

TRAVAIL DE BACHELOR

MÉMOIRE

---

## **Conception d'un système de suivi temps réel LoRa pour compétitions sportives**

---

*Auteur :*

Léonard BISE

*Conseiller :*

Pierre BRESSY

Informatique Embarquée (ISEC)

Mars à Septembre 2018



# Travail de Bachelor

# Préambule

Ce travail de Bachelor est réalisé en vue de l'obtention du titre de Bachelor of Sciences en Ingénierie.

Son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Aucune utilisation, même partielle, de ce travail ne peut être faite sans l'autorisation écrite préalable de la Direction. Toutefois, l'entreprise ou l'organisation qui a confié un sujet de travail de Bachelor peut utiliser les résultats du travail pour ses propres besoins.

## Doyenne du Centre Formation de Base

L. Larghi

Yverdon-les-Bains, novembre 2017



## TRAVAIL DE BACHELOR 2017 - 2018

### ***Conception d'un système de suivi temps réel LoRa pour compétitions sportives***

---

#### Résumé publiable

Les compétitions sportives, de course à pied ou de cyclisme par exemple, sont des événements dont le déroulement n'est pas toujours facile à suivre pour les spectateurs qui sont sur place. Une fois le départ de la course donné, on peut vite perdre de vue les concurrents. Il faut donc se déplacer pour pouvoir suivre son déroulement et même dans ce cas, vu que le nombre de sportifs est souvent élevé, on peut difficilement se faire une idée globale sur la position de chacun d'eux ou du classement à un instant précis.

Ce projet propose la réalisation d'un système permettant le suivi en temps réel de compétitions sportives au travers d'une application mobile Android afin d'impliquer d'avantage les spectateurs. Grâce à une carte sur laquelle il est possible de voir le parcours de la course en entier, l'application tiendra à jour la position GPS de chaque coureur en temps réel au travers de données transmises par un capteur porté par les sportifs. L'interface propose également d'autres informations : le nom du sportif, son pays d'origine, son numéro de dossard, la distance parcourue, la vitesse moyenne, son rythme cardiaque et sa cadence (nombre de pas par unité de temps). Enfin, l'application permet de choisir ses coureurs favoris afin d'en faciliter le suivi ainsi que l'administration du système telles que la création des courses ou l'inscription des coureurs par exemple.

Pour que le système puisse remplir sa tâche, les éléments suivant ont été développés :

- un capteur qui est porté par les athlètes et qui permet l'acquisition des données,
- une passerelle qui est placée le long du parcours et qui centralise les données produites par les capteurs dans une base de données,
- une application mobile qui permet la visualisation des données produites.

La communication entre le capteur et la passerelle est assurée grâce à la technologie sans-fil LoRa, un protocole qui est prévu pour être utilisé par de petits systèmes avec peu de ressources sur des distances de plusieurs kilomètres, et qui ne nécessite pas de surcoût pour son utilisation contrairement aux communications mobiles GSM, car il utilise les bandes de fréquences libres ISM.

Le capteur utilise un micro-contrôleur ATSAMD21 avec un cœur ARM Cortex-M0+ sur lequel s'exécute le logiciel développé pendant le travail de Bachelor. Il est écrit en C et utilise le

système d'exploitation temps réel Zephyr qui propose tous les mécanismes et services nécessaires pour le développement d'une application embarquée temps réel.

La passerelle est construite à partir d'un Raspberry Pi 3 model B+ exécutant le système d'exploitation Linux ainsi que d'une carte de gestion de la couche radio LoRa. Une application serveur, codée en C++, se charge de récupérer les paquets reçus par l'interface LoRa et d'enregistrer les données décodées dans la base de données.

L'application mobile est développée en Java sur Android Studio et utilise le framework "Maps SDK for Android" de Google afin de gérer la carte et les éléments affichés dessus.

**Candidat**

Bise Léonard Date: 21.09.2018

Signature:

**Responsable**

Bressy Pierre Date: 14. IX . 2018

Signature:







# Table des matières

<b>Table des figures</b>	<b>xiv</b>
<b>Liste des tableaux</b>	<b>xviii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Énoncé du problème . . . . .	1
<b>2 Vue d'ensemble du système</b>	<b>3</b>
2.1 Les interactions . . . . .	4
2.2 LoRa & LoRaWAN . . . . .	4
<b>3 Développement du projet</b>	<b>9</b>
3.1 Phase #1 . . . . .	9
3.2 Phase #2 . . . . .	9
3.3 Phase #3 . . . . .	10
3.4 Environnement de développement . . . . .	11
<b>4 Description du capteur</b>	<b>13</b>
4.0.1 Les contraintes . . . . .	13
4.1 Le matériel . . . . .	13
4.1.1 Le module GPS UBloxEVA8M . . . . .	14
4.1.2 Le module LoRa RN2483 . . . . .	15
4.1.3 Le module rythme cardiaque Adafruit . . . . .	16
4.1.4 Le module accéléromètre LSM303AGR . . . . .	16
4.1.5 L'accumulateur . . . . .	16

4.2 Le système d'exploitation Zephyr . . . . .	17
4.2.1 Configuration Zephyr pour la carte SODAQ One . . . . .	17
4.3 Le logiciel embarqué . . . . .	19
4.3.1 Architecture logiciel . . . . .	20
4.3.2 Race Sensor Manager . . . . .	22
4.3.3 Heart Rate . . . . .	22
4.3.4 Cadence . . . . .	23
4.3.5 Debug . . . . .	25
4.3.6 Race Sensor Shell . . . . .	25
4.4 Les drivers . . . . .	26
4.4.1 Driver $I^2C$ ATSAMD21G18 . . . . .	27
4.4.2 Driver UBloxEVA8M . . . . .	29
4.4.3 Driver LSM303AGR . . . . .	31
4.4.4 Driver RN2483 . . . . .	33
4.4.5 Driver LEDs . . . . .	35
4.4.6 Driver External Interrupt Controller . . . . .	36
4.5 Paquet de donnée . . . . .	37
4.6 Le boîtier . . . . .	38
<b>5 Description de la passerelle</b>	<b>41</b>
5.1 Le matériel . . . . .	41
5.2 Le packet forwarder . . . . .	42
5.3 Le serveur d'application . . . . .	44
5.3.1 Architecture logiciel . . . . .	44

5.3.2 Les librairies externes . . . . .	47
5.3.3 Les classes . . . . .	49
<b>6 Description de la base de données</b>	<b>57</b>
6.1 L'extension PostGIS . . . . .	57
6.2 Les tables . . . . .	58
6.2.1 competition . . . . .	58
6.2.2 competitor . . . . .	59
6.2.3 country . . . . .	59
6.2.4 competitor_registration . . . . .	59
6.2.5 data_point . . . . .	60
6.2.6 track_point . . . . .	60
<b>7 Description de l'application mobile</b>	<b>63</b>
7.1 Architecture logiciel . . . . .	63
7.2 Les librairies externes . . . . .	66
7.3 Accès à la base de données . . . . .	67
7.4 Les activités . . . . .	67
7.4.1 MainActivity . . . . .	68
7.4.2 ViewRaceSelectorActivity . . . . .	68
7.4.3 ViewRaceActivity . . . . .	68
7.4.4 ReplayRaceActivity . . . . .	69
7.4.5 ManageRaceSelectorActivity . . . . .	70
7.4.6 CreateNewRaceActivity . . . . .	70
7.4.7 RaceLocationPickerActivity . . . . .	70

7.4.8 CreateNewCompetitorActivity . . . . .	71
7.4.9 RegistrationActivity . . . . .	71
7.4.10 RegistrationEditActivity . . . . .	71
7.4.11 StartEndRaceActivity . . . . .	71
7.4.12 SettingsActivity . . . . .	73
7.5 Les fragments . . . . .	73
7.5.1 DatePickerFragment . . . . .	73
7.5.2 TimePickerFragment . . . . .	73
7.6 Les classes . . . . .	74
7.6.1 RaceTrackerCompetition . . . . .	74
7.6.2 RaceTrackerCompetitionAdapter . . . . .	74
7.6.3 RaceTrackerCompetitions . . . . .	74
7.6.4 RaceTrackerCompetitor . . . . .	74
7.6.5 RaceTrackerCompetitorAdapter . . . . .	75
7.6.6 RaceTrackerCompetitors . . . . .	75
7.6.7 RaceTrackerCountry . . . . .	76
7.6.8 RaceTrackerCountryAdapter . . . . .	76
7.6.9 RaceTrackerCountries . . . . .	76
7.6.10 RaceTrackerRegistration . . . . .	76
7.6.11 RaceTrackerRegistrationAdapter . . . . .	76
7.6.12 RaceTrackerRegistrations . . . . .	76
7.6.13 RaceTrackerDataPoint . . . . .	76
7.6.14 RaceTrackerDataPoints . . . . .	77
7.6.15 RaceTrackerTrackPoint . . . . .	77

7.6.16 RaceTrackerTrackPoints . . . . .	77
7.6.17 RaceTrackerDBConnection . . . . .	77
7.6.18 RaceTrackerQuery . . . . .	77
7.6.19 RaceTrackerExecuteQuery . . . . .	78
7.6.20 RaceTrackerExecuteUpdate . . . . .	78
7.6.21 LatLngWraper . . . . .	78
7.6.22 RecyclerTouchListener . . . . .	78
7.7 Les interfaces . . . . .	86
7.7.1 OnQueryResultReady . . . . .	86
7.7.2 OnQueryExecuted . . . . .	86
7.7.3 OnUpdateDone . . . . .	86
<b>8 Problèmes et solutions</b>	<b>87</b>
8.1 Capteur : Driver $I^2C$ manquant . . . . .	87
8.2 Capteur : Driver GPIO incomplet . . . . .	87
8.3 Passerelle : Problème de connexion réseau . . . . .	87
8.4 Capteur : Interruption accéléromètre . . . . .	88
<b>9 Test phase #1</b>	<b>89</b>
9.1 Scénarios . . . . .	90
9.2 Résultats . . . . .	90
9.2.1 Test sur piste . . . . .	91
9.2.2 Test de distance . . . . .	91
9.3 Conclusions . . . . .	94
<b>10 Test phase #2</b>	<b>95</b>

10.1 Scénarios . . . . .	96
10.2 Résultats . . . . .	96
10.3 Conclusions . . . . .	98
<b>11 Test phase #3</b>	<b>99</b>
11.1 Scénarios . . . . .	99
11.2 Résultats . . . . .	100
11.2.1 Test de validation du système . . . . .	100
11.2.2 Test obstruction passerelle . . . . .	101
11.3 Conclusions . . . . .	104
<b>12 Considérations pour le développement d'un produit</b>	<b>105</b>
<b>13 Dossier de gestion</b>	<b>107</b>
<b>14 Conclusion</b>	<b>109</b>
<b>15 Annexes</b>	<b>111</b>
<b>Bibliographie</b>	<b>113</b>

## Table des figures

2.1	Interactions des acteurs du système . . . . .	4
2.2	Infrastructure LoRaWAN . . . . .	7
4.1	Schéma block du capteur SODAQ One . . . . .	15
4.2	Architecture du système d'exploitation Zephyr - <a href="http://www.zephyrproject.org">http://www.zephyrproject.org</a> . . . . .	18
4.3	Processus général du capteur . . . . .	20
4.4	Architecture statique du logiciel embarqué sur le capteur . . . . .	21
4.5	Architecture dynamique du logiciel embarqué sur le capteur . . . . .	22
4.6	Diagramme de classe du Race Sensor Manager . . . . .	23
4.7	Diagramme de séquence de l'initialisation du Race Sensor Manager . . . . .	24
4.8	Diagramme de séquence du thread du Race Sensor Manager . . . . .	25
4.9	Diagramme de classe du module Heart Rate . . . . .	25
4.10	Valeurs des échantillons de l'accéléromètre sur une période de 10 secondes . . . . .	26
4.11	Diagramme de classe du module Cadence . . . . .	26
4.12	Schéma d'un bus $I^2C$ . . . . .	27
4.13	Les messages $I^2C$ . . . . .	28
4.14	Diagramme de class du driver $I^2C$ . . . . .	28
4.15	Diagramme de class du driver UBloxEVA8M . . . . .	30
4.16	Diagramme de class du driver LSM303AGR . . . . .	32
4.17	Diagramme de class du driver RN2483 . . . . .	34
4.18	Diagramme de classe du driver LED . . . . .	35
4.19	Diagramme de class du driver External Interrupt Controller . . . . .	36

4.20 Format du paquet de donnée LoRa . . . . .	37
4.21 Image du concept de la boîte du capteur . . . . .	39
5.1 Schéma block de la passerelle . . . . .	42
5.2 Architecture statique du serveur d'application . . . . .	45
5.3 Flux des données du serveur d'application . . . . .	46
5.4 Architecture dynamique du serveur d'application . . . . .	47
5.5 Diagramme de classe de Race App Server . . . . .	49
5.6 Diagramme de séquence des opérations du thread de la classe Race App Server . .	50
5.7 Diagramme de classe de Race Tracker Data . . . . .	50
5.8 Diagramme de classe de rxpk Handler . . . . .	51
5.9 Diagramme de classe de Race Mode Handler et Race Mode Record . . . . .	51
5.10 Diagramme de classe de Test Mode Handler et Test Mode Record . . . . .	52
5.11 Diagramme de classe de LoRa Packet Forwarder Parser . . . . .	53
5.12 Diagramme de classe de LoRa Push Data Parser . . . . .	54
5.13 Diagramme de classe de LoRa rxpk Parser . . . . .	55
5.14 Diagramme de classe de Vector Reader . . . . .	55
5.15 Diagramme de classe de Shell et Shell Command . . . . .	56
5.16 Diagramme de classe de Logger . . . . .	56
6.1 Diagramme relationnel de la base de données . . . . .	57
7.1 Architecture statique de l'application mobile . . . . .	64
7.2 Architecture dynamique de l'application mobile . . . . .	65
7.3 MainMenuActivity . . . . .	68
7.4 ViewRaceSelectorActivity . . . . .	68

7.5 ViewRaceActivity . . . . .	69
7.6 ReplayRaceActivity . . . . .	69
7.7 ViewRaceActivity . . . . .	70
7.8 CreateNewRaceActivity . . . . .	70
7.9 RaceLocationPickerActivity . . . . .	71
7.10 CreateNewCompetitorActivity . . . . .	71
7.11 RegistrationActivity . . . . .	72
7.12 RegistrationEditActivity . . . . .	72
7.13 StartEndRaceActivity . . . . .	72
7.14 SettingsActivity . . . . .	73
7.15 DatePickerFragment . . . . .	73
7.16 TimePickerFragment . . . . .	74
7.17 RaceTrackerCompetition . . . . .	75
7.18 RaceTrackerCompetitor . . . . .	79
7.19 Diagramme de classe de RaceTrackerCompetitors . . . . .	80
7.20 RaceTrackerCountry . . . . .	80
7.21 RaceTrackerRegistration . . . . .	81
7.22 RaceTrackerDataPoint . . . . .	82
7.23 RaceTrackerTrackPoint . . . . .	83
7.24 Diagramme de classe de RaceTrackerDBConnection . . . . .	83
7.25 RaceTrackerQuery . . . . .	84
7.26 Diagramme de classe de LatLngWraper . . . . .	85
7.27 Diagramme de classe de RecyclerTouchListener . . . . .	85
7.28 Diagramme de classe de OnQueryResultReady . . . . .	86

7.29 Diagramme de classe de OnQueryExecuted . . . . .	86
7.30 Diagramme de classe de OnUpdateDone . . . . .	86
9.1 Format du paquet test1 . . . . .	89
9.2 Positions GPS des tests piste . . . . .	92
9.3 Positions GPS des tests distance . . . . .	93
10.1 Visualisation des positions reçues du capteur depuis l'application mobile . . . . .	97
11.1 Visualisation de l'application mobile pour le test 3 . . . . .	102
11.2 Résultats du test d'obstruction . . . . .	103

## Liste des tableaux

2.1 Quota d'utilisation de la bande ISM . . . . .	6
3.1 Dates clefs phase #1 . . . . .	9
3.2 Dates clefs phase #2 . . . . .	9
3.3 Dates clefs phase #3 . . . . .	10
3.4 Outils de développement généraux . . . . .	11
3.5 Outils de développement pour le capteur . . . . .	11
3.6 Outils de développement pour la passerelle . . . . .	12
3.7 Outils de développement pour l'application mobile . . . . .	12
4.1 Caractéristiques de la carte SODAQ One v3 . . . . .	14
4.2 Caractéristiques des threads du capteur . . . . .	21
4.3 Analyse des valeurs des échantillons . . . . .	24
4.4 Détails des champs du paquet de données radio LoRa . . . . .	38
5.1 Caractéristiques du Raspberry Pi 3 Model B+ . . . . .	42
5.2 Caractéristiques du Dragino LoRa Hat . . . . .	42
9.1 Résultats des tests phase 1 - Piste . . . . .	91
9.2 Résultats des tests phase 1 - Distance . . . . .	91
11.1 Comparaison des résultats . . . . .	101



# 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 cartes électroniques présentant des systèmes temps réel performants capables de communiquer avec une interface sans-fil 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 destiné à 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 pour les spectateurs d'avoir une vue d'ensemble de la situation de la course ou d'un classement précis, ce qui peut rendre l'événement parfois ennuyeux ou difficile à suivre.

Afin d'essayer de rendre de tels événements plus vivants, 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 leur temps de course, la distance parcourue ou leur 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 l'application mobile 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

Lors d'événements sportifs comme des courses à pied ou de vélo tout terrain, une fois le départ donné les spectateurs sont parfois loin de l'action pour une longue durée.

Afin de rendre ce temps mort plus intéressant, ce projet propose le développement d'un système de tracking des athlètes en direct. Grâce à un capteur placé sur chaque concurrent, il devient possible d'afficher sur une carte la situation globale de la course à tout moment.

L'objectif de ce système est de permettre de récupérer et centraliser la position GPS et le rythme cardiaque de chaque athlète équipé d'un capteur et d'afficher ces informations sur une carte géographique.

Le système est composé de 3 éléments distincts : un capteur, un gateway et une application.

Le capteur doit embarquer un capteur de rythme cardiaque et un système de positionnement GPS. De plus, il doit avoir sa propre source d'énergie avec une autonomie permettant son fonctionnement pendant l'entièreté d'une course.

Le gateway est le système qui récupère les données produites par les capteurs et les stocke dans une base de données située sur un serveur.

Les capteurs et le gateway communiqueront en utilisant le protocole LoRa (Long-Range) sur la bande de fréquence 868Mhz. De son côté, le gateway se connecte à la base de données en utilisant un réseau WiFi.

L'application est en charge de l'affichage de la position des coureurs sur la carte ainsi que de leur rythme cardiaque. Une estimation de la vitesse et de la distance parcourue est également affichée. En plus, l'application permet de rejouer une course qui s'est tenue dans le passé.

Une contrainte liée au capteur est qu'il doit être suffisamment petit pour ne pas gêner le sportif lors de son effort et être utilisable en extérieur.

## 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 qui 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 détail 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 12 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 simplification, la couche MAC LoRaWAN n'est pas employée, elle prendrait cependant tout son sens dans une optique de développement d'un produit.

La figure 2.1 montre les interactions existantes entre les acteurs du système.

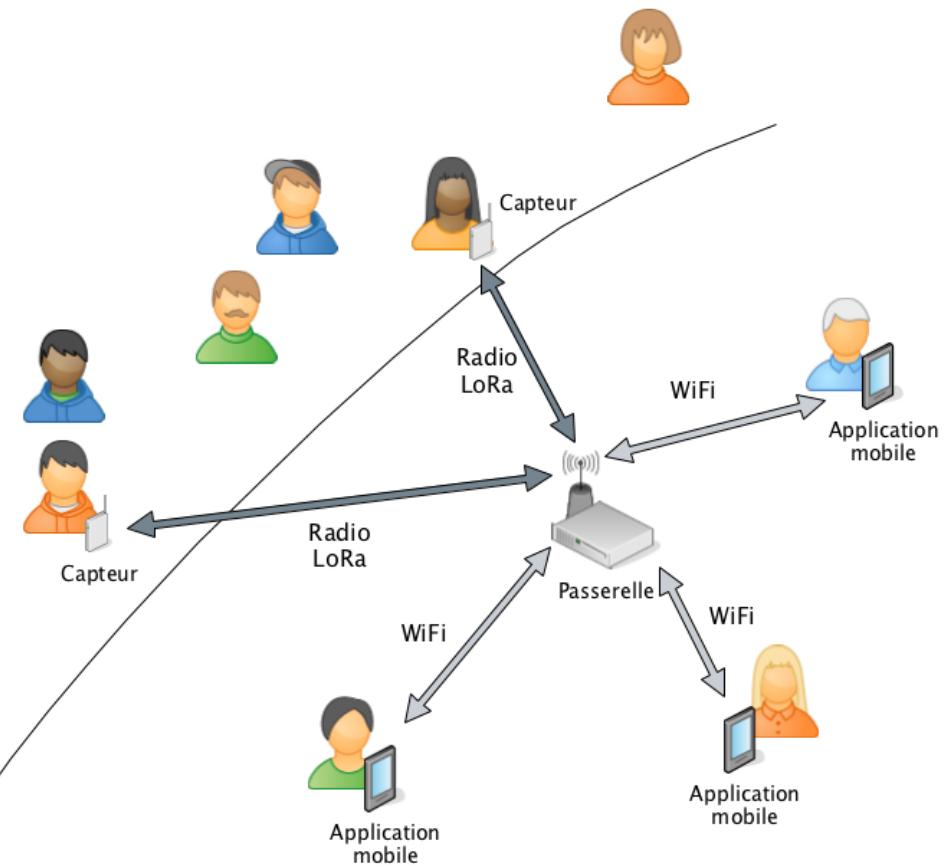


FIGURE 2.1 – Interactions des acteurs du système

TODO: Data flow?

## 2.2 LoRa & LoRaWAN

Lorsque l'on parle de LoRa, souvent une confusion est faite entre deux entités, d'un côté la couche LoRa et de l'autre la couche MAC LoRaWAN.

La couche physique LoRa est une technique de modulation pour les communications sans fil permettant l'envoie de données à bas débit à des distances de plusieurs kilomètres et le tout en consommant peu d'énergie. Ses caractéristiques rendent cette technologie très attractive pour des projets dans le monde de l'embarqué où les systèmes disposent de très peu de ressources et qui fonctionnent souvent avec des accumulateurs comme seule source d'énergie.

A son fondement, la couche physique est basée sur un type de modulation appelé modulation à étalement de spectre. Son principe premier est d'élargir artificiellement la bande de fréquence utilisée par le signal à envoyé ce qui le rend du coup plus résilient aux interférences volontaires ou involontaires.

Une des techniques très connues dans ce type de modulation est l'étalement de spectre à séquence directe (Direct Sequence Spread Spectrum). Comme expliqué du fait de l'utilisation de l'étalement de spectre ce signal est plus robuste aux interférences par contre il nécessite que le transmetteur et le récepteur se mettent d'accord sur la séquence à utiliser pour encoder les signaux. Cette technique est beaucoup utilisée, notamment par la norme 802.11b (WiFi). Elle comporte comme gros désavantage le fait qu'elle nécessite une horloge de base très précise et que le décodage du signal sur le récepteur peut être long en fonction de la séquence de codage utilisée. Tout ceci rend cette technique peu pratique pour des systèmes à bas coût et à basse consommation comme ceux du domaine de l'Internet of Things (IoT).

Dans les années 40, une autre technique se basant sur l'étalement de spectre a été développée pour être utilisée dans les applications radars, le Chirp Spread Spectrum (CSS). Elle utilise toujours le principe d'étalement de spectre afin d'être robuste aux interférences, cependant l'utilisation d'une séquence pseudo aléatoire qui est utilisé pour distinguer le signal du bruit, comme dans les techniques Direct Sequence Spread Spectrum (DSSS) ou Frequency Hopping Spread Spectrum (FHSS), ne se fait plus. En plus des avantages inhérents à l'étalement de spectre les changements liés à cette technique la rendent résistante à la dégradation des données causées par les trajets multiples (multipath) et l'évanouissement (fading). [1]

Un autre élément intéressant est apporté par la couche physique LoRa, le facteur d'étalement du spectre. Ce paramètre configurable définit comme son nom l'indique le facteur d'étalement du spectre lors de la modulation du signal. Plus le facteur est grand plus la transmission de l'information sera lente mais par contre la portée de réception des données sera grande, plus le facteur est petit et plus le taux de transfert sera important par contre la portée en sera grandement réduite. Ce facteur peut prendre une valeur entre 7 et 12 ce qui permet de faire un choix entre une distance accrue ou un taux de transfert plus élevé, entre 0.3 kbps avec un facteur de 12 à 27 kbps pour l'autre extrême. Enfin ce paramètre influence également sur la taille de la charge utile qu'il est possible d'envoyer dans un seul message, entre 51 octets pour la valeur la plus faible à 222 octets maximum. [2]

La modulation LoRa utilise les bandes de fréquences ISM (Industrielle, scientifique, médicale), qui sont libre d'utilisation et sans surcoût, à condition de respecter les duty cycle assignés. En effet puisque cette bande de fréquence n'est pas contrôlée, des quota de taux d'utilisation ont été définis afin de garantir que tout les utilisateurs puissent envoyés leurs données et ainsi éviter qu'un petit nombre de capteur ne monopolise la bande de fréquence. [3]

La table 2.1 présente les quota pour quelques bandes de fréquence.

TABLE 2.1 – Quota d'utilisation de la bande ISM

Code	Fréquence [Mhz]	Duty cycle
g	863.0 à 868.0	1%
g1	868.0 à 868.6	1%
g2	868.7 à 869.2	0.1%
g3	869.4 à 869.65	10%

LoRaWAN, qui est souvent également appelé à tort LoRA est la couche MAC qui s'appuie directement sur la couche radio. Elle propose un set de fonctionnalités supplémentaires permettant de faciliter la gestion d'un grand nombre de capteur et d'améliorer le niveau de sécurité. Son concept est de s'appuyer sur un réseau de passerelles placées afin de produire une couverture maximale pour les utilisateurs, les passerelles récupèrent les paquets et les transmettent à un serveur centrale, le serveur réseau, qui gère le protocole dans son ensemble. Il s'occupe de générer les paquets de réponses qui sont envoyés en retour aux passerelles et il permet également l'envoie des données reçues des capteurs à des serveurs d'applications qui vont ensuite exploiter les données. LoRaWAN permet l'encryption des données transitant entre le serveur réseau et les capteurs, qui sont les deux seuls éléments à connaître les clefs de chiffrement, ce qui permet d'assurer la confidentialité des informations traversant le réseau. Une autre fonctionnalité intéressante que ce protocole propose est le Adaptive Data Rate (ADR), en fonction des paramètres reçus des passerelles et des informations concernant la qualité de transmission des capteurs, le serveur réseau va pouvoir altérer les débit de transfert des capteurs. Un nœud qui sera très distant de la passerelle verra son taux de transfert baissé, ce qui permettra de rendre la communication plus robuste aux interférences. [4]

Pour des raisons de simplification le projet du travail de Bachelor utilise uniquement la couche radio LoRa sans la couche protocolaire LoRaWAN. En effet, la couche LoRaWAN est très intéressante dans le cas d'utilisation d'un produit où plusieurs capteurs devraient être gérés en parallèle. Dans ce cas, il est obligatoire de stocker les données sur internet et LoRaWAN propose toute une infrastructure permettant de le faire, en plus cela permet d'utiliser des technologies web moderne pour accéder aux données, des API REST par exemple. Cependant puisque dans la mesure ou ce projet à pour objectif de valider les études que j'ai effectuées dans la filière informatique embarquée j'ai préféré me concentrer sur le développement dans ce domaine et donner une priorité plus basse aux éléments qui en sortiraient.

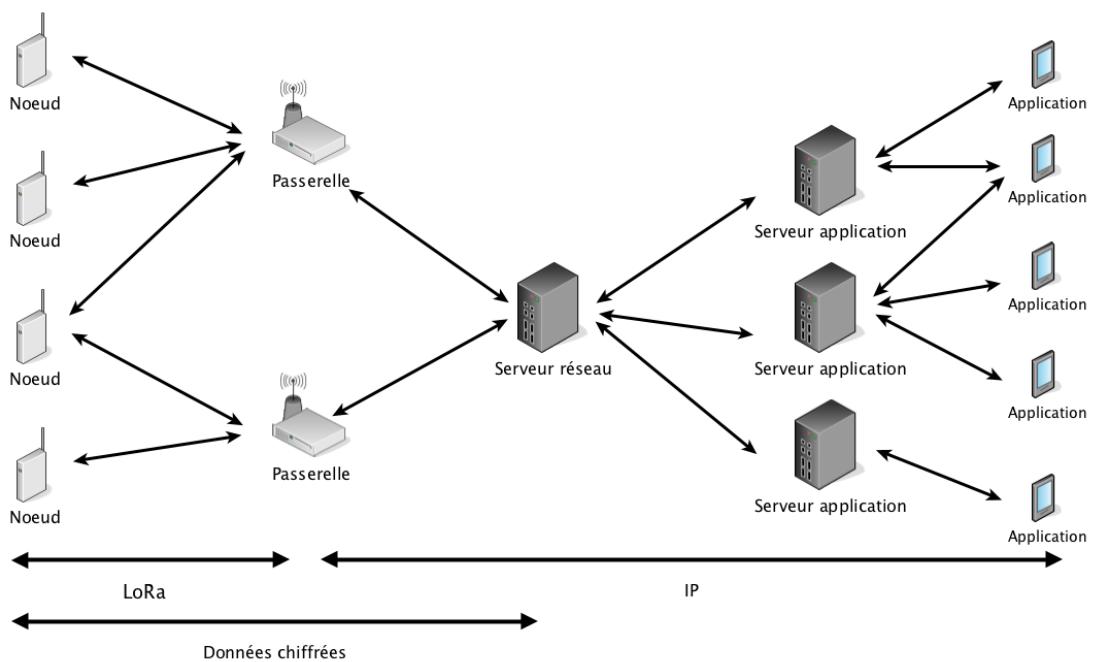


FIGURE 2.2 – Infrastructure LoRaWAN



## 3 Développement du projet

Durant la pré-étude, j'ai pris la décision de découper le développement du projet en trois phases distinctes afin de pouvoir gérer le travail au mieux et pouvoir réagir rapidement en cas d'imprévu ou de problèmes. Pour chaque phase, des objectifs ont été établis afin de pouvoir, à son terme, en valider l'exécution et ainsi passer à la phase suivante.

Les sections suivantes décrivent les trois phases de développement du projet.

### 3.1 Phase #1

TABLE 3.1 – Dates clefs phase #1

Début	Validation
9 Juillet 2018	13 Juillet 2018

La première phase de développement qui a commencé dès le début du projet, c'est à dire le 9 Juillet 2018, avait pour objectif la validation de la transmission des données avec la technologie LoRa et la solution matériel choisie pour le capteur et la passerelle. Durant cette phase, seuls le capteur et la passerelle ont été testés.

Pour ce faire, un programme simple de test a été écrit pour le capteur en se basant sur le framework Arduino, facilitant et accélérant grandement le développement car déjà tous les modules et drivers nécessaires étaient disponibles, le but étant de permettre de faire des tests de transmissions de paquet rapidement et efficacement. En ce qui concerne la passerelle, le début du serveur d'application a été développé qui permet la récupération des données transmises par le capteur. Un gestionnaire spécial pour ce test a été développé qui permet de vérifier si des paquets ont été perdus lors de leur transmission afin de jauger de la qualité de la transmission LoRa.

Au terme de ce développement un test a été effectué en extérieur afin de s'assurer que le matériel était capable d'envoyer et de transmettre des paquets à des distances proches de l'utilisation finale du projet. Dans le cas où les résultats du test ne seraient pas satisfaisants, une nouvelle étude sur le matériel devrait être faite afin de trouver une solution fonctionnant dans les conditions d'utilisation du projet. Le test a été effectué le 13 Juillet 2018 et a permis de valider la phase #1 avec succès ouvrant ainsi le développement pour la deuxième phase.

Pour plus d'information sur le test effectué et ses résultats voir le chapitre 9.

### 3.2 Phase #2

TABLE 3.2 – Dates clefs phase #2

Début	Validation
14 Juillet 2018	7 Septembre 2018

La deuxième phase du projet a été beaucoup plus conséquente en terme de travail à accom-

plir. Son objectif a été de valider la chaîne complète de communication du système, tous les éléments ont été développés dans leur forme simplifiée afin de pouvoir effectuer les tâches nécessaires au bon fonctionnement du système. Durant cette phase, l'accent a été mis sur la transmission de la position GPS, les autres paramètres ont été laissés de côté afin de pouvoir se concentrer sur les éléments les plus importants.

Pour le capteur, le système d'exploitation Zephyr a été pris en main afin d'y ajouter la configuration de la carte SODAQ One et les drivers manquants,  $I^2C$  et GPS, ont été développés et testés afin de pouvoir interagir avec le modules GPS. Enfin le cœur de l'application du capteur a été écrit, qui consiste principalement à récupérer la position GPS au travers du bus  $I^2C$  en utilisant le driver GPS et puis créer un paquet de données pour enfin l'envoyer grâce à la couche radio LoRa, ceci en boucle.

La structure de la base de données a été définie au moyen d'un diagramme UML et les scripts associés permettant la création des différentes tables ont également été écrits. Le logiciel de gestion de base de données a ensuite été installé sur la passerelle et les tables créées.

Le serveur d'application a été développé d'avantage afin d'y intégrer la gestion du mode "race" qui permet la réception des données envoyées par le capteur, l'extraction des paramètres intéressants pour finalement aller écrire le tout dans la base de données.

L'application mobile a été mise en place, permettant la connexion à la base de données afin d'y récupérer les données nécessaires. La gestion de la carte bien connue de Google a été implémenté afin de pouvoir y afficher les positions récupérées depuis la base de données. Cependant, l'interface n'a été que très peu développée, l'accent étant mis sur le développement du fonctionnement de base plutôt que sur l'esthétique.

Cette phase a été terminée après un test en extérieur sur un anneau de distance similaire à une piste d'athlétisme permettant de valider le bon fonctionnement de la chaîne complète du système.

Pour plus d'information sur le test effectué et ses résultats voir le chapitre 10.

### 3.3 Phase #3

TABLE 3.3 – Dates clefs phase #3

Début	Validation
20 Août 2018	22 Septembre 2018

Une fois que le fonctionnement du système dans son ensemble a été validé, la troisième phase a pu commencer. Son objectif est la finalisation du système dans son entier, c'est à dire d'implémenter les éléments restants, comme l'acquisition du rythme cardiaque et de la cadence et leur écriture dans la base de données. L'interface graphique de l'application mobile ainsi que les fonctionnalités restantes comme la gestion des courses et la gestion graphique de la carte sont finalisées.

Il est à noter que cette phase a commencé plus tôt que la date du test formel de la phase #2, ceci étant dû au fait que le test formel a dû être repoussé à plusieurs reprises à cause de pro-

blèmes techniques au niveau de la connexion de l'application mobile et de la passerelle. Afin de pouvoir respecter les contraintes de temps liées au projet, il a été décidé de commencer le développement relatif à la phase #3 avant le passage formel du test de validation #2.

Au terme de la phase #3, un test faisant office de démonstration de l'utilisation du système a été fait afin de valider son fonctionnement global.

Pour plus d'information sur le test effectué et ses résultats voir le chapitre 11.

### 3.4 Environnement de développement

Dans la manière du possible, des outils modernes, performants, gratuits et open source ont été utilisés dans le cadre du développement du projet. L'entier du travail effectué est hébergé sur la plateforme [github.com](https://github.com/leodido99/TravailBachelor), en accès libre, à l'adresse [github.com/leodido99/TravailBachelor](https://github.com/leodido99/TravailBachelor).

Les outils et librairies utilisés pour le développement des différents acteurs du projet figure ci-dessous.

TABLE 3.4 – Outils de développement généraux

Nom	URL	Description
LaTeX	<a href="http://www.latex-project.org">www.latex-project.org</a>	Système open source pour l'écriture de documents
git	<a href="http://git-scm.com">git-scm.com</a>	Outil open source de gestion de configuration
github	<a href="http://www.github.com">www.github.com</a>	Plateforme d'hébergement gratuite pour des projets git
yEd	<a href="http://www.yworks.com">www.yworks.com</a>	Outil de dessin de schéma
StarUML	<a href="http://staruml.io">staruml.io</a>	Outils de création de diagramme UML
Fusion 360	<a href="http://www.autodesk.com">www.autodesk.com</a>	Outil de dessin 3D
Matlab	<a href="http://www.mathworks.com">www.mathworks.com</a>	Outil d'analyse de données

TABLE 3.5 – Outils de développement pour le capteur

Nom	URL	Description
Eclipse	<a href="http://www.eclipse.org">www.eclipse.org</a>	IDE pour le développement en langage C/C++
Zephyr Project	<a href="http://www.zephyrproject.org">www.zephyrproject.org</a>	Système d'exploitation temps réel sur lequel l'application du capteur se base
GNU ARM Embedded Toolchain	<a href="http://developer.arm.com">developer.arm.com</a>	Toolchain fournit gcc et les outils associés et permettant la compilation pour les cibles de types Arm Cortex-M
cmake	<a href="http://cmake.org">cmake.org</a>	Évolution de make permettant la gestion du processus de compilation
BOSSA	<a href="https://github.com/shumatech/BOSSA">github.com/shumatech/BOSSA</a>	Outils permettant de programmer la mémoire des microcontrôleurs Atmel SAM

TABLE 3.6 – Outils de développement pour la passerelle

Nom	URL	Description
Eclipse	<a href="http://www.eclipse.org">www.eclipse.org</a>	IDE pour le développement en langage C/C++
GNU ARM RPI Toolchain	<a href="https://github.com/raspberrypi">github.com/raspberrypi</a>	Toolchain fournissant gcc et les outils associés et permettant la compilation pour les cibles de types ARMv7 comme le Raspberry Pi <b>TODO:</b>
make	<a href="http://www.gnu.org/software/make">www.gnu.org/software/make</a>	Permet la gestion du processus de compilation
Raspbian	<a href="http://www.raspbian.org">www.raspbian.org</a>	Distribution Linux basée sur Debian spécialement optimisé pour fonctionner sur le Raspberry Pi
PostgreSQL	<a href="http://www.postgresql.org">www.postgresql.org</a>	Système open source de gestion de base de données relationnelle
PostGIS	<a href="http://postgis.net">postgis.net</a>	PostGIS est une extension pour les base de données PostgreSQL qui ajoute des fonctionnalités permettant de faire des requêtes et des opérations avec des positions géographique
Single channel packet forwarder	<a href="https://github.com/hallard/RPI-Lora-Gateway">github.com/hallard/RPI-Lora-Gateway</a>	Logiciel packet forwarder permettant de récupérer les paquets depuis le module LoRa au travers du bus SPI
libpqxx	<a href="http://pqxx.org/development/libpqxx">pqxx.org/development/libpqxx</a>	API C++ pour client PostgreSQL permettant d'exécuter des requête sur une base de données
base64	<a href="https://github.com/Lora-net">github.com/Lora-net</a>	Classe permettant le décodage de données de type base64
rapidjson	<a href="http://rapidjson.org">rapidjson.org</a>	Librairie utiliser pour générer ou parser des chaînes de caractère de type json

TABLE 3.7 – Outils de développement pour l'application mobile

Nom	URL	Description
Android Studio	<a href="https://developer.android.com/studio">developer.android.com/studio</a>	IDE pour le développement en langage java sur la plateforme Android
Maps for Android SDK	<a href="https://developers.google.com/maps">developers.google.com/maps</a>	SDK permettant la création et la manipulation de carte type Googlemaps
Driver JDBC PostgreSQL	<a href="https://jdbc.postgresql.org">jdbc.postgresql.org</a>	Driver JDBC pour PostgreSQL permettant l'exécution de requête sur des base de données de type PostgreSQL

## 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 ce 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 grand nombre de modules, 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 produites. 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 décrites 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 battements 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'envoi 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 connectés 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.

#### 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.

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

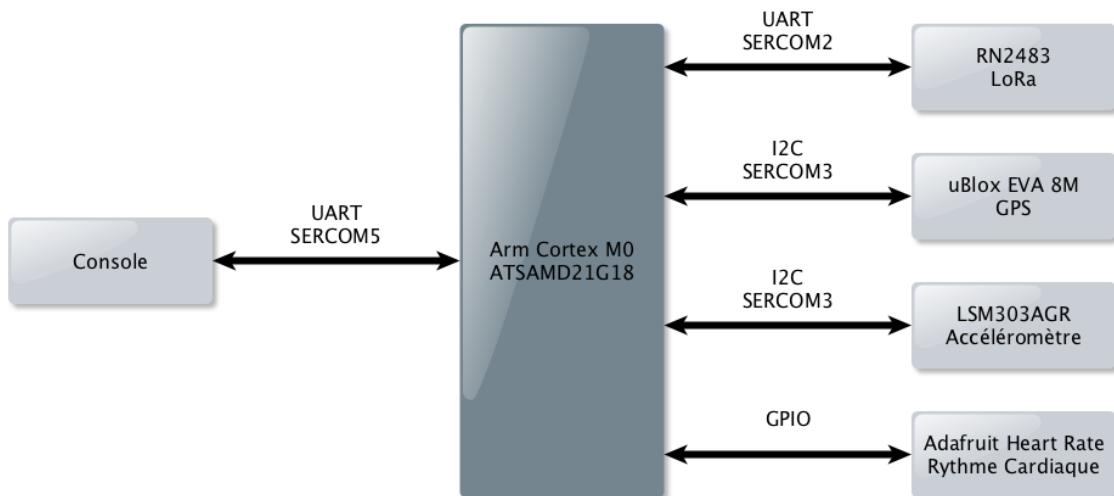


FIGURE 4.1 – Schéma block du capteur SODAQ One

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 voulus à une certaine fréquence. En fonction des messages différentes informations seront disponibles. 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 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 [5] ainsi que dans le document d'explication de son protocole [6].

#### 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'envoi 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 décrites dans la datasheet du composant [7].

#### 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 d'autre part le récepteur qui permettra au micro-contrôleur de détecter les battements du cœur. La sangle et le récepteur utilisent le protocole sans-fil WeakLink+ de la compagnie Polar, ce qui permet au récepteur de générer une impulsion sur une entrée du micro-contrôleur lorsqu'un battement est transmis par la sangle. Cette impulsion est détectée par le micro-contrôleur grâce au driver External Interrupt Controller spécialement développé pour ce projet et qui permet de déclencher des interruptions lors de la détection d'un flanc montant par exemple.

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 axe. 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 L'accumulateur

TODO: figure

L'accumulateur utilisé pour alimenter la carte SODAQ One est de type lithium-polymère, solution assurant une sécurité accrue mais distribuant un peu moins d'énergie qu'un accumulateur de type lithium-ion par exemple.

Cet accumulateur est capable de délivrer environ 3.7V et 1200 mAh.

TODO: Discharge figure?

## 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 projets dans les domaines aérospatial, militaire et automobile. Plusieurs architectures de micro-contrôleur sont prises en charge, comme ARM, RISC ou x86 par exemple, et plusieurs dizaines de configurations pour différentes cartes du marché existent.

Ce RTOS est également très facile à configurer pour ses propres besoins au moyen de fichiers de configuration où 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. [8]

Autour de cet RTOS, il existe une importante communauté qui travaille 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 modification 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 existants pour le micro-contrôleur utilisé par le SODAQ One, un nouveau driver  $I^2C$  ainsi qu'un driver de gestion des interruptions externes ont été réalisés car il n'en existait aucun au moment où j'ai commencé le travail de Bachelor. Le driver  $I^2C$  est utilisé afin de pouvoir communiquer avec les modules GPS et accéléromètre. Le driver External Interrupt Controller est utilisé afin de détecter les battements du cœur - lors de cet événement une impulsion est générée par le module rythme cardiaque qui, grâce à ce driver, permet de déclencher une interruption. 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.

### 4.2.1 Configuration Zephyr pour la carte SODAQ One

Zephyr utilise un système de configuration qui permet de définir les périphériques qui sont présents sur une certaine carte. Cela permet ensuite aux utilisateurs de pouvoir accéder aux différents éléments grâce aux drivers proposés par le système d'exploitation. En plus de cela il est également nécessaire de configurer la fonction de chaque pins que l'on souhaite utiliser car

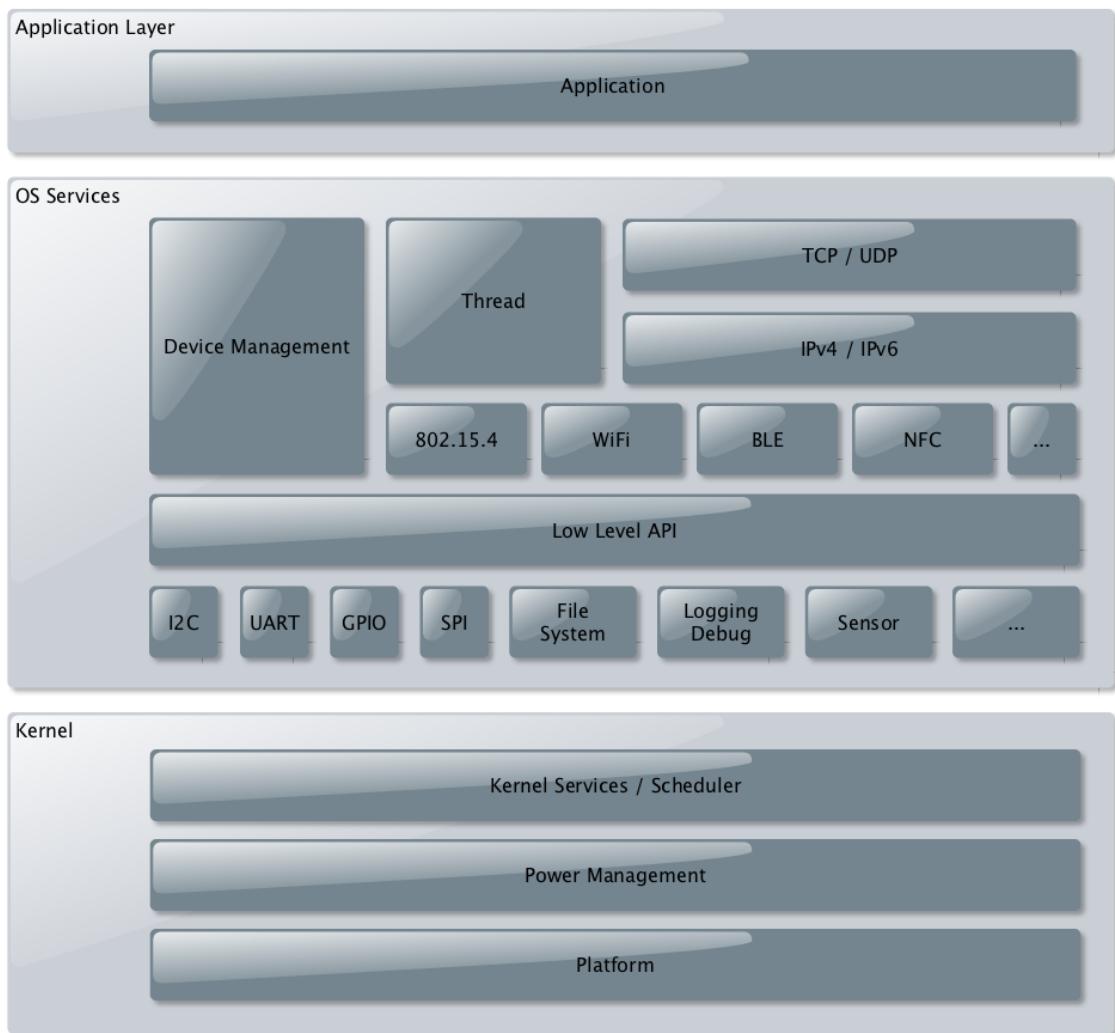


FIGURE 4.2 – Architecture du système d'exploitation Zephyr - <http://www.zephyrproject.org>

elles ont souvent la possibilité d'en avoir plusieurs. Enfin l'on peut spécifier une configuration de base, par exemple en activant par défaut la gestion du bus  $I^2C$ .

Dans le cadre du projet, une nouvelle configuration a dû être définie pour la carte SODAQ One.

Un fichier important est de type Device Tree (dts), c'est un type de fichier qui permet de définir avec des nœuds les composants disponibles. Les informations contenues dans ce fichier sont spécifiques à une certaine carte car elles dépendent de la façon dont les périphériques sont connectés au micro-contrôleur. Ce type de fichier est beaucoup utilisé par Linux par exemple.

Pour la carte SODAQ One, le fichier est principalement utilisé afin de définir le type de micro-contrôleur présent sur la carte ainsi que les composants SERCOM utilisés pour les divers périphérique. Enfin, on peut spécifier quel composant doit être utilisé par Zephyr pour la console.

Un extrait de ce fichier est présenté ci-dessous.

```
1  /*
2   * Copyright (c) 2018 Leonard Bise
3   *
4   * SPDX-License-Identifier: Apache-2.0
5   */
6
```

```
7 /dts-v1/;
8 #include <atmel/samd21.dtsi>
9
10 / {
11     model = "SODAQ One v3";
12     compatible = "sodaq,one-v3", "atmel,samd21g18a",
13         "atmel,samd21";
14
15     chosen {
16         zephyr,console = &sercom5;
17         zephyr,sram = &sram0;
18         zephyr,flash = &flash0;
19         zephyr,code-partition = &code_partition;
20     };
21 };
22
23 &sercom0 {
24     status = "ok";
25     compatible = "atmel,sam0-spi";
26     #address-cells = <1>;
27     #size-cells = <0>;
28 };
29
30 &sercom2 {
31     status = "ok";
32     compatible = "atmel,sam0-uart";
33     current-speed = <57600>;
34 };
35
36 &sercom3 {
37     status = "ok";
38     compatible = "atmel,sam0-i2c";
39     clock-frequency = <I2C_BITRATE_FAST>;
40 };
41
42 &sercom5 {
43     status = "ok";
44     compatible = "atmel,sam0-uart";
45     current-speed = <115200>;
46 };
47
48 ...
```

### 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 décrits dans la section 4.4, c'est lui qui va cadencer les acquisitions ainsi que l'envoi des paquets LoRa à la passerelle. Le logiciel embarqué ainsi que les drivers sont entièrement écrits 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 généré 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 générale et de façon simplifiée, on peut définir le processus du capteur comme dans la figure 4.3.

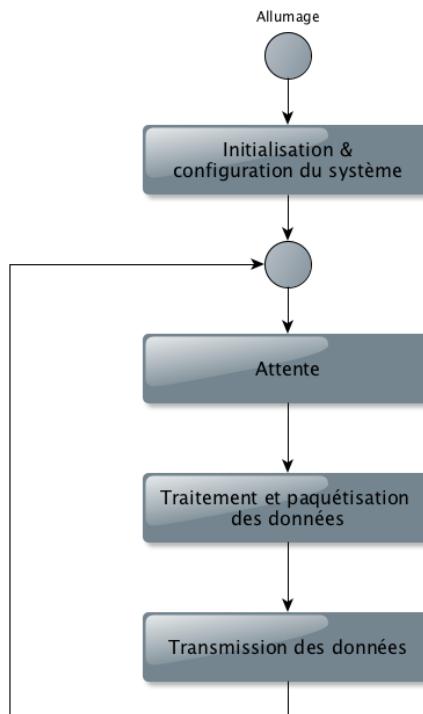


FIGURE 4.3 – Processus général du capteur

#### 4.3.1 Architecture logiciel

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

Les éléments qui composent la couche application du logiciel embarqué sont décrits 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 facilitent le debug de l'application.
- Heart Rate : Compte le nombre de battements du cœur par unité de temps en s'appuyant sur l'accéléromètre Adafruit Heart Rate starter pack
- Cadence : Permet le comptage des pas du sportif par unité de temps grâce à l'accéléromètre LSM303AGR

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

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

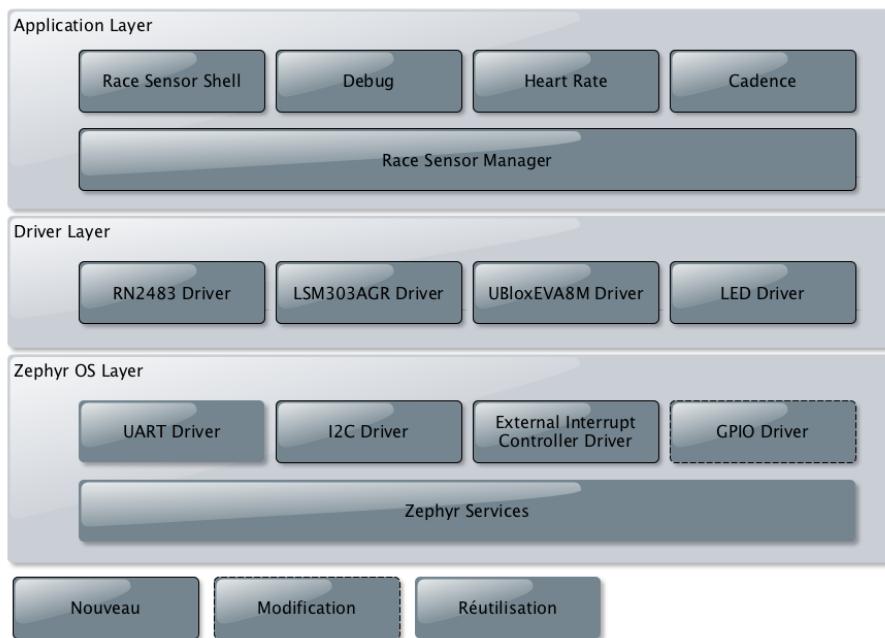


FIGURE 4.4 – Architecture statique du logiciel embarqué sur le capteur

thread, valeur entière, peut être soit positive ou négative. Une valeur de priorité plus petite est 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 thread en utilisant la valeur de priorité - des threads dit coopératifs et des threads pré-emptibles.

Les threads coopératifs ont une valeur de priorité négative. Une fois qu'un thread coopératif devient 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 thread 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. [8]

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

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	500 ms
Cadence sampling	Basse - 3	50 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 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

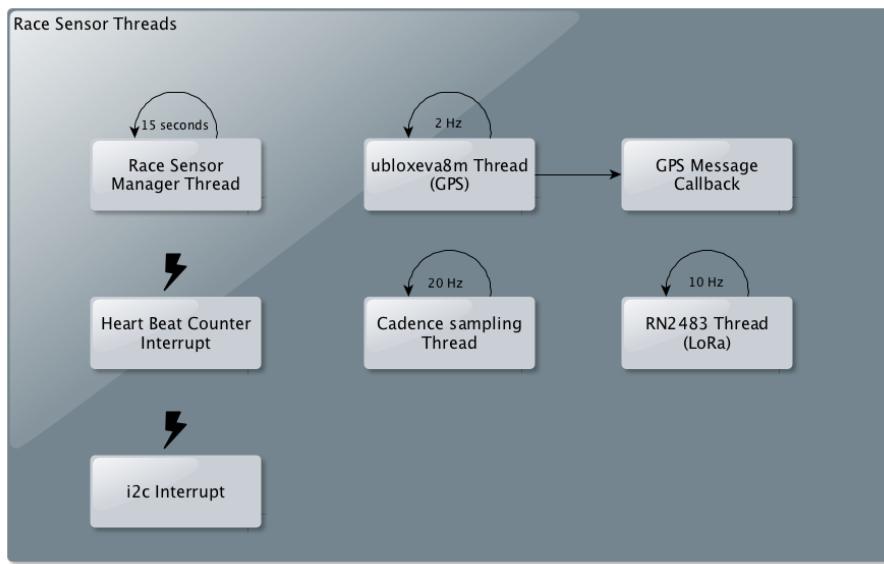


FIGURE 4.5 – Architecture dynamique du logiciel embarqué sur le capteur

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 principal 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 initialisés et les interfaces de communications sont configurées. Au terme de cette opération, le thread est lancé qui s'occupera du séquencement des opérations. Les opérations d'initialisation sont décrites 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

Le module Heart Rate est responsable de la gestion et du calcul du rythme cardiaque de l'athlète portant le capteur. Grâce au dispositif électronique permettant l'interfaçage avec la ceinture pectorale, ce module va placer une interruption, grâce au driver External Interrupt Controller et au driver GPIO de Zephyr, modifié pour l'occasion, sur la pin sur laquelle une impulsion est déclenchée lors d'un battement du cœur. En mesurant le temps écoulé ainsi que le nombre de battement il est ensuite possible de calculer le rythme cardiaque qui est ensuite transmis dans les paquets à destination de la passerelle.

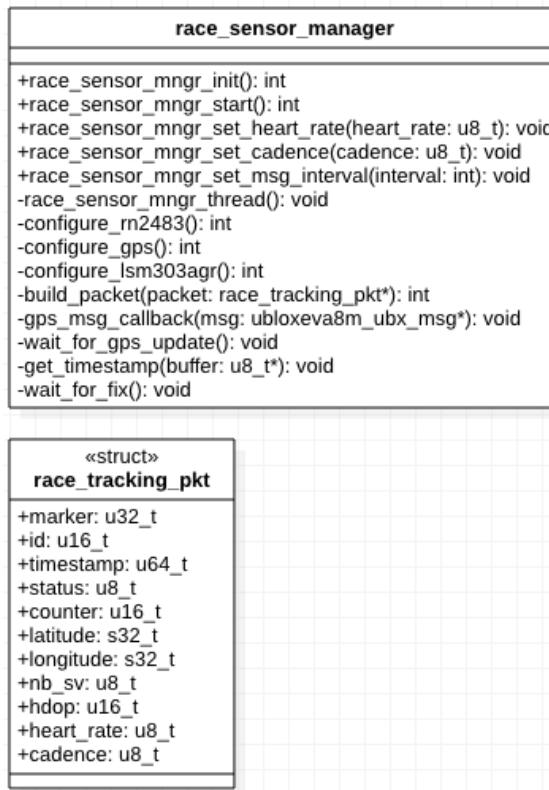


FIGURE 4.6 – Diagramme de classe du Race Sensor Manager

La figure 4.9 montre le diagramme de classe du module Heart Rate.

#### 4.3.4 Cadence

La cadence du sportif est déterminé grâce à ce module. En se basant sur les données de l'accéléromètre, qui est capable de déterminer les accélérations sur les 3 axes x,y et z, il est possible de trouver le moment où un pas est effectué et donc de les compter.

Au début du projet, il était prévu d'utiliser une fonctionnalité de l'accéléromètre LSM303AGR afin de générer une interruption lors du dépassement d'un certain seuil d'accélération sur un des axe. Cependant à cause d'un problème récalcitrant que je n'ai pas eu le temps résoudre dans le temps imparti, j'ai décidé de me replier sur une technique d'échantillonnage. Un thread est alloué à la récupération des échantillons à une fréquence de 20 Hz, tous les 20 échantillons, une analyse des sample est faite afin d'essayer de détecter un pas.

Afin de déterminer la façon dont on va pouvoir détecter un pas, la première étape a été de récupérer des données permettant de visualiser la valeur des échantillons sur une période de temps lorsqu'une personne équipé du capteur marche. Pour se faire, une application permettant la récupération des valeurs de l'accéléromètre a été écrite. Elle permet de récupérer un nombre d'échantillons correspondant à 10 secondes de marche. Les données ont ensuite été envoyées par le port série afin de pouvoir les analyser dans un logiciel graphique, Matlab en l'occurrence. Lors de l'acquisition des échantillons, le capteur a été tenu sans modifier son orientation et en marchant normalement.

Les résultats de ce test sont présentés dans la figure 4.10.

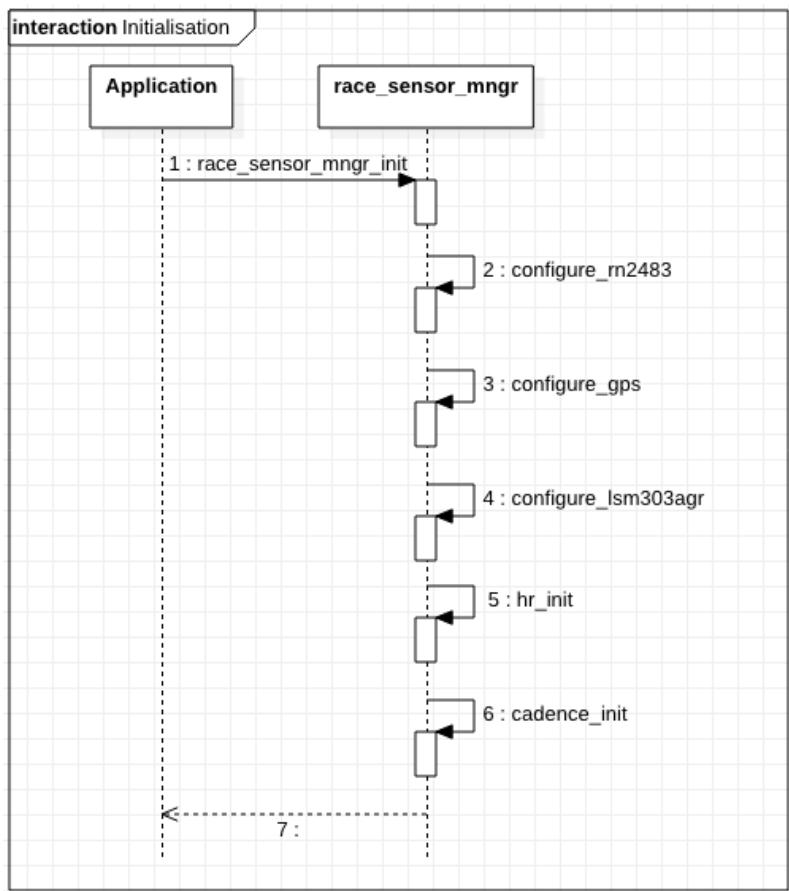


FIGURE 4.7 – Diagramme de séquence de l’initialisation du Race Sensor Manager

Lorsque le capteur est au repos, c'est à dire sans mouvement, et posé à plat, c'est à dire avec l'axe Z perpendiculaire au sol, les axes X et Y ont des valeurs proche de 0 g, la valeur de l'axe Z est quand à elle au alentour de -1 g ce qui correspond à la force de gravité. Si l'inclinaison est modifiée alors la valeur de la gravité se distribue entre les axes sur lesquels le mouvement s'opère.

La table 4.3 présente les valeurs minimum, maximum et médiane pour chaque axe en g.

TABLE 4.3 – Analyse des valeurs des échantillons

Axe	Min	Médian	Max
x	-0.288	-0.092	0.216
y	-0.008	0.196	0.352
z	0.708	0.938	1.444

Si le capteur est posé à plat sans bouger et que l'on marche sans modifier son orientation la tâche consistant à compter le nombre de pas est assez facile, il suffit de détecter lorsque la valeur de l'axe donné dépasse un certain seuil. En l'occurrence si l'on analyse les données on peut remarquer qu'une valeur dépassant les 1.125 g semble permettre de pouvoir compter le nombre de pas.

Puisque le capteur sera porté sur le bras des sportifs on ne peut pas partir du principe que l'orientation fixe puisqu'elle sera à même de changer pendant les mouvements, ceci rend la

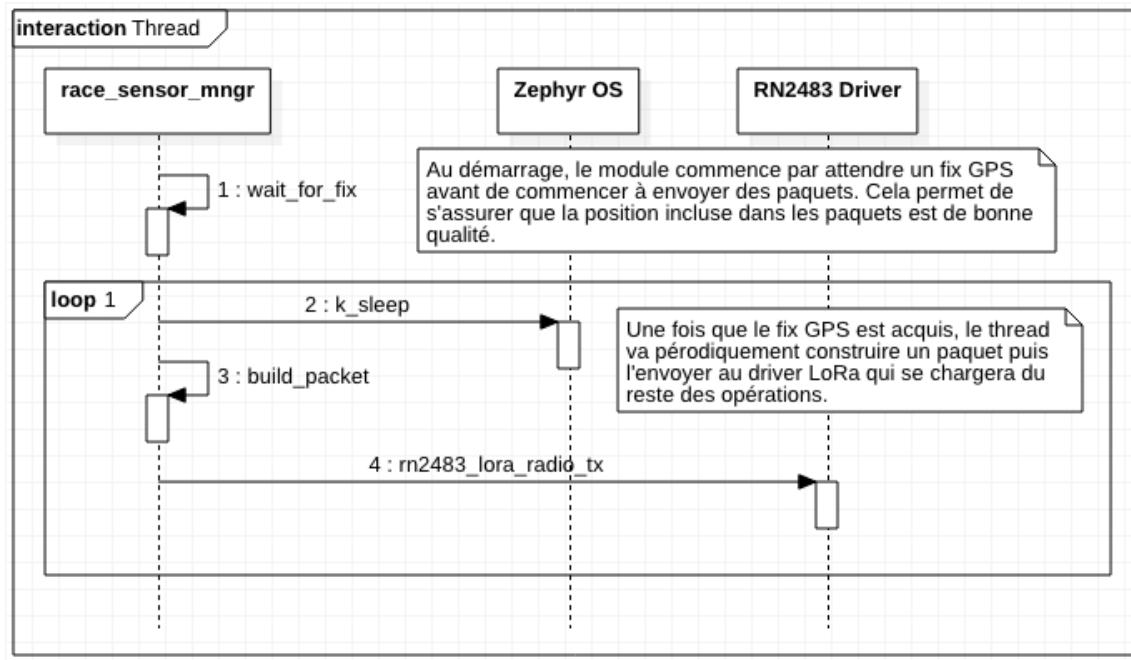


FIGURE 4.8 – Diagramme de séquence du thread du Race Sensor Manager

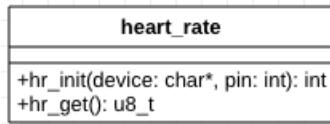


FIGURE 4.9 – Diagramme de classe du module Heart Rate

tâche de détection des pas beaucoup plus ardue.

Une approche qui peut être étudié est de calculer la moyenne des échantillons pour chaque axe, cette opération peut permettre de trouver quels sont les axes qui doivent être analysé afin de détecter les piques et également de connaître le seuil auquel on doit les compter comme pas.

La figure 4.11 montre le diagramme de classe du module Cadence.

### 4.3.5 Debug

Ce module propose quelques fonctions et macros qui sont utiles 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 disponibles dans le shell du capteur :

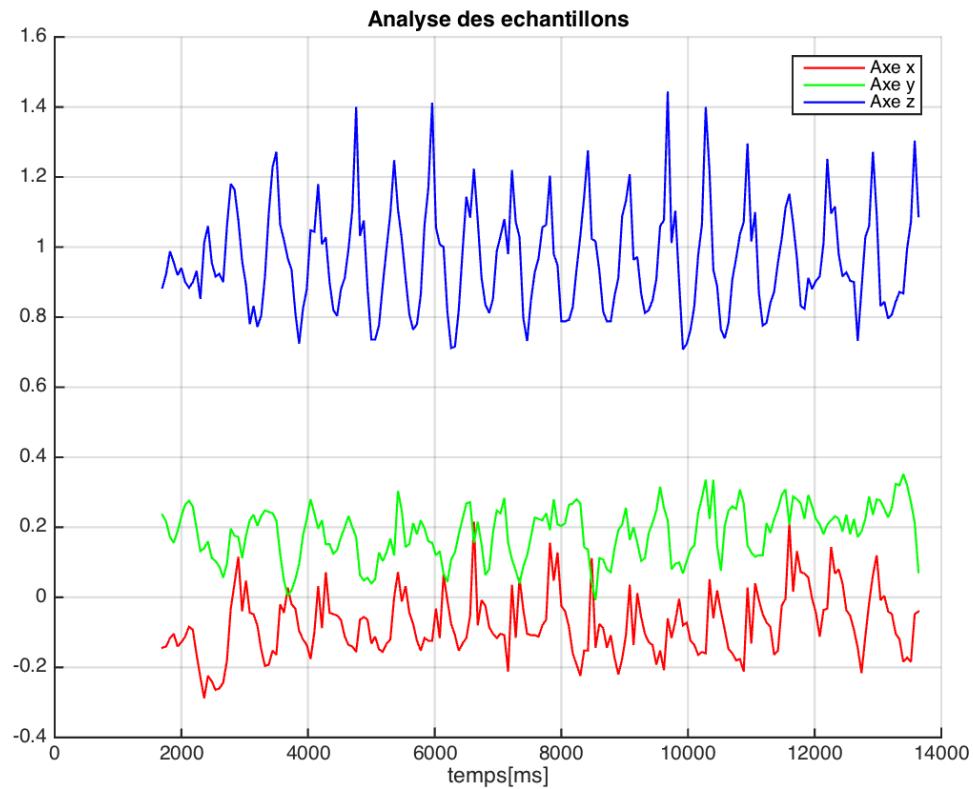


FIGURE 4.10 – Valeurs des échantillons de l'accéléromètre sur une période de 10 secondes

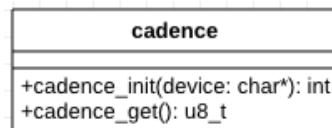


FIGURE 4.11 – Diagramme de classe du module Cadence

- 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 l'envoi de deux messages

## 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
- Driver External Interrupt Controller permet de déclencher une interruption en fonction de l'état d'un I/O qui est utilisé pour pouvoir compter les battements du cœur

Enfin le Driver GPIO de Zephyr a dû être modifié afin de pouvoir utiliser les interruptions sur les I/O proposés par le driver External Interrupt Controller.

#### 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 [9]. Les composants SERCOM  $I^2C$  sont décrits à partir de la page 545. L'application note [10] a également été consultée.

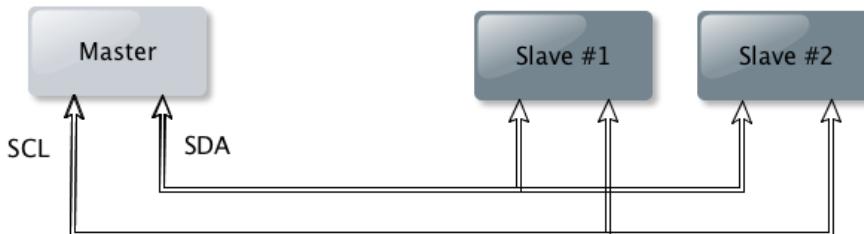
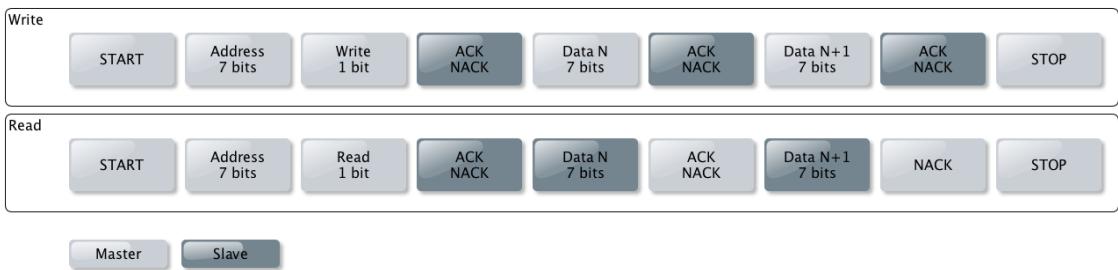


FIGURE 4.12 – Schéma d'un bus  $I^2C$

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 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 envoyer un message de type NACK suivi d'un STOP afin d'informer l'esclave de la fin du transfert. La figure 4.13 propose les deux types de messages qu'il existe.

Le driver  $I^2C$  est responsable du pilotage des SERCOM que propose le micro-contrôleur, ce sont des modules qui peuvent être configurés 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 de proposer une interface standardisée qui permet au système d'exploitation d'initier les opérations suivantes :

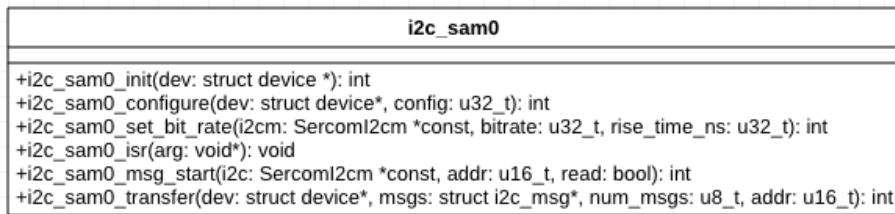
FIGURE 4.13 – Les messages  $I^2C$ 

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

Pour ce 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 qui va envoyer les codes (START ou STOP) suivis des données. Il produira les acquittements nécessaire et il vérifiera également que l'esclave en question valide bien les messages. L'utilisation d'une 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'envoi du prochain message.

La figure 4.14 présente le diagramme de classe du driver  $I^2C$ .

FIGURE 4.14 – 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 msg[0] = 0x11;
26 msg[1] = 0x22;
27 msg[2] = 0x33;
28 msg[3] = 0x44;
29
30 /* Send message to i2c slave */
31 if (i2c_write(i2c_dev, msg, 4, I2C_DEVICE_ADDR)) {
32     DBG_PRNTK("%s: I2C access failed\n", __func__);
33 }
34
35 /* Read from i2c slave */
36 if (i2c_read(i2c_dev, msg, 4, I2C_DEVICE_ADDR)) {
37     DBG_PRNTK("%s: I2C access failed\n", __func__);
38     return 0;
39 }
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 idéale pour des capteurs de petite taille. Il est décrit plus en détail 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 formés de caractère ASCII. Il est idéal 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 envoi vers le récepteur) qui permet au récepteur 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 [6].

Une fois que le module a été configuré, il va produire les messages qui ont été activés et contenant diverses informations et les placer dans une FIFO. Au travers du bus  $I^2C$ , le thread du driver va interroger le module périodiquement afin de savoir si des messages sont prêts à être lus, si c'est le cas le driver va lire les messages disponibles et les stocker dans une structure de type ubloxeva8m\_ubx\_msg. L'utilisateur récupère les messages reçus 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 nécessaires 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 contenues dans ce message, pour plus de détails voir [6, p. 307].

- La date
- L'heure actuelle
- Le type de fix GPS
- La validité et la précision des informations contenues 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 présents dans la FIFO, si c'est le cas ils sont lus et ensuite décodés. Lorsque les messages sont prêts, ils sont ensuite transférés à l'application au travers d'une fonction de callback.

La figure 4.15 présente le diagramme de classe du driver UBloxEVA8M.

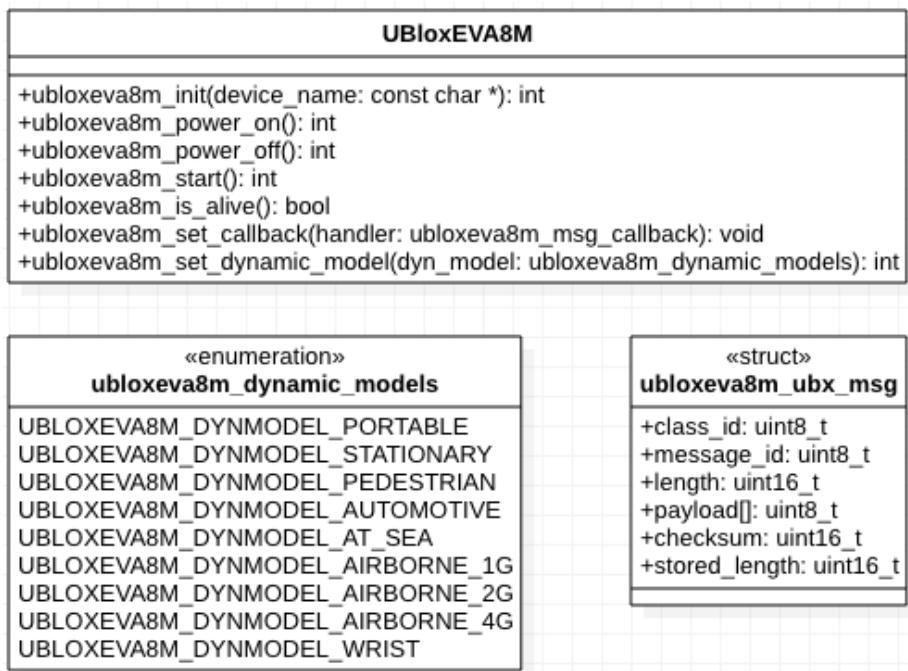


FIGURE 4.15 – Diagramme de class du driver UBloxEVA8M

Un exemple d'utilisation du driver UBloxEVA8M est proposé ci-dessous. **TODO: Le principe d'utilisation et la**  
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_PRNTIK("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);

```

```

6 }
7
8 /* Callback function on GPS message */
9 static void gps_msg_callback(ubloxeva8m_ubx_msg* msg)
10 {
11     ubloxeva8m_nav_pvt_t pvt_msg;
12
13     /* Check message type */
14     if (msg->class_id == UBLOXEVA8M_CLASS_NAV && msg->message_id == UBLOXEVA8M_MSG_NAV_PVT)
15     {
16         /* Copy message in struct */
17         memcpy(&pvt_msg, msg->payload, sizeof(ubloxeva8m_nav_pvt_t));
18         print_nav_pvt_msg("UBX-NAV-PVT: ", &pvt_msg);
19     }
20 }
21 void main(void)
22 {
23     int err;
24
25     /* Initialize the gps */
26     err = ubloxeva8m_init(CONFIG_I2C_SAM0_SERCOM3_LABEL);
27     if (err) {
28         DBG_PRNTK("%s: Can't initialize UBlxEVA8M %d\n", __func__, err);
29         return;
30     }
31
32     /* Set callback function */
33     ubloxeva8m_set_callback(gps_msg_callback);
34
35     /* Start module */
36     err = ubloxeva8m_start();
37     if (err) {
38         DBG_PRNTK("%s: Can't start GPS module %d\n", __func__, err);
39         return;
40     }
41
42     /* Set the dynamic model used by the GPS */
43     err = ubloxeva8m_set_dynamic_model(UBLOXEVA8M_DYNMODEL_AUTOMOTIVE);
44     if (err) {
45         DBG_PRNTK("%s: Can't set dynamic model %d\n", __func__, err);
46         return;
47     }
48 }
```

#### 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 [11].

Le composant LSM303AGR est configuré au travers d'une liste de registre, il est également possible de lire les différentes valeurs mesurées au travers d'un autre groupe de registre. Tous les registres disponibles sont détaillés dans le document [11, 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 noter 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 utilisée afin de détecter les pas effectués par le porteur de capteur.

La figure 4.16 présente le diagramme de classe du driver LSM303AGR.

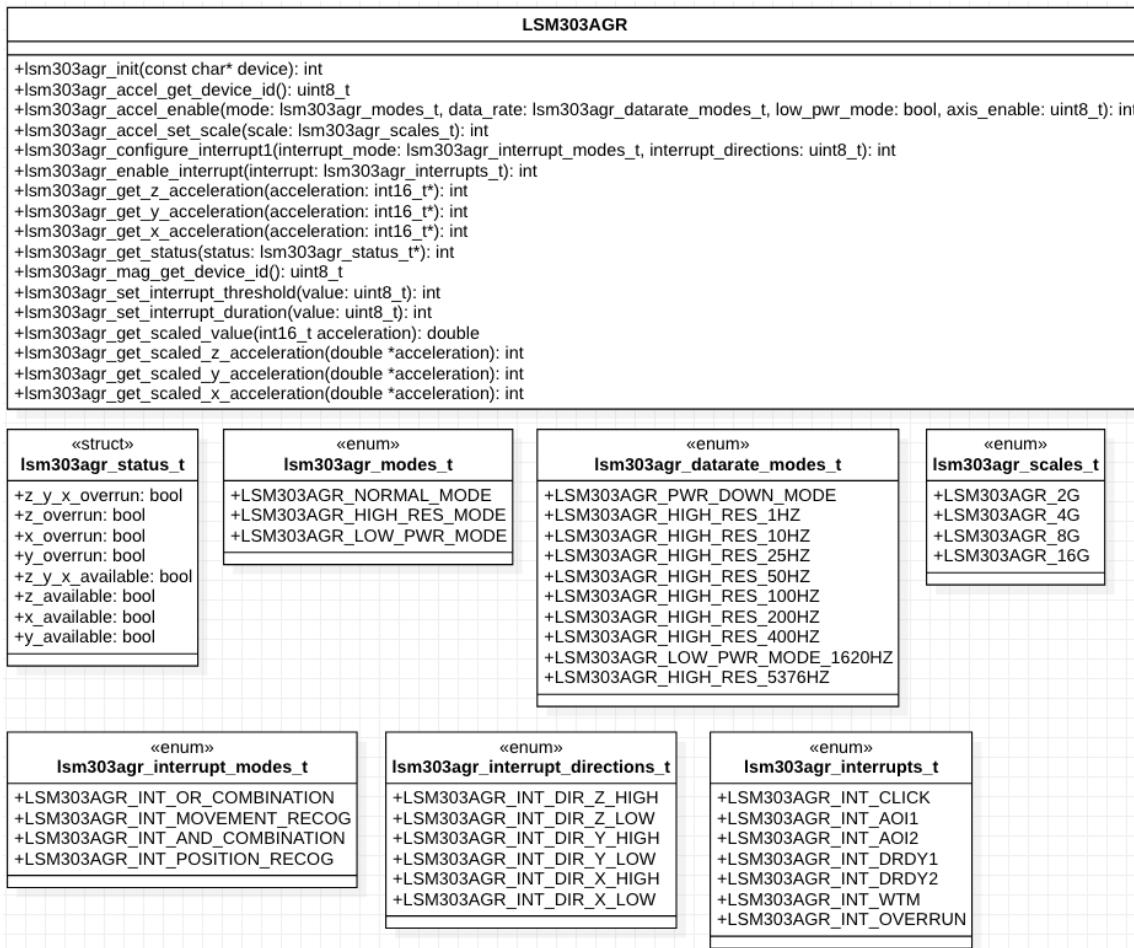


FIGURE 4.16 – Diagramme de class du driver LSM303AGR

Un exemple d'utilisation du driver est proposé 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 }
```

#### 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 permet d'effectuer les différentes opérations requises pour l'envoi de données.

Le protocole est détaillé dans le document Command Reference User's Guide [7].

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 4.17 présente le diagramme de classe du driver RN2483.

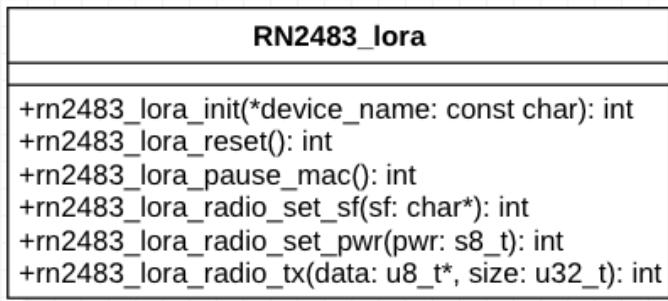


FIGURE 4.17 – Diagramme de class du driver RN2483

Un exemple d'utilisation du driver est proposé 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 bleu, 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 4.18 présente le diagramme de classe du driver LED.

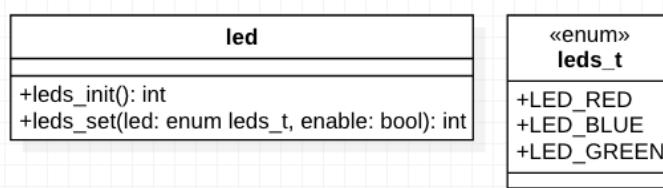


FIGURE 4.18 – Diagramme de classe du driver LED

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

```

1 #include "led.h"
2
3 void main(void)
4 {
5     int err;
6
7     err = leds_init();
8     if (err) {
9         DBG_PRNTK("%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 }
```

#### 4.4.6 Driver External Interrupt Controller

Lorsqu'un battement du cœur du sportif est détecté par le module rythme cardiaque, une impulsion est générée sur une ligne connectée à un GPIO du micro-contrôleur. Afin de pouvoir compter proprement les battements, une interruption doit pouvoir être déclenchée lorsque l'impulsion est détectée. Dans Zephyr, cela se traduit par la configuration au travers du driver GPIO, on peut spécifier si l'on souhaite déclencher une interruption et dans quelles conditions, flanc montant ou descendant ou alors lors d'un niveau haut ou bas. Cependant cette fonctionnalité n'était pas présente au moment du travail de Bachelor. Afin de pouvoir proposer cette fonctionnalité, un driver de gestion des interruptions externe ou External Interrupt Controller (EIC) a dû être développé. Une fois le driver développé, il a fallu ensuite modifier le driver GPIO afin de pouvoir proposer cette fonctionnalité.

La figure 4.19 présente le diagramme de classe du driver External Interrupt Controller.

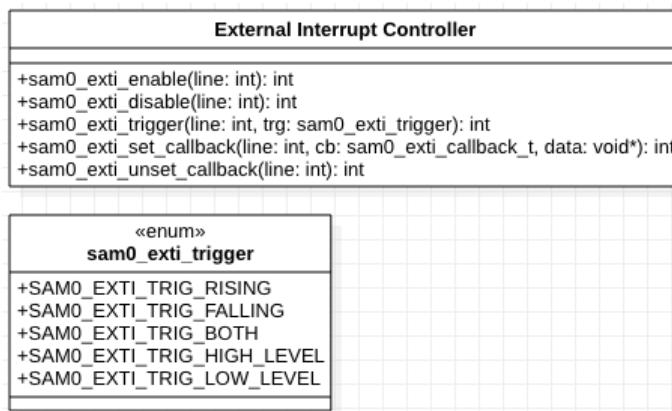


FIGURE 4.19 – Diagramme de class du driver External Interrupt Controller

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

```

1 #include "exti_sam0.h"
2
3 void gpio_callback(int line, void *user)
4 {
5     DBG_PRNTK("GPIO interruption triggered");
6 }
7
8 void main(void)
9 {
10     int err;
11     int line = 1;
```

```

12
13     err = sam0_exti_enable(line);
14     if (err) {
15         DBG_PRNTK("%s: Cannot enable external interrupt\n", __func__);
16         return;
17     }
18
19     err = sam0_exti_trigger(line, SAM0_EXTI_TRIG_RISING);
20     if (err) {
21         DBG_PRNTK("%s: Cannot set interrupt trigger type\n", __func__);
22         return;
23     }
24
25     err = sam0_exti_set_callback(line, gpio_callback);
26     if (err) {
27         DBG_PRNTK("%s: Cannot set interrupt trigger type\n", __func__);
28         return;
29     }
30
31     while (1) {
32         k_sleep(K_SECONDS(1));
33     }
34 }
```

## 4.5 Paquet de donnée

Le capteur transmet les données qui sont acquises au cours de la course grâce à un paquet de donnée LoRa. Le format et le contenu de ce paquet sont présentés ci-dessous.

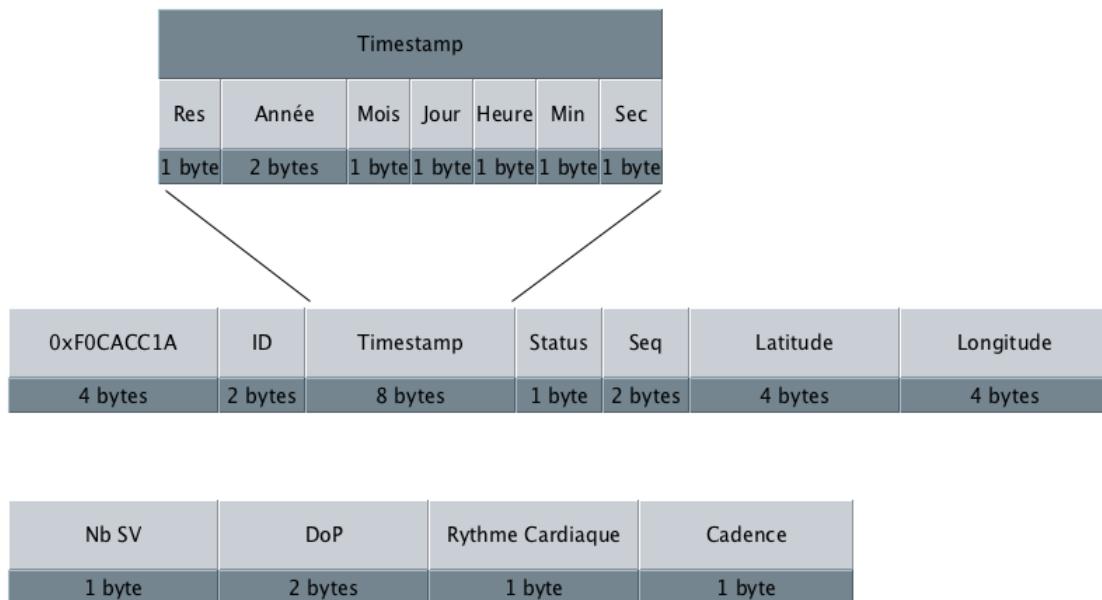


FIGURE 4.20 – Format du paquet de donnée LoRa

Le paquet fait une taille totale de 30 bytes et est composé des éléments suivants.

TABLE 4.4 – Détails des champs du paquet de données radio LoRa

Champs	Taille	Type	Description
Marqueur	4	uint	Une valeur fixe servant de marqueur et permettant d'identifier que le paquet provient bien d'un capteur du système
ID	1	uint	Identifiant du capteur qui a généré le paquet
Timestamp	8	uint	Heure et date au moment de la génération du paquet
Status	1	uint	Status du capteur
Seq	1	uint	Numéro de séquence du paquet. Chaque fois qu'un paquet est généré le compteur est incrémenté de un, cela permet de détecter lorsque des paquets sont perdus
Latitude	4	int	Latitude du capteur reporté directement depuis le module GPS
Longitude	4	int	Longitude du capteur reporté directement depuis le module GPS
Nb SV	1	uint	Nombre de satellites GPS en vue
DoP	2	int	Dissolution of Position, la qualité de la précision de la position GPS
Rythme cardiaque	1	uint	Rythme cardiaque de l'athlète
Cadence	1	uint	Cadence de l'athlète

## 4.6 Le boîtier

Afin de pouvoir contenir le capteur ainsi que les éléments associés, une boîte a été construite puis imprimée sur une imprimante 3D. Le logiciel utilisé pour la création des plans s'appelle Fusion 360 de la compagnie Autodesk.

La boîte permet d'insérer la batterie dans un compartiment, fermable grâce à un clapet, afin de la protéger et d'éviter qu'elle ne bouge durant la course du sportif. Une bande en tissu élastique permet d'attacher le capteur à son bras ce qui permet de ne pas trop gêner le porteur.

**TODO:** Images vrai boîte

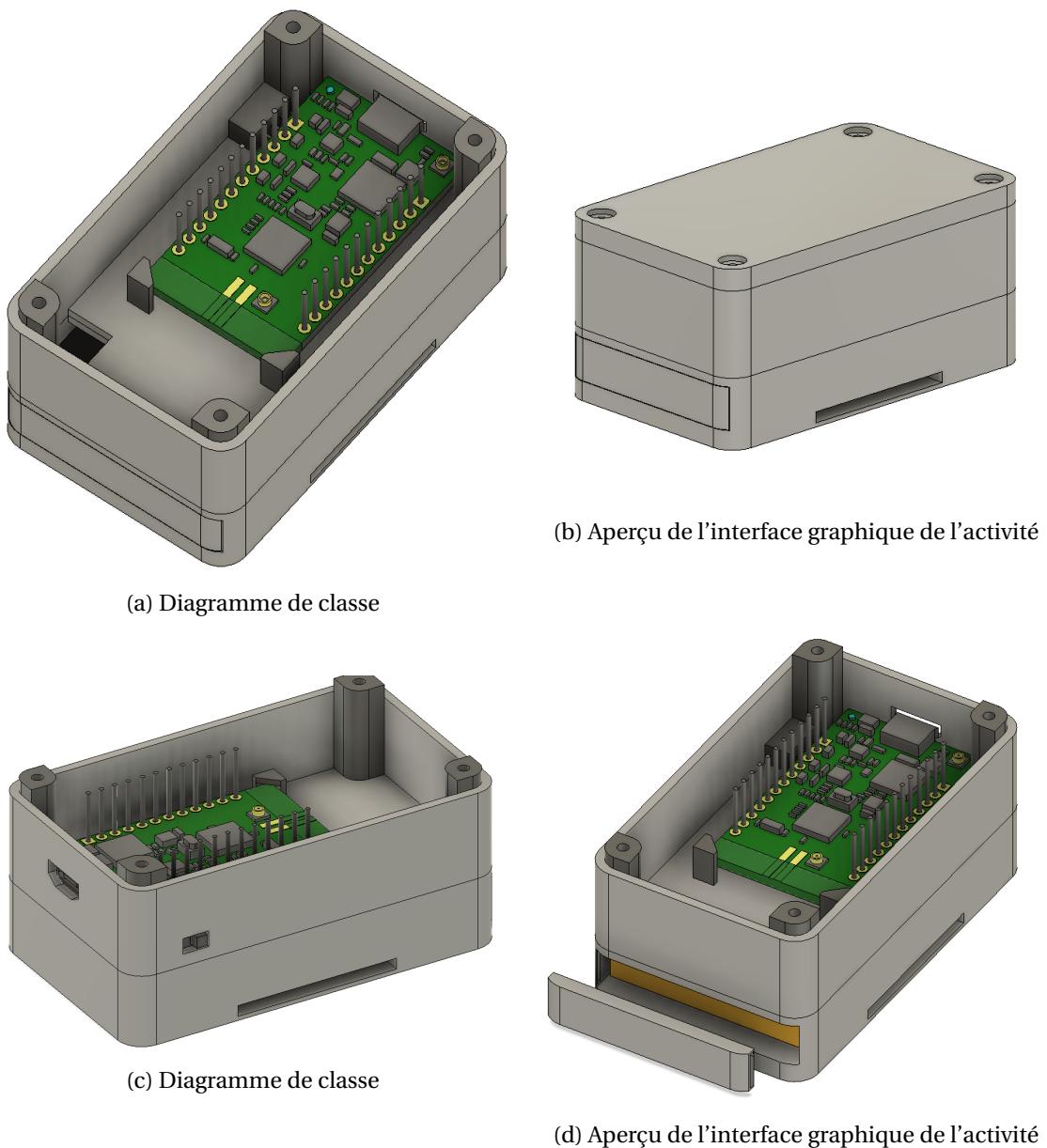


FIGURE 4.21 – Image du concept de la boîte du capteur



## 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.

D'autre part, la passerelle fait également office d'access point WiFi, les applications qui souhaitent pouvoir accéder à la base de données se connectent directement à ce réseau. Ce sera le cas de l'application mobile par exemple.

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 a été 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 nécessaire 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.

Pendant la pré-étude, deux concentrateurs différents avaient été étudiés, le choix final s'est porté sur la solution d'un concentrateur moins couteux, 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 est assemblé. Le Dragino LoRa HAT est un module d'extension pour le 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 5.1 – 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

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 effectuée par le logiciel packet forwarder détaillé dans la chapitre 5.2.

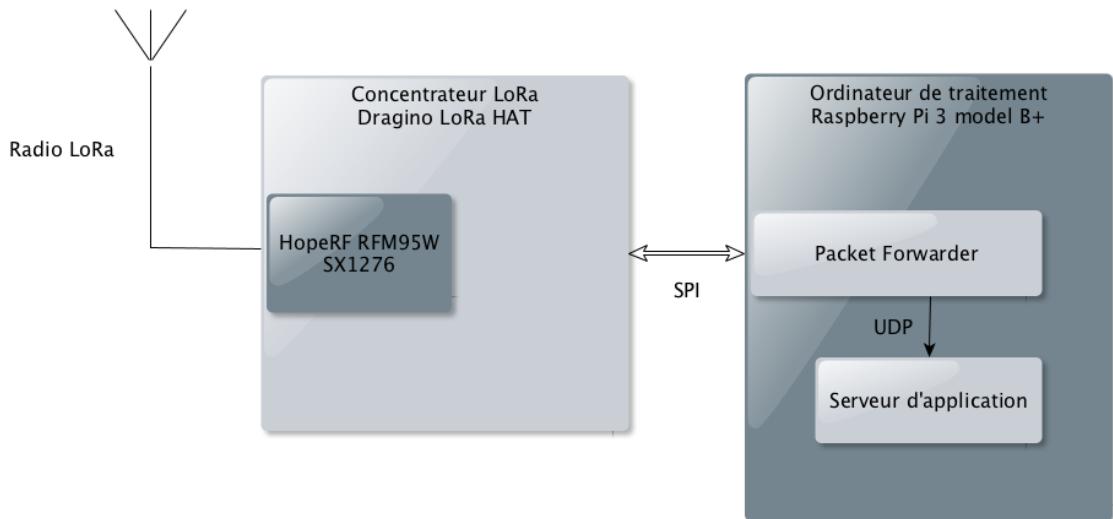


FIGURE 5.1 – Schéma block de la passerelle

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 et en analyser le contenu afin de pouvoir créer un objet json conte-

nant 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 transformées en json, le tout est envoyé au moyen d'un paquet UDP à un serveur, dans notre cas le serveur d'application. La liste suivante décrit les informations contenues 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 créés 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.

```

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 détail sur la page github du packet forwarder de Semtech, voir

[12].

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 objets json envoyés 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 fonctionnalités 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. [13]

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, course 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 ré-



FIGURE 5.2 – Architecture statique du serveur d'application

ceptionne 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
- Race Mode Handler : Gestionnaire du mode "Race", une fois que les données du paquet reçu sont extraites, c'est cette classe qui stocke 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 reçus 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 parsé, 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 decodage de chaîne de caractère encodée en base64
- rapidjson : Permet la manipulation d'objet de type json, développé par la société Tencent
- vector reader : Une classe qui permet la lecture de données depuis un vecteur d'octet

- 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 différente

La figure 5.3 montre le flux des données reçues par le serveur d'application et quelle classe est responsable de sa gestion.

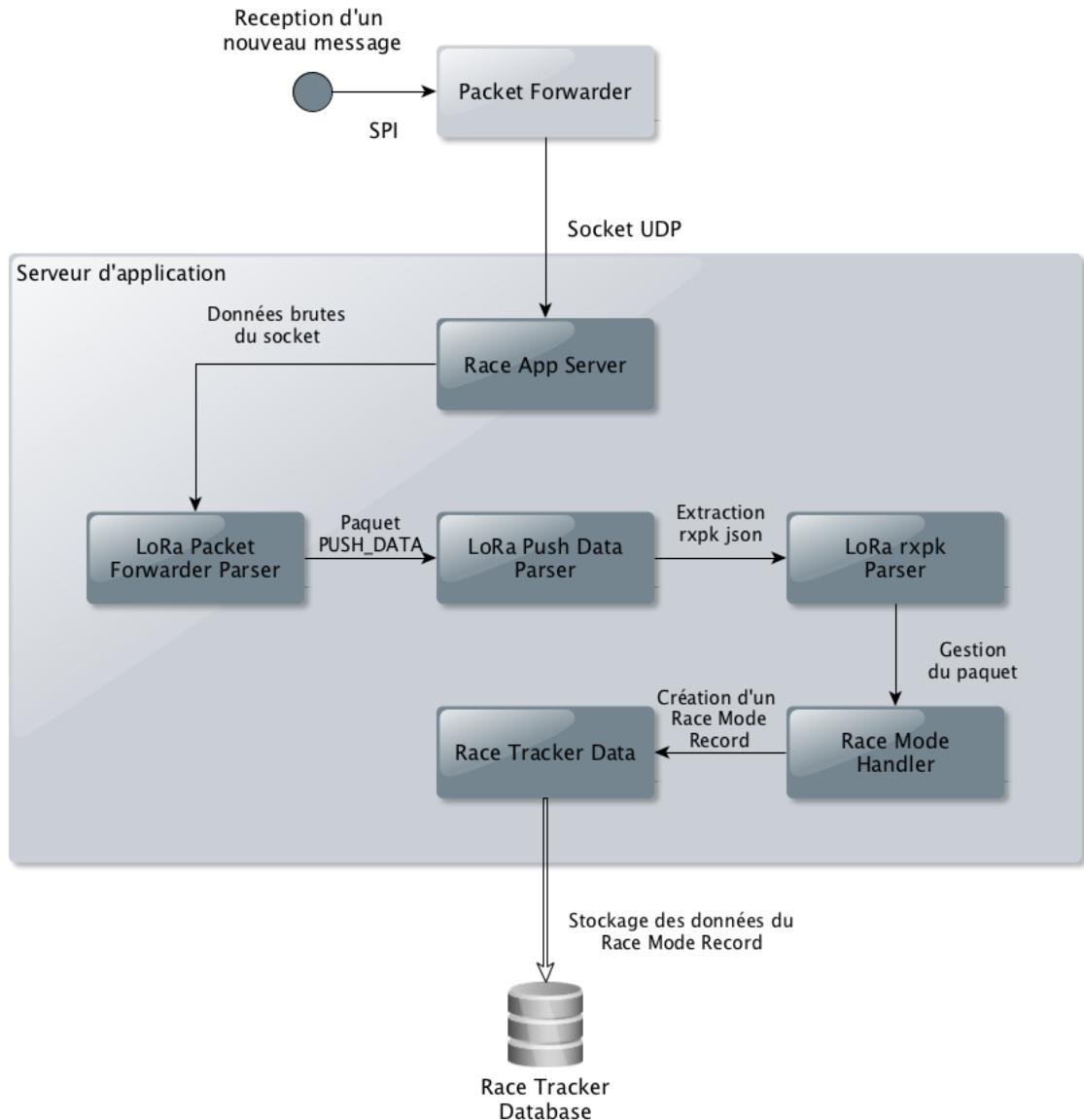


FIGURE 5.3 – Flux des données du serveur d'application

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 5.4 montre l'architecture dynamique du serveur d'application.

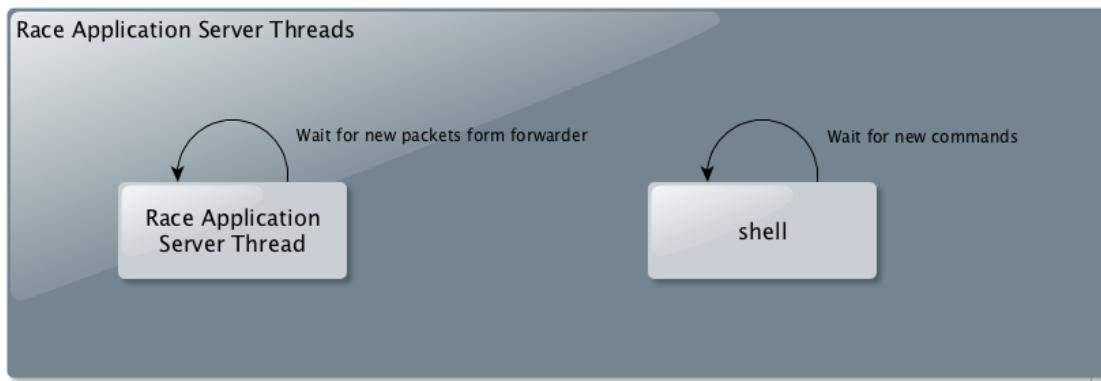


FIGURE 5.4 – Architecture dynamique du serveur d'application

### 5.3.2 Les librairies externes

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 relationnelles. Elle est open source, multi-plateforme et disponible gratuitement avec une licence BSD sur internet à l'adresse <http://pqxx.org/development/libpqxx/>.

Un exemple simple d'exécution d'une requête est présenté ci-dessous.

```

1 #include <pqxx/pqxx>
2
3 int main(void) {
4     pqxx::connection c("dbname = race_tracker_db user = pi password = heig hostaddr =
5     127.0.0.1 port = 5432");
6     pqxx::work t{c};
7
8     if (!c.is_open()) {
9         throw std::runtime_error("Cannot connect to database " + connection_str);
10    }
11
12    c.prepare("print_competitions", "SELECT * FROM race_tracker.competition;");
13
14    pqxx::result r = t.prepared("print_competitions").exec();
15
16    for (auto row: r) {
17        std::cout << "Competition id=" << row["competition_id"] << "name=" << row["name"] <<
18        std::endl;
19    }
20
21    return 0;
22 }
```

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](https://github.com/Lora-net/packet_forwarder/blob/master/lora_pkt_fwd/src/base64.c)

```

1 #include "base64.h"
```

```
2
3 #define BUF_SIZE 256
4
5 int main(void) {
6     uint8_t buffer[BUF_SIZE];
7     int nb_bytes_conv;
8     char b64_str[] = { "-DS4CGaDCdG+48eJNM3Vai-zDpsR71Pn9CPA9uCON84" };
9     int bin_size = 32;
10
11    /* Decode data from base64 */
12    nb_bytes_conv = b64_to_bin(b64_str, strlen(b64_str), buffer, BUF_SIZE);
13    if (nb_bytes_conv == -1 || (unsigned int)nb_bytes_conv != bin_size) {
14        throw std::runtime_error("Couldn't convert the base64 data (actual=" + std::to_string(
15            nb_bytes_conv) + " expected=" + std::to_string(bin_size)));
16    }
17
18    std::cout << "Decoded data=";
19    for (int i = 0; i < bin_size; i++) {
20        std::cout << (unsigned)buffer[i];
21    }
22
23    std::cout << std::endl;
24 }
```

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/>.

```
1 #include <rapidjson/document.h>
2
3 int main(void) {
4     std::string json_str =
5     {
6         "hello": "world",
7         "t": true,
8         "f": false,
9         "n": null,
10        "i": 123,
11        "pi": 3.1416,
12        "a": [1, 2, 3, 4]
13    };
14    rapidjson::Document doc;
15    static const char* kTypeNames[] = { "Null", "False", "True", "Object", "Array", "String",
16                                      "Number" };
17
18    rapidjson::StringStream packet_stream(json_string.c_str());
19    doc.ParseStream(packet_stream);
20
21    /* Print all json members name and type */
22    for (Value::ConstMemberIterator itr = document.MemberBegin(); itr != document.MemberEnd()
23        (); ++itr)
24    {
25        printf("Type of member %s is %s\n", itr->name.GetString(), kTypeNames[itr->value.
26        GetType()]);
27    }
28 }
```

```
26     return 0;  
27 }
```

### 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 5.5 montre le diagramme de séquence de la classe.

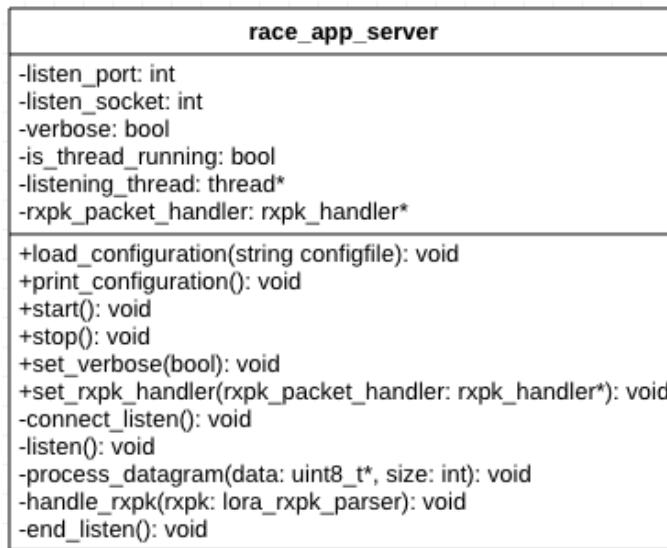


FIGURE 5.5 – Diagramme de classe de Race App Server

Les opérations effectuées pour traiter un nouveau paquet reçu sont décrites dans le diagramme de séquence 5.6.

#### Race Tracker Data

L'interface entre la base de données et le serveur d'application est gérée par la classe Race Tracker Data. Par le biais de librairie pqxx, elle va effectuer les requêtes SQL nécessaires 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 5.7 montre le diagramme de classe de Race Tracker Data.

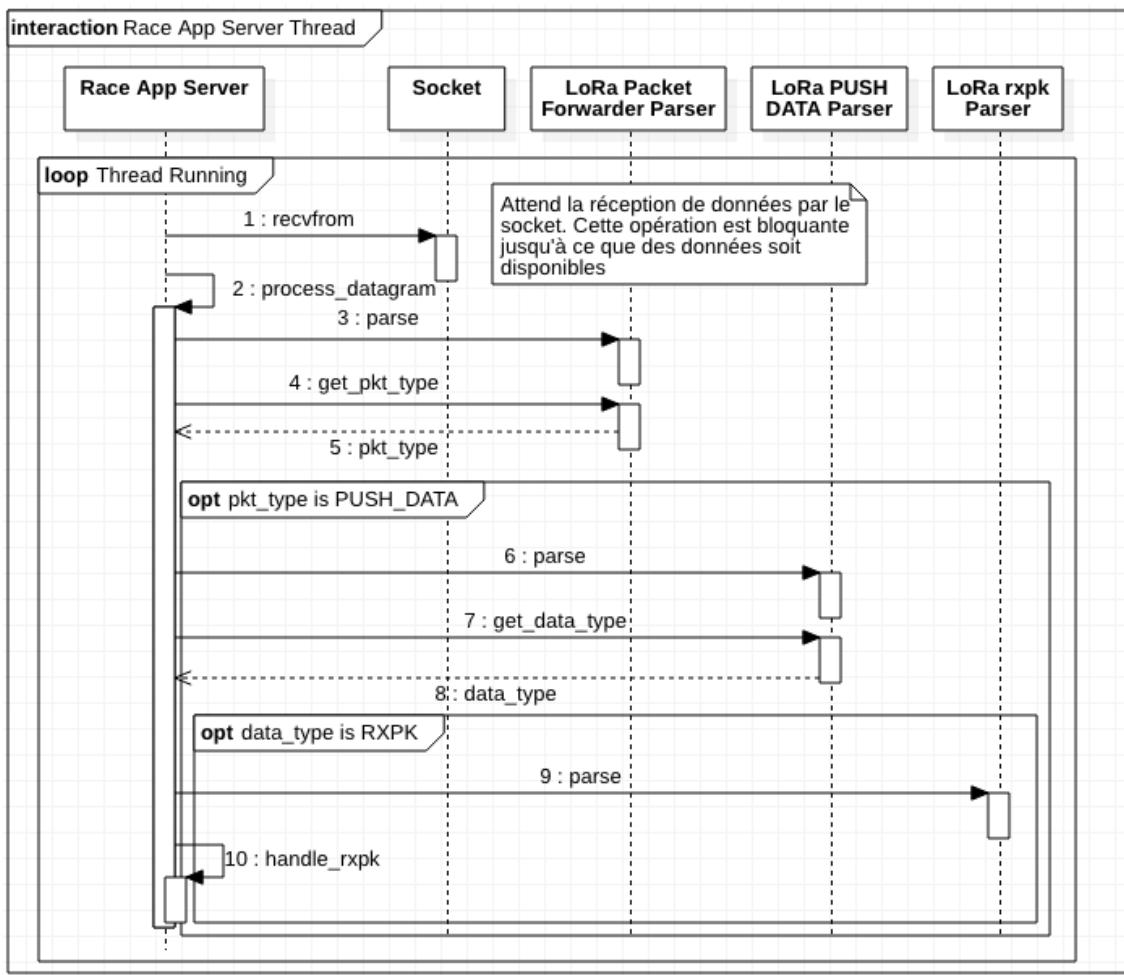


FIGURE 5.6 – Diagramme de séquence des opérations du thread de la classe Race App Server

### 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 5.8 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

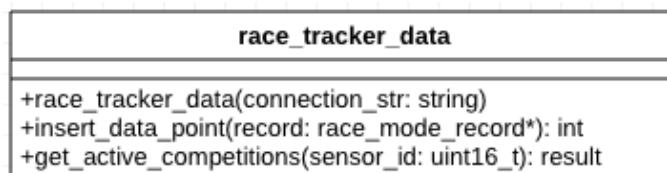


FIGURE 5.7 – Diagramme de classe de Race Tracker Data

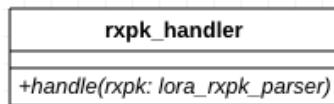


FIGURE 5.8 – Diagramme de classe de rxpk Handler

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 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.

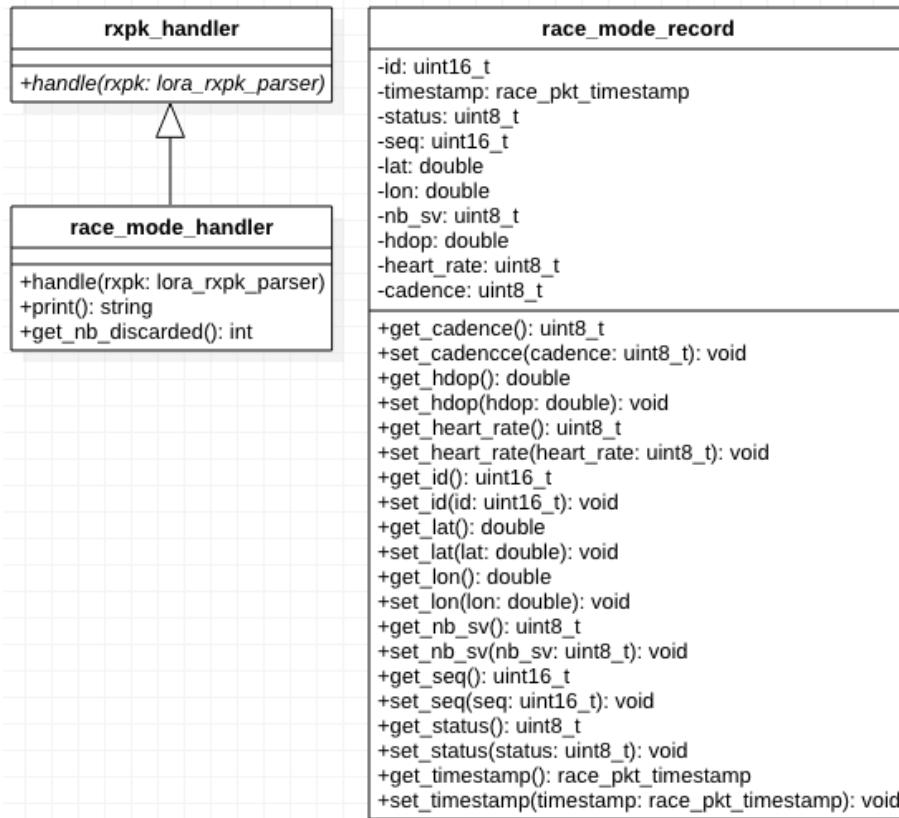


FIGURE 5.9 – Diagramme de classe de Race Mode Handler et Race Mode Record

### 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 9 et contrairement au mode "Race" il ne sauvegarde pas les données ainsi reçues dans la base de données, mais uniquement localement en mémoire et, si demandé par l'utilisateur, dans un fichier.

La figure 5.10 montre le diagramme de classe de Test Mode Handler et Test Mode Record.

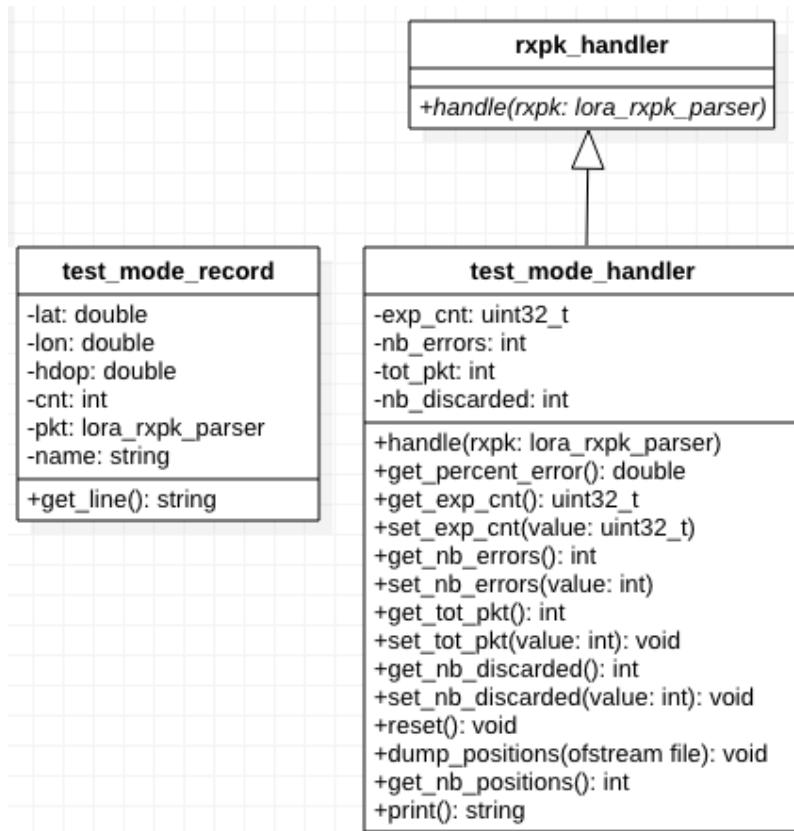


FIGURE 5.10 – Diagramme de classe de Test Mode Handler et Test Mode Record

### LoRa Packet Forwarder Parser

Le LoRa Packet Forwarder Parser permet de parser les données reçues par le socket et de déterminer si c'est un paquet de type PUSH\_DATA.

La figure 5.11 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 5.12 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 intéressante qui sont contenues dans l'objet json. Grâce aux fonctionnalités de la librairie 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.

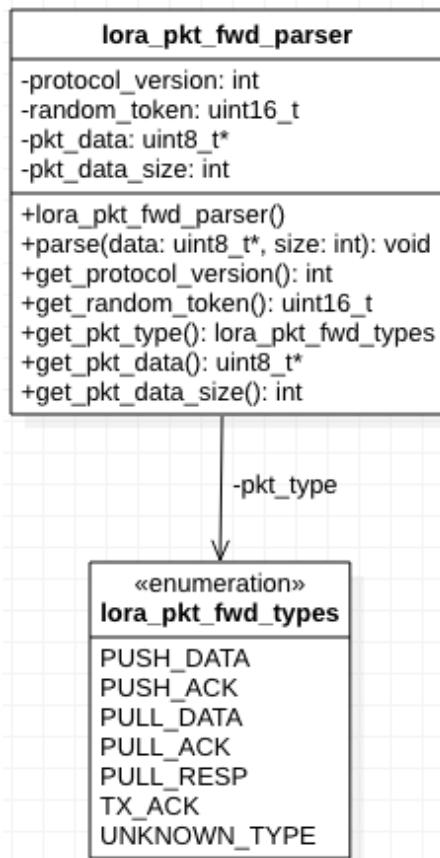


FIGURE 5.11 – Diagramme de classe de LoRa Packet Forwarder Parser

La figure 5.13 montre le diagramme de classe de LoRa rxpk Parser.

### Vector Reader

Le Vector Reader est une classe qui permet l'extraction de données depuis un vecteur de byte.

**TODO:** Les données qui sont reçues dans les paquets sous forme d'objet json contient les données encodée en base64. Une fois les données décodées, elles sont transformées en vecteur de byte et c'est donc grâce à la classe Vector Reader que l'on peut en extraire les différents paramètres.

La figure 5.14 montre le diagramme de classe de Vector Reader.

### Shell & Shell Command

La classe Shell est responsable de la gestion du shell de la passerelle qui permet à l'utilisateur d'exécuter des commandes afin d'effectuer diverses opérations. C'est cette classe qui dispose d'un thread en écoute sur l'entrée du clavier, lorsqu'une commande est entrée le Shell va aller regarder dans sa liste de commande si elle existe et l'exécuter. La Shell Command est une classe abstraite qui permet de définir le nom et le comportement d'une commande. Une fois définie, elle est ajoutée au Shell au moyen de la méthode *add\_cmd()*.

La figure 5.15 montre le diagramme de classe de Shell et Shell Command.

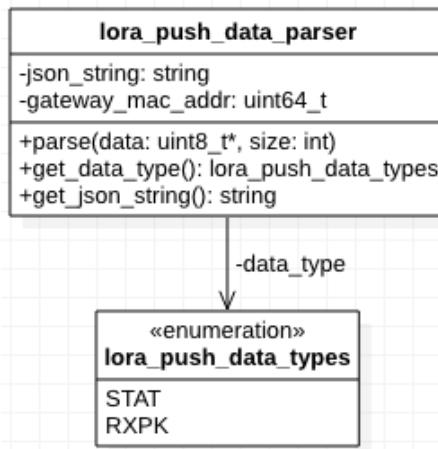


FIGURE 5.12 – Diagramme de classe de LoRa Push Data Parser

### Logger

Le logger est une simple classe qui permet l'affichage de message de log sur la sortie standard. Elle est principalement utilisée pour des questions de debug. Il est possible de loguer différents types de messages, information, warning, erreur et également de filtrer certains types de messages que l'on ne souhaite pas recevoir par exemple.

La figure 5.16 montre le diagramme de classe de Logger.

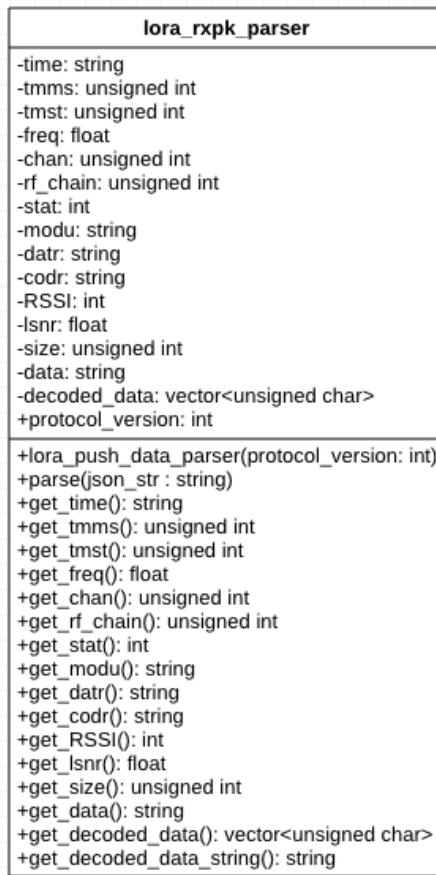


FIGURE 5.13 – Diagramme de classe de LoRa rxpk Parser

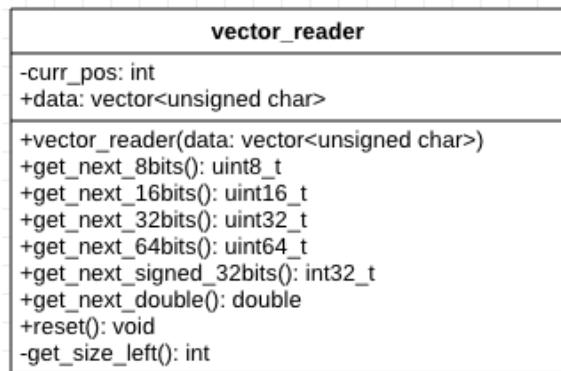


FIGURE 5.14 – Diagramme de classe de Vector Reader

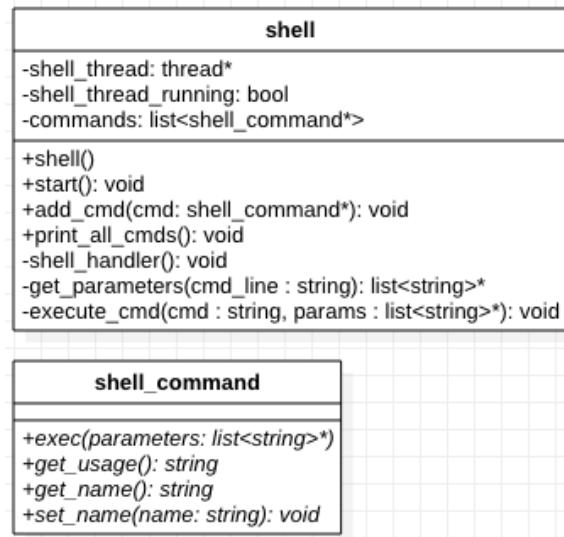


FIGURE 5.15 – Diagramme de classe de Shell et Shell Command

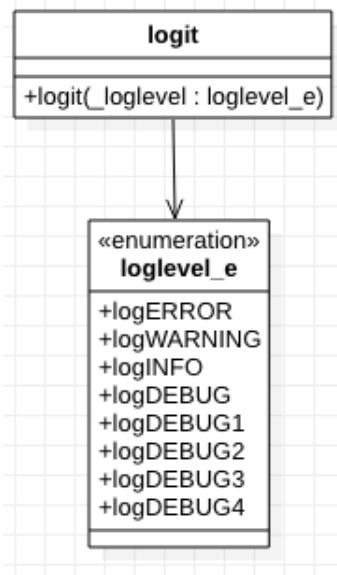


FIGURE 5.16 – Diagramme de classe de Logger

## 6 Description de la base de données

La base de données est le cœur du système, c'est elle qui permet la pérennisation des données. L'application mobile peut ensuite exploiter ces données afin de gérer la carte et afficher les diverses informations à l'utilisateur.

Elle est de type PostgreSQL et elle est hébergée directement sur la passerelle.

La base de données permet de stocker les informations suivantes :

- Information sur les compétiteurs
- Information sur les compétitions
- Inscriptions des compétiteurs à des compétitions
- Enregistrement de chaque paquet de données reçu des capteurs

Le diagramme relationnel de la base de données est disponible sur la figure 6.1.

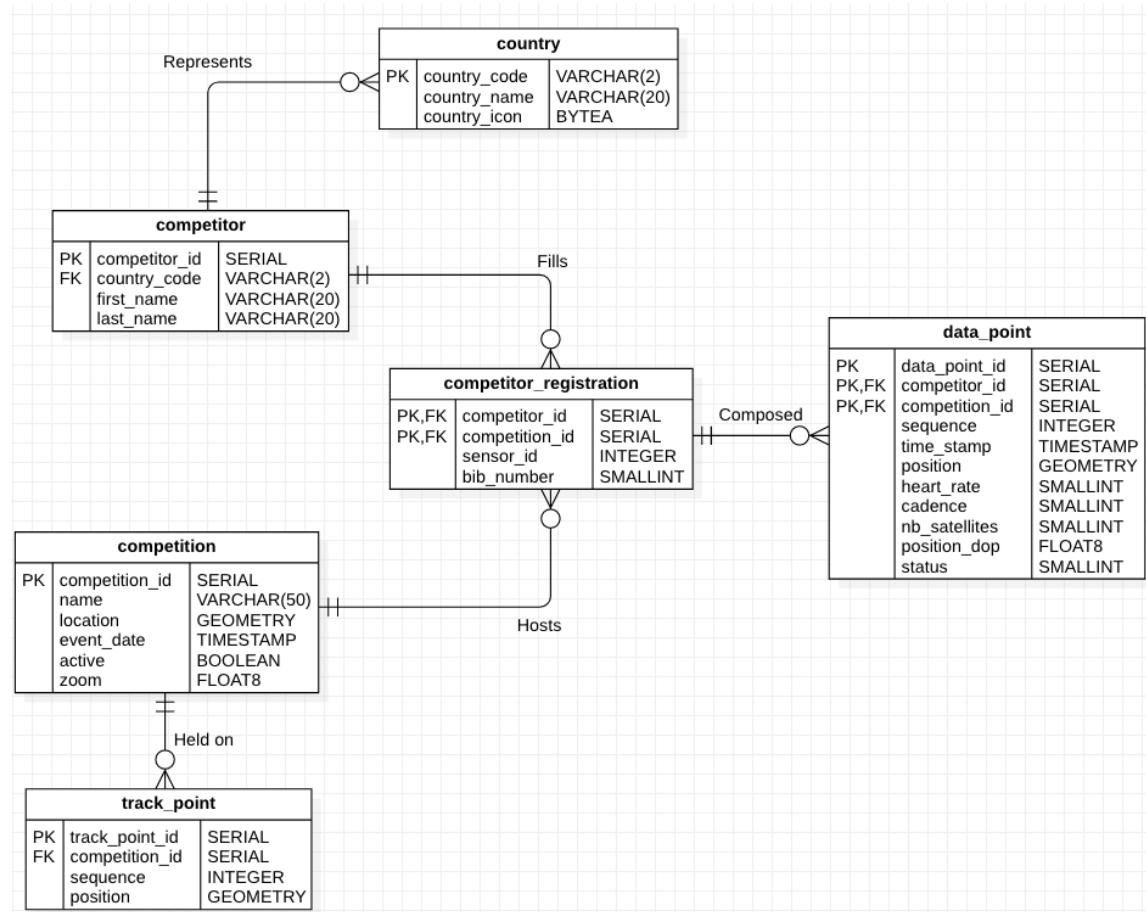


FIGURE 6.1 – Diagramme relationnel de la base de données

### 6.1 L'extension PostGIS

La base de données utilise une extension nommée PostGIS, qui rajoute une série de fonctionnalités qui permettent la gestion d'objet géographique dans la base, ce qui permet de faire des requêtes en utilisant des positions GPS par exemple.

La base de données utilise principalement le type ST\_POINT qui permet de définir des points en spécifiant la latitude et la longitude. Il existe deux types de points différents, geography et geometry. Les points de type geography permettent de prendre en compte la curvature de la planète lors du calcul de distance par exemple. Ce type de point est à préférer lorsque l'application doit faire des opérations entre des points qui sont très éloignés, deux continents différents par exemple. Le type geometry ne prend pas en compte cette information qui complexifie pas-sablement les calculs, mais tire simplement une ligne droite entre les deux points afin de faire le calcul de distance, ce qui convient parfaitement pour des points qui sont faiblement espacés comme c'est le cas pour se travail de Bachelor.

Un exemple de l'utilisation de point est présenté ci-dessous.

```
1 INSERT INTO a_table (gps) VALUES (ST_MakePoint(46.9933, 6.91612));  
2  
3 SELECT ST_X(gps) as latitude, ST_Y(gps) as longitude FROM a_table;
```

L'extension propose également des fonctions intéressantes comme la possibilité de déterminer la distance entre deux points géographiques.

```
1 SELECT ST_Distance(ST_MakePoint(46.9933, 6.91612), ST_MakePoint(46.781036, 6.647138));
```

## 6.2 Les tables

Grâce au modèle relationnel de la base de données ainsi que de l'extension PostGIS, il est ensuite facile d'insérer ou d'extraire des données dans les différentes tables au moyen de requêtes SQL.

Cette section décrit chaque table ainsi que les requêtes SQL qui y sont associées.

### 6.2.1 competition

Cette table permet de stocker toutes les informations relatives aux compétitions.

- competition\_id : L'identifiant de la compétition
- name : Le nom de la course
- location : L'emplacement de la course. Cette position géographique est utilisée par l'application mobile afin de centrer la vue de la carte correctement
- event\_date : La date et l'heure du déroulement de la course
- active : Définit si la course est en direct ou si l'événement est déjà passé
- zoom : Le zoom que l'application utilise lors du centrage de la vue sur la course

La requête SQL suivante définit la façon d'ajouter une nouvelle compétition à la base de données.

```
1 INSERT INTO race_tracker.competition (name, location, event_date) VALUES ('Nom de la  
course', ST_MakePoint(latitude, longitude), '2018-08-01 10:00:00-00');
```

Lorsque la passerelle reçoit un paquet de données, elle doit déterminer le competition\_id et le competitor\_id correspondant au coureur qui est défini par le numéro de capteur (sensor\_id). Pour ce faire, la requête suivante est exécutée. Elle permet de trouver le ou les competition\_id du capteur inscrit dans une course qui se déroule actuellement (active = True).

```
1 SELECT race_tracker.competitor_registration.competition_id, race_tracker.  
competitor_registration.competitor_id FROM race_tracker.competitor_registration INNER  
JOIN race_tracker.competitor ON (race_tracker.competitor_registration.competitor_id =  
race_tracker.competitor.competitor_id) INNER JOIN race_tracker.competition ON (  
race_tracker.competition.competition_id = race_tracker.competitor_registration.  
competition_id) WHERE race_tracker.competitor_registration.sensor_id = 1234 AND  
race_tracker.competition.active = True;
```

## 6.2.2 competitor

La table competitor contient tous les sportifs enregistrés dans le système. Une fois enregistrés, ils peuvent participer à des compétitions.

- competitor\_id : L'identifiant du compétiteur
- country\_code : Son pays d'origine
- first\_name : Son prénom
- last\_name : Son nom de famille

Un nouveau compétiteur est rajouté à la base de données en utilisant la requête suivante.

```
1 INSERT INTO race_tracker.competitor (first_name, last_name, country_code) VALUES ('Dilyana  
' , 'Petrova' , 'BG');
```

## 6.2.3 country

Elle contient tous les pays dont les compétiteur peuvent provenir ainsi que les icônes associés.

- country\_code : Le code du pays par exemple CH
- country\_name : Le nom du pays
- country\_icon : L'icône représentant le drapeau du pays

Un pays est rajouté grâce à la requête suivante.

```
1 INSERT INTO race_tracker.country VALUES ('BG' , 'Bulgaria' , bytea('country/bulgaria.svg'));
```

## 6.2.4 competitor\_registration

La table competitor\_registration permet d'effectuer l'inscription des sportifs à une compétition.

- competitor\_id : L'identifiant du compétiteur inscrit
- competition\_id : L'identifiant de la compétition auquel le compétiteur est inscrit
- sensor\_id : L'identifiant du capteur que le compétiteur porte pendant la course. Cet identifiant est envoyé dans tous les paquets envoyés par le capteur
- bib\_number : Le numéro de dossard du compétiteur

Une inscription s'effectue de la manière suivante.

```
1 INSERT INTO race_tracker.competitor_registration (competitor_id, competition_id, sensor_id  
, bib_number) VALUES (1, 1, 1234, 57);
```

L'application mobile a besoin de savoir tous les concurrents qui sont inscrits à une certaine course, la requête suivante permet de récupérer ces informations.

```
1 SELECT * FROM race_tracker.competitor_registration INNER JOIN race_tracker.competitor ON (
competitor_registration.competitor_id = competitor.competitor_id) WHERE
competitor_registration.competition_id = 1;
```

### 6.2.5 data\_point

Chaque data point correspond à un paquet émis par un capteur et contient toutes les informations. L'application mobile exploite les data points afin de pouvoir afficher les informations à l'utilisateur.

- data\_point\_id : Identifiant du data point
- competitor\_id : Identifiant du compétiteur dont le data point provient
- competition\_id : L'identifiant de la compétition à laquelle le data point est rattaché
- sequence : Un numéro de séquence qui est envoyé dans le paquet. Cette information permet de savoir si des paquets ont été perdus lors de la transmission
- time\_stamp : Le timestamp envoyé dans le paquet qui correspond au moment de la génération du paquet
- position : La position GPS du compétiteur
- heart\_rate : Le rythme cardiaque du compétiteur
- cadence : La cadence du compétiteur
- nb\_satellites : Le nombre de satellites en vue au moment de la génération du paquet
- position\_dop : Dissolution of precision, c'est à dire le degré de certitude de la précision de la position GPS
- status : L'état du capteur au moment de la génération du paquet

La requête suivante permet d'ajouter un nouveau data point dans la base de données. Cette opération est effectuée par la passerelle à chaque fois qu'un nouveau paquet est reçu.

```
1 INSERT INTO race_tracker.data_point (competitor_id, competition_id, sequence, position,
time_stamp) VALUES (1, 2, 1, ST_MakePoint(46.7856, 6.6424), '2018-10-15 00:00:00');
```

Durant son exécution, l'application mobile va exécuter périodiquement une requête, qui permet de connaître le numéro de séquence du dernier data point pour chaque concurrent inscrit à une certaine compétition. La requête suivante permet d'effectuer cette opération. Cette requête est en fait deux requêtes imbriquées, la première permet l'affichage de toutes les informations du data point du point contenant le numéro de séquence le plus élevé pour chaque concurrent (WHERE therank = 1), la deuxième va récupérer tous les data point et les trier par ordre décroissant.

```
1 SELECT data_point_id, competitor_id, competition_id, sequence, time_stamp, ST_X(position)
as lat, ST_Y(position) as lon, heart_rate, cadence, nb_satellites, position_dop, status
FROM (SELECT rank() OVER (PARTITION BY data_point.competitor_id ORDER BY sequence DESC)
AS therank, * FROM race_tracker.data_point WHERE data_point.competition_id = 1) t WHERE
therank = 1;
```

### 6.2.6 track\_point

Les track point correspondent à chaque point qui constitue le tracé de la course, l'application mobile va connecter chaque point par une ligne ce qui permettra de dessiner le tracé de la course sur la carte.

- track\_point\_id : Identifiant du track point
- competition\_id : L'identifiant de la compétition du track point
- sequence : Le numéro de séquence du track point
- position : La position géographique du point

Un nouveau track point peut être rajouté très simplement grâce à la requête suivante.

```
1 INSERT INTO race_tracker.track_point (competition_id, sequence, position) VALUES (3, 1,  
ST_MakePoint(46.7856, 6.6424));
```



## 7 Description de l'application mobile

L'application mobile, aussi appelé RaceTracker dans ce document, permet de visualiser, en temps réel, les informations produites par les capteurs porté par les sportifs. En plus de cela elle permet également d'administrer les compétitions.

Elle est codée en langage Java et utilise le système Android et également le "Maps SDK for Android" de Google qui permet d'interagir avec les cartes ce qui permet l'ajout de marqueurs ou de dessiner des formes géométriques par exemple.

L'environnement de développement Android Studio a été utilisé pour le développement de cette application qui permet également de simuler l'exécution de l'application et facilite ainsi le débogage.

TODO: Image GUI

### 7.1 Architecture logiciel

L'architecture logiciel de l'application mobile est présentée ci-dessous. Une application Android est composée de "Activities", qui représentent chacun des écrans disponibles dans l'application. Des "Fragments" sont également utilisés, ce sont des morceaux d'interface graphique qui peuvent être insérés dans des "Activities". Enfin l'application mobile utilise des classes qui sont responsable de la gestion de l'application et de l'accès à la base de données.

La figure 7.1 présente l'architecture statique de l'application mobile Android.

Durant l'exécution de l'application mobile, le thread principal est entièrement géré par le système d'exploitation Android, il est en charge de la mise à jour de l'interface graphique et de tout ce qui est en lien comme par exemple le changement d'activité lors de l'appui d'un bouton. Lorsque l'application désire envoyer des requêtes à la base de données, elle utilise la classe RaceTrackerDBAsyncTask qui hérite de AsyncTask. Cette classe permet d'effectuer une tâche en arrière-plan de manière asynchrone et de gérer le résultat une fois la tâche terminée. Enfin, lorsque l'utilisateur est en train de visionner une course, un objet de type Handler est utilisé pour faire la requête du dernier point de donnée à intervalle régulier.

La figure 7.2 montre l'architecture dynamique de l'application mobile.

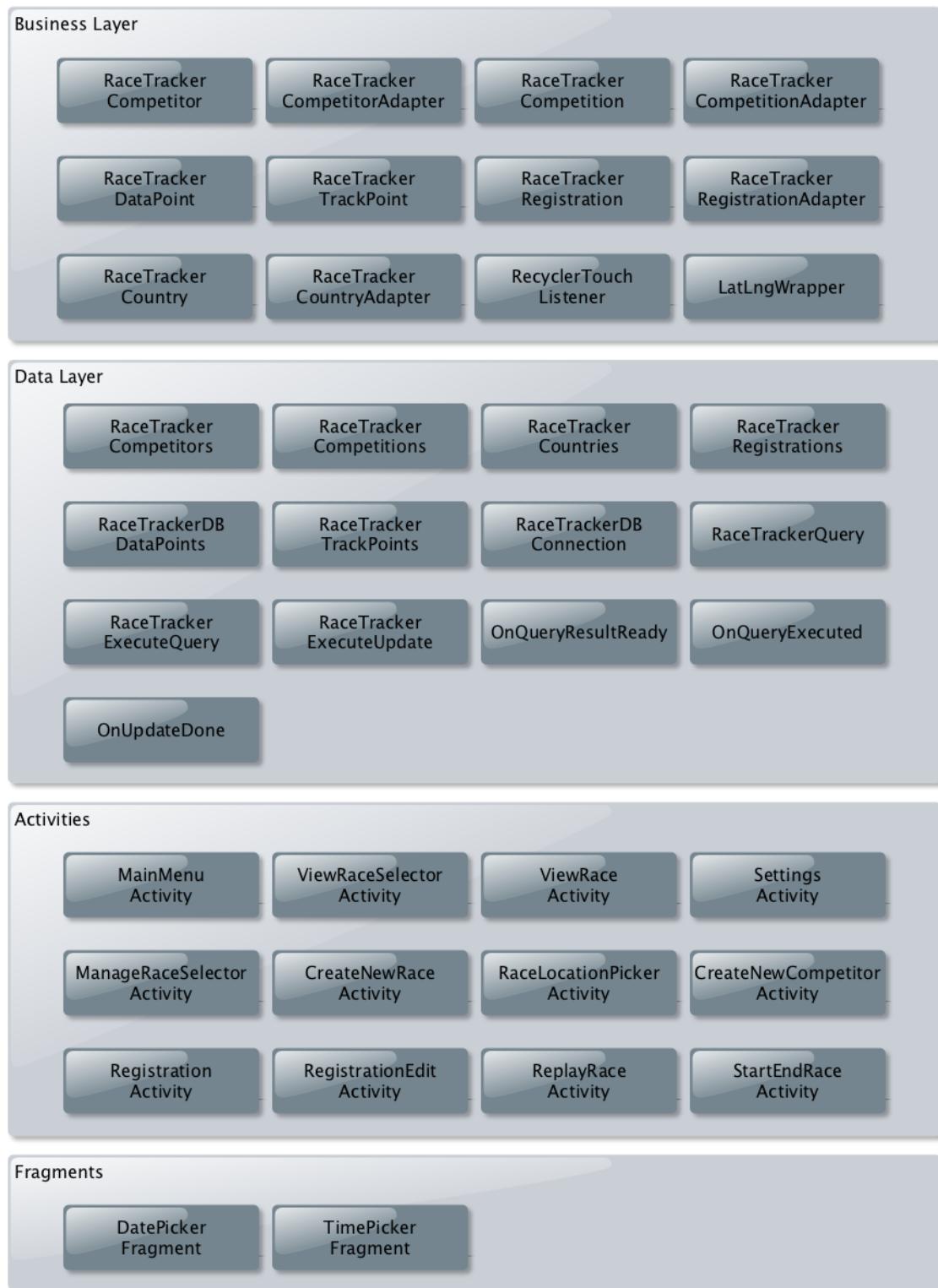


FIGURE 7.1 – Architecture statique de l'application mobile

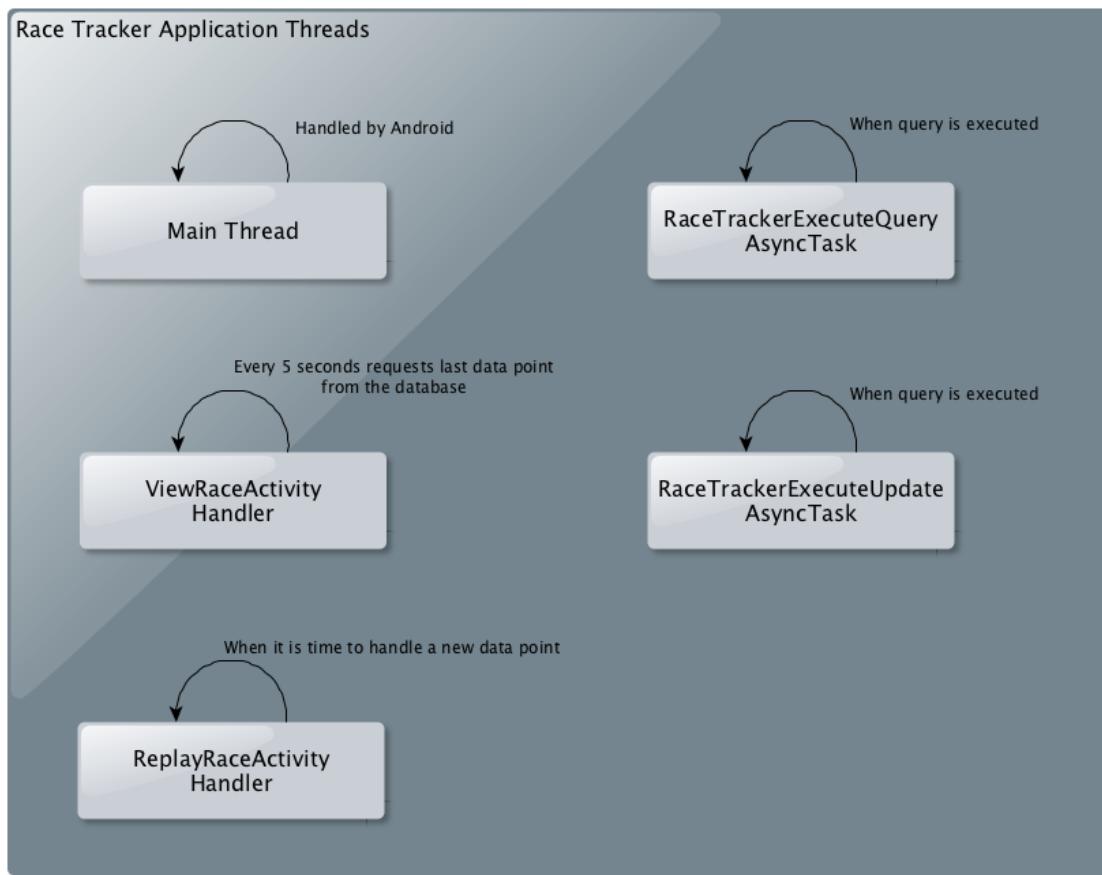


FIGURE 7.2 – Architecture dynamique de l'application mobile

## 7.2 Les librairies externes

L'application mobile utilise principalement deux librairies externes. La première est le "Maps for Android SDK". Cette librairie permet d'ajouter à son application mobile une carte de type google maps et d'interagir avec elle. Elle permet d'ajouter des marqueurs à des positions (latitude/longitude) spécifiques, de créer des lignes entre les points ou encore de modifier le comportement de la carte. La librairie est disponible sur <https://cloud.google.com/maps-platform/>.

Un exemple d'utilisation est proposé ci-dessous.

```
1 ...
2 private GoogleMap mMap;
3
4 protected void onCreate(Bundle savedInstanceState) {
5     /* Retrieve the map fragment when the map is ready */
6     MapFragment mapFragment = (MapFragment) getSupportFragmentManager().findFragmentById(R.id.
mapView);
7     mapFragment.getMapAsync(this);
8 }
9
10 @Override
11 public void onMapReady(GoogleMap googleMap) {
12     mMap = googleMap;
13
14     /* Change map type */
15     mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
16
17     /* Add a marker in Sydney, Australia */
18     LatLng sydney = new LatLng(-34, 151);
19     mMap.addMarker(new MarkerOptions().position(sydney).title("Marker in Sydney"));
20
21     /* Move camera and zoom */
22     mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(sydney, -12));
23 }
24 ...
```

La deuxième librairie externe utilisée est un driver de type JDBC permettant de s'interfacer avec une base de données PostgreSQL. C'est grâce à ce composant que l'application mobile va interroger la base de données afin de récupérer la liste des compétiteurs, les données relatives aux compétitions ou encore les points de données à afficher sur la carte. Le driver JDBC est téléchargeable gratuitement à l'adresse <https://jdbc.postgresql.org/>.

```
1 ...
2 public ResultSet executeQuery(String connection, String user, String password, String
query) {
3     Statement st;
4     ResultSet results
5
6     Connection conn = DriverManager.getConnection(connection, user, password);
7     st = conn.createStatement();
8
9     results = st.executeQuery(query);
10    st.close();
11
12    return results;
13 }
```

```
14
15 public void myQuery() {
16     String myQuery = "SELECT * FROM my_table;" ;
17     ResultSet results ;
18
19     results = executeQuery("jdbc:postgresql://192.168.1.4:5432/mydatabase", "me", "1234",
20     myQuery) ;
21
22     while (results .getNext() .next()) {
23         System.out.println("Field test1: " + results .getString("field_test_1"));
24         System.out.println("Field test2: " + results .getInt("field_test_2"));
25     }
26
27     results .close();
28 }
```

### 7.3 Accès à la base de données

Les accès à la base de données sont fait de manière similaire pour toute la requête en elle-même. La seule différenciation est entre les requêtes de type SELECT et les mises à jour des données (UPDATE, DELETE ou INSERT). Le comportement est un peu différent dans la mesure où une requête de type SELECT attend en retour un résultat qui comporte les données demandées à la base. Dans le cas d'une mise à jour, seul le nombre d'élément dans la base est retourné.

Afin d'exécuter une requête l'on créera une instance d'une classe RaceTrackerQuery qui contient la requête en elle-même ainsi que les fonctions de callback qui seront appelés au fil de l'évolution de l'exécution de la requête. La classe RaceTrackerQuery utilise les classes RaceTrackerExecuteQuery et RaceTrackerExecuteUpdate en fonction du type de requête. Ces deux objets utilisent le concept de AsyncTask afin d'exécuter la requête en arrière plan et ainsi ne pas bloquer le thread principal pendant trop de temps.

Afin de pouvoir récupérer les résultats d'une requête de type SELECT, les classes devront implémenter les interfaces OnQueryResultsReady et OnQueryExecuted. La première est appelée lorsque les résultats venant de la base de données sont prêts à être analysés. La deuxième interface est appelée lorsque l'exécution de la requête est terminée.

Si l'on désire exécuter une requête de mise à jour alors il faudra implémenter OnUpdateDone qui permettra, au terme de l'exécution de la requête, combien d'éléments ont été modifiés.

### 7.4 Les activités

Dans le monde Android, une activité est une chose singulière que l'utilisateur peut effectuer. La majorité des activités interagissent avec l'utilisateur à travers de l'interface graphique afin d'effectuer une certaine tâche. [?] Dans le cas de l'application développée dans le cadre du travail de diplôme les activités utilisées sont décrites ci-dessous.

### 7.4.1 MainMenuActivity

Le point d'entrée de l'application Android est le premier écran qui se lance, et qui permet de choisir si l'on souhaite visionner une course ou alors faire des opérations d'administration sur la base de données. Lorsque l'utilisateur a fait son choix, elle lance l'activité, suivante c'est à dire soit ViewRaceSelectorActivity ou ManageRaceSelectorActivity.

La figure 7.3a montre le diagramme de classe de MainMenuActivity.

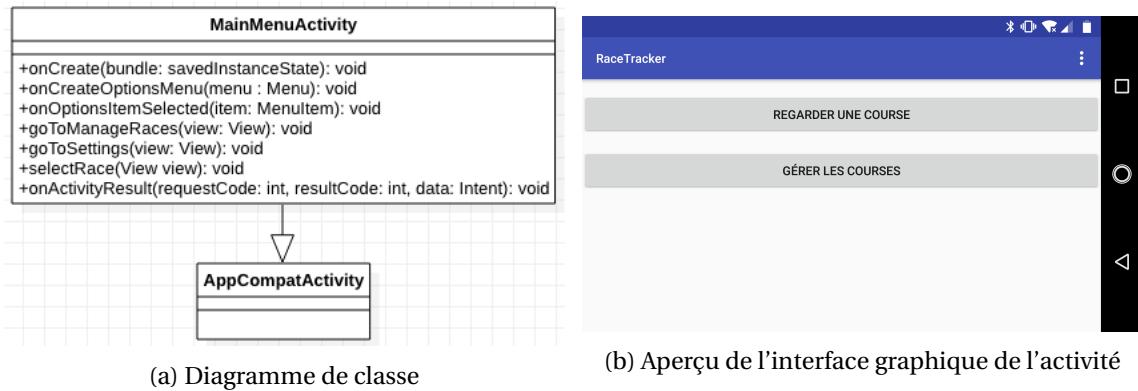


FIGURE 7.3 – MainMenuActivity

### 7.4.2 ViewRaceSelectorActivity

Cette activité permet à l'utilisateur de sélectionner une course. Lorsqu'elle se lance, elle va aller interroger la base de données afin de récupérer la liste de toutes les compétitions et l'afficher à l'utilisateur qui peut ensuite sélectionner une course en cliquant sur l'objet de son choix. Une fois la course choisie, un objet de type RaceTrackerCompetition contenant les informations relatives à la course est renvoyé au créateur de cette activité.

La figure 7.4a montre le diagramme de classe de ViewRaceSelectorActivity.

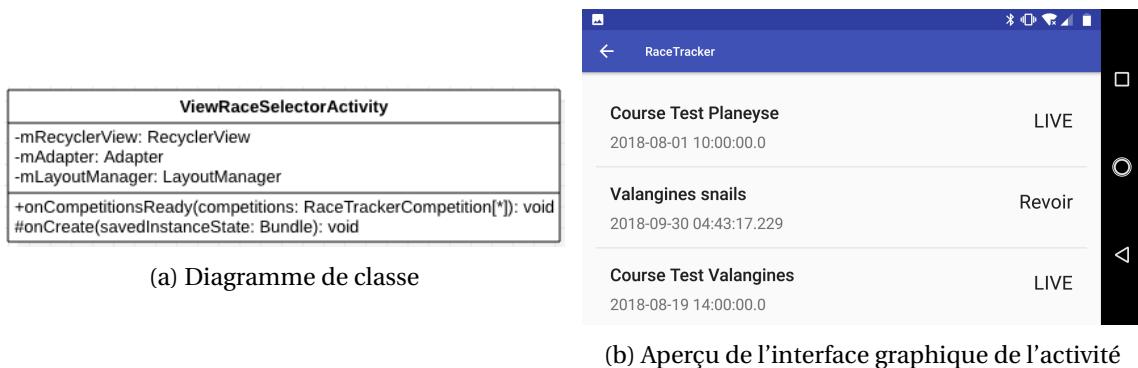


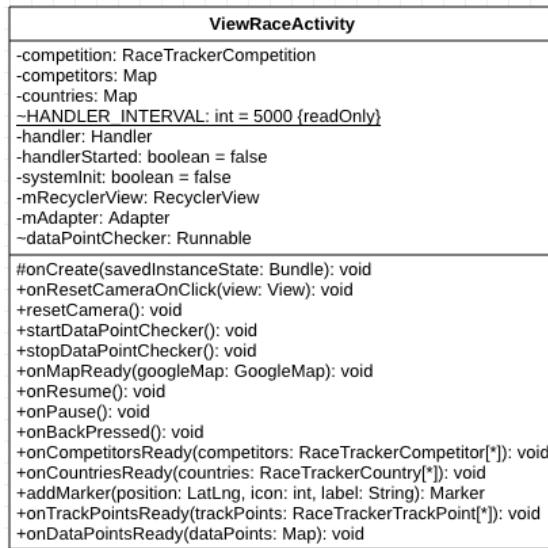
FIGURE 7.4 – ViewRaceSelectorActivity

### 7.4.3 ViewRaceActivity

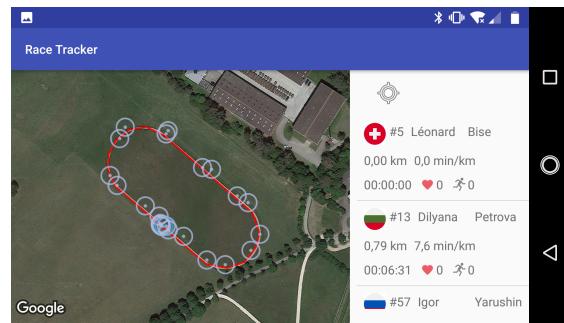
Lorsque l'utilisateur désire regarder une course, c'est cette activité qui va être lancée. Elle est responsable de la gestion de l'affichage des informations sur la carte et également de la liste des compétiteurs. Périodiquement cette activité va aller chercher dans la base de données les

points de données relatifs à une course et vérifier si de nouveaux points ont été ajoutés, auquel cas elle met à jour la carte avec la nouvelle position. Toutes ces informations sont récupérées depuis la base de données.

La figure 7.5a montre le diagramme de classe de ViewRaceActivity.



(a) Diagramme de classe



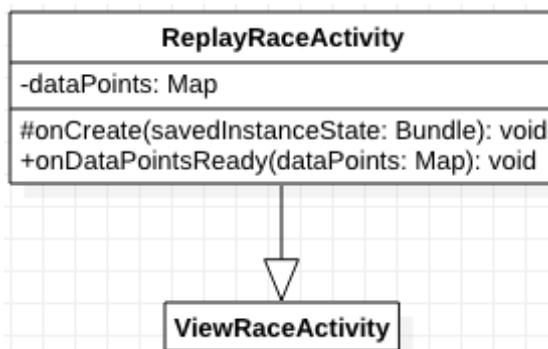
(b) Aperçu de l'interface graphique de l'activité

FIGURE 7.5 – ViewRaceActivity

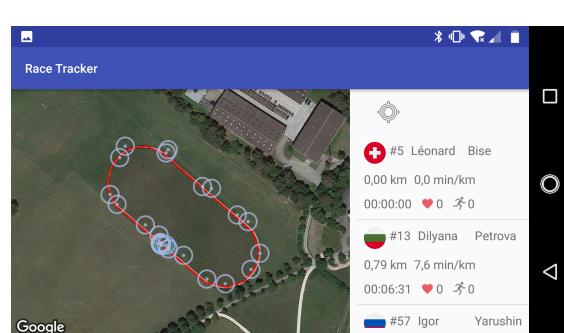
#### 7.4.4 ReplayRaceActivity

L'activité ReplayRaceActivity est très semblable et donc hérite de ViewRaceActivity. Son but est d'afficher une course déjà terminée et donc de reconstituer la course comme lorsqu'elle s'est passée. Elle va récupérer tous les points de données puis calculer le temps qui est passé entre chacun d'eux afin de pouvoir simuler l'évolution de la course comme en direct.

La figure 7.6a montre le diagramme de classe de ReplayRaceActivity.



(a) Diagramme de classe



(b) Aperçu de l'interface graphique de l'activité

FIGURE 7.6 – ReplayRaceActivity

**TODO: Maj GUI image**

### 7.4.5 ManageRaceSelectorActivity

La classe ManageRaceSelectorActivity permet à l'utilisateur de sélectionner l'action d'administration qu'il désire effectuer. Une fois la sélection faite, l'activité correspondante est lancée.

La figure 7.7a montre le diagramme de classe de ManageRaceSelectorActivity.

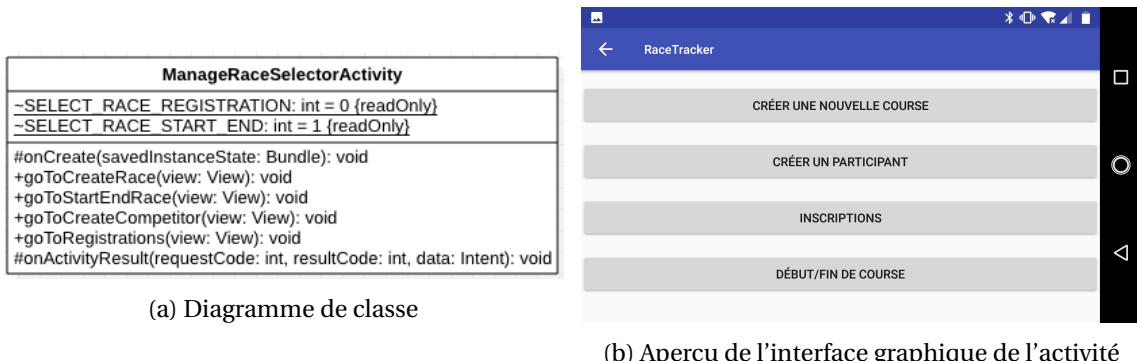


FIGURE 7.7 – ViewRaceActivity

### 7.4.6 CreateNewRaceActivity

L'activité CreateNewRaceActivity permet à l'utilisateur de créer une nouvelle compétition dans la base de données. Une fois que l'utilisateur a rentré toutes les informations, une requête d'insertion est envoyée à la base de données.

La figure 7.8a montre le diagramme de classe de CreateNewRaceActivity.

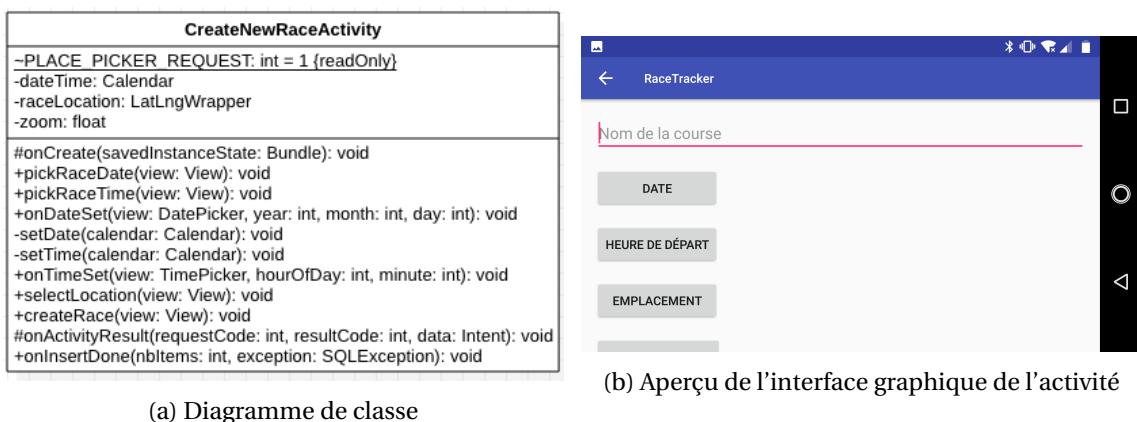
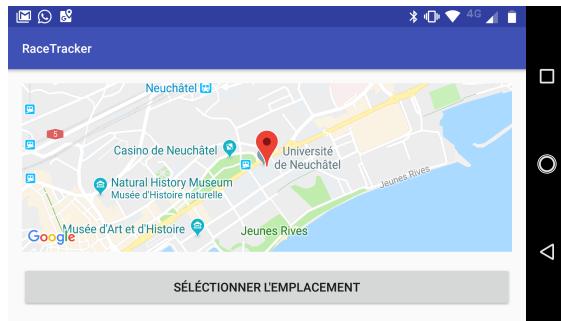
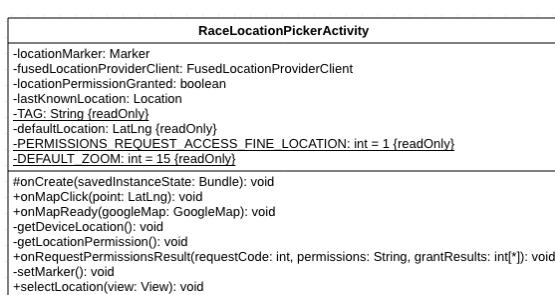


FIGURE 7.8 – CreateNewRaceActivity

### 7.4.7 RaceLocationPickerActivity

Permet la sélection de la position où se tient la compétition lors de la création de nouvelles courses. Cette information est utilisée pour pouvoir centrer la vue sur le bon emplacement.

La figure 7.9a montre le diagramme de classe de RaceLocationPickerActivity.



(a) Diagramme de classe

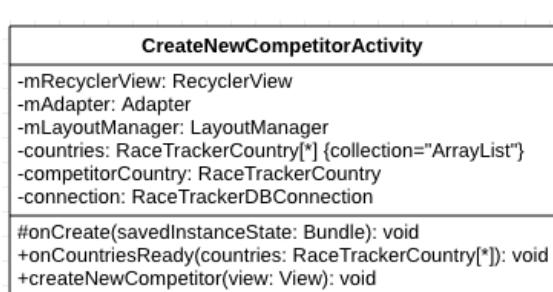
(b) Aperçu de l'interface graphique de l'activité

FIGURE 7.9 – RaceLocationPickerActivity

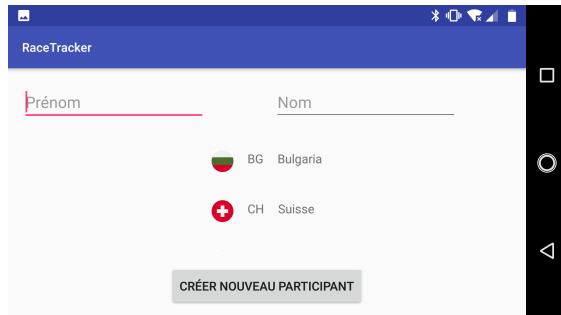
#### 7.4.8 CreateNewCompetitorActivity

Cette activité permet la création d'un nouveau compétiteur et de son ajout dans la base de données. Une fois rajouté, il pourra ensuite être inscrit à des compétitions.

La figure 7.10a montre le diagramme de classe de CreateNewCompetitorActivity.



(a) Diagramme de classe



(b) Aperçu de l'interface graphique de l'activité

FIGURE 7.10 – CreateNewCompetitorActivity

#### 7.4.9 RegistrationActivity

Liste les compétiteur déjà inscrits à la compétition sélectionnée. En plus de cela il est possible d'ajouter un nouveau participant, ce qui sera géré par l'activité RegistrationEditActivity.

La figure 7.11a montre le diagramme de classe de RegistrationActivity.

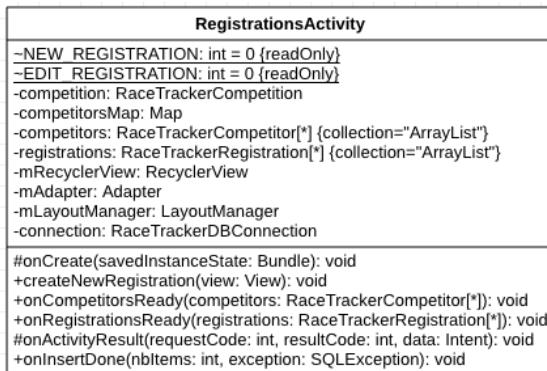
#### 7.4.10 RegistrationEditActivity

Permet l'inscription des concurrents à une certaine course. Un numéro de dossard ainsi que le numéro du capteur qu'ils vont porter durant la course sera également enregistré.

La figure 7.12a montre le diagramme de classe de RegistrationEditActivity.

#### 7.4.11 StartEndRaceActivity

Cette activité donne la possibilité d'altérer l'état d'une course. Il existe deux états possibles, active ou inactive. Une course active est entrain de se dérouler en direct alors qu'une course

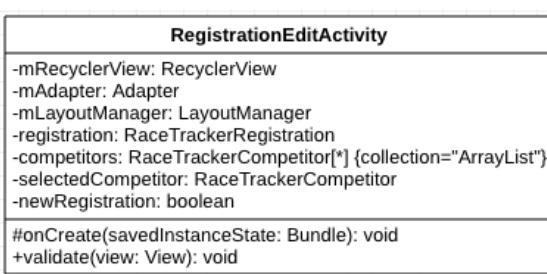


(a) Diagramme de classe

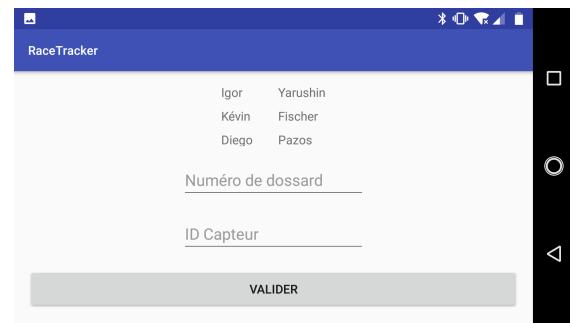


(b) Aperçu de l'interface graphique de l'activité

FIGURE 7.11 – RegistrationActivity



(a) Diagramme de classe

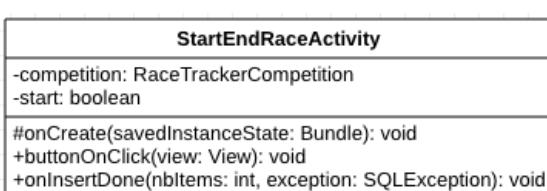


(b) Aperçu de l'interface graphique de l'activité

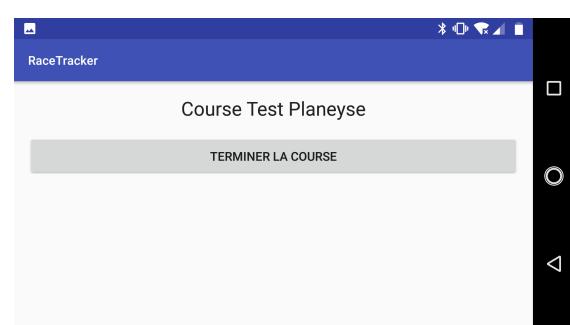
FIGURE 7.12 – RegistrationEditActivity

inactive est terminée. Dans le cas d'une course en direct, l'activité ViewRaceActivity va alors rechercher les nouveaux points ajoutés à la course et les mettre à jour en temps réel. Pour une course inactive, c'est à dire terminée, c'est une autre activité qui servira à sa visualisation, ReplayRaceActivity. Elle simulera l'affichage des points comme lorsque la course s'est réellement passée, ce qui permet de revoir l'évolution de la compétition.

La figure 7.13a montre le diagramme de classe de StartEndRaceActivity.



(a) Diagramme de classe



(b) Aperçu de l'interface graphique de l'activité

FIGURE 7.13 – StartEndRaceActivity

### 7.4.12 SettingsActivity

Cette classe est en charge de la gestion de l'affichage du menu des paramètres. L'utilisateur peut modifier les paramètres relatifs à l'application grâce au menu proposé par cette activité.

La figure 7.14a montre le diagramme de classe de SettingsActivity.

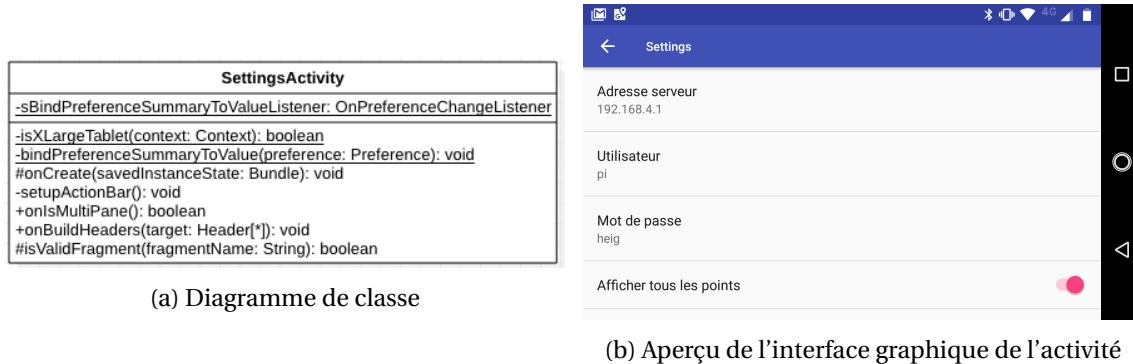


FIGURE 7.14 – SettingsActivity

## 7.5 Les fragments

Les fragments représentent le comportement d'une partie de l'interface graphique. Il est possible d'avoir plusieurs fragments sur une même activité.

### 7.5.1 DatePickerFragment

Le fragment DatePickerFragment est utilisé pour récupérer une date entrée par l'utilisateur, l'interface habituelle pour entrer une date est présentée à l'utilisateur et son choix est retourné.

La figure ?? montre le diagramme de classe de DatePickerFragment.

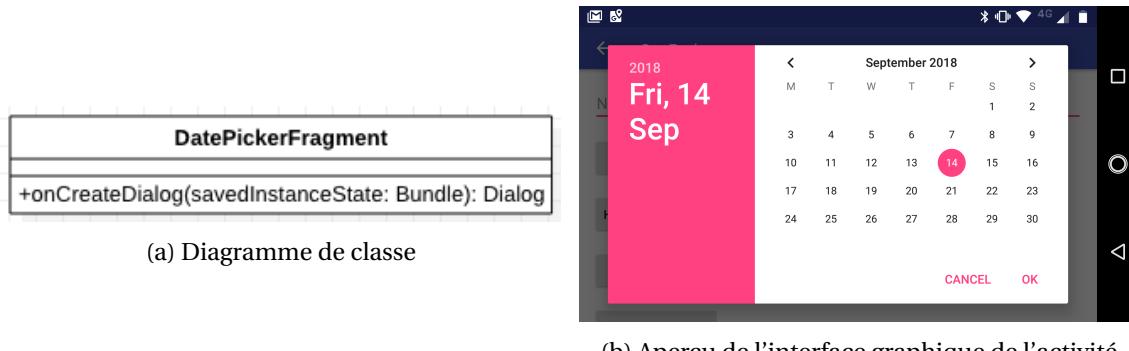


FIGURE 7.15 – DatePickerFragment

### 7.5.2 TimePickerFragment

Le fragment TimePickerFragment permet la sélection d'une heure, il présente à l'utilisateur l'interface usuelle de sélection de l'heure puis en retourne le choix.

La figure 7.16a montre le diagramme de classe de TimePickerFragment.

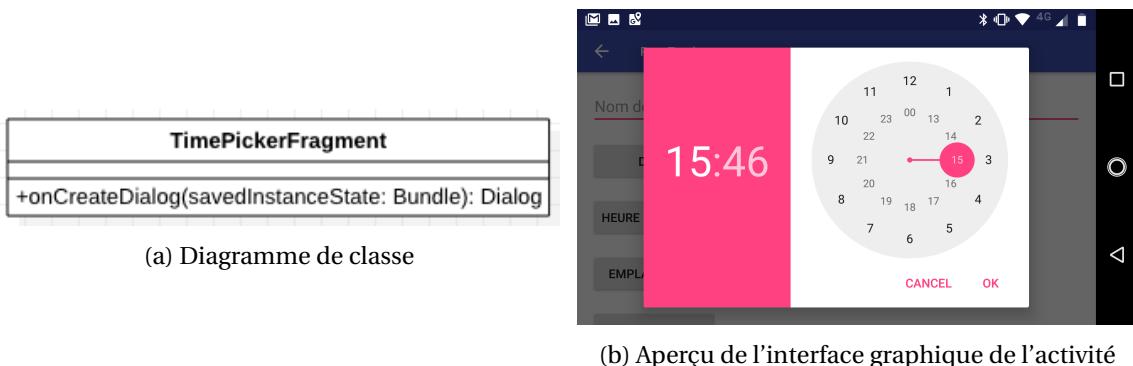


FIGURE 7.16 – TimePickerFragment

## 7.6 Les classes

L'application mobile RaceTracker est composée de plusieurs classes qui lui permettent d'effectuer ses différentes tâches. Elles sont décrites dans cette section.

### 7.6.1 RaceTrackerCompetition

Cette classe contient les données relatives à une compétition. Elle peut être initialisée directement en lui passant le résultat d'une requête à une base de données (ResultSet).

La figure 7.17a montre le diagramme de classe de RaceTrackerCompetition.

### 7.6.2 RaceTrackerCompetitionAdapter

Cette classe permet de faire la gestion de l'affichage d'une liste d'instance de RaceTrackerCompetition dans un objet de type RecyclerView.

La figure 7.17b montre le diagramme de classe de RaceTrackerCompetitionAdapter.

### 7.6.3 RaceTrackerCompetitions

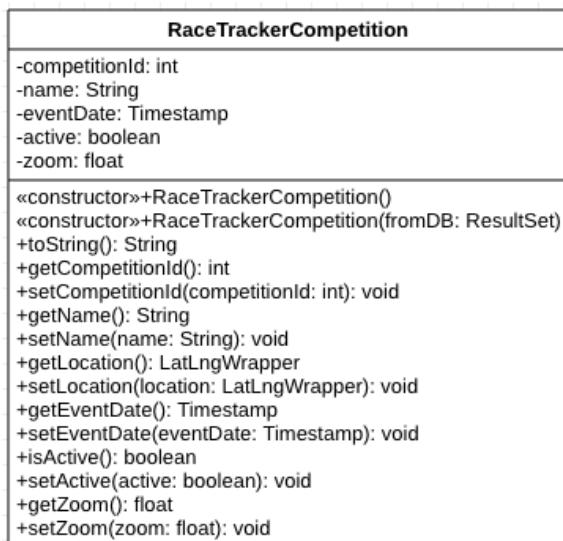
Cette classe permet de faire le lien avec la base de données. Elle permet de récupérer la liste de toutes les compétitions mais également l'ajout de nouvelles courses ou la modification de courses existantes.

La figure 7.17c montre le diagramme de classe de RaceTrackerCompetitions.

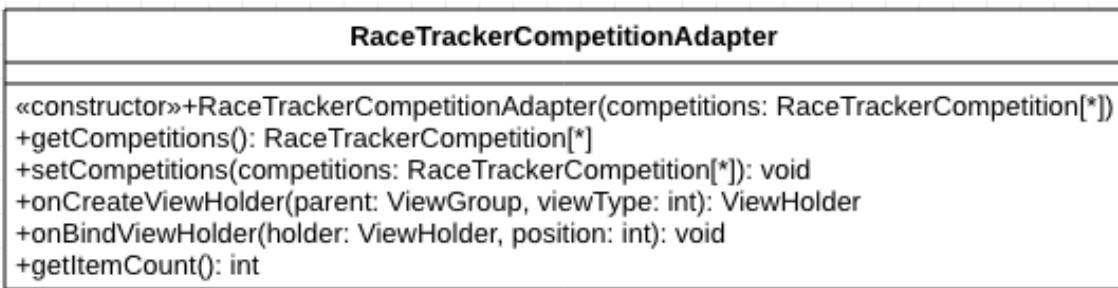
### 7.6.4 RaceTrackerCompetitor

Cette classe contient les données relatives à un compétiteur. Elle peut être initialisée directement en lui passant le résultat d'une requête à une base de données (ResultSet).

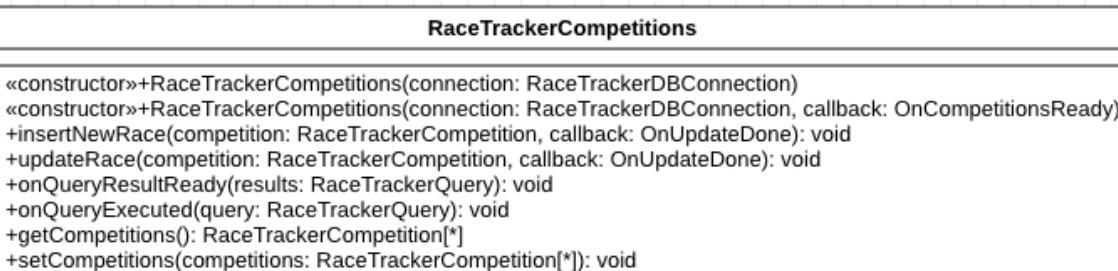
La figure 7.18a montre le diagramme de classe de RaceTrackerCompetitor.



(a) Diagramme de classe



(b) Diagramme de classe



(c) Diagramme de classe

FIGURE 7.17 – RaceTrackerCompetition

### 7.6.5 RaceTrackerCompetitorAdapter

Cette classe permet de faire la gestion de l'affichage d'une liste d'instance de RaceTrackerCompetitor dans un objet de type RecyclerView.

La figure 7.18b montre le diagramme de classe de RaceTrackerCompetitorAdapter.

### 7.6.6 RaceTrackerCompetitors

Permet de faire la liaison avec la base de données. Elle permet la récupération de tous les compétiteurs existants dans la base ainsi que l'ajout de nouveaux compétiteurs.

La figure 7.19 montre le diagramme de classe de RaceTrackerCompetitors.

### 7.6.7 RaceTrackerCountry

Représente un pays dont un compétiteur peut être originaire. Cette classe s'occupe également de la gestion de l'icône du drapeau du pays qui est directement récupéré depuis la base.

La figure 7.20a montre le diagramme de classe de RaceTrackerCountry.

### 7.6.8 RaceTrackerCountryAdapter

Cette classe permet de faire la gestion de l'affichage d'une liste d'instance de RaceTrackerCountry dans un objet de type RecyclerView.

La figure 7.20c montre le diagramme de classe de RaceTrackerCountryAdapter.

### 7.6.9 RaceTrackerCountries

Permet la gestion de la liste de tous les pays qu'elle récupère directement depuis la base de données.

La figure 7.20b montre le diagramme de classe de RaceTrackerCountries.

### 7.6.10 RaceTrackerRegistration

La classe RaceTrackerRegistration contient les informations relatives à une inscription à une course spécifique.

La figure 7.21a montre le diagramme de classe de RaceTrackerRegistration.

### 7.6.11 RaceTrackerRegistrationAdapter

Cette classe permet de faire la gestion de l'affichage d'une liste d'instance de RaceTrackerRegistration dans un objet de type RecyclerView.

La figure 7.21b montre le diagramme de classe de RaceTrackerRegistrationAdapter.

### 7.6.12 RaceTrackerRegistrations

Permet la récupération d'une liste de RaceTrackerRegistration pour une certaine course depuis la base de données. C'est également cette classe qui permet l'insertion de nouvelles inscriptions.

La figure 7.21c montre le diagramme de classe de RaceTrackerRegistrations.

### 7.6.13 RaceTrackerDataPoint

La classe RaceTrackerDataPoint contient les données relatives à un point de données sur le parcours. C'est ce type de classe que le ViewRaceActivity utilise afin d'afficher à l'utilisateur la position et les informations associées aux coureurs. Elle peut être initialisée directement en lui passant le résultat d'une requête à une base de données (ResultSet). Chaque data point corres-

pond à un paquet de donnée envoyé par le capteur et contient toutes les données qui étaient parties de la charge utile du paquet.

La figure 7.22a montre le diagramme de classe de RaceTrackerDataPoint.

#### 7.6.14 RaceTrackerDataPoints

Permet de récupérer une liste de RaceTrackerDataPoint initialisée directement depuis les données reçues depuis la base de données.

La figure 7.22b montre le diagramme de classe de RaceTrackerDataPoints.

#### 7.6.15 RaceTrackerTrackPoint

Les track points représentent toutes les positions qui sont utilisées pour pouvoir dessiner le tracé du parcours de la course. Ils sont stockés dans la base de données et il est possible de les lire grâce à la classe RaceTrackerTrackPoints.

La figure 7.23a montre le diagramme de classe de RaceTrackerTrackPoint.

#### 7.6.16 RaceTrackerTrackPoints

Cette classe permet la récupération de tous les points du parcours de la course depuis la base de données. Ils sont ensuite passés à la carte Google map afin d'y dessiner le parcours.

La figure 7.23b montre le diagramme de classe de RaceTrackerTrackPoints.

#### 7.6.17 RaceTrackerDBConnection

La classe qui contient les informations relatives à la connexion à la base de données, c'est à dire adresse, user name et mots de passe. Elle permet ensuite de construire la chaîne de caractères utilisée par le driver JDBC.

La figure 7.19 montre le diagramme de classe de RaceTrackerCompetitors.

#### 7.6.18 RaceTrackerQuery

Contient une requête pour la base de donnée, son résultat, la fonction de callback de l'utilisateur et éventuellement l'exception si la requête n'a pas pu être exécutée correctement. Cette classe permet d'exécuter deux types de requêtes, les requêtes dites de mise à jour (DELETE, UPDATE ou INSERT) ou de sélection (SELECT).

Deux types de fonction callback sont utilisées pour informer l'utilisateur de l'évolution de la requête. OnQueryResultReady est déclenché lorsque les résultats d'une sélection sont prêts à être analysés. OnQueryExecuted est déclenché au terme de l'opération de sélection ou de mise à jour.

La figure 7.25a montre le diagramme de classe de RaceTrackerQuery.

### 7.6.19 RaceTrackerExecuteQuery

Cette classe, qui hérite de AsyncTask, permet l'exécution d'une requête RaceTrackerQuery de type mise à jour de manière asynchrone dans le but de ne pas bloquer le thread principal d'affichage d'Android.

La figure 7.25b montre le diagramme de classe de RaceTrackerExecuteQuery.

### 7.6.20 RaceTrackerExecuteUpdate

Cette classe, qui hérite de AsyncTask permet l'exécution d'une requête RaceTrackerQuery de type sélection de manière asynchrone dans le but de ne pas bloquer le thread principal d'affichage d'Android.

La figure 7.25b montre le diagramme de classe de RaceTrackerExecuteUpdate.

### 7.6.21 LatLngWraper

Cette classe permet d'encapsuler un objet de type LatLng. Cette classe est utilisée pour rendre un objet LatLng serialisable.

La figure 7.26 montre le diagramme de classe de LatLngWraper.

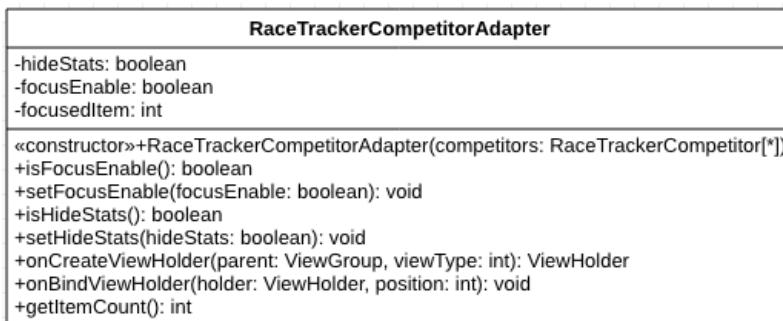
### 7.6.22 RecyclerTouchListener

Cette classe permet de faciliter la gestion des événements OnClick des RecyclerView.

La figure 7.27 montre le diagramme de classe de RecyclerTouchListener.



(a) Diagramme de classe



(b) Diagramme de classe

FIGURE 7.18 – RaceTrackerCompetitor

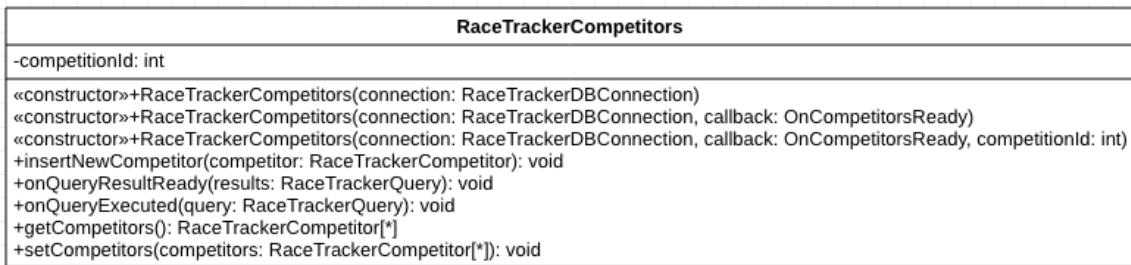
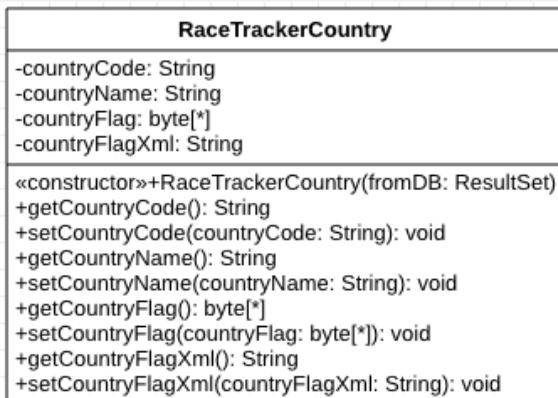
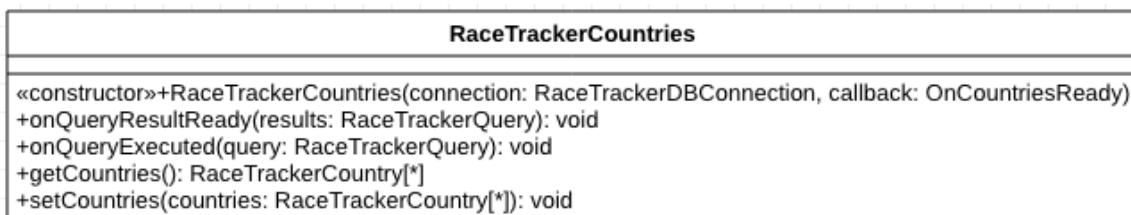


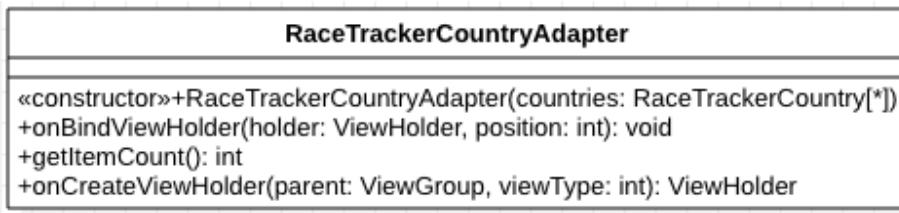
FIGURE 7.19 – Diagramme de classe de RaceTrackerCompetitors



(a) Diagramme de classe

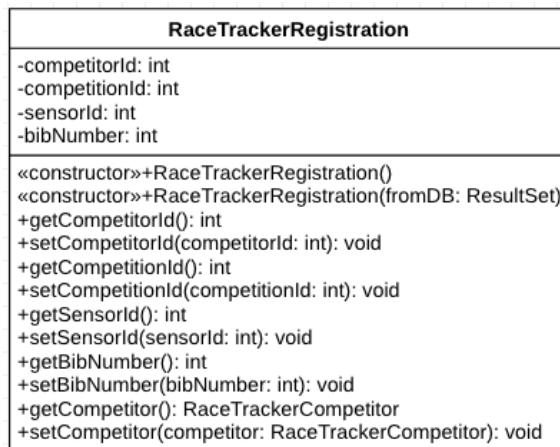


(b) Diagramme de classe

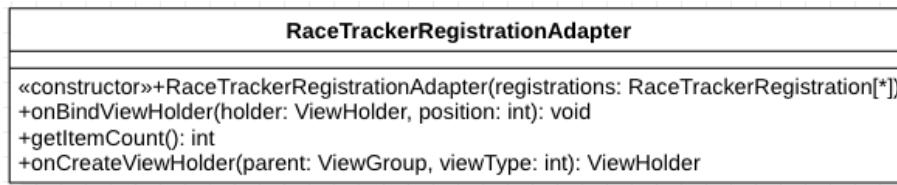


(c) Diagramme de classe

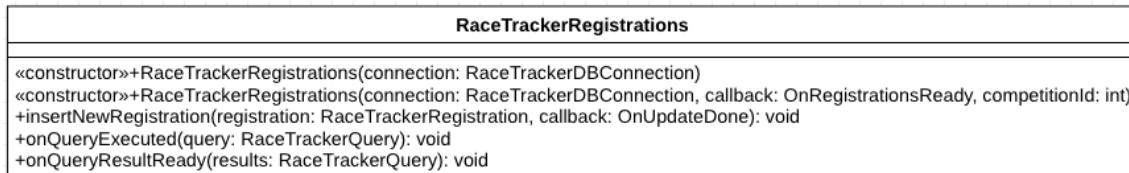
FIGURE 7.20 – RaceTrackerCountry



(a) Diagramme de classe

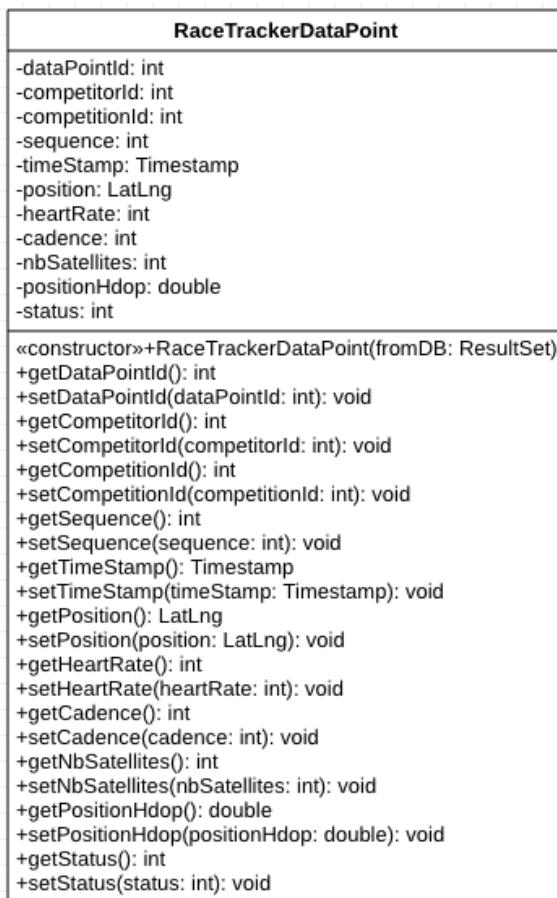


(b) Diagramme de classe

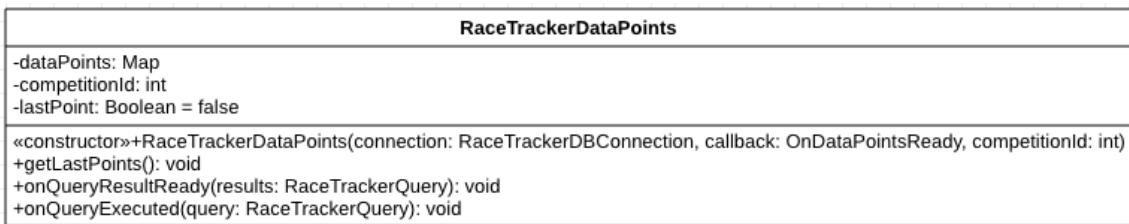


(c) Diagramme de classe

FIGURE 7.21 – RaceTrackerRegistration

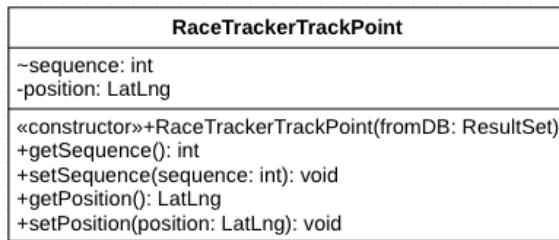


(a) Diagramme de classe

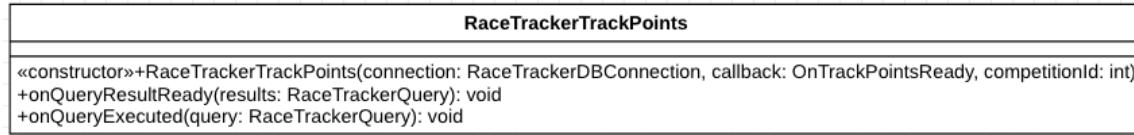


(b) Diagramme de classe

FIGURE 7.22 – RaceTrackerDataPoint



(a) Diagramme de classe



(b) Diagramme de classe

FIGURE 7.23 – RaceTrackerTrackPoint

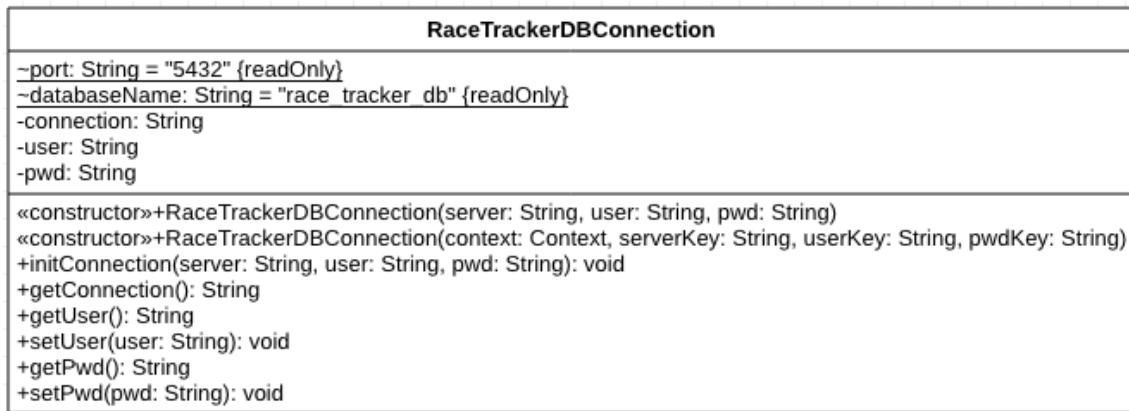
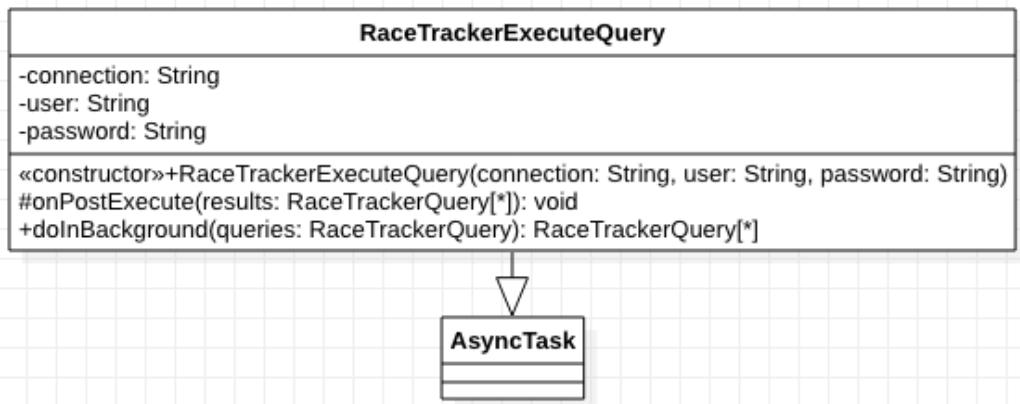


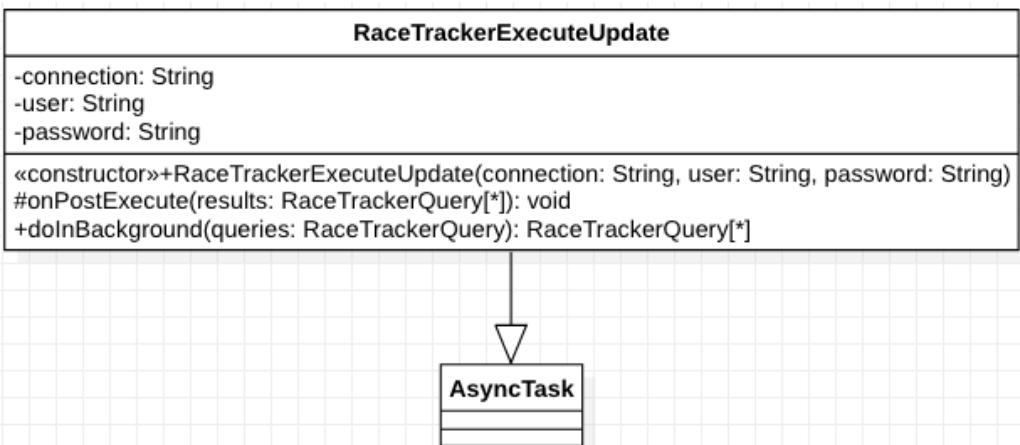
FIGURE 7.24 – Diagramme de classe de RaceTrackerDBConnection



(a) Diagramme de classe



(b) Diagramme de classe



(c) Diagramme de classe

FIGURE 7.25 – RaceTrackerQuery

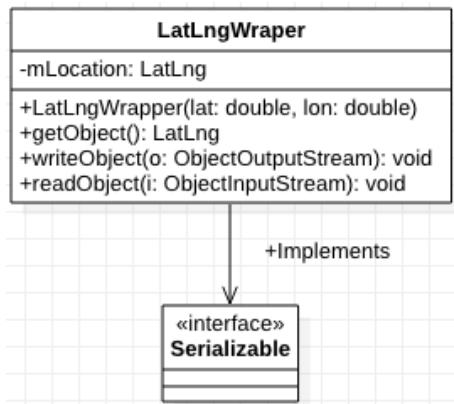


FIGURE 7.26 – Diagramme de classe de **LatLngWraper**

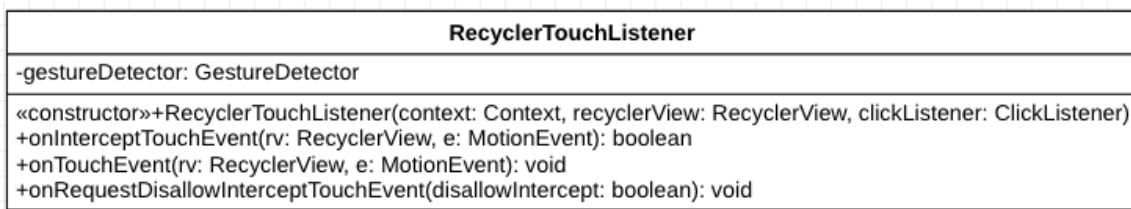


FIGURE 7.27 – Diagramme de classe de **RecyclerTouchListener**

## 7.7 Les interfaces

### 7.7.1 OnQueryResultReady

OnQueryResultReady est une interface que l'utilisateur doit implémenter afin de pouvoir recevoir les résultats des requêtes envoyées à la base de données. La fonction est automatiquement appelée lorsque les résultats de la requête associée sont prêts à être consultés. Cette interface est appelée depuis le contexte d'exécution de AsyncTask, c'est à dire en arrière-plan (et non sur le thread d'affichage de l'UI, ce qui ne permet donc pas d'interagir avec les éléments graphiques lors de son appel).

La figure 7.28 montre le diagramme de classe de OnQueryResultReady.

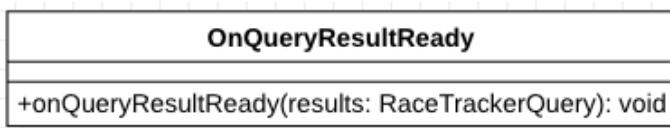


FIGURE 7.28 – Diagramme de classe de OnQueryResultReady

### 7.7.2 OnQueryExecuted

Cette interface permet à l'utilisateur de savoir lorsque l'exécution d'une requête est terminée. L'interface OnQueryExecuted est appelée depuis le contexte d'exécution du thread d'affichage de l'UI et donc permet à l'utilisateur de modifier des éléments d'affichage ou de remplir des listes de type RecyclerView.

La figure 7.29 montre le diagramme de classe de OnQueryExecuted.

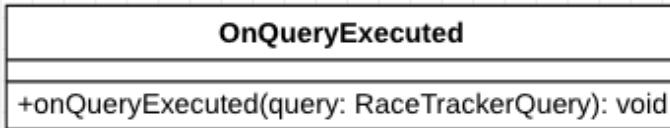


FIGURE 7.29 – Diagramme de classe de OnQueryExecuted

### 7.7.3 OnUpdateDone

OnUpdateDone est appelé au terme de l'exécution d'une requête de type mise à jour. Elle permet de connaître le nombre d'éléments qui ont été mis à jour ainsi que l'éventuelle exception qui serait survenue durant son exécution.

La figure 7.30 montre le diagramme de classe de OnUpdateDone.

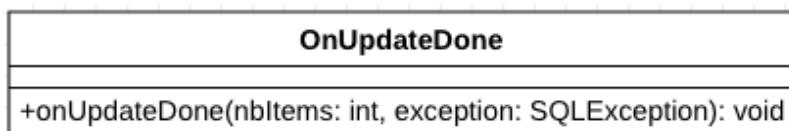


FIGURE 7.30 – Diagramme de classe de OnUpdateDone

## 8 Problèmes et solutions

Ce chapitre décrit les problèmes majeurs rencontrés pendant le développement du projet et les solutions qui ont été employées afin d'y pallier.

### 8.1 Capteur : Driver $I^2C$ manquant

Au début du développement du projet, j'ai décidé d'utiliser le système d'exploitation temps réel Zephyr. En consultant la page de la carte utilisant le même micro-contrôleur que le capteur je n'ai malheureusement pas remarqué qu'il n'y avait aucunes mentions d'un driver  $I^2C$  existant. Je suis parti du principe qu'il en existait un. Lors du développement, afin de communiquer avec les modules GPS et accéléromètre, j'ai alors constaté ce manque à mes dépend.

La solution a été simplement de m'armer de la data sheet du micro-contrôleur et d'en écrire un moi-même. Ce driver est décrit en détail dans le chapitre 4.4.

### 8.2 Capteur : Driver GPIO incomplet

Lors de l'implémentation du module rythme cardiaque, il m'a fallu brancher une interruption sur un I/O afin de pouvoir détecter les battement du cœur. Après avoir écrit tout le code je me suis rendu compte à l'exécution que le programme me retournais une erreur du type "not supported". En résumé, le driver GPIO SAM ne proposait pas cette fonctionnalité.

Afin de pallier à ce problème j'ai implémenté cette fonctionnalité manquante. Pour se faire j'ai du écrire un driver External Interrupt Controller qui permet de gérer le module du micro-contrôleur permettant d'ajouter des interruptions sur des lignes d'entrées sorties. Une fois ce driver développé il a fallu modifier le driver GPIO afin de utiliser ce nouveau driver et ainsi permettre l'ajout d'interruption sur I/O.

### 8.3 Passerelle : Problème de connexion réseau

Pendant le développement du projet, un des domaines où j'ai rencontré beaucoup de problème est la configuration réseau de la passerelle. J'ai voulu utiliser l'interface WiFi du Raspberry Pi en conjonction avec un dongle WiFi afin d'avoir deux interfaces qui me permette sur une de créer une access point afin de pouvoir me connecter avec mon téléphone mobile et sur l'autre de ce connecter à un réseau WiFi. Je n'ai jamais réussi à faire fonctionner ce modèle comme je le voulais, probablement en partie du à la mise à jour de Raspbian qui à rajouté pas mal de changements à ce niveau là.

Après plusieurs dizaines d'essais de configuration différentes j'ai décidé de ne plus utiliser le dongle et de le remplacer par un cable Ethernet, ce qui me permet de facilement me connecter à la passerelle depuis mon réseau personnel.

## 8.4 Capteur : Interruption accéléromètre

Afin de pouvoir détecter les pas du porteur du capteur, j'avais pris la décision d'utiliser un système proposé par l'accéléromètre placé sur la carte qui permet de le configurer afin qu'il déclenche une interruption lorsqu'une condition est détectée. Dans le cas du projet je voulais pouvoir déclencher une interruption lorsque l'accélération sur l'un des trois axes dépasse un certain seuil. Le driver LSM303AGR a été modifié afin de pouvoir permettre la configuration de ce type d'interruption, problème aucunes interruptions n'était déclenchées lorsque le capteur était mis en mouvement.

Après beaucoup de temps perdu à essayer de débusquer le bug et à cause du fait que le temps commençait à me manquer, j'ai pris la décision de changer d'approche et d'utiliser un thread afin d'effectuer l'échantillonnage des données de l'accéléromètre.

## 9 Test phase #1

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 la 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.

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 9.1.

0xFFEDEAD	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 9.1 – 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 suivie 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 ajouté 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 transmettent 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 émis 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 pa-

quet 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.

Le code utilisé durant ce test correspond au tag de la version v0.1 sur git.

## 9.1 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.

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

### 9.2.1 Test sur piste

TABLE 9.1 – 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 9.2a et 9.2b 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.

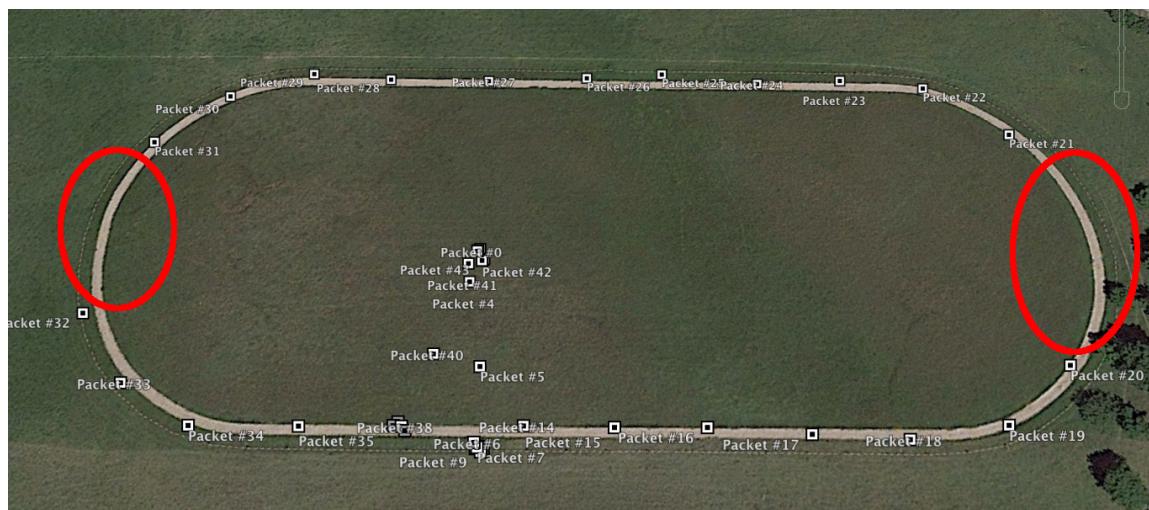
### 9.2.2 Test de distance

TABLE 9.2 – Résultats des tests phase 1 - Distance

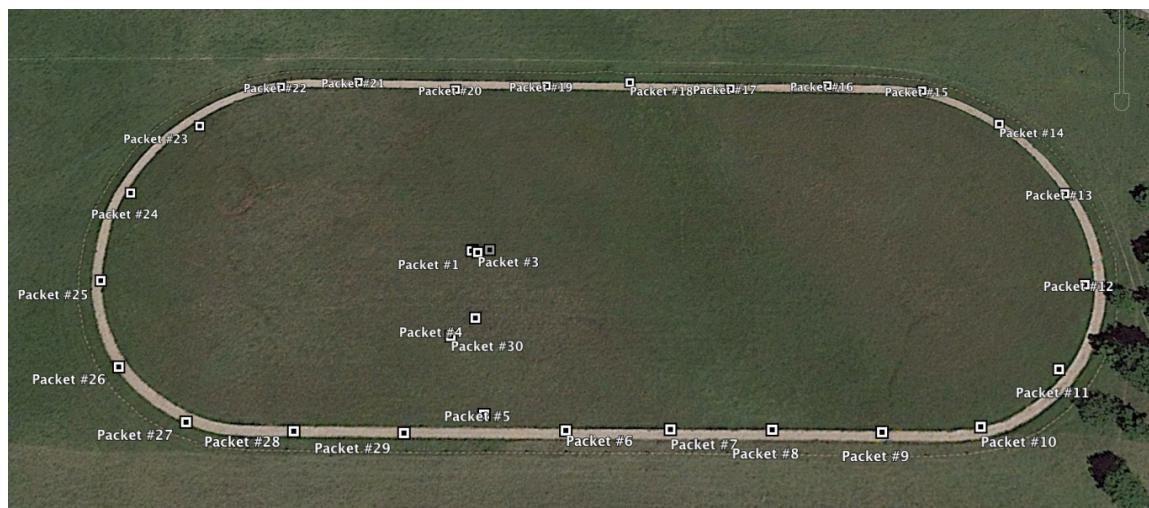
Tests Distance				Planeyse 13.07.2018		
Nom	Configuration	HDOP Moy	Nb Sat Moy	Nb reçu	Nb perdu	% perdu
Test #3	SF7 - PWR -0.6 dBm	1.37	7.88	32	10	23.8%
Test #4	SF12 - PWR -0.6 dBm	0.93	9.32	37	1	2.6%

Les figures 9.3a et 9.3b 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.

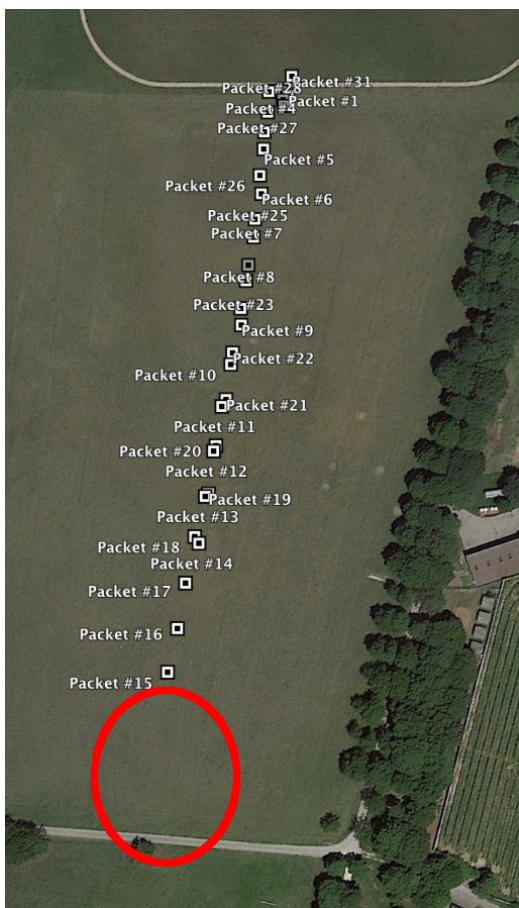


(a) Test #1

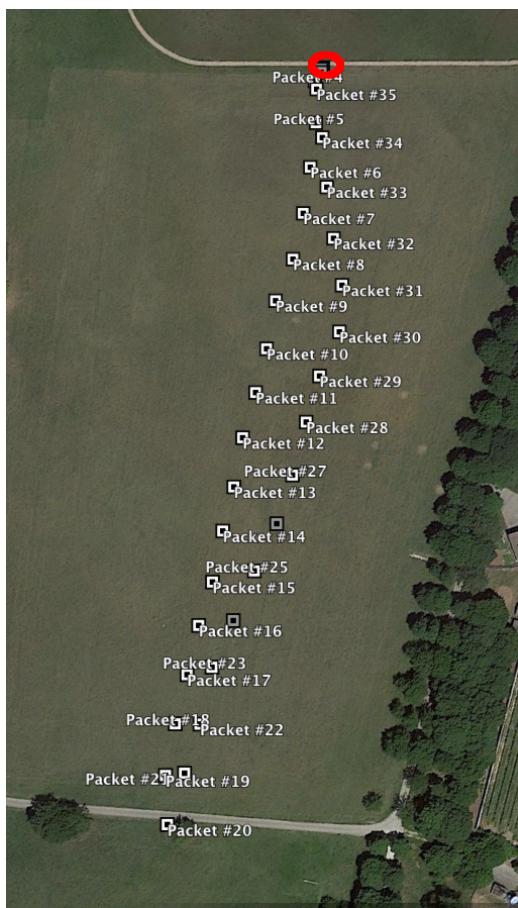


(b) Test #2

FIGURE 9.2 – 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 9.3 – Position GPS de chaque paquet reçu durant le test de distance - Images capturés grâce au logiciel © Google Earth

## 9.3 Conclusions

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.

## 10 Test phase #2

Pour clôturer la phase de développement #2 du projet, un test est effectué avec pour objectif de valider que le concept dans son ensemble fonctionne. Ce test a permis de mettre ensemble tout les acteurs du système, le capteur, la passerelle, la base de données et enfin l'application mobile et vérifier leur bon fonctionnement.

Le test s'est concentré sur la transmission de la position GPS depuis le capteur jusqu'à l'application mobile, les autres éléments du système n'étant pas encore implémentés et de vérifier que la mise à jour de la position du coureur se passe bien.

Afin de pouvoir faire ce test les éléments suivant ont été développés.

- Firmware du capteur avec l'utilisation de Zephyr et envoie de la position GPS dans un paquet
- Création de la base de données
- Implémentation de la réception, décodage et stockage des paquets sur la passerelle
- Création de la base de l'application mobile avec visualisation des positions GPS sur la carte

On comprend que cette phase de développement est un gros morceau du développement complet du projet car tous les composants du système sont développés. Dans un premier temps seul les fonctionnalités de bases sont implémentées, ce qui permet de s'assurer que toutes les interactions se passent comme prévues avant de poursuivre le travail pour la phase #3.

Pour le capteur, la base du firmware comprenant le système d'exploitation temps réel Zephyr est mis en place. Pour l'instant il ne fait que de récupérer périodiquement la position GPS grâce au driver  $I^2C$  implémenté pour l'occasion et l'envoyer dans un paquet LoRa à destination de la passerelle.

Le système PostgreSQL est installé sur la passerelle puis la base de données est conçue et créée ce qui permet le stockage des données venant du capteur. Le logiciel serveur d'application est d'avantage développé afin de pouvoir récupérer les paquets, qui sont d'un format différent que celui du test de la phase #1, puis grâce à la librairie pqxx, les stocker dans la base.

Le cœur de l'application mobile est écrit, permettant de périodiquement envoyer une requête à la passerelle pour récupérer d'éventuelles nouvelles données et les afficher sur la carte Google maps afin de pouvoir visualiser l'évolution du coureur.

Puisque l'objectif du test est de se mettre dans des conditions réelles, les résultats du test sont directement sauvegardés dans la base de données, permettant ainsi de pouvoir consulter toutes les positions facilement depuis l'application mobile. Le téléphone mobile sur lequel s'exécute l'application est connecté à la passerelle grâce à l'access point qu'il propose et fait les requêtes directement à la passerelle.

Le code utilisé durant ce test correspond au tag de la version v0.2 sur git.

TODO: pictars of test, running goo?

## 10.1 Scénarios

Le scénarios de ce test est très simple, il consiste à recréer les conditions réel d'utilisation du système. Pour se faire un cobaye coureur muni du capteur va courir le long d'un parcours définit afin de vérifier que la position est correctement mise à jour au fil de son évolution.

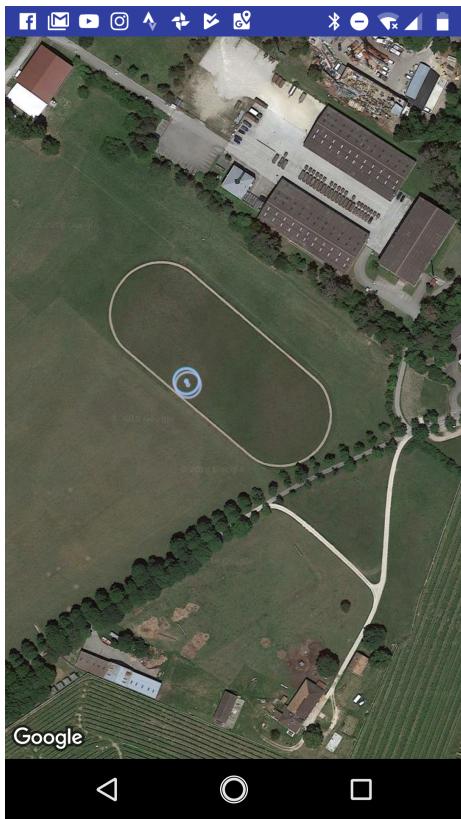
## 10.2 Résultats

Après plusieurs échec dus aux problèmes de connections entre l'application mobile et la passerelle, le test a finalement pu se dérouler sans embûche.

Le test a été effectué le 7 Septembre 2018 sur la piste finlandaise de la place d'arme de Planeyse à Neuchâtel.

Le capteur et la passerelle sont configuré pour utiliser un facteur d'étalement maximal de sf12 afin de s'assurer d'avoir la portée maximum.

Les positions récoltées durant le tests peuvent être consultées sur la figure 10.1.



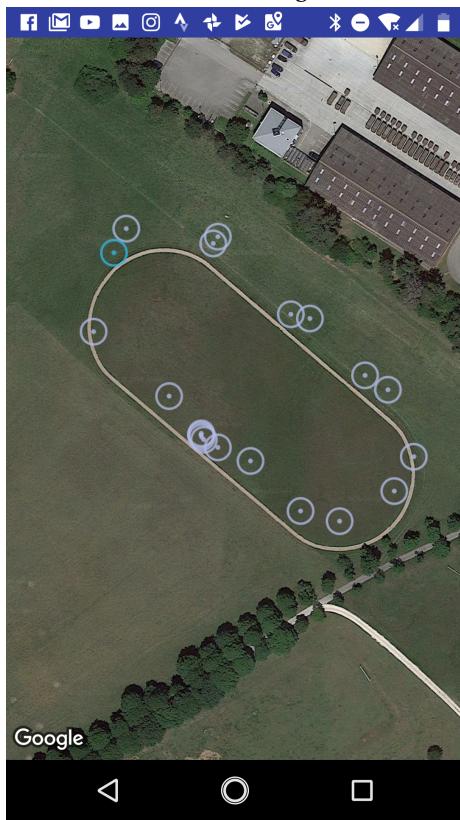
(a) Test #2 - Image #1



(b) Test #2 - Image #2



(c) Test #2 - Image #3



(d) Test #2 - Image #4

FIGURE 10.1 – Visualisation des positions reçues du capteur depuis l'application mobile

### 10.3 Conclusions

On remarque que les positions reportées sur la carte ne semblent pas être alignées avec la piste. Après des investigations ceci est la cause d'un bug au niveau de l'affichage des marqueurs sur la carte, en effet la position est appliquée à partir du milieu inférieur de l'image ce qui fait qu'il n'est pas totalement aligné. Après correction du bug les positions sont correctement affichées.

Le test phase #2 a permis de s'assurer que les positions GPS acquises et transmises dans des paquets LoRa avec le firmware Zephyr permet d'avoir des précisions très similaires à celle constatées durant le test phase #1 qui utilisait un logiciel écrit avec le framework Arduino. De manière générale aucunes grosses surprises n'ont été révélées par ce test, les performances sont très satisfaisantes également lorsque le porteur du capteur cours ce qui est très encourageant pour la suite du développement du projet.

On peut constater que les données reçues par la passerelle dans les paquets LoRa, leur extractions et stockage fonctionne conformément aux attentes. La façon d'accéder à la base de données depuis l'application mobile est également validée, à savoir la connexion à l'access point proposé par la passerelle puis envoie des requêtes à la base. Aucuns problèmes n'ont été mis en lumière de ce côté là.

Grâce aux résultats du test on peut conclure que le concept du système de base est parfaitement fonctionnelle et qu'aucuns problèmes majeurs n'ont été révélés durant l'exécution du test ce qui permet donc de valider le développement de la phase #2 et de passer à la phase suivante qui consiste en la finalisation du système dans son ensemble avec l'implémentation de toutes les fonctionnalités restantes.

## 11 Test phase #3

Le travail effectué pendant le projet est validé dans son ensemble par le test de la phase #3. Ce test permet de vérifier que tout les éléments développés durant l'entier de la période de réalisation du projet sont à la hauteur de ce qui est attendu et remplisse leur tâche correctement. Il permet également de jauger si la problématique au cœur du projet est en partie solutionné par le système proposé.

Les éléments développés pour ce test sont les suivants.

- Ajout de l'acquisition du rythme cardiaque et de la cadence au capteur
- Gestion et stockage des nouveaux paramètres ajoutés au capteur par la passerelle
- Finalisation de l'application mobile

Le capteur est modifié afin de pouvoir compter les battements du cœur ainsi que l'intervalle de temps afin de permettre le calcul du nombre de battement par minute. De même pour la cadence, l'accéléromètre est interrogé périodiquement pour récupérer les valeurs d'accélération et tenter de détecter les pas. Ces données sont ensuite transmises à la passerelle à l'intérieur des paquets LoRa. Une boîte permettant d'insérer le capteur ainsi que tout les autres éléments, l'accumulateur, le module rythme cardiaque, l'antenne GPS et LoRa, est construite puis imprimée grâce à une imprimante 3D. Une sangle élastique permet de fixer le capteur au bras du cobaye.

La configuration de la passerelle est modifiée pour automatiquement lancer l'exécution du packet forwarder et du serveur d'application lors de son lancement, ce qui permet de ne pas devoir effectuer d'autres opérations que de la mettre sous tension pour qu'elle soit opérationnelle. Le traitement des nouvelles données reçues dans les paquets est implémenté dans le code du serveur d'application de la passerelle. Une fois les données extraites le tout est sauvegardé dans la base de données.

Pour ce test le gros du travail est centré autour de l'application mobile. Toutes les fonctions de gestion de course sont implémenté, comme la création de nouvelles courses ainsi que l'inscription des coureurs. Les modes de visualisation de course est complété avec l'affichage de toute les nouvelles informations à disposition, le rythme cardiaque et la cadence, mais également les informations qui sont calculées à partir des données reçues, la distance parcourue, la vitesse moyenne et le temps de course. L'interface graphique est finalisée afin de pouvoir montrer toutes les informations au spectateur et une phase de affinage est effectué pour rendre le tout cohérent et agréable à l'utilisation.

Le code utilisé durant ce test correspond au tag de la version v1.0 sur git.

### 11.1 Scénarios

Une nouvelle fois le système est mis à l'utilisation dans un cadre permettant un test en conditions réelles. Avant le démarrage du test, la course est créée directement depuis l'application mobile, pour les test des phases précédente les courses étaient créées manuellement dans la base. L'inscription de la coureuse est également fait dans l'interface de l'application, enfin la course est démarrée.

Au terme de la création de la course, deux tests sont effectués.

Le premier test consiste à mettre le capteur sous tension grâce à l'accumulateur puis d'attendre la synchronisation GPS. Lorsque le module GPS est prêt à acquérir des positions avec assez de précision la coureuse s'élance alors sur la piste afin de faire quelques tours de la piste d'athlétisme permettant ainsi de vérifier que tout fonctionne bien.

Le deuxième test consiste à voir comment le capteur se comporte lorsqu'il n'a plus la vue direct sur la passerelle mais qu'elle est obstruée par un obstacle comme un bâtiment par exemple.

Durant les deux tests les parcours ont également été enregistrés avec une montre GPS sur la plateforme Strava qui permet d'enregistrer ses courses. Cela permettra de comparer les résultats de chaque application et vérifier qu'aucunes grosses erreurs n'existent.

## 11.2 Résultats

Les deux tests ont pu être conduits à bien le 22 Septembre 2018 au stade d'athlétisme de Colombier.

### 11.2.1 Test de validation du système

Pendant ce test, la coureuse a fait 3 fois le tour de l'anneau d'athlétisme, ce qui a permis de vérifier que les paramètres rapportés par l'application mobile étaient vraisemblables.

Afin de pouvoir comparer les résultats obtenues par le système, la course enregistrée sur la plateforme Strava peut être consultée à l'adresse <https://www.strava.com/activities/1858813564>.

Sur la figure 11.1 on peut voir que très rapidement des valeurs de rythme cardiaque et de cadence apparaissent sur l'interface de l'application mobile. En comparant les valeurs de rythme cardiaque avec celle de la plateforme Strava, elles semblent être conformes à la réalité. Par contre on peut remarquer que les valeurs de cadences sont totalement fausses, elles devraient être au alentours de 160-170 pas par minute au lieu de 11-48.

En ce qui concerne les positions GPS, comme durant le test phase #2 on peut voir que la précision est très bonne et que les positions sont conformes au tracé effectué durant le test. On remarque que comme montré dans la figure 11.1c, après 2 tours de piste, c'est à dire 800 m de course l'application montre une distance de 0.79 km ce qui est très proche de la réalité.

Au terme de l'exercice, on peut voir les valeurs totale enregistrée par l'application sur la figure 11.1d, une comparaison entre les résultats obtenus et ceux de la plateforme Strava sont disponible dans la table 11.1. On peut voir que les résultats des deux applications sont très proches ce qui permet de dire que la précision des données calculées et acquises par le système sont bonnes. Si on prend en considération qu'une montre GPS enregistre beaucoup plus de position que le capteur qui n'en enregistre qu'une toutes les 15 secondes on peut comprendre l'erreur qu'il y a entre les deux systèmes, en effet le fait de d'acquérir beaucoup plus de point permet d'avoir un calcul de la distance totale plus proche de la réalité car elle est calculé en tirant une droite entre deux points du tracé.

TABLE 11.1 – Comparaison des résultats

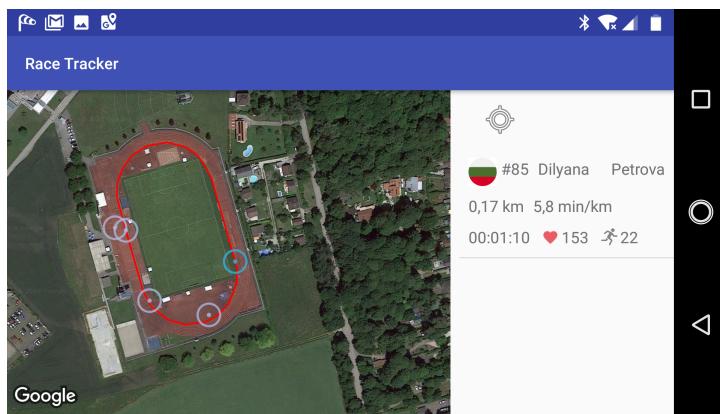
Plateform	Distance	Temps	Vitesse Moyenne
Strava	1.2 km	6.08 min	5.07 min/km
TB	1.1 km	6.15 min	5.4 min/km

### 11.2.2 Test obstruction passerelle

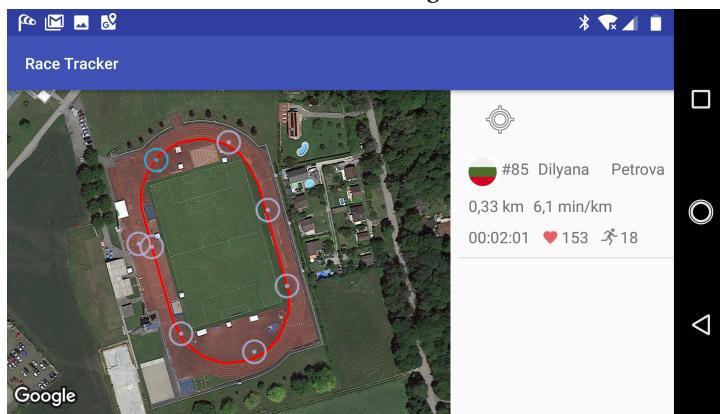
Le test d'obstruction consiste à courir dans un emplacement où la passerelle et le capteur ne sont plus en vue l'un de l'autre. Le but étant de vérifier si dans ces conditions les positions étaient en mesure d'atteindre la passerelle.

Il est possible de consulter la course de comparaison à l'adresse <https://www.strava.com/activities/1858813761>.

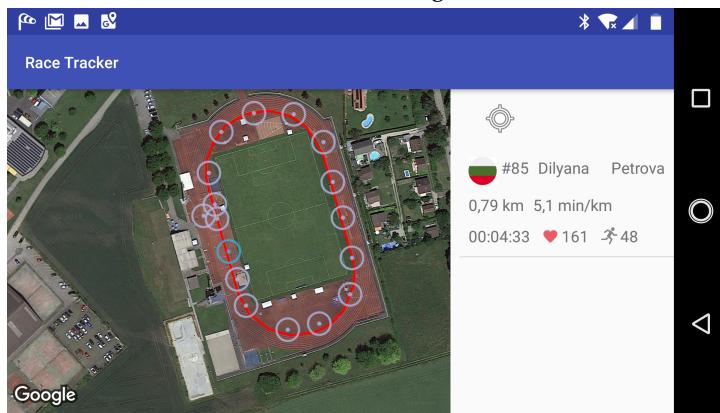
La figure 11.2 montre en rouge le tracé du chemin emprunté par la coureuse. On remarque que les premiers points sont bien reçus puis très rapidement la trace du capteur est perdu. Cet exercice permet de constater que malgré les arguments marketing qui promettent d'arriver à atteindre des distances de communication de plusieurs kilomètre en ville, un simple bâtiment peut rapidement empêcher toute communication de se faire.



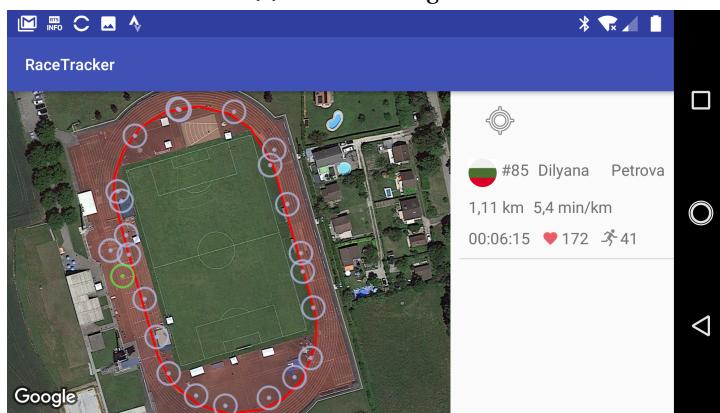
(a) Test #3 - Image #1



(b) Test #23- Image #2



(c) Test #3 - Image #3



(d) Test #23- Image #4

FIGURE 11.1 – Visualisation de l'application mobile pour le test 3

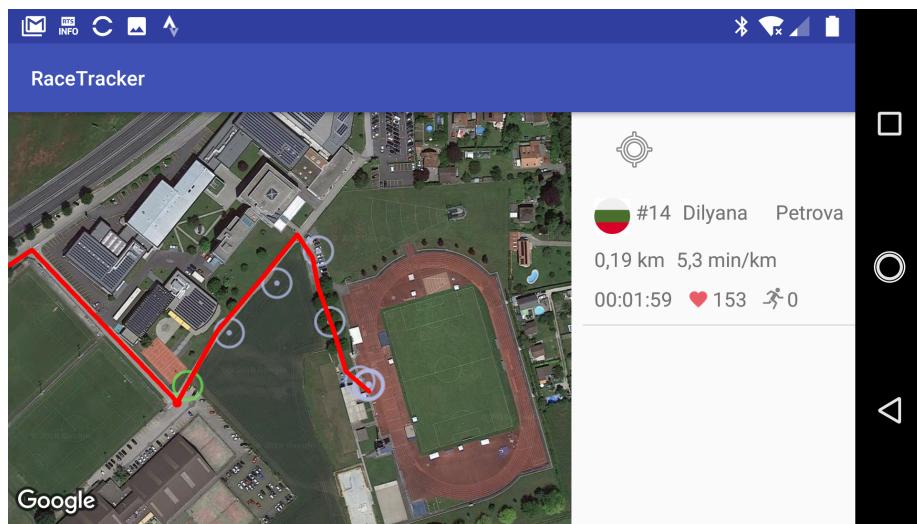


FIGURE 11.2 – Résultats du test d'obstruction

### 11.3 Conclusions

Le test de la phase #3 a permis de valider le fonctionnement du système dans son ensemble. Premier point positif, on peut déjà dire que malgré quelques problèmes le concept du système fonctionne bien, les positions GPS sont mis à jour correctement et les positions sont justes. Le rythme cardiaque, la distance totale ainsi que la vitesse moyenne montrent des valeurs proche de la réalité ce qui permet d'affirmer que le gros du système est à la hauteur des attentes.

Pour ce qui est de la cadence, même si on voit que des valeurs sont remontées à l'application ce qui est quand même encourageant, on remarque que les valeurs en elle-même sont totalement fausses, ceci étant dû au fait que les pas ne sont pas du tout détectés avec assez de précision ce qui fait que le compte est faux et donc la cadence également. L'algorithme de détection des pas requiert plus de travail afin de pouvoir correctement faire son travail et ainsi calculer une cadence correcte.

Le test a également mis en lumière que la communication LoRa semble être plus sensible qu'escompté aux obstructions entre l'émetteur et le récepteur, cet aspect demande plus de test afin de pouvoir dire précisément les effets de l'obstruction cependant cet élément peut devenir problématique pour le système et son utilisation. La seule solution qui pourrait permettre de pallier à ce problème serait l'utilisation d'un grand nombre de passerelle afin d'assurer que le capteur aie toujours la vue sur une d'entre elles afin de pouvoir transmettre les informations.

Dans l'ensemble le test de la phase #3 est un succès même si les résultats ne sont pas entièrement parfait, le gros du système fonctionne bien et l'utilisation de l'application mobile donne la satisfaction attendue.

## 12 Considérations pour le développement d'un produit



## 13 Dossier de gestion



## 14 Conclusion



## 15 Annexes

TODO: Doxygen



## Bibliographie

- [1] Semtech, “Lora modulation basics,” Semtech, Application Note AN1200.22, 05 2015.
- [2] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melià-Seguí, and T. Watteyne, “Understanding the limits of lorawan,” vol. 55, 06 2017.
- [3] E. C. of Postal and T. A. (CEPT), “Erc recommendation 70-03,” May 1997. [Online]. Available : <https://www.ecodocdb.dk/download/25c41779-cd6e/Rec7003.pdf>
- [4] L. A. T. Committee, “Lorawan™ 1.1 specification,” LoRa Alliance, Inc, Specification, 10 2017.
- [5] *EVA-8M u-blox 8 GNSS module Data Sheet*, Ubx-16009928 - r03 ed., UBlox, July 2016.
- [6] U-Blox, *u-blox 8 / u-blox M8 - Receiver Description Including Protocol Specification*, ubx-13003221 - r15 ed., March 2018.
- [7] *RN2483 LoRa Technology Module Command Reference User's Guide*, Ds40001784b revision b ed., Microchip Technology Inc., March 2015.
- [8] T. L. Foundation, “Zephyr project rtos.” [Online]. Available : <http://www.zephyrproject.org>
- [9] *SAMD21 Family Datasheet*, Ds40001882c rev. c ed., Microchip Technology Inc., June 2018.
- [10] *AT11628 : SAM D21 SERCOM I2C Configuration*, 42631st ed., Atmel, December 2015.
- [11] *LSM303AGR Accelerometer/Magnetometer Datasheet*, Docid027765 rev 9 ed., ST, September 2016.
- [12] Semtech, “Basic communication protocol between lora gateway and server.” [Online]. Available : [https://github.com/Lora-net/packet\\_forwarder/blob/master/PROTOCOL.TXT](https://github.com/Lora-net/packet_forwarder/blob/master/PROTOCOL.TXT)
- [13] S. T. T. C. H. J. L. Sech, “Single channel lorawan gateway.” [Online]. Available : [https://github.com/hallard/single\\_chan\\_pkt\\_fwd](https://github.com/hallard/single_chan_pkt_fwd)



### Authentification

Le soussigné, Léonard Bise, atteste par la présente avoir réalisé seul ce travail et n'avoir utilisé aucune autre source que celle expressément mentionnées, si ce n'est les connaissances acquises durant ses études et son expérience acquise dans une activité professionnelle.

Neuchâtel, le 21 Septembre 2018.



L. Bise