifications pour adapter la configuration du RTOS au SODAQ One. Cette configuration consiste à définir, grâce à des Device Tree, quels sont les composants que la carte embarque et sur quels ports ou pins ils sont connectés, cela permet ensuite au système d'exploitation de pouvoir les piloter correctement.

Même si beaucoup d'éléments sont déjà existant pour le micro-contrôleur utilisé par le SODAQ One, le développement d'un driver I^2C a été réalisé car il n'en n'existe aucun au moment où j'ai commencé le travail de Bachelor. Ce driver est utilisé afin de pouvoir communiquer avec les modules GPS et accéléromètre. Les drivers développés dans le cadre du travail de diplôme sont décrits en détail dans la section 4.4.

La figure 4.2 présente l'architecture du système d'exploitation temps-réel Zephyr.

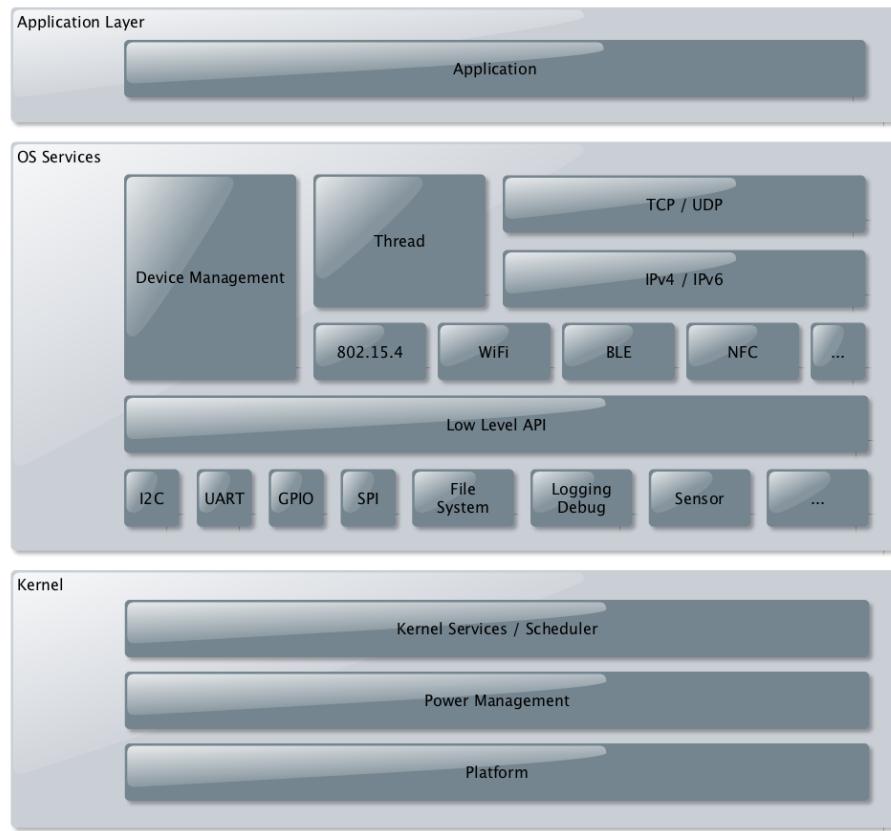


FIGURE 4.2 – Architecture du système d'exploitation Zephyr - <http://www.zephyrproject.org>

4.3 Le logiciel embarqué

Cette section décrit le logiciel embarqué sur le capteur dans son ensemble. En se basant sur les fonctionnalités proposées par Zephyr ainsi que les drivers, qui sont décrit dans la section 4.4, c'est lui qui va cadencer les acquisitions ainsi que l'envoie des paquets LoRa à la passerelle. Le logiciel embarqué ainsi que les drivers sont entièrement écrit en langage C. Il en va de même pour le système d'exploitation Zephyr qui contient cependant certaines parties écrites en assembleur.

La liste suivante présente toutes les acquisitions qui sont gérées par le logiciel.

- Position GPS message générée par le module à une fréquence de 1 Hz
- Rythme cardiaque, une interruption est déclenchée à chaque battement du cœur
- Comptage du nombre de pas, une interruption est déclenchée lorsqu'un pas est détecté par l'accéléromètre

De manière général et de façon simplifiée on peut définir le processus du capteur comme dans la figure 4.3.

4.3.1 Architecture logiciel

La figure 4.4 montre l'architecture statique du logiciel embarqué, tout les modules dont il est composé sont présentés.

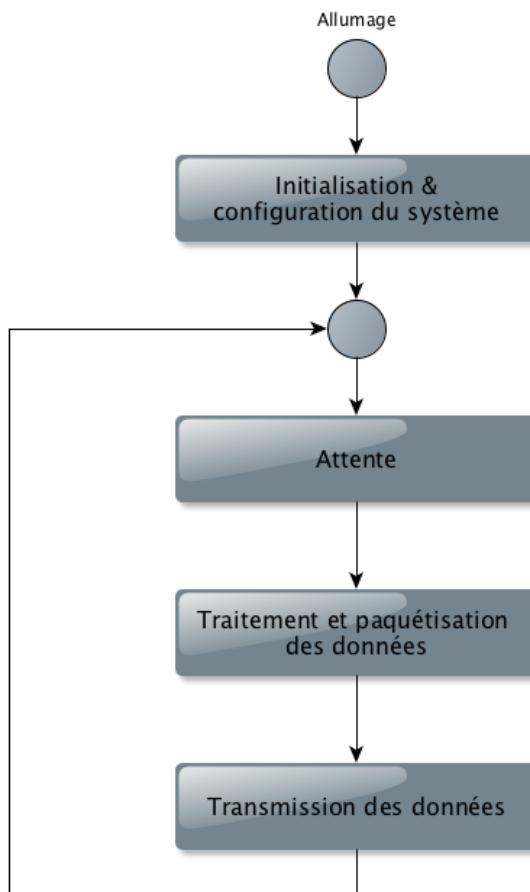


FIGURE 4.3 – Processus général du capteur

Les éléments qui compose la couche application du logiciel embarqué sont décrit en plus de détail dans la liste ci-dessous.

- Race Sensor Manager : C'est le module responsable de la gestion du capteur, au moyen d'un thread c'est lui qui déclenche les acquisitions et qui paquetise les données afin de les transmettre dans des paquets LoRa.
- Race Sensor Shell : Le shell du capteur, il est principalement utile pour les tests et le debug. Il propose plusieurs commandes qui permettent par exemple de modifier la configuration de la liaison radio LoRa.
- Debug : Ce module propose des fonctions qui facilite le debug de l'application.
- Heart Rate : Compte le nombre de battement du cœur par unité de temps en s'appuyant sur l'accéléromètre Adafruit Heart Rate starter pack
- Cadence : Permet le comptage des pas du sportifs par unité de temps grâce à l'accéléromètre LSM303AGR

La figure 4.5 décrit les éléments dynamiques qui compose le logiciel du capteur.

TODO: Update when finished

Zephyr propose une implémentation de la priorité des threads intéressante qui permet en fonction de la valeur d'également modifier le comportement de l'ordonnanceur. La priorité d'un thread, valeur entière, peut être soit positive ou négative. Une valeur de priorité plus petite est

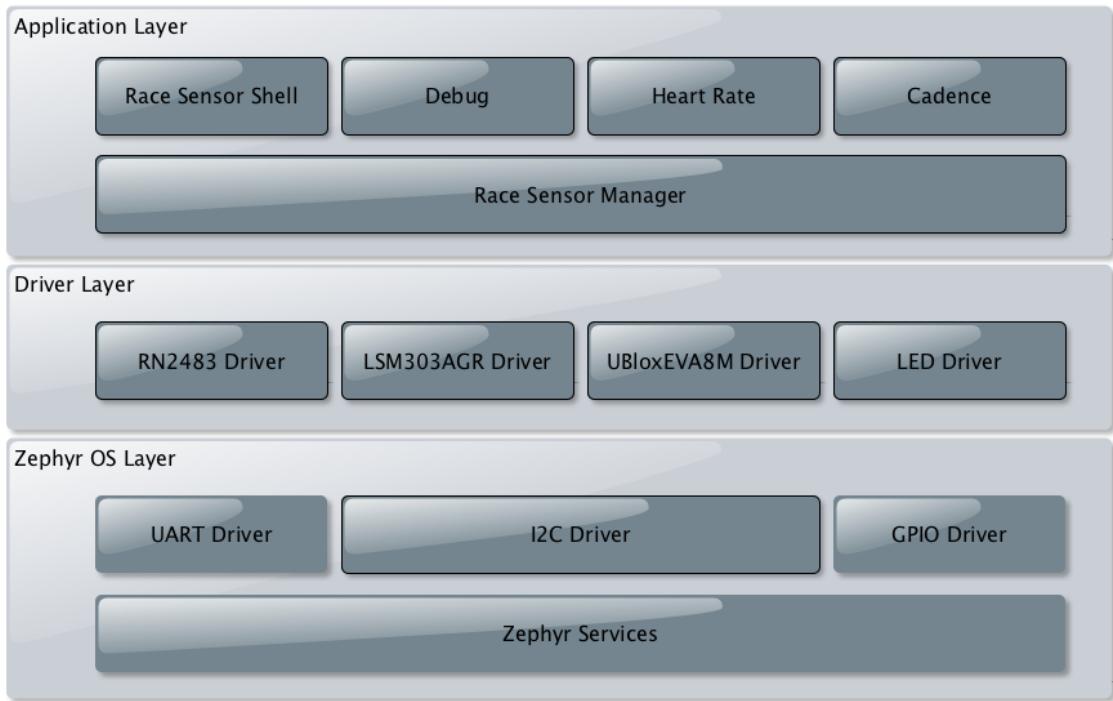


FIGURE 4.4 – Architecture statique du logiciel embarqué sur le capteur

plus importante qu'une valeur numériquement plus grande, ainsi un thread de priorité -2 est plus prioritaire qu'un de priorité 7.

En plus de cela, l'ordonnanceur distingue deux catégories de threads en utilisant la valeur de priorité. Des threads dit coopératif et des threads pré-emptible.

Les threads coopératif ont une valeur de priorité négative, une fois qu'ils deviennent le thread exécutant, il le reste jusqu'à ce qu'il effectue une action qui le mettrait dans l'état non prêt, c'est à dire l'acquisition d'un sémaphore qui est utilisé ou une attente par exemple.

Un threads pré-emptible a une valeur de priorité non négative. Lors de son exécution ce genre de thread peut à tout moment être mis dans l'état non prêt par l'ordonnanceur afin de permettre à un thread coopératif ou à un thread pré-emptible de priorité plus haute ou égale de s'exécuter. [4]

La table 4.2 décrit en plus de détail les caractéristiques de chaque threads.

TABLE 4.2 – Caractéristiques des threads du capteur

Thread	Priorité	Période
Race Sensor Manager	Haute - 0	15 s
RN2483 (LoRa)	1	100 ms
Driver UBloxEVA8M (GPS)	2	100 ms
Accéléromètre	Basse - 3	100 ms

Le thread le plus prioritaire est celui du gestionnaire du capteur, en effet étant donné qu'il a une période qui est longue, lorsqu'il est prêt il doit directement prendre la main sur les autres.

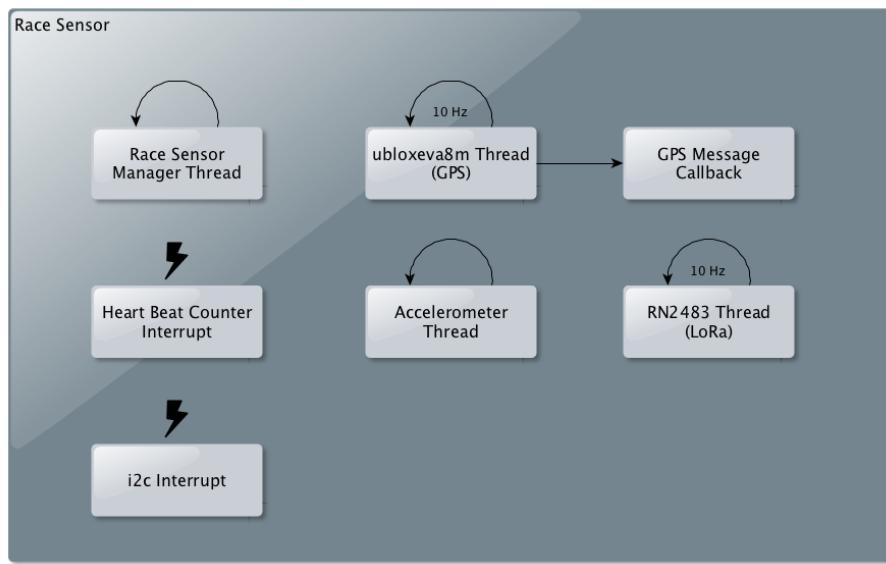


FIGURE 4.5 – Architecture dynamique du logiciel embarqué sur le capteur

threads afin de garantir la fréquence de transmission des données à la passerelle. Le thread responsable de la transmission des données au module LoRa doit également faire son travail en priorité afin de garantir le transfert des données rapidement. Enfin les thread de gestion du module GPS et de l'accéléromètre dispose de priorité plus faible car leur importance est moindre dans le processus du capteur.

4.3.2 Race Sensor Manager

Le Race Sensor Manager est le module principale du capteur, lors de l'allumage du capteur, c'est lui qui commence par configurer tous les éléments nécessaires au fonctionnement du capteur, comme les drivers par exemple. Une fois le système initialisé, le thread du Race Sensor Manager va s'occuper d'envoyer les données acquises dans des paquets LoRa à intervalle régulier.

La figure 4.6 présente le diagramme de classe du Race Sensor Manager.

Lors du lancement du firmware du capteur, le Race Sensor Manager est initialisé puis lancé grâce aux fonctions `race_sensor_mngr_init()` et `race_sensor_mngr_start()`. Durant l'initialisation du module tous les drivers sont initialisé et les interfaces de communications sont configuré, au terme de cette opération le thread est lancé qui s'occupera du séquencement des opérations. Les opérations d'initialisation sont décrite dans le diagramme de séquence 4.7.

Le thread va s'occuper de paquetiser et d'envoyer les données collectées régulièrement, le comportement est décrit dans le diagramme de séquence 4.8.

4.3.3 Heart Rate

TODO:

4.3.4 Cadence

TODO:

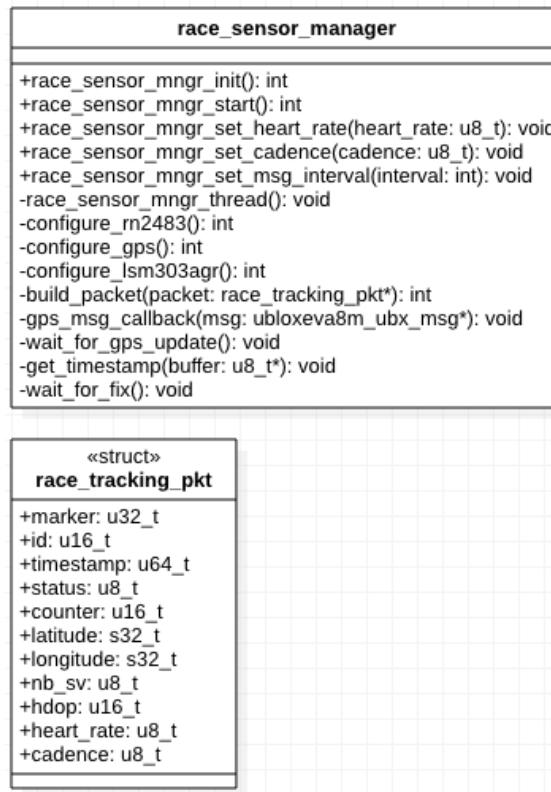


FIGURE 4.6 – Diagramme de classe du Race Sensor Manager

4.3.5 Debug

Ce module propose quelque fonction et macros qui sont utile pendant le debug du capteur comme l'affichage de message sur une console ou l'affichage de la liste de tous les threads qui sont actifs.

4.3.6 Race Sensor Shell

Le Race Sensor Shell permet l'utilisation d'un shell exécuté directement sur le capteur qui permet l'ajout de diverses commandes permettant la configuration du capteur pendant l'exécution du firmware. Il s'appuie sur le module shell proposé par Zephyr qui prend en charge la gestion du shell, il suffit de lui donner des pointeurs de fonction ainsi que des noms de fonction et le reste est pris en charge par le système d'exploitation.

La liste suivante montre les commandes disponible dans le shell du capteur :

- set_lora_sf : Permet de changer le facteur d'étalement de la couche radio LoRa
- set_lora_pwr : Permet de changer la puissance de sortie du signal de la couche radio LoRa
- set_msg_interval : Permet de modifier l'intervalle entre deux messages envoyé à la passerelle

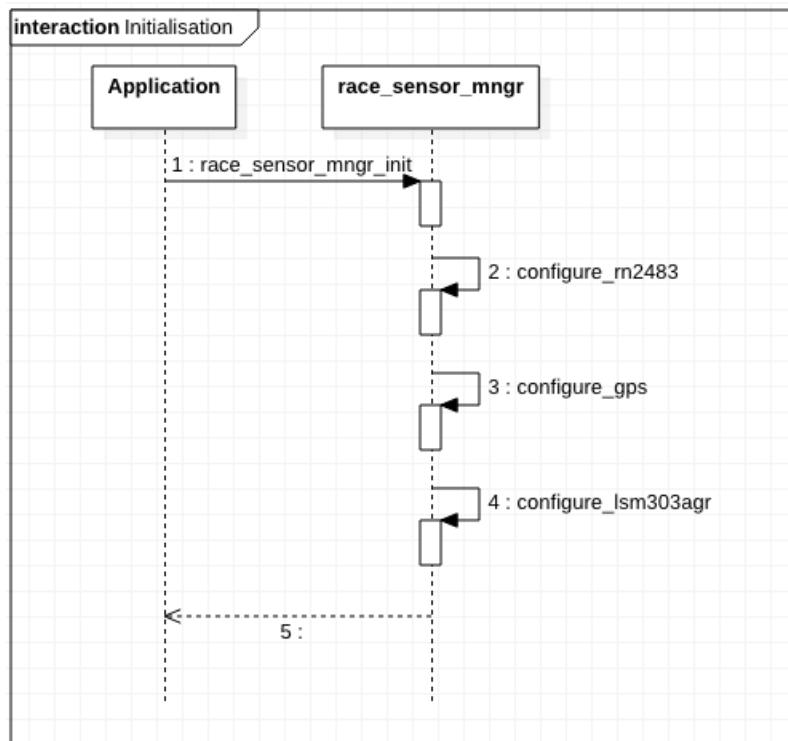


FIGURE 4.7 – Diagramme de séquence de l'initialisation du Race Sensor Manager

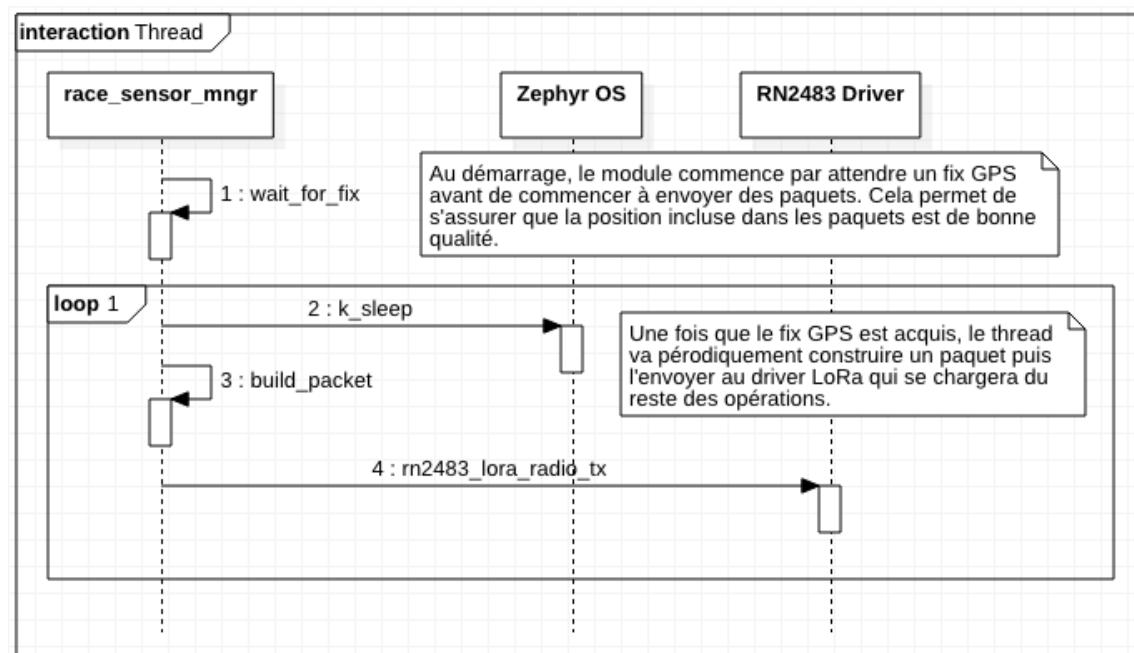


FIGURE 4.8 – Diagramme de séquence du thread du Race Sensor Manager

4.4 Les drivers

Afin de pouvoir utiliser tous les modules requis par le projet ainsi que le RTOS Zephyr il a été nécessaire d'écrire plusieurs drivers qui sont décrits dans cette section.

Les drivers présentés dans la liste suivante ont été développés dans le cadre du travail de Bachelor.

- Driver I^2C pour micro-contrôleur ATSAMD21G18 intégré au système d'exploitation Zephyr
- Driver UBloxEVA8M pour piloter le module GPS qui se base lui-même sur le driver I^2C
- Driver LSM303AGR pour le module accéléromètre qui se base également sur le driver I^2C
- Driver RN2483 LoRa permettant d'exploiter la communication LoRa et qui se base sur le driver UART existant de Zephyr
- Driver pour piloter les 3 LEDs de la carte SODAQ One au travers de GPIOs

4.4.1 Driver I^2C ATSAMD21G18

I^2C est un bus qui permet de connecter plusieurs esclaves à un maître. Le maître est le seul à initier les accès aux esclaves qui disposent chacun de leur adresse spécifique, lorsque le maître veut lire ou écrire sur un esclave il va envoyer un message I^2C contenant l'adresse de l'esclave en question, suivi des données à écrire ou de l'adresse à lire par exemple. Avantage de taille est qu'il ne nécessite que deux connections pour fonctionner, une qui permet la distribution du signal de synchronisation qui est uniquement contrôlé par le maître et une autre pour le transfert des données qui peut être contrôlé soit par le maître lors d'une écriture ou de l'esclave pour une lecture.

L'écriture de ce driver se base sur les informations disponibles dans la datasheet du micro-contrôleur ATSAMD21G18 [5]. Les composants SERCOM I^2C sont décrits à partir de la page 545. L'application note [6] a également été consultée.

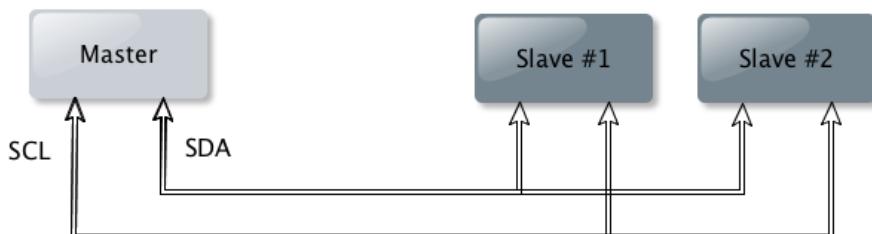
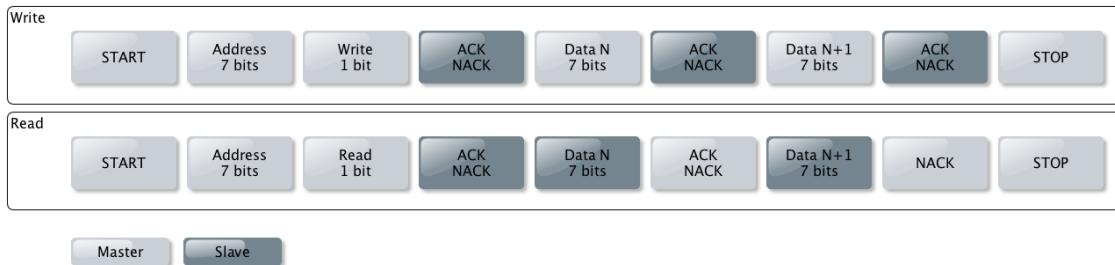


FIGURE 4.9 – Schéma d'un bus I^2C

Le bus I^2C utilise un protocole pour la communication entre un maître et un esclave, en fonction de l'opération voulue le message ne sera pas tout à fait le même. Le maître commence toujours par envoyer un START suivi de l'adresse de l'esclave à qui est destiné le message et puis enfin du type d'opération, lecture ou écriture. Les données sont ensuite placées sur le bus par le maître dans le cas d'une écriture ou par l'esclave lors d'une lecture. chose importante lorsque le maître a terminé le processus de lecture, il doit envoyé un message de type NACK suivi d'un STOP afin d'informer l'esclave de la fin du transfert. La figure 4.10 propose les deux types de messages qu'il existe.

FIGURE 4.10 – Les messages I^2C

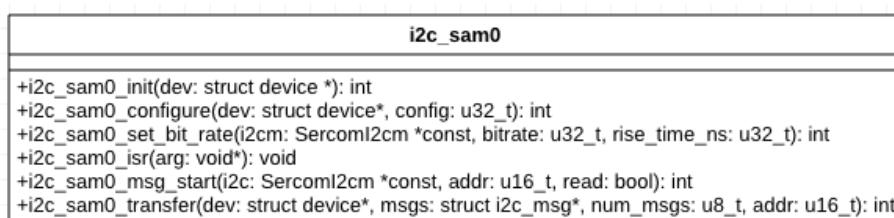
Le driver I^2C est responsable du pilotage des SERCOM que propose le micro-contrôleur, ce sont des modules qui peuvent être configuré afin de proposer une interface de type UART, I^2C ou SPI. Étant donné qu'il est intégré directement à Zephyr, il doit respecter les contraintes qui y sont liées c'est à dire d'être placé dans le dossier I^2C du système d'exploitation : zephyr/drivers/i2c/i2c_sam0.c et qu'il doit proposer une interface standardisée qui permet au système d'exploitation d'initier les opérations suivantes.

- Configuration de l'interface
- Transfert de donnée (Lecture/Écriture)

Pour se faire, il faut remplir une structure avec des pointeurs de fonctions qui sont ensuite appelés par le système d'exploitation au besoin. En plus de cela, il faut définir et configurer un device, c'est la structure utilisée par Zephyr qui sera ensuite utilisée par les applications afin de pouvoir communiquer sur le bus I^2C .

C'est le driver qui en fonction de l'opération à effectuer va envoyer les codes (START ou STOP) suivie des données. Il produira les acquittement nécessaire et Il vérifiera également que l'esclave en question valide bien les messages. L'utilisation d'une l'interruption permet au driver de savoir, après avoir écrit un message dans le SERCOM, quand il a été effectivement envoyé ceci afin de pouvoir initier l'envoie du prochain message.

La figure 4.11 présente le diagramme de classe du driver I^2C .

FIGURE 4.11 – Diagramme de class du driver I^2C

Un exemple d'utilisation du driver I^2C est disponible ci-dessous.

```

1 #include <i2c.h>
2
3 /* Address of the i2c slave */
4 #define I2C_DEVICE_ADDR 0x20
5
6 #define BUFFER_SIZE 128
7
8 void main(void) {

```

```
9  /* Get binding on i2c device */
10 struct device* i2c_dev = device_get_binding(CONFIG_I2C_SAM0_SERCOM3_LABEL);
11 /* i2c device configuration */
12 u32_t i2c_cfg = I2C_SPEED_SET(I2C_SPEED_FAST) | I2C_MODE_MASTER;
13 uint8_t msg[BUFFER_SIZE];
14
15 /* Check if binding succeeded */
16 if (!i2c_dev) {
17     DBG_PRNTK("%s: Binding to i2c failed\n", __func__);
18     return;
19 }
20 /* Configure I2C Device */
21 if (i2c_configure(i2c_dev, i2c_cfg)) {
22     DBG_PRNTK("%s: i2c configuration failed\n", __func__);
23     return;
24 }
25
26 msg[0] = 0x11;
27 msg[1] = 0x22;
28 msg[2] = 0x33;
29 msg[3] = 0x44;
30
31 /* Send message to i2c slave */
32 if (i2c_write(i2c_dev, msg, 4, I2C_DEVICE_ADDR)) {
33     DBG_PRNTK("%s: I2C access failed\n", __func__);
34 }
35
36 /* Read from i2c slave */
37 if (i2c_read(i2c_dev, msg, 4, I2C_DEVICE_ADDR)) {
38     DBG_PRNTK("%s: I2C access failed\n", __func__);
39     return 0;
40 }
41 }
```

4.4.2 Driver UBloxEVA8M

Le driver UBloxEVA8M permet le pilotage du module GPS du même nom qui se trouve connecté sur le bus I^2C . Ce module est intégré directement à la carte SODAQ One et propose une gestion GPS complète, il est capable de recevoir les signaux GPS, GLONASS, QZSS et SBAS, il est extrêmement sensible, très rapide et de petite taille, en somme une solution idéal pour des capteurs de petite taille. Il est décrit en plus détails à la section 4.1.1.

Le module utilise trois types de protocole pour la transmission des données et la configuration du module, NMEA, UPX et RTCM. NMEA est un protocole à base de texte qui envoie des messages formée de caractère ASCII il est idéale lorsque le module est connecté sur un UART. UPX est un protocole compact car il utilise des mots binaires qui sont protégés par des checksum. Enfin le protocole Radio Technical Commission for Maritime Services (RTCM) est unidirectionnel (seulement envoie vers le receveur) qui permet au receveur l'utilisation des corrections du positionnement relatif uniquement. Le driver utilise les messages du protocole UPX car il est le mieux adapté pour l'utilisation sur un bus I^2C , il est décrit en détail dans le document [2].

Une fois que le module a été configuré, il va produire les messages qui ont été activés et contenant diverses informations et les placé dans une FIFO. Au travers du bus I^2C , le thread du dri-

ver va interroger le module périodiquement afin de savoir si des messages sont prêt à être lu, si c'est le cas le driver va lire les messages disponible et les stocker dans une structure de type ubloxeva8m_ubx_msg. L'utilisateur récupère les messages reçu grâce à une fonction callback qui est passée au driver et qu'il appelle à chaque fois qu'un message est prêt.

Dans le cadre du projet, le message UBX-NAV-PVT est utilisé pour récupérer les informations nécessaire au projet, c'est à dire la position, l'évaluation de la précision de la position ainsi que les informations de temps qui servent à synchroniser le capteur. La liste suivante présente un résumé des informations contenue dans ce message, pour plus de détails voir [2, p. 307].

- La date du jour
- L'heure actuelle
- Le type de fix GPS
- La validité et la précision des informations contenus dans le message
- La latitude et longitude
- Le nombre de satellite actuellement vu
- La vitesse

En utilisant le driver I^2C décrit à la section précédente il va permettre l'initialisation, la configuration et la récupération des messages GPS du composant. Une fois le module GPS initialisé et configuré, le driver utilise un thread qui va périodiquement aller voir si de nouveaux messages sont présent dans la FIFO, si c'est le cas ils sont lus et ensuite décodé. Lorsque les messages sont prêt ils sont ensuite transférer à l'application au travers d'une fonction de callback.

La figure 4.12 présente le diagramme de classe du driver UBloxEVA8M.

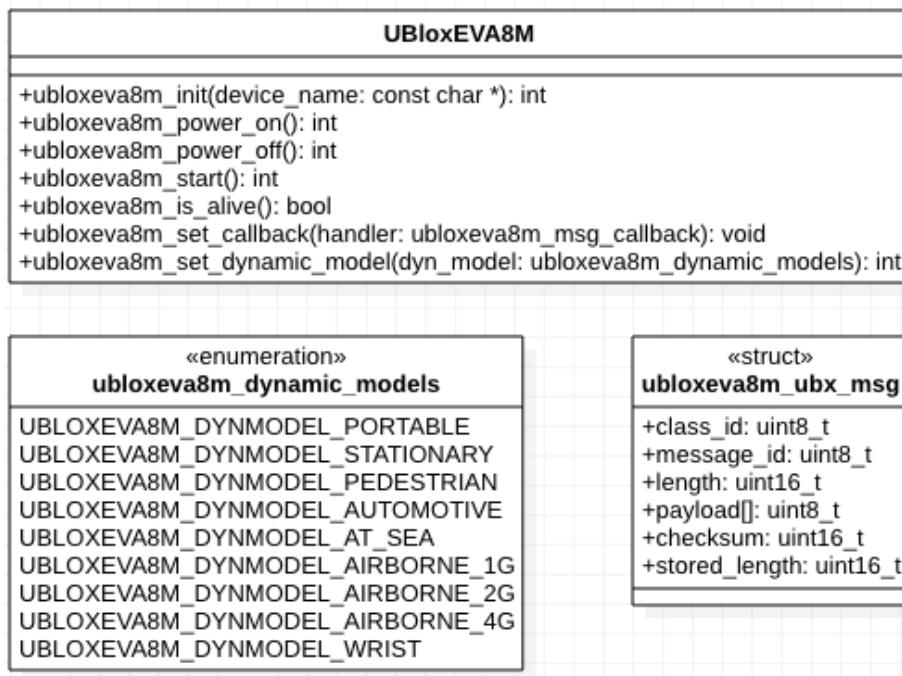


FIGURE 4.12 – Diagramme de class du driver UBloxEVA8M

Un exemple d'utilisation du driver UBloxEVA8M est proposée ci-dessous. Le principe d'utilisation et la configuration d'une fonction qui est appelé par le driver lorsqu'un nouveau message est reçu, une fonction callback. Dans cette fonction, l'utilisateur peut vérifier de quel type de

message il s'agit, dans le cas où le message l'intéresse il peut ensuite utiliser les structures permettant le décodage des différents messages.

```
1 #include "UBloxEVA8M.h"
2
3 /* Function to print a UBX-NAV-PVT message */
4 static void print_nav_pvt_msg(char *txt, ubloxeva8m_nav_pvt_t* msg) {
5     DBG_PRNTK("UBX-NAV-PVT: %d.%d.%d %02d:%02d:%03d validity=%d fixType=%d numSV=%d
6         lat=%d lon=%d \n", msg->day, msg->month, msg->year, msg->hour, msg->minute, msg->seconds
7         , msg->nano, msg->valid, msg->fixType, msg->numSV, msg->lat, msg->lon);
8 }
9
10 /* Callback function on GPS message */
11 static void gps_msg_callback(ubloxeva8m_ubx_msg* msg)
12 {
13     ubloxeva8m_nav_pvt_t pvt_msg;
14
15     /* Check message type */
16     if (msg->class_id == UBLOXEVA8M_CLASS_NAV && msg->message_id == UBLOXEVA8M_MSG_NAV_PVT)
17     {
18         /* Copy message in struct */
19         memcpy(&pvt_msg, msg->payload, sizeof(ubloxeva8m_nav_pvt_t));
20         print_nav_pvt_msg("UBX-NAV-PVT: ", &pvt_msg);
21     }
22 }
23
24 void main(void)
25 {
26     int err;
27
28     /* Initialize the gps */
29     err = ubloxeva8m_init(CONFIG_I2C_SAM0_SERCOM3_LABEL);
30     if (err) {
31         DBG_PRNTK("%s: Can't initialize UBloxEVA8M %d\n", __func__, err);
32         return;
33     }
34
35     /* Set callback function */
36     ubloxeva8m_set_callback(gps_msg_callback);
37
38     /* Start module */
39     err = ubloxeva8m_start();
40     if (err) {
41         DBG_PRNTK("%s: Can't start GPS module %d\n", __func__, err);
42         return;
43     }
44
45     /* Set the dynamic model used by the GPS */
46     err = ubloxeva8m_set_dynamic_model(UBLOXEVA8M_DYNMODEL_AUTOMOTIVE);
47     if (err) {
48         DBG_PRNTK("%s: Can't set dynamic model %d\n", __func__, err);
49         return;
50     }
51 }
```

4.4.3 Driver LSM303AGR

Le driver LSM303AGR est le moyen de communiquer avec le module accéléromètre et magnétomètre qui est connecté au bus I^2C . Dans le cadre du projet seule la partie qui gère l'accéléromètre a été développée puisque le magnétomètre n'est pas utilisé. Le composant est décrit en détail dans le datasheet associé [7].

Le composant LSM303AGR est configuré au travers d'une liste de registre, il est également possible de lire les différentes valeurs mesurer au travers d'un autre groupe de registre. Tout les registres disponible sont détaillés dans le document [7, p. 43].

Puisque le composant LSM303AGR se trouve placé sur le bus I^2C , ce driver utilise également le driver pour ce bus afin de pouvoir communiquer avec le module. La majorité des opérations consiste à écrire ou lire des valeurs dans les différents registres à disposition afin de configurer ou de récupérer les données voulues. Il est à noté qu'il est également possible de configurer le module afin qu'il produise des interruptions lorsque certains seuils sont dépassés sur un certain axe. Cette fonctionnalité est utilisé afin de détecter les pas effectuer par le porteur de capteur.

TODO: Revoir diagramme après implémentation finale

La figure 4.13 présente le diagramme de classe du driver LSM303AGR.

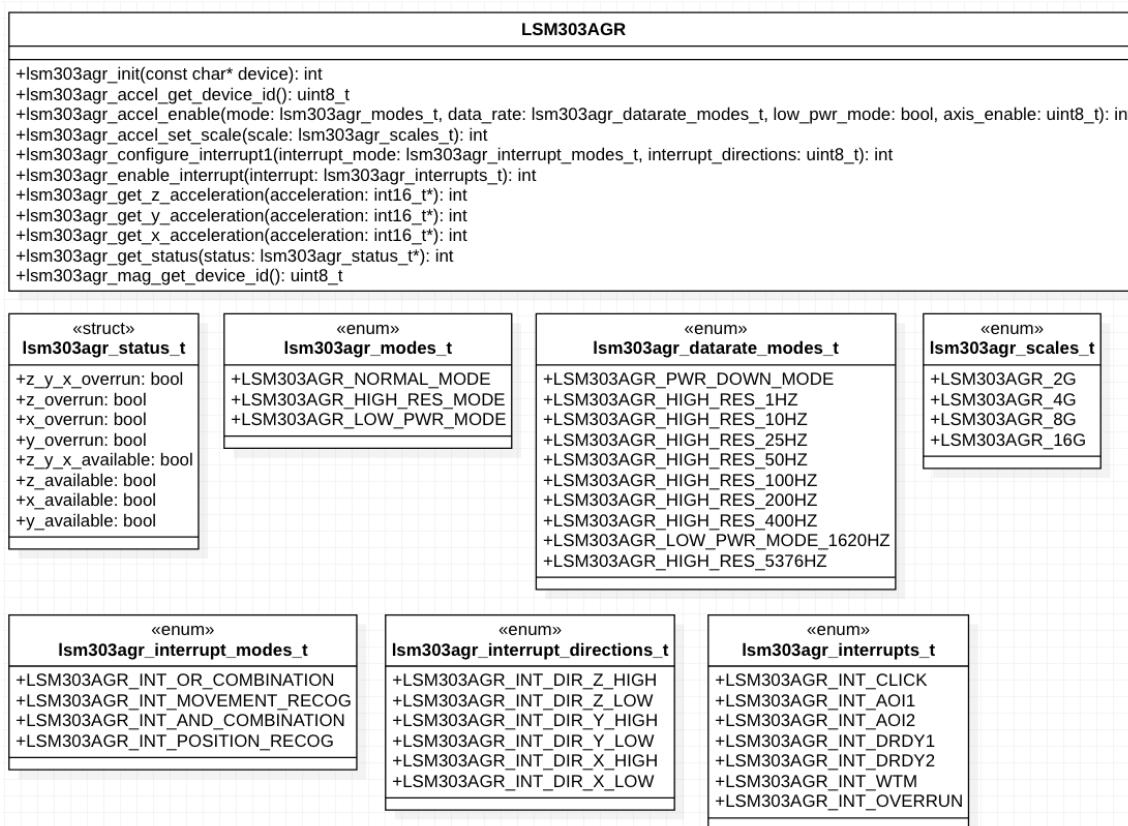


FIGURE 4.13 – Diagramme de class du driver LSM303AGR

Un exemple d'utilisation du driver est proposée ci-dessous.

1 #include "LSM303AGR.h"

2

```
3 void main(void)
4 {
5     int err;
6     lsm303agr_status_t lsm_status;
7     int16_t accel_x;
8     int16_t accel_y;
9     int16_t accel_z;
10
11    /* Initialize the LSM303AGR module */
12    err = lsm303agr_init(CONFIG_I2C_SAM0_SERCOM3_LABEL);
13    if (err) {
14        DBG_PRNTK("%s: Can't init LSM303AGR %d\n", __func__, err);
15        return;
16    }
17
18    /* Enable the accelerometer */
19    err = lsm303agr_accel_enable(LSM303AGR_NORMAL_MODE, LSM303AGR_HIGH_RES_100HZ, false, (
20        LSM303AGR_Z_AXIS | LSM303AGR_Y_AXIS | LSM303AGR_X_AXIS));
21    if (err) {
22        DBG_PRNTK("%s: Can't enable accelerometer %d\n", __func__, err);
23        return;
24    }
25
26    /* Set accelerometer scale */
27    if (lsm303agr_accel_set_scale(LSM303AGR_8G)) {
28        DBG_PRNTK("Couldn't set accelerometer scale\n");
29        return;
30    }
31
32    /* Get accelerometer status */
33    if (lsm303agr_get_status(&lsm_status)) {
34        DBG_PRNTK("Couldn't get status\n");
35        return;
36    }
37
38    /* Get current acceleration on x axis */
39    if (lsm303agr_get_x_acceleration(&accel_x)) {
40        DBG_PRNTK("Couldn't get acceleration\n");
41        return;
42    }
43
44    /* Get current acceleration on y axis */
45    if (lsm303agr_get_y_acceleration(&accel_y)) {
46        DBG_PRNTK("Couldn't get acceleration\n");
47        return;
48    }
49
50    /* Get current acceleration on z axis */
51    if (lsm303agr_get_z_acceleration(&accel_z)) {
52        DBG_PRNTK("Couldn't get acceleration\n");
53        return;
54    }
```

TODO: Add setup of IRQ

4.4.4 Driver RN2483

Le driver RN2483 permet de piloter le composant du même nom qui permet la communication radio LoRa. Le module est connecté au micro-contrôleur par un UART et il est piloté grâce à un protocole de commande texte qui permette d'effectuer les différentes opérations requises pour l'envoie de donnée.

Le protocole est détaillé dans le document Command Reference User's Guide [3].

Grâce au driver il est possible à l'utilisateur d'effectuer certaines opérations, le driver se charge de créer les bonnes commandes et de les envoyer sur l'UART. Une fois la commande exécutée par le module il vérifie également le bon acquittement de celle-ci.

Il est aussi possible de gérer la couche MAC LoRaWAN grâce au composant, puisque cette couche n'est pas utilisée dans le cadre du projet, elle est désactivée au moyen de la fonction rn2483_lora_pause_mac() et seule la couche radio LoRa est utilisée pour envoyer des messages, cela permet de simplifier grandement la gestion du protocole de communication.

La figure 4.14 présente le diagramme de classe du driver RN2483.

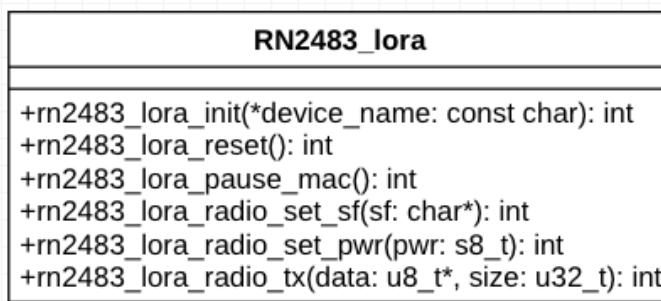


FIGURE 4.14 – Diagramme de class du driver RN2483

Un exemple d'utilisation du driver est proposée ci-dessous.

```

1 #include "RN2483.h"
2
3 /**
4 * LoRa spreading factor
5 * (Between sf7 and sf12)
6 */
7 #define LORA_SPREADING_FACTOR "sf7"
8
9 /**
10 * LoRa power output
11 */
12 #define LORA_POWER_OUTPUT 1
13
14 #define BUFFER_SIZE 56
15
16 void main(void)
17 {
18     int err;
19     u8_t buffer[BUFFER_SIZE];
20
21     /* Initialize the UART */

```

```

22     err = rn2483_lora_init(CONFIG_UART_SAM0_SERCOM2_LABEL);
23     if (err) {
24         DBG_PRNTK("%s: Can't init RN2483 %d\n", __func__, err);
25         return;
26     }
27
28     /* Pause mac layer */
29     err = rn2483_lora_pause_mac();
30     if (err) {
31         DBG_PRNTK("%s: Can't pause mac layer %d\n", __func__, err);
32         return;
33     }
34
35     /* Set spreading factor */
36     err = rn2483_lora_radio_set_sf(LORA_SPREADING_FACTOR);
37     if (err) {
38         DBG_PRNTK("%s: Can't set spreading factor %d\n", __func__, err);
39         return;
40     }
41
42     /* Set power output */
43     err = rn2483_lora_radio_set_pwr(LORA_POWER_OUTPUT);
44     if (err) {
45         DBG_PRNTK("%s: Can't set power output %d\n", __func__, err);
46         return;
47     }
48
49     /* Send data */
50     buffer[0] = 0x11;
51     buffer[1] = 0x22;
52     buffer[2] = 0x33;
53     buffer[3] = 0x44;
54
55     if (rn2483_lora_radio_tx(buffer, 4) {
56         DBG_PRNTK("Couldn't send packet\n");
57         return;
58     }
59 }
```

4.4.5 Driver LEDs

La carte SODAQ One est équipée de 3 LEDs, une rouge, une verte, une bleue, qui sont connectées à des GPIOs. Ce driver, très simple, permet d'abstraire la gestion des GPIO et propose une interface facilitant la gestion des LEDs.

La figure 4.15 présente le diagramme de classe du driver LED.

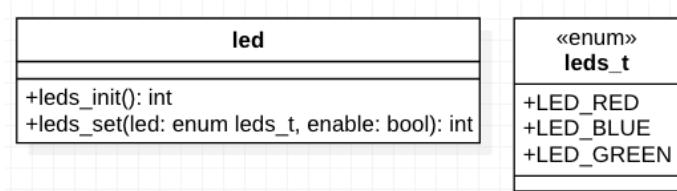


FIGURE 4.15 – Diagramme de classe du driver LED

```
1 #include "led.h"
2
3 void main(void)
4 {
5     int err;
6
7     err = leds_init();
8     if (err) {
9         DBG_PRINK(">%s: Cannot initialize LEDs\n", __func__);
10        return;
11    }
12
13    while(1) {
14        /* Switch-on LED */
15        leds_set(LED_RED, true);
16        /* Delay */
17        k_sleep(K_SECONDS(1));
18        /* Switch-off LED */
19        leds_set(LED_RED, false);
20        /* Delay */
21        k_sleep(K_SECONDS(1));
22    }
23 }
```

4.5 Paquet de donnée

TODO:

4.6 Le déboggage

TODO:

4.7 Le boîtier

TODO:

5 Description de la passerelle

La passerelle est chargée du traitement des paquets reçus par les capteurs et du stockage de ces données dans la base de données.

Elle se compose de deux parties physiques, d'une part un module de gestion de la communication LoRa, le concentrateur, qui se charge de la réception des paquets radio LoRa. L'autre partie est le micro-ordinateur, lorsqu'un paquet de données est reçu par le concentrateur, l'ordinateur de traitement le récupère puis se charge de décoder les données pour ensuite les stocker.

La passerelle utilise une distribution du système d'exploitation Linux sur lequel sont exécutés deux programmes : le packet forwarder et le serveur d'application.

Le packet forwarder est un logiciel qui est disponible sur internet gratuitement, son but est de récupérer les messages LoRa reçus par le concentrateur et de transformer les paquets en objets de type json qui sont ensuite transmis au travers d'un socket sous la forme d'un paquet UDP.

Le serveur d'application qui a été développé spécialement pour le travail de Bachelor, est le programme qui se charge du traitement des paquets UDP envoyés par le packet forwarder. C'est lui qui est responsable d'extraire les données des objets json et de les stocker dans la base de données.

Afin de pouvoir respecter les contraintes de temps du TB, une seule passerelle sera assemblée pour le projet.

La figure 5.1 présente le schéma block de la passerelle.

5.1 Le matériel

L'ordinateur de traitement des paquets LoRa se base sur le micro-ordinateur très connu Raspberry Pi. Il dispose de toutes les ressources nécessaires pour les besoins du travail de Bachelor, de plus la communauté et la documentation à son sujet est très développée.

Pour rappel les caractéristiques du Raspberry Pi sont résumées dans la table 5.1.

TABLE 5.1 – Caractéristiques du Raspberry Pi 3 Model B+

Dimensions	85mm x 49mm
Microcontrôleur	Broadcom BCM2837B0 – Cortex-A53 64-bit
Oscillateur	1.4 Ghz
Stockage	Carte SD
RAM	1 GB SDRAM
WiFi	802.11 b/g/n/ac
Prix	34.50 CHF

Pendant la pré-étude deux concentrateurs différents avaient été étudiés, le choix final s'est porté sur la solution d'un concentrateur moins coûteux, le Dragino LoRa HAT. C'est un concentrateur de type simple canal, c'est à dire qu'il n'est capable d'écouter qu'un seul canal de fréquence à

la fois, ce qui convient parfaitement pour un prototype comme celui développé pour le projet puisque un seul capteur sera assemblé. Le Dragino LoRa HAT est un module d'extension pour la Raspberry Pi, il est conçu pour se fixer au dessus de lui facilement. En plus de la gestion de la couche radio LoRa le module propose également un module GPS qui pourrait se rendre utile afin de pouvoir déterminer la position des passerelles, cet axe pourrait être étudié d'avantage pour le développement d'un produit.

Pour rappel les caractéristiques du Dragino Hat sont résumées dans la table 5.2.

TABLE 5.2 – Caractéristiques du Dragino LoRa Hat

Dimensions	60mm x 53mm x 25mm
LoRa	SX1276
Type de passerelle	Simple canal
Prix	38.90 CHF

Le Raspberry Pi et le Dragino LoRa HAT communiquent au travers d'un bus de type SPI. La gestion de la communication est entièrement gérée par le logiciel packet forwarder détaillé dans la chapitre 5.2.

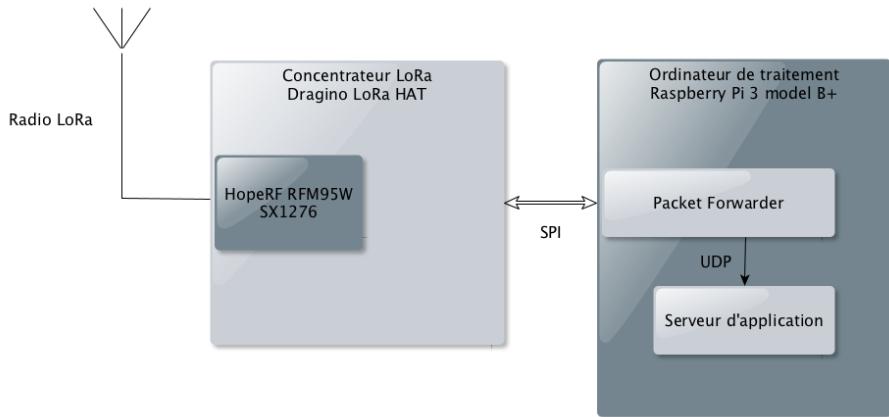


FIGURE 5.1 – Schéma block de la passerelle

TODO: Figure RPI + Dragino

5.2 Le packet forwarder

Le packet forwarder est le logiciel qui se place en amont du serveur d'application, il est en charge de la gestion du Dragino LoRa HAT, c'est à dire qu'il va régulièrement interroger le module d'extension pour savoir si de nouveaux paquets sont disponibles, si c'est le cas, le packet forwarder va les récupérer en analyser le contenu afin de pouvoir créer un objet json contenant toutes les informations. En plus de cela il va aussi rajouter des meta-données, comme le temps de réception du paquet ainsi que la fréquence et le facteur d'étalement par exemple.

Une fois les données transformé en json, le tout est envoyé au moyen d'un paquet UDP à un serveurs, dans notre cas le serveur d'application. La liste suivante décrit les informations contenu dans le paquet.

- La version du protocole utilisée
- Un jeton aléatoire utilisé pour marquer les paquets
- Un identifiant du type de message
- Un identifiant unique de la passerelle
- L'objet json

Pour les paquets de type PUSH_DATA, qui sont les seuls paquets utilisés dans le cadre du projet et qui sont créé pour les données du flux montant (noeud -> passerelle), l'objet json contient un tableau nommé rxpk. Chaque élément du tableau peut contenir les informations suivantes.

- time : Heure UTC à la réception du paquet
- tmms : Temps GPS à la réception du paquet (nombre de ms depuis le 6 janvier 1980)
- tmst : Temps interne de l'événement "RX finished"
- freq : La fréquence centrale en Mhz à la réception
- chan : Le canal de réception
- rfch : La chaîne radio fréquence utilisée pour la réception
- stat : Status du CRC du paquet (1 = OK, -1 = NOK, 0 = Pas de CRC)
- modu : Modulation LORA ou FSK
- datr : Le taux de transfert LoRa (par exemple SF12BW500, facteur d'étalement 12, largeur de bande 500Mhz)
- codr : Identifiant du taux LoRa ECC
- rssi : RSSI (Received Signal Strength Indication) en dBm
- lsnr : SNR (Signal to Noise Ratio) LoRa en dB
- size : La taille en byte de la charge utile du paquet radio LoRa
- data : La charge utile du paquet encodée en Base64

Un exemple d'objet json envoyé par le packet forwarder est présenté ci-dessous.

TODO: Mettre vrai exemple

```

1  {
2      "rxpk": [
3          {
4              "time": "2013-03-31T16:21:17.528002Z",
5              "tmst": 3512348611,
6              "chan": 2,
7              "rfch": 0,
8              "freq": 866.349812,
9              "stat": 1,
10             "modu": "LORA",
11             "datr": "SF7BW125",
12             "codr": "4/6",
13             "rssi": -35,
14             "lsnr": 5.1,
15             "size": 32,
16             "data": "-DS4CGaDCdG+48eJNM3Vai-zDpsR71Pn9CPA9uCON84"
17         }
18     ]
}
```

Le protocole est décrit en grand détails sur la page github du packet forwarder de Semtech voir [8].

A la base le packet forwarder a été développé par la société Semtech, tenante de la patente du protocole de communication LoRa, c'est également cette société qui a défini le format des

strings json envoyées dans les paquets UDP. Cependant la version qui est utilisée dans le cadre du projet de Bachelor, est un fork sur lequel plusieurs personnes ont travaillé afin de rajouter un certain nombre de fonctionnalité comme l'utilisation de fichiers de configuration ou le support de concentrateurs divers. Les principaux acteurs du développement de ce logiciel sont la société Semtech, Thomas Telkamp, Charles Hallard et Julien Le Sech. C'est le fork de Charles Hallard qui est employé par le projet car il supporte le Dragino LoRa HAT, il est disponible gratuitement sur github. [9]

Les tâches en lien avec le packet forwarder sont très simples, il s'agit d'abord de cloner le repository git, de compiler le programme puis de configurer le packet forwarder au moyen d'un fichier json, principalement pour sélectionner la fréquence et le facteur d'étalement sur lequel il doit écouter ainsi que l'adresse et le port du serveur auquel on souhaite envoyer les paquets UDP générés. On peut ensuite exécuter simplement le packet forwarder et dès la réception de paquets ils seront automatiquement transférés.

5.3 Le serveur d'application

Le serveur d'application est le logiciel principal de gestion de la passerelle. Au démarrage il se connecte à un port du packet forwarder, ce qui lui permet ensuite de recevoir les paquets de données LoRa sous la forme d'objet json. D'autre part il va également gérer la connexion à la base de données afin de pouvoir y sauver les informations qui auront été extraites des paquets.

Afin de rendre le serveur d'application plus flexible, un shell est intégré au programme, ce qui permet d'exécuter diverses commandes pendant son exécution afin d'acquérir des informations sur l'état du serveur ou d'écrire toutes les positions GPS acquises dans un fichier par exemple afin d'aider durant le debug du système.

Le serveur d'application est écrit en C++ et s'exécute sur le système d'exploitation Linux.

5.3.1 Architecture logiciel

L'architecture logiciel du serveur d'application se compose de 3 couches différentes, la couche application, la couche paquet LoRa et la couche outils. Le serveur d'application peut fonctionner en deux modes, race ou test. Le mode course est le mode standard, dans ce mode le serveur d'application récupère les paquets envoyés par le capteur, extrait les informations et les stocke dans la base de données. Dans le mode test, les paquets sont récupérés mais ne sont stockés que localement afin de pouvoir faire différents tests sur le système. Il est possible de changer de mode en utilisant le shell du serveur d'application.

La figure 5.2 présente l'architecture statique du serveur d'application.

Les différents modules qui composent le serveur d'application sont résumés dans la liste suivante.

- Race App Server : C'est la classe principale du serveur d'application, c'est elle qui réceptionne les paquets en provenance du packet forwarder et qui déclenche la chaîne de gestion des paquets
- Race Tracker Data : L'interface entre la base de données et le serveur d'application permet l'exécution de requête SQL grâce à la librairie pqxx



FIGURE 5.2 – Architecture statique du serveur d'application

- Race Mode Handler : Gestionnaire du mode "Race", une fois que les données du paquet reçu sont extraites c'est cette classe qui stock les données dans la base
- Race Mode Record : L'enregistrement du mode "Race", cette classe contient les données extraites du paquet
- Test Mode Handler : Gestionnaire du mode "Test", vérifie que les paquets reçu se suivent et garde ces informations en mémoire
- Test Mode Record : L'enregistrement du mode "Test", contient les données extraites du paquet test
- LoRa Packet Forwarder Parser : Parse les données brutes reçues depuis le socket et permet de savoir si le paquet est de type PUSH_DATA
- LoRa Push Data Parser : Lorsque le paquet reçu est de type PUSH_DATA, c'est cette classe qui extrait les informations du paquet, dont l'objet json qui nous intéresse
- LoRa rxpk Parser : Une fois le paquet PUSH_DATA parse, c'est cette classe qui s'occupe d'extraire les informations de l'objet json envoyé dans le paquet et qui est ensuite traité par les classes Race Mode Handler ou Test Mode Handler
- base64 : Une classe développée par la société Semtech qui permet le décodage de chaînes de caractères encodées en base64
- rapidjson : Permet la manipulation d'objets de type json, développé par la société Tencent
- vector reader : Une classe qui permet la lecture de données depuis un vecteur d'octets
- shell & shell command : Le module shell qui permet la création de commandes, leur gestion et exécution
- logger : Une classe permettant de logger des messages de criticité différentes

TODO: Add lib extern to biblio with url

Le serveur d'application est composé de deux threads qui sont gérés par le système d'exploitation, le premier est responsable de la réception des paquets du packet forwarder, c'est à dire qu'il attend sur un socket que des données soient disponibles. L'autre thread est utilisé pour le

shell, il attend une entrée au clavier et l'exécute.

Les threads du serveur d'application sont gérés par la librairie standard pthread, ils n'ont pas de priorité spécifique et sont de type asynchrone, c'est à dire que tout deux attendent un événement précis avant de se débloquer.

La figure 5.3 montre l'architecture dynamique du serveur d'application.

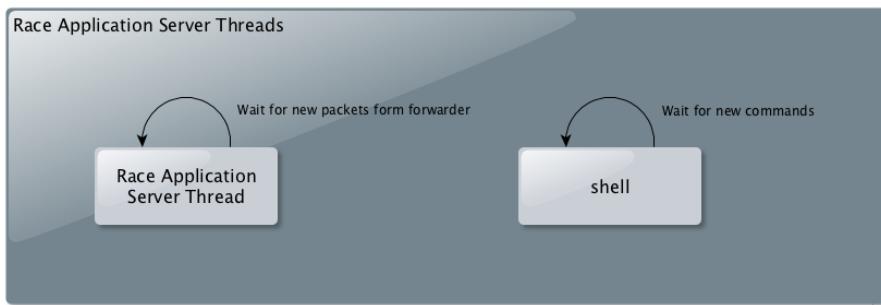


FIGURE 5.3 – Architecture dynamique du serveur d'application

5.3.2 Les librairies externes

TODO: Ajout petit exemple code pour chaque

Le serveur d'application utilise plusieurs librairies externes qui sont décrites dans cette section.

La librairie pqxx est la librairie officielle qui permet d'exécuter simplement des requêtes PostgreSQL sur des bases de données de type relationnelle. Elle est open source, multi-plateforme et disponible gratuitement avec une licence BSD sur internet à l'adresse <http://pqxx.org/development/libpqxx/>.

base64 est une classe écrite par la société Semtech également proposée en open source avec une licence de type revised BSD. https://github.com/Lora-net/packet_forwarder/blob/master/lora_pkt_fwd/src/base64.c

La librairie rapidjson est un parser et générateur de chaîne JSON rapide et efficient qui propose une API de style SAX/DOM. Elle est développée par la société Tencent et open source. Disponible à l'adresse <http://rapidjson.org/>.

5.3.3 Les classes

Cette section décrit les classes développées dans le cadre du serveur d'application.

Plus d'informations sur les classes sont disponibles en annexe de ce document dans la documentation Doxygen. **TODO: doxygen**

Race App Server

La classe Race App Server est la classe centrale du serveur d'application, lors de son initialisation elle crée un socket et le configure puis un thread. Lorsqu'elle reçoit un paquet du packet

forwarder elle le parse puis le gère en utilisant les autres classes. Lorsque le nouveau paquet est géré, elle attend le suivant.

La figure 5.4 montre le diagramme de séquence de la classe.

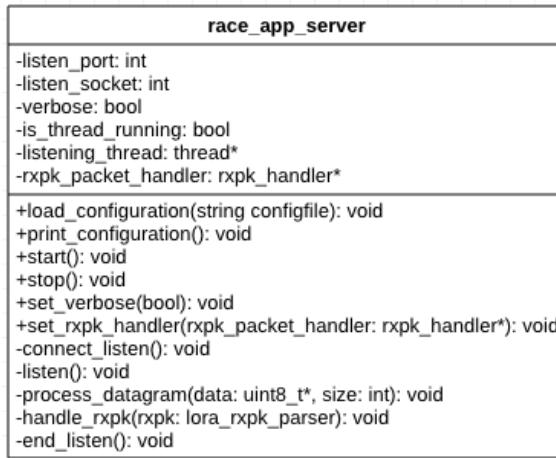


FIGURE 5.4 – Diagramme de classe de Race App Server

Les opérations effectué pour traiter un nouveau paquet reçu sont décrites dans le diagramme de séquence 5.5.

Race Tracker Data

L'interface entre la base de données et le serveur d'application est géré par la classe Race Tracker Data. Par le biais de librairie pqxx, elle va effectuer les requêtes SQL nécessaire afin de stocker une nouvelle position.

La principale fonction de cette classe est de transformer une classe de type Race Mode Record en requête SQL et d'exécuter la requête afin de stocker les informations dans la base de données.

La figure 5.6 montre le diagramme de classe de Race Tracker Data.

rxpk Handler

La classe rxpk Handler est une classe abstraite qui permet de définir la méthode nécessaire qui permet de faire la gestion d'un paquet reçu. Les classes Race Mode Handler et Test Mode Handler héritent toutes deux de la classe rxpk Handler.

La figure 5.7 montre le diagramme de classe de rxpk Handler.

Race Mode Handler & Record

Comme expliqué précédemment le serveur d'application peut fonctionner en deux modes, les classes Race Mode Handler & Race Mode Record gèrent le mode "Race". Dans ce mode les paquets reçus sont analysés, les données extraites puis stockées dans la base. La classe Race Mode Handler reçoit une instance de LoRa rxpk Parser contenant toutes les données brutes envoyées par le capteur, au moyen de la classe Vector Reader elle va extraire les différents champs et créer une

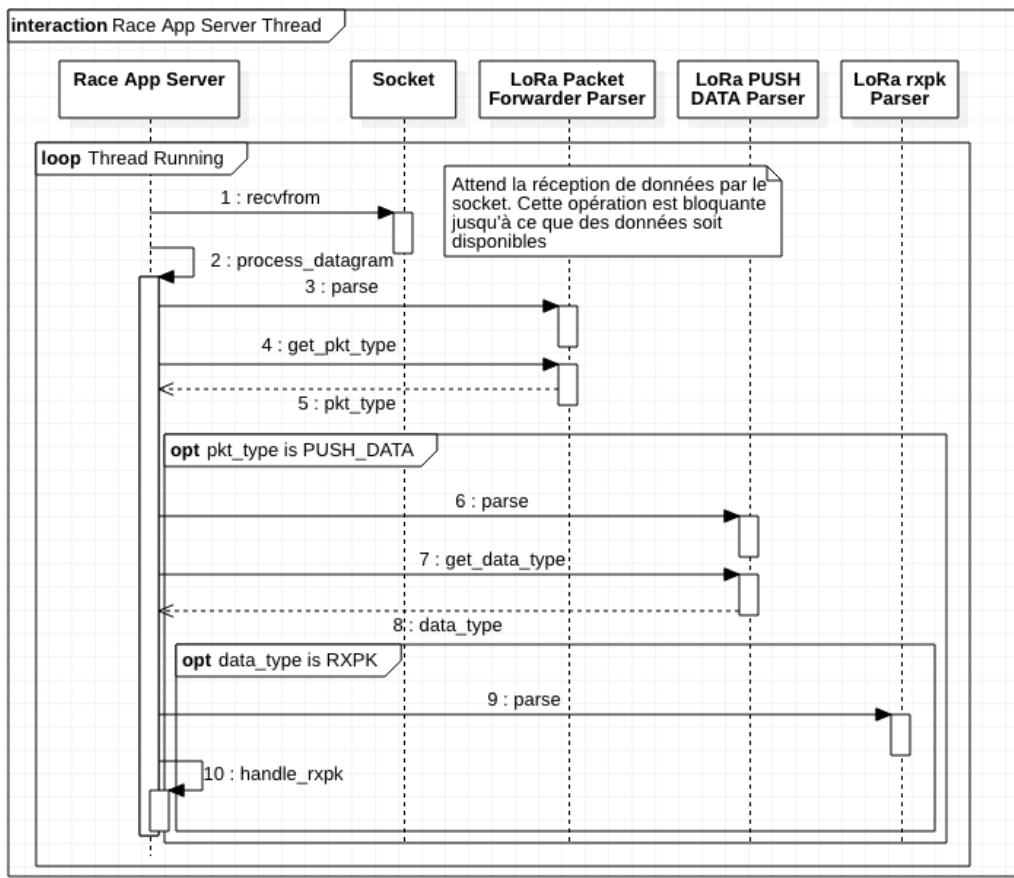


FIGURE 5.5 – Diagramme de séquence des opérations du thread de la classe Race App Server

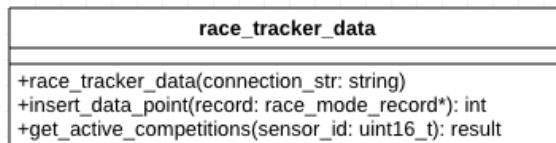


FIGURE 5.6 – Diagramme de classe de Race Tracker Data

instance de Race Mode Record contenant toutes les données décodées, ensuite grâce à la classe Race Tracker Data le tout est enregistré dans la base de données.

La figure 5.9 montre le diagramme de classe de Race Mode Handler et Race Mode Record.

Test Mode Handler & Record

Les classes Test Mode Handler et Test Mode Record permettent la gestion du mode "Test". Ce mode utilise un format de paquet différent du mode "Race" qui est décrit dans le chapitre 8 et contrairement au mode "Race" il ne sauvegarde pas les données ainsi reçus dans la base de données, mais uniquement localement en mémoire et si demandé par l'utilisateur dans un fichier.

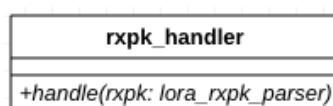


FIGURE 5.7 – Diagramme de classe de rxpk Handler

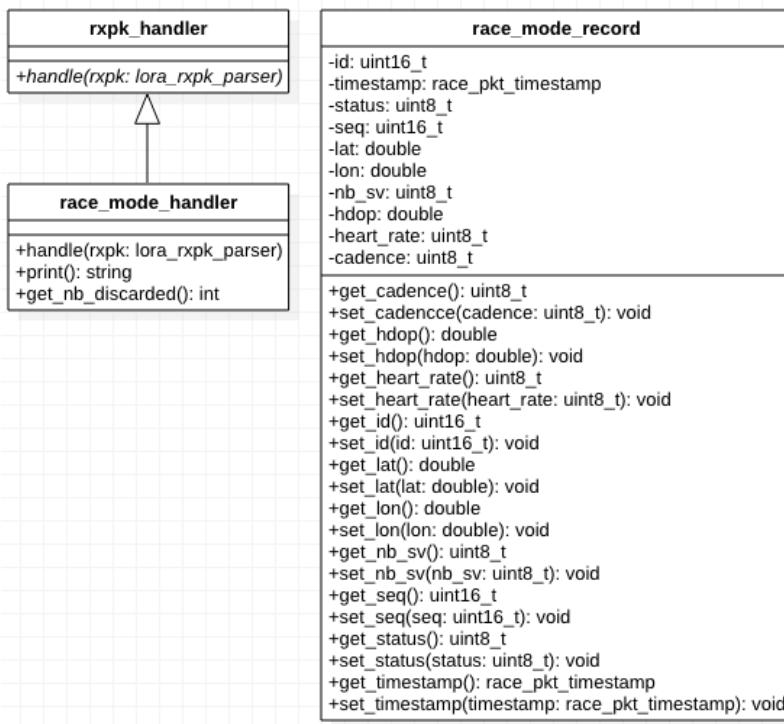


FIGURE 5.8 – Diagramme de classe de Race Mode Handler et Race Mode Record

La figure ?? montre le diagramme de classe de Test Mode Handler et Test Mode Record.

LoRa Packet Forwarder Parser

Le LoRa Packet Forwarder Parser permet de parser les données reçus par le socket et de déterminer si c'est un paquet de type PUSH_DATA.

La figure 5.10 montre le diagramme de classe de LoRa Packet Forwarder Parser.

LoRa Push Data Parser

Si le paquet est de type PUSH_DATA, comme reporté par la classe LoRa Packet Forwarder Parser, alors le reste des données peut être extrait en utilisant la classe LoRa Push Data Parser. Cette classe permet de savoir si le contenu du paquet correspond à un objet json nommée rxpk (Réception de donnée) ou stat (Statistiques envoyé périodiquement par le packet forwarder).

La figure 5.11 montre le diagramme de classe de LoRa Push Data Parser.

LoRa rxpk Parser

Une fois que l'on a déterminé que le paquet est de type PUSH_DATA et qu'il contient bien un objet json rxpk, le type de paquet qui nous intéresse, alors la classe LoRa rxpk Parser permet d'extraire les informations intéressantes qui sont contenues dans l'objet json. Grâce aux fonctionnalités de la bibliothèque rapidjson, les différents champs composant le tableau json rxpk sont extraits et décodés afin d'être facilement accessibles. Les données du paquet, qui sont encodées en base64 sont également décodées pendant cette étape.

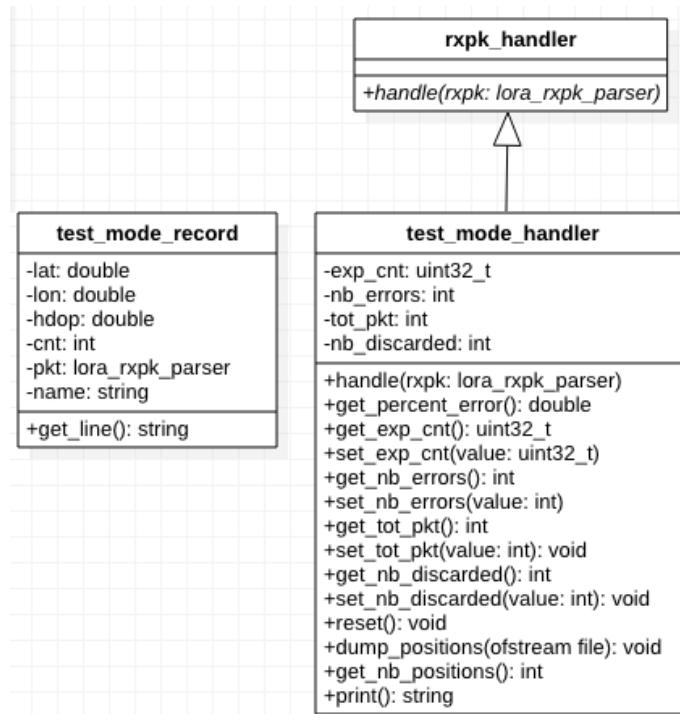


FIGURE 5.9 – Diagramme de classe de Test Mode Handler et Test Mode Record

La figure 5.12 montre le diagramme de classe de LoRa rxpk Parser.

Vector Reader

TODO:

Shell & Shell Command

TODO:

Logger

TODO:

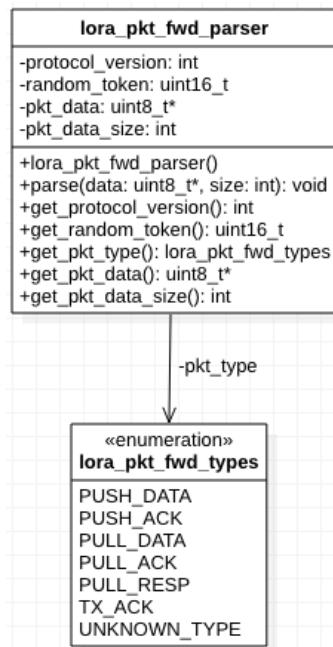


FIGURE 5.10 – Diagramme de classe de LoRa Packet Forwarder Parser

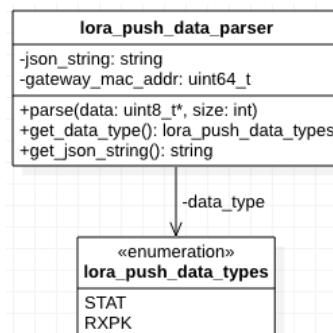


FIGURE 5.11 – Diagramme de classe de LoRa Push Data Parser

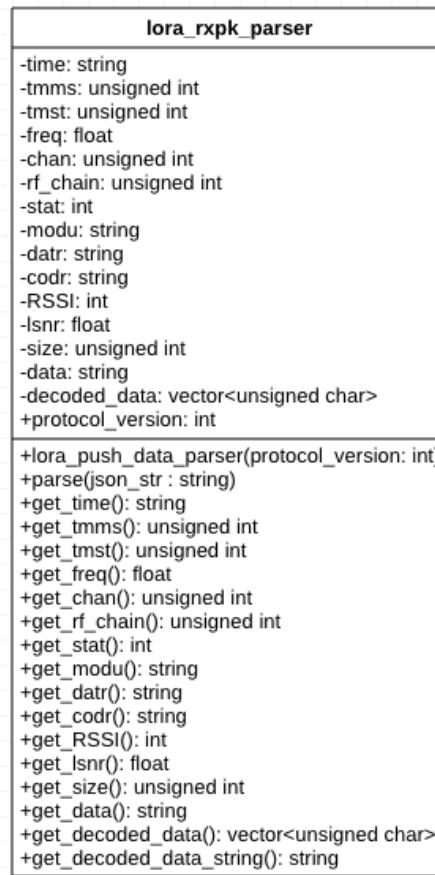


FIGURE 5.12 – Diagramme de classe de LoRa rxpk Parser

6 Description de la base de données

ADd info to install DB

7 Description de l'application mobile

8 Test phase #1

Comme décrit dans la pré-étude, afin de valider la phase 1 du développement du projet, un test impliquant le capteur choisi ainsi que la passerelle est effectué afin de s'assurer que les deux éléments sont capable de remplir les tâches qui leurs sont attribuées pour le projet. Si le test est concluant alors la solution matérielle choisie peut être validée pour la suite du développement. Dans le cas contraire le matériel doit être changé afin de pouvoir garantir une solution adéquate.

L'objectif est de vérifier le bon fonctionnement du capteur et de la passerelle dans les conditions finale d'utilisation, c'est à dire une réception adéquate des données envoyées par le capteur en extérieur et en mouvement. De plus ce test va également permettre de choisir la configuration initiale à utiliser pour la transmission des paquets LoRa, en particulier le facteur d'étalement ainsi que la puissance de transmission du signal de sortie à utiliser. On rappelle qu'un petit facteur d'étalement permettra un taux de transfert plus élevé sur une distance moindre, alors qu'un grand facteur permettra l'envoie de données à des distances accrues mais à un taux de transfert plus bas. En ce qui concerne la puissance de sortie, l'objectif est de trouver la valeur minimale qui permet une bonne réception des données à la distance d'utilisation. Ceci permettra d'optimiser l'utilisation de la batterie pour garantir la durée d'utilisation requise qui est de 10h.

Pour pouvoir effectuer ce test les éléments suivants ont été réalisés.

- Mise en place et assemblage du matériel du capteur et de la passerelle
- Développement d'un programme de test pour le capteur
- Installation et configuration du packet forwarder de la passerelle
- Développement d'une partie du serveur d'application de la passerelle

Afin de pouvoir s'assurer de la bonne réception des données le capteur, à intervalles réguliers, va envoyer un paquet de données à destination de la passerelle. Le format ainsi que le contenu du paquet envoyé par le capteur est décrit dans la figure 8.1.

0xFFEEDDEAD	0xACABFACE	Latitude	Longitude	Nombre de satellite en vue	HDOP	Compteur
4 bytes	4 bytes	8 bytes	8 bytes	1 byte	8 bytes	4 bytes

FIGURE 8.1 – Format du paquet test1

Un programme de test, se basant sur le système de développement Arduino IDE proposant un framework pour les cartes Arduino, est réalisé. Son comportement est très simple, il se contente d'envoyer un paquet de données LoRa puis d'attendre un certain temps, au terme duquel le cycle recommence. Le paquet envoyé par le capteur commence par deux valeurs fixes suivis de la latitude/longitude du capteur au moment de l'envoi du paquet. Ces deux éléments sont suivis du HDOP, ou Horizontal Dilution of Precision, qui exprime le degré de précision de la position GPS. Pour terminer, la valeur du compteur est ajoutée au paquet ce qui permettra à la passerelle de détecter quand un paquet est perdu et ainsi garder des statistiques afin de pouvoir jauger la qualité de la transmission.

Du côté de la passerelle, le packet forwarder, logiciel repris depuis internet, est configuré et mis en œuvre. Il récupère les paquets LoRa reçus, les transforme en chaîne de text de type json et

les transmets par le biais d'un paquet UDP. Une partie du serveur d'application est développée qui permet à la passerelle de récupérer les paquets LoRa émit par le packet forwarder au travers d'un socket et d'en analyser le contenu. A chaque paquet reçu la passerelle s'assure que le paquet est en provenance du capteur en vérifiant la valeur des deux marqueurs de début, ensuite la valeur du compteur est vérifiée pour s'assurer que c'est bien celle attendue, si ce n'est pas le cas cela signifie qu'un ou plusieurs paquets ont été perdus dans l'intervalle. Cette partie du serveur d'application servira de base pour le développement final de l'application. Au moyen d'un shell implémenté dans le serveur de paquet, il est possible à tout moment de sauvegarder le contenu des paquets reçus jusqu'ici dans un fichier, cela permet ensuite d'en extraire les positions GPS afin de les afficher dans un logiciel comme Google Earth par exemple qui permettra la visualisation de toutes les positions acquises durant le test.

Afin de pouvoir récupérer les logs relatifs aux tests et contrôler la réception des paquets, la passerelle est configurée afin de faire office de access point WiFi. Cela permet à l'ordinateur portable de se connecter à la passerelle au moyen de ssh et d'effectuer les opérations nécessaires. Enfin, le capteur est alimenté par l'accumulateur polymer-ion et la passerelle, elle, est alimentée par l'USB de l'ordinateur portable.

8.1 Scénarios

Deux scénarios distincts sont réalisés en utilisant le système expliqué dans la section précédente. Ils seront effectués deux fois chacun, une fois avec la valeur d'étalement de spectre (spreading factor) avec la plus petite valeur et une fois avec la plus grande valeur, cela permettra de jauger quelle configuration sera nécessaire pour la version finale du capteur.

Le premier test est le test sur piste, il consiste à prendre le capteur et ensuite de marcher le long du parcours d'une piste d'athlétisme. L'objectif de ce test est de voir si dans des conditions proches de l'utilisation finale pour le projet les données sont reçues correctement et de pouvoir également juger de la configuration finale que le système devra utiliser.

Le deuxième test est appelé test de distance, l'objectif est de pouvoir évaluer la distance maximum de fonctionnement jusqu'à laquelle les paquets sont bien reçus. Pour se faire, le capteur sera déplacé sur une ligne droite jusqu'à un point fixé puis il sera ensuite retourné au point de départ.

8.2 Résultats

Les résultats des tests décrits dans cette section ont été réalisés à la place d'arme de Planeyse à Neuchâtel le 13 Juillet 2018. Cet endroit dispose de grandes surfaces plates et également d'une piste proposant des conditions très proches d'une piste d'athlétisme. C'est donc un endroit idéal réunissant les conditions nécessaires pour faire les tests.

Les tables suivantes présentent les résultats des deux tests, sur piste et de distance. La colonne configuration spécifie les paramètres utilisés pour la communication LoRa, SF voulant dire spreading factor (facteur d'étalement) et PWR signifiant power (le niveau de puissance du signal en sortie). Le facteur d'étalement peut être paramétré entre SF7 et SF12, le niveau de puissance quant à lui peut être configuré dans des valeurs entre -4.0 à +14.1 dBm. Pour finir elles présentes

également le nombre total de paquet reçu ainsi que le nombre de paquets perdus.

8.2.1 Test sur piste

TABLE 8.1 – Résultats des tests phase 1 - Piste

Tests Piste			Planeyse 13.07.2018			
Nom	Configuration	HDOP Moy	Nb Sat Moy	Nb reçu	Nb perdu	% perdu
Test #1	SF7 - PWR -0.6 dBm	0.97	8.74	46	3	6.1%
Test #2	SF12 - PWR -0.6 dBm	0.92	8.88	32	0	0

Les figures 8.2a et 8.2b permettent de visualiser les positions GPS reçu dans chaque paquet LoRa. Lors du test, la passerelle était positionnée vers le centre de la piste, c'est de là que je suis parti avec le capteur ce qui explique les premiers points qui ne se trouvent pas sur la piste.

On remarque que durant le test #1 des paquets ont été perdus, dans les deux zones rouge, lorsque le capteur se trouvait aux extrémités de la piste. Lorsqu'on augmente la valeur du facteur d'étalement, dans le test #2, on remarque que le problème n'apparaît plus.

8.2.2 Test de distance

TABLE 8.2 – Résultats des tests phase 1 - Distance

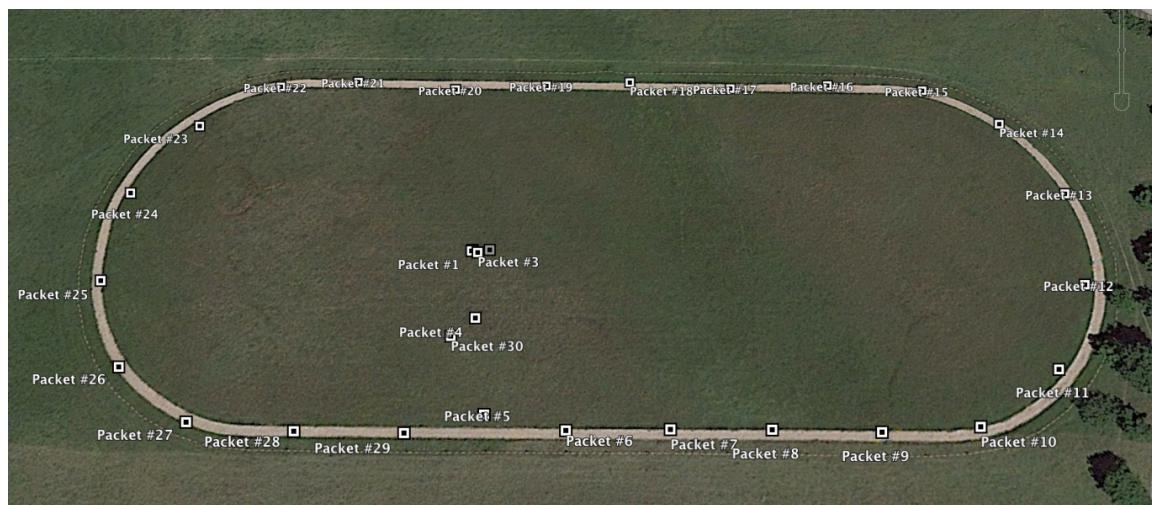
Tests Distance			Planeyse 13.07.2018			
Nom	Configuration	HDOP Moy	Nb Sat Moy	Nb reçu	Nb perdu	% perdu
Test #3	SF7 - PWR -0.6 dBm	1.37	7.88	32	10	23.8%
Test #4	SF12 - PWR -0.6 dBm	0.93	9.32	37	1	2.6%

Les figures 8.3a et 8.3b permettent de visualiser les positions GPS reçu dans chaque paquet LoRa.

Pendant les deux tests quelques paquets ont été perdus dans les zones marquées en rouge. Cependant si on analyse les résultats en plus de détails on remarque que durant le test #4, un seul paquet a été perdu et au moment où le capteur était très proche de la passerelle, on peut donc négliger cette perte qui est probablement due à un masquage de l'antenne de la passerelle. Lors du test #3 par contre, la distance limite qu'il est possible d'atteindre avec la configuration SF7 et puissance à -0.6 dBm a été atteinte après environ 200m de distance entre le capteur et la passerelle.

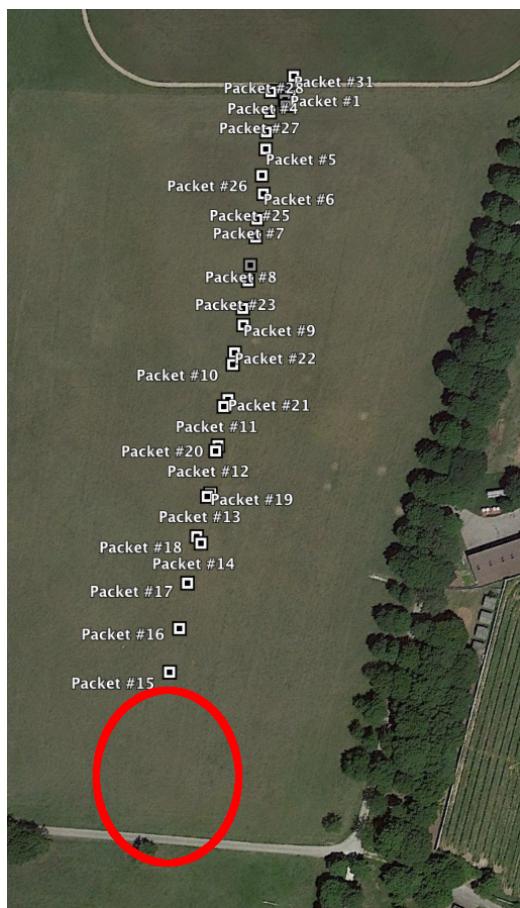


(a) Test #1

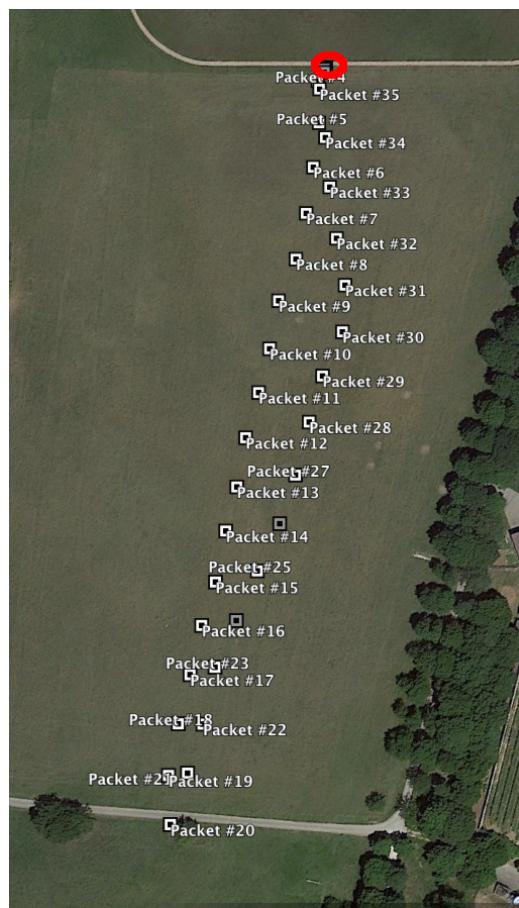


(b) Test #2

FIGURE 8.2 – Position GPS de chaque paquet reçu durant le test sur piste - Images capturés grâce au logiciel © Google Earth



(a) Test #3



(b) Test #4

FIGURE 8.3 – Position GPS de chaque paquet reçu durant le test de distance - Images capturés grâce au logiciel © Google Earth

8.3 Conclusion test phase #1

Au terme du test phase #1 on peut conclure que :

- La précision des positions GPS fournis par le module GPS est suffisante pour l'application visée
- Le fait que le capteur soit en mouvement ne pose pas de problème au niveau de la couche radio LoRa ou de la qualité des positions GPS fournit
- L'alimentation du capteur par l'accumulateur et la passerelle par l'USB fonctionne correctement
- La configuration de la couche radio devra être adaptée, le facteur d'étalement SF7 et puissance à -0.6 dBm n'étant pas suffisant pour un taux de réception de paquet satisfaisant

Grâce à ses éléments, on peut conclure que le matériel choisis est adéquat, la phase #1 du développement du projet est donc validée ce qui permet donc de passer à la phase #2. Dans cette phase du projet, la base de données, l'application du capteur et une ébauche de l'application mobile seront développé, ce qui permettra de pouvoir tester la chaîne de communication complète du système.

9 Test phase #2

10 Test phase #3

11 Considérations pour le développement d'un produit

12 Conclusion

13 Annexes

TODO: Doxygen

14 Dossier de gestion

Bibliographie

- [1] *EVA-8M u-blox 8 GNSS module Data Sheet*, Ubx-16009928 - r03 ed., UBlox, July 2016.
- [2] U-Blox, *u-blox 8 / u-blox M8 - Receiver Description Including Protocol Specification*, ubx-13003221 - r15 ed., March 2018.
- [3] *RN2483 LoRa Technology Module Command Reference User's Guide*, Ds40001784b revision b ed., Microchip Technology Inc., March 2015.
- [4] T. L. Foundation, "Zephyr project rtos." [Online]. Available : <http://www.zephyrproject.org>
- [5] *SAMD21 Family Datasheet*, Ds40001882c rev. c ed., Microchip Technology Inc., June 2018.
- [6] *AT11628 : SAM D21 SERCOM I2C Configuration*, 42631st ed., Atmel, December 2015.
- [7] *LSM303AGR Accelerometer/Magnetometer Datasheet*, Docid027765 rev 9 ed., ST, September 2016.
- [8] Semtech, "Basic communication protocol between lora gateway and server." [Online]. Available : https://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT
- [9] C. H. J. L. S. Semtech, Thomas Telkamp, "Single channel lorawan gateway." [Online]. Available : https://github.com/hallard/single_chan_pkt_fwd

Authentification