Cahier des Charges Techniques

Equipe AquaDrone

COLLOMB Jérémie - GOASDOUE Jérôme - PASCOLI Théo - STROCK Philippe - SZYMANSKI Loïc - VANDERPERRE François - YOU Thibault

2 février 2017

Responsable projet ONEMA LICCARDI Alexandre

Versions

Version	Date	Auteur	Modification
0.1	01/11/2016	Equipe projet	Version initiale
1.0	25/11/2016	Equipe projet	Version pour diffusion
1.1	25/11/2016	Equipe projet	Amélioration et Correction
1.2	13/12/2016	Chef de projet	Ajout de l'environnement de test
1.3	30/12/2016	Equipe projet	Refonte de l'ordre
1.4	02/02/2017	Chef de projet	Corrections mineures pour diffusion

Approbations

Etapes	Auteur	Rôle	Date
Ecriture	Equipe projet	-	02/02/2017
Relecture	Equipe projet	-	02/02/2017
Validation	Jérôme	Responsable Qualité et Documentation	02/02/2017
Approbation	Jérémie	Chef de projet	02/02/2017

Table des matières

Table des illustrations

Introduction

Le cahier des charges technique permet de réaliser une analyse technique complète du présent projet : AquaDrone (pour l'ONEMA).

Le but de ce document est de déterminer les besoins matériels et humains permettant d'atteindre les objectifs définis dans le cahier des charges fonctionnelles. La finalité du projet prend en compte les contraintes et les exigences de l'ONEMA. Les contraintes sont aussi bien logicielles et matérielles, que financières.

Les solutions retenues dans ce document sont celles qui sont développées lors de la phase de développement (« Rush »). Ce document décrit et documente le fonctionnement de chaque composant de l'application. Il permet ainsi de justifier chaque solution retenue.

Note importante
Le drone et le matériel l'entourant ont été choisi en collaboration avec l'ONEMA. Cependant, le choix a été validé avant la rédaction de ce document. Par conséquent, le cahier des charges techniques ne détaille pas la partie choix du matériel du drone.

1. Glossaire

Drone : Objet inhabité, piloté à distance, semi-autonome ou autonome, susceptible d'emporter différentes charges utiles le rendant capable d'effectuer des tâches spécifiques pendant une durée pouvant varier en fonction de ses capacités.

Capteur : Système servant à détecter un phénomène physique, souvent sous forme de signal électrique dans le but de le représenter.

GPS : « Global Positioning System », système de géolocalisation utilisant des signaux satellites pour identifier une position.

Bathymétrie : Science de la mesure des profondeurs et du relief de l'océan pour déterminer la topographie du sol de la mer.

Topographie : Science qui permet la mesure puis la représentation sur un plan ou une carte des formes et détails visibles sur un terrain. Ils peuvent être naturels (notamment le relief et l'hydrographie) ou artificiels (comme les bâtiments, les routes, etc.).

Temps réel : En informatique, un système temps réel est un système capable de contrôler un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé. Par exemple, les mesures réalisées par un drone sont affichées en direct (sans latence) sur un écran déporté, plus ou moins loin de la source du drone.

Turbidité : Caractéristique optique de l'eau, il s'agit de sa capacité à diffuser ou absorber la lumière provenant du « ciel ».

Salinité : Teneur en sel d'un milieu.

Luminométrie : Mesure de l'intensité lumineuse.

Waterproof : Terme employé pour parler de quelque chose d'imperméable, d'étanche à l'eau.

Géomatique : Ensemble des technologies permettant de modéliser, représenter et analyser le territoire pour en faire des représentations virtuelles.

IGN : "Institut Géographique Nationale". C'est l'organisme qui s'occupe de fournir les informations géographiques de référence en France.

POC : "Proof of concept". Il s'agit d'apporter des preuves sur la faisabilité ou non d'un projet.

Plongée : Passage du drone de la surface de l'eau à un milieu complètement aquatique.

Remonté : Passage d'un milieu complétement aquatique à la surface de l'eau.

Système : Le système permet de diriger le drone, de le localiser, d'effectuer des mesures, d'envoyer des informations au pilote ou à l'analyste

Analyste : L'analyste va traiter l'ensemble des informations affichées par le système, et va les afficher.

Pilote : Personne physique qui va diriger le drone et les mesures à l'aide des informations du système et du pilote.

Zone de récupération : Lieu où la personne en charge du robot peut le récupérer

Accéléromètre : Instrument permettant de mesurer l'accélération d'un mouvement et d'étudier l'intensité des chocs et vibrations qui en résultent

Gyroscope : Appareil indiquant une direction constante grâce à un axe autour duquel il tourne.

Compas : Le compas est un instrument de navigation qui donne une référence de direction (le nord) sur le plan horizontal.

Station de supervision : équipement recevant et traitant les informations captées par le drone et les capteurs.

Station de contrôle/pilotage : équipement permettant à une personne physique de diriger directement ou indirectement (par le biais de programme) le drone et les capteurs.

GPLv3 (Open-Source) : Licence open source la plus permissive sur le marché.

Trace (Log) : Données sauvegardées afin de pouvoir les réutiliser.

Calibrage : Géolocalisation de l'ensemble du trajet effectué.

Canvas : Fenêtre du logiciel QGIS affichant les données géolocalisées.

Flux MAVLINK : Micro Air Vehicle Link est un protocole de communication pour les drones. Il est utilisé pour la communication entre le drone et la station de pilotage.

WGS84 : Système géodésique de référence pour le GPS. Les coordonnées issues du GPS seront reprojeter en Lambert 93 pour respecter la réglementation Française

Lambert-93 : Système de projection Français de référence

Géométrie : Représentations ponctuelles, linéaires ou polygones servant à représenter, sur une interface cartographique, des données ayant un champ Geometry.

IMU : Acronyme pour représenter l'accéléromètre, le gyroscope et le compas (boussole).

SIREN : ou Système d'Interconnexion des Ressources de l'Environnement et de la Navigation est le nom de l'application développée.

2. Architecture

2.1. Globale

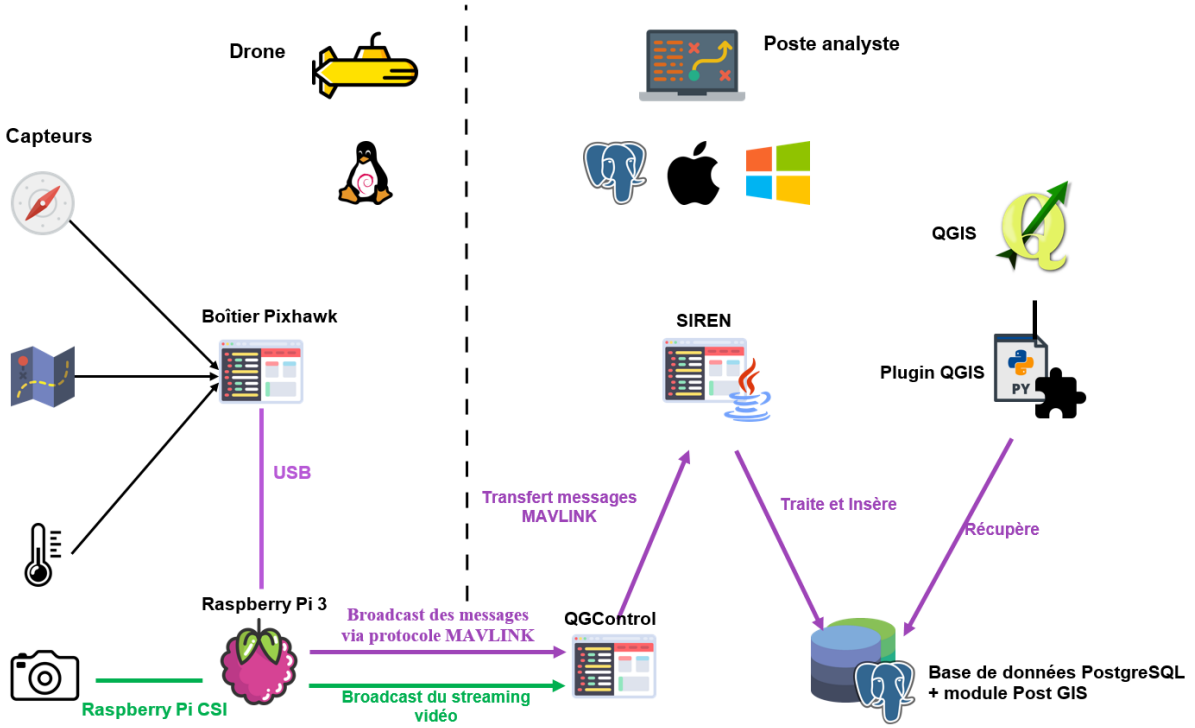


Figure 1 - Architecture fonctionnelle globale

L'architecture de l'application est complexe. En effet, le drone fonctionne de base sur une architecture multi couche. Le schéma ci-dessus décrit le fonctionnement de l'application qui sera développé.

Le premier élément important est le boitier Pixhawk. Ce composant est situé sur le drone. Celui-ci permet de réaliser le pilotage du drone. Il héberge le système du drone : ArduSub. Cependant, nous devons y intégrer ce système, puisque le code source ArduSub n'est pas présent de base sur le Pixhawk. Il ne sert pas uniquement au pilotage : il est utilisé pour les mesures. En effet, tous les capteurs sont connectés à ce composant. Par conséquent, le Pixhawk se doit de récupérer et de transférer les mesures. Pour ce faire, le composant utilise le protocole MAVLink (cf. 4.1.8 Le protocole MAVLink pour une description plus ample). Ce protocole est utilisé sur la liaison USB reliant le Pixhawk au Raspberry.

Le second composant important du drone est le Raspberry. Ce composant sert de relai. Il reçoit le flux MAVLink via un port USB et le retransmet en broadcast sur le port Ethernet. Cependant, son rôle ne s'arrête pas là. Le mini-ordinateur a la responsabilité de recevoir le flux vidéo émis par la caméra du drone (sur le port Raspberry PI CSI) pour le retransmettre sur le port Ethernet. Le flux vidéo et le flux MAVLink cohabitent donc sur la liaison Ethernet.
Le flux MAVLink est bidirectionnel. Par conséquent, le Raspberry peut recevoir des messages MAVLink sur le port Ethernet. Ces messages sont alors automatiquement retransmis au Pixhawk.

Le composant principal présent sur le poste client est le QGControl. Ce logiciel permet de piloter le drone. Ce pilotage est réalisé grâce au flux vidéo et au flux de messages MAVLink. Le logiciel permet de rediriger le flux MAVLink vers un autre client. Cette possibilité est utilisée dans le cadre de notre

application. Les messages sont alors envoyés (une copie) vers SIREN (Système d'Interconnexion des Ressources de l'Environnement et de la Navigation – Application principale). De plus, nous allons développer une surcouche en Java afin de pouvoir y insérer de nouveaux indicateurs.

Le premier composant développé dans le cadre du projet est l'application SIREN. Après analyse (cf. 3.5.3 Calcul de la position), il est plus avantageux que l'application calcule toutes positions. Cette application permet donc de calculer la position précise du drone. Chaque mesure est alors associée à une position. Ces informations de positions et de mesures proviennent des messages MAVLink provenant du QGControl. Une fois le calcul des positions/mesures effectives terminé, l'application ajoute les données dans la base de données PostGIS.

La base de données PostGIS est le point central de stockage. C'est dans cette base que toutes les mesures et que toutes les positions sont stockées.

Le module QGIS permet de réaliser l'affichage des mesures stockées en base de données. L'affichage est réalisé de deux manières : directe ou différée. En cas d'affichage direct, le module se charge de détecter les modifications en base de données avant de les afficher. Dans le cadre de l'affichage différé, une récupération simple des mesures réalisées est effectuée.

Une fois la récupération des données effectives sur le module QGIS, un affichage est réalisé dans le logiciel de cartographie QGIS.

Le protocole UDP ne garantit pas le transit d'un datagramme d'une source à une destination (cf. 4.1.8.3 Configuration – aucune garantie d'acheminement et d'ordre d'arrivée). Nous avons donc réalisé différents choix :

- La perte d'un message MAVLink est impossible à détecter. En effet, la détection de la perte devrait être réalisée au travers d'un numéro de séquence manquant (ou équivalent). Le protocole de communication MAVLink n'est pas étudié pour ce type de situation. La perte d'un paquet est donc indétectable. Cette situation reste donc sans solution.
- UDP ne garantit pas l'ordre d'arrivée des messages MAVLink. Par conséquent, certains paquets plus anciens peuvent arriver après des paquets plus récents. Dans cette situation, l'équipe projet a décidé d'ignorer les messages « du passé ». La détection de ce type de situation est très simple : un champ représentant l'heure d'envoi du message est présent. Il suffit donc de comparer l'heure du message reçu à l'instant, et l'heure du précédent message.

Au vu de l'architecture, les pertes et l'arrivée dans le désordre des paquets sont extrêmement rare. En effet, aucun élément de routage (pouvant perturber la communication) n'est présent entre :

- *Le drone (Raspberry Pi) et la station QGControl*
- *La station QGControl et SIREN.*

2.2. Simulateur

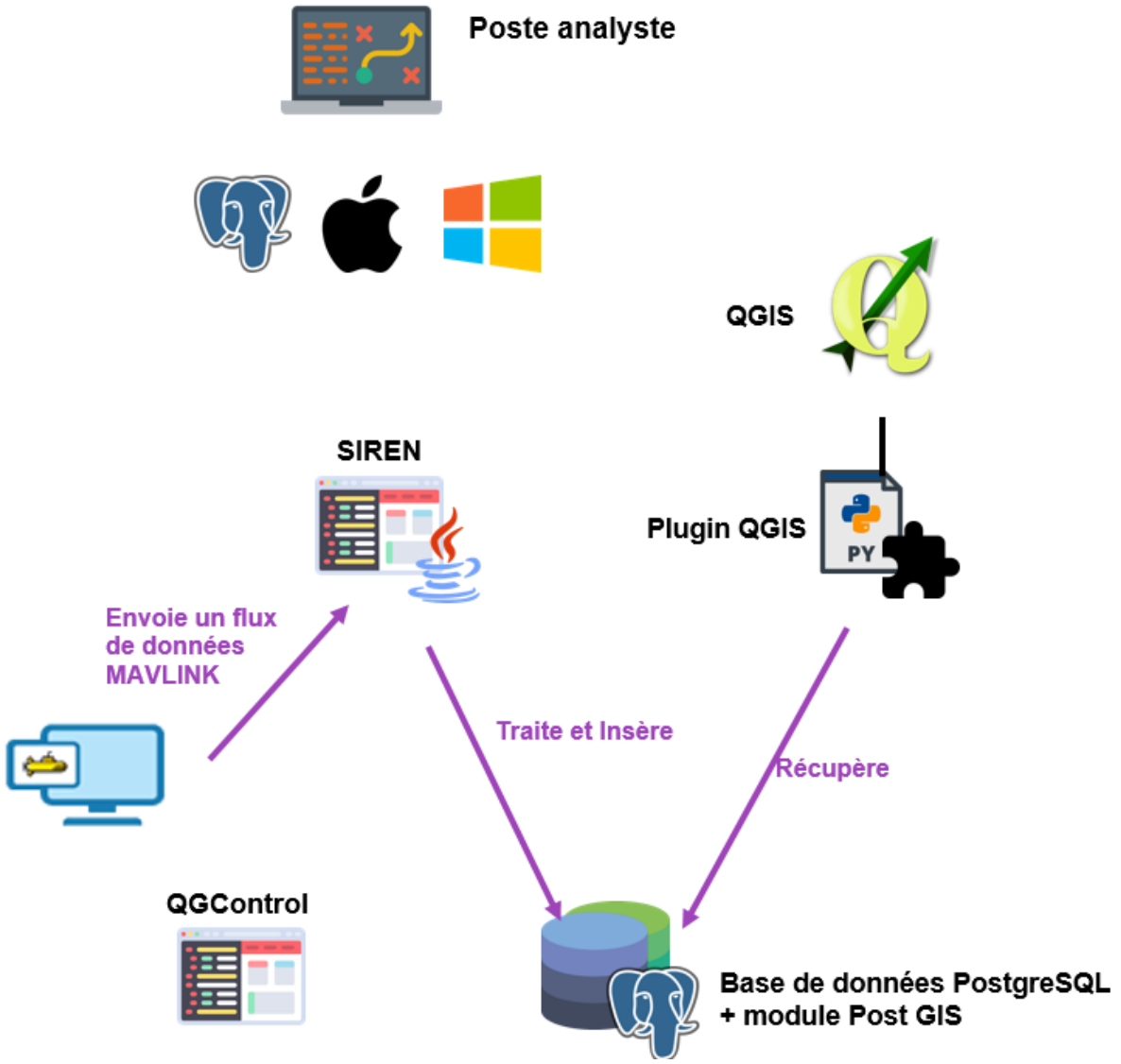


Figure 2 - Architecture fonctionnelle du Simulateur

Le rôle du simulateur est de nous permettre de vérifier la pertinence de l'algorithme utilisé pour le positionnement et le bon fonctionnement des outils mis en place pour l'affichage des relevés en temps réel, le tout en s'affranchissant du drone et des contraintes physiques qu'il peut présenter (en termes de temps et d'espace).

Ainsi l'architecture fonctionnelle est quasiment identique à celle du cas réel, le simulateur vient simplement se brancher en entrée de l'application de traitements des données à la place de QGControl afin de fournir le flux de données MAVLINK qui doit être traité.

Ce flux de données générées correspond à une recopie des messages MAVLink provenant du QGControl. Ainsi, tous les messages sortant du simulateur représentent une situation pouvant se produire sur le drone.
Il est important, pour que le simulateur puisse remplir son rôle, d'être en mesure de simuler avec précision le flux (qualitativement et quantitativement) que pourrait recevoir l'application dans le cadre d'une utilisation réelle.

2.3. Communication/QGIS

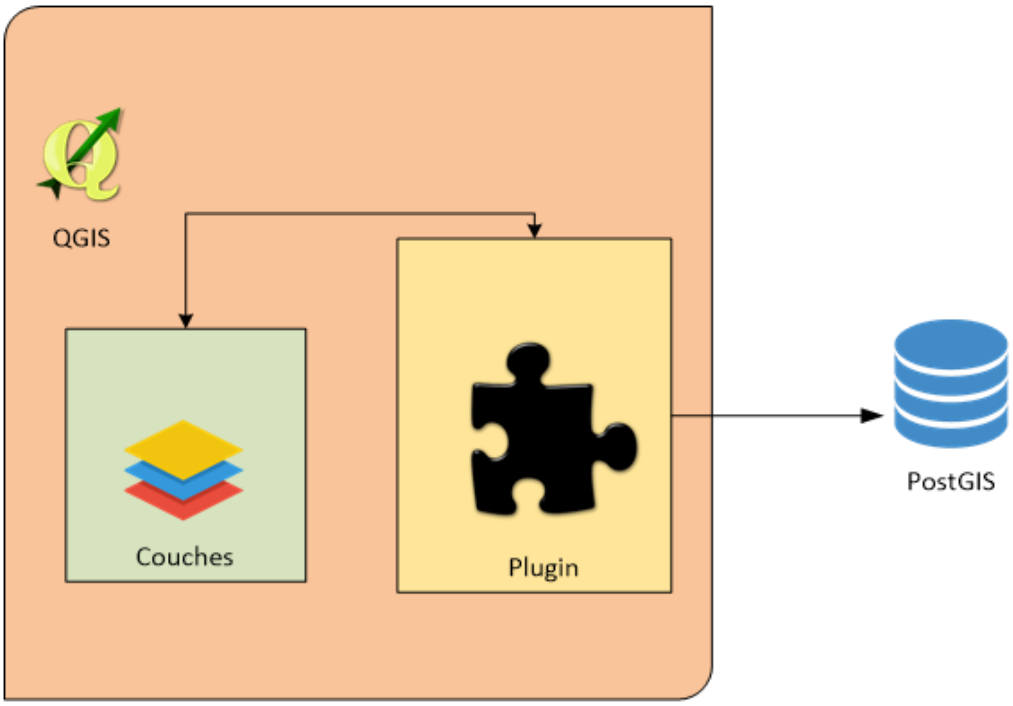


Figure 3 - Architecture fonctionnelle du module QGIS

Le composant QGIS permet de visualiser les différents éléments relatifs à l'affichage des positions/mesures sur l'application cartographique QGIS. Deux situations sont possibles pour le module développé : visualiser en direct (live) les mesures réalisées par le drone ; visualiser une plongée déjà réalisée.

Dans le premier cas, visualiser en direct :

- Le plugin QGIS se charge de détecter les modifications réalisées dans la base de données PostGIS (cf. 3.4 Base de données pour plus d'informations).
- Lorsqu'il détecte des modifications, le plugin met à jour l'affichage de QGIS. La mise à jour est uniquement réalisée sur les points (mesures) ayant été modifiés/ajoutés.

Dans le second cas, visualiser une plongée précédemment réalisée, le plugin aura la responsabilité de récupérer les données en base, puis de les afficher.

Dans tous les cas, le plugin QGIS offre différentes possibilités. L'analyste (utilisateur du plugin QGIS) a la possibilité de choisir s'il affiche les données brut (sans recalcul, c'est à dire les données positionnées sans prendre en compte les contraintes dues au courant) ou les données recalculées. Cette possibilité permet à l'utilisateur de réaliser des analyses différentes. L'analyste peut alors se rendre compte de la précision du drone dans la plongée affichée. Il sera également possible d'afficher ces données sur plusieurs couches.

2.4. Développement

2.4.1. Global

Parmi tous les outils et applications que le projet va utiliser, certains existent déjà et d'autres devront être développé par l'équipe. Parmi les outils qui seront programmer, nous avons tout d'abord SIREN. SIREN permet de calculer la position précise du drone. L'application QGControl envoi à SIREN les informations de position. SIREN traite ces données et les transmet vers la base de données PostGIS.

L'équipe AquaDrone développera également une interface s'ajoutant à QGControl. Cette interface sera indépendante de QGControl, mais sera utilisée en parallèle de celui-ci. Elle affichera des indications pour le pilote comme la durée de la plongée en cours.

Un plugin QGis permettant l'affichage des couches métiers sera également développé dans le cadre du projet.

Les programmes ArduSub et QGControl existent déjà et ne seront pas modifiés dans le cadre du projet.

QGControl est un programme open source qui permet le pilotage du drone. Nous avons préféré programmer des fenêtres qui seront superposées à QGControl plutôt que de modifier le code source de l'application. En effet, le développement de nouvelle fonctionnalité dans l'interface de pilotage requiert un temps de développement que l'équipe ne possède pas.

ArduSub est également open source. ArduSub permet le pilotage du drone et il ne nous est pas nécessaire de le modifier. Cependant, une documentation complète des modifications à réaliser (sur le code source d'ArduSub) sera rédigée afin d'expliquer la procédure pour intégrer un nouveau capteur.

2.4.2. Simulateur

L'équipe projet développera également un simulateur qui aura la responsabilité de représenter, fidèlement, les messages MAVLink qui sont envoyés par QGControl (lorsque celui-ci est connecté au drone – situation réel de prise de mesure). Ce simulateur se chargera de générer des données de capteurs en essayant de reproduire au mieux la réalité et de les convertir en format MAVLink pour l'envoi à l'application SIREN.

3. Mise en œuvre

3.1. Diagrammes de Séquence

Pour faciliter la compréhension des Cas d'Utilisations (CU), une étiquette a été ajoutée dans le nom de ceux-ci.

L'étiquette « Simulation » indique que ce CU est présent uniquement lors de l'utilisation de la simulation par les testeurs.

L'étiquette « Drone » indique que ce CU est toujours présent. C'est-à-dire que celui-ci sera utilisé que l'on soit en simulation, ou dans un environnement « réel » et naturel.

3.1.1. Simulation : Générer les entrées et les fichiers de simulation

Identifiant : 1

Type : Primaire

Acteurs :

- **Principal** : Testeur

Résumé : Avant de réaliser la simulation, il est nécessaire de générer la configuration et le déroulement de la simulation. Le testeur fournit à l'application un fichier contenant les positions réelles du drone, et le système fournit en sortie les différentes informations que devraient fournir les capteurs dans une situation réel.

Précondition : Le testeur possède le fichier d'entrée* (ou est en mesure de le créer).

Post-condition : Le simulateur est capable de lire les données du fichier et d'en déduire des informations.

Fichier d'entré (CSV) :

Champ	Description
Timestamp	Identifiant dans le temps. Dans le cadre de la simulation, le champ est incrémenté de 1 à chaque ligne.
Latitude	Position transversale du drone par rapport au référentiel WSG84
Longitude	Position longitudinale du drone par rapport au référentiel WSG84
Altitude	Altitude du drone en mètre par rapport au niveau de la mer
Cap	Cap magnétique du drone (orientation)
Mesure 1 (température)	Température mesurée en degré Celsius

Tableau 1 - UC 1 : fichier d'entrée

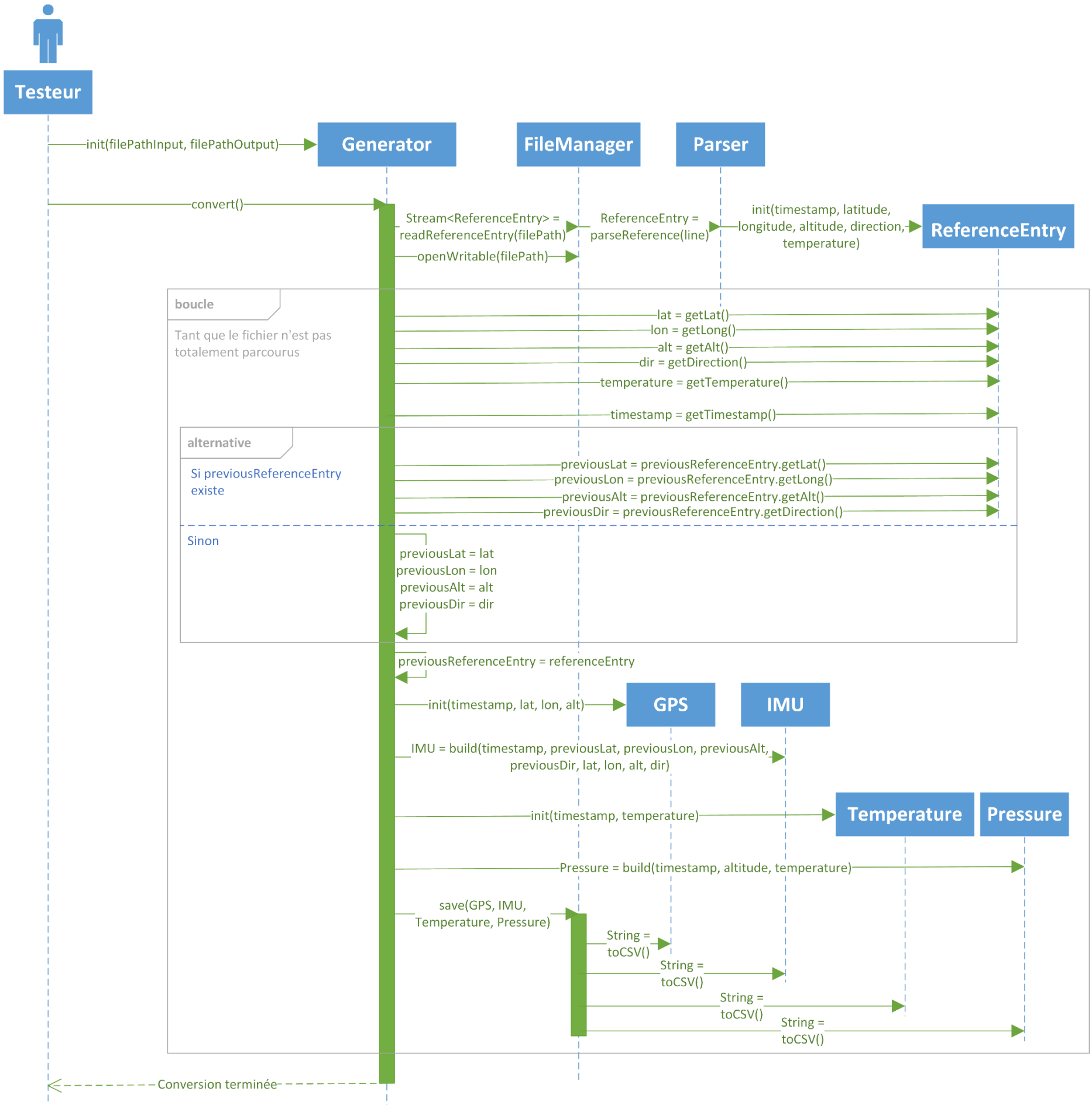
***Exemple** : 1, 33.654533, -1.3547486, -2.5, 258, 16.9*

Fichier de sortie (CSV) :

Champ	Description
Timestamp	Identifiant dans le temps. Dans le cadre de la simulation, le champ est incrémenté de 1 à chaque ligne. (Identique au fichier d'entrée)
GPS longitude	Position longitudinale fournie par le GPS (en surface – début et fin de plongée)
GPS latitude	Position transversale fournie par le GPS (en surface – début et fin de plongée)
GPS altitude	Altitude fournie par le GPS (en surface – début et fin de plongée)
Accélération X	Accélération (translation) sur l'axe X (mg - accélération)
Accélération Y	Accélération (translation) sur l'axe Y (mg - accélération)
Accélération Z	Accélération (translation) sur l'axe Z (mg - accélération)
Rotation X	Vitesse de rotation sur l'axe X (millirad /sec)
Rotation Y	Vitesse de rotation sur l'axe Y (millirad /sec)
Rotation Z	Vitesse de rotation sur l'axe Z (millirad /sec)
Cap X	Orientation magnétique du capteur (sens du drone) sur l'axe X par rapport au Nord magnétique (milli tesla).
Cap Y	Orientation magnétique du capteur (sens du drone) sur l'axe Y par rapport au Nord magnétique (milli tesla).
Cap Z	Orientation magnétique du capteur (sens du drone) sur l'axe Z par rapport au Nord magnétique (milli tesla).
Pression	Pression Absolue mesurée (hectopascal)
Mesure 1 (température)	Température mesurée en degré Celsius (identique au fichier d'entrée)
Latitude	Position transversale du drone par rapport au référentiel WSG84 (identique au fichier d'entrée)
Longitude	Position longitudinale du drone par rapport au référentiel WSG84 (Identique au fichier d'entrée)
Altitude	Altitude du drone en mètre par rapport au niveau de la mer (Identique au fichier d'entrée)
Cap réel	Cap magnétique du drone (orientation) (identique au fichier d'entrée)

Tableau 2 - UC 1 : fichier de sortie

***Exemple** : 1, 33.654533, -1.3547486, -2.5, 0, 0, 0, 0, 0, 0, 25, 25, 0, 1015.50, 16.9, 33.654533, -1.3547486, -2.5, 258*


Testeur
Generator
FileManager
Parser
ReferenceEntry
GPS
IMU
Temperature
Pressure
init(filePathInput, filePathOutput)
convert()
Stream<ReferenceEntry> = readReferenceEntry(filePath)
ReferenceEntry = parseReference(line)
init(timestamp, latitude, longitude, altitude, direction, temperature)
openWritable(filePath)
boucle
Tant que le fichier n'est pas totalement parcourus
lat = getLat()
lon = getLong()
alt = getAlt()
dir = getDirection()
temperature = getTemperature()
timestamp = getTimestamp()
alternative
Si previousReferenceEntry existe
previousLat = previousReferenceEntry.getLat()
previousLon = previousReferenceEntry.getLong()
previousAlt = previousReferenceEntry.getAlt()
previousDir = previousReferenceEntry.getDirection()
Sinon
previousLat = lat
previousLon = lon
previousAlt = alt
previousDir = dir
previousReferenceEntry = referenceEntry
init(timestamp, lat, lon, alt)
IMU = build(timestamp, previousLat, previousLon, previousAlt, previousDir, lat, lon, alt, dir)
init(timestamp, temperature)
Pressure = build(timestamp, altitude, temperature)
save(GPS, IMU, Temperature, Pressure)
String = toCSV()
String = toCSV()
String = toCSV()
String = toCSV()
Conversion terminée

3.1.2. Simulation : Effectuer une simulation

Identifiant : 2

Type : Primaire

Acteurs :

- **Principal** : Testeur

Résumé : Le testeur charge une simulation puis l'exécute. Le système simule alors le fonctionnement du drone et des capteurs pour les transmettre à l'application. Cette simulation permet de vérifier le bon fonctionnement des différents algorithmes de l'application.

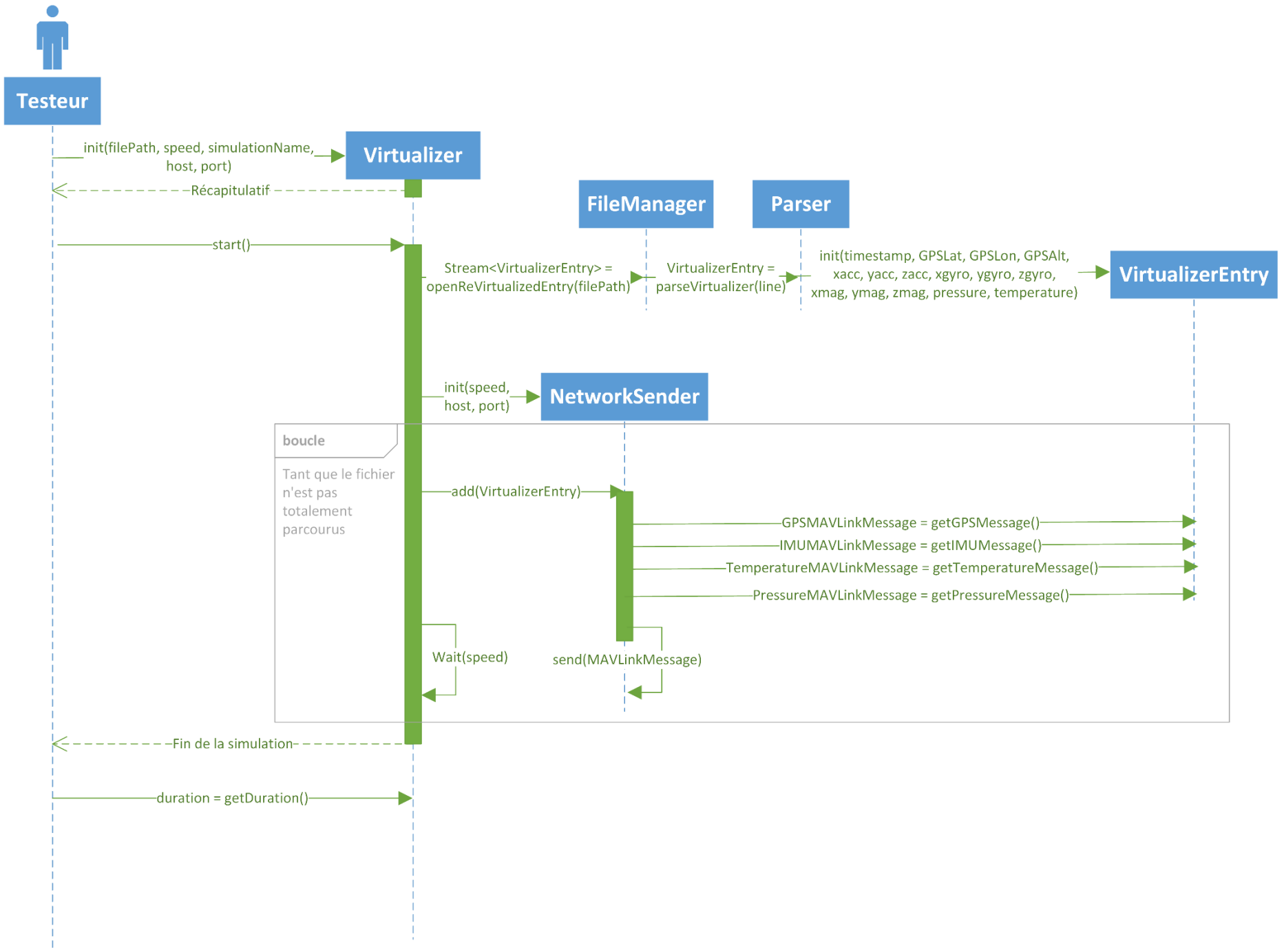
Précondition : UC-1 Simulation : générer les entrées et les fichiers de simulations / posséder un export d'une situation réelle (au format du fichier de sortie de l'UC-1).

Post condition : La simulation est terminée. Un indice de cohérence des données obtenues via l'algorithme de positionnement est donné ainsi que les informations de l'environnement dans un fichier de logs.

Fichier d'entré (CSV) :

Champ	Description
Timestamp	Identifiant dans le temps. Dans le cadre de la simulation, le champ est incrémenté de 1 à chaque ligne. (Identique au fichier d'entrée)
GPS longitude	Position longitudinale fournie par le GPS (en surface – début et fin de plongée)
GPS latitude	Position transversale fournie par le GPS (en surface – début et fin de plongée)
GPS altitude	Altitude fournie par le GPS (en surface – début et fin de plongée)
Accélération X	Accélération (translation) sur l'axe X (mg - accélération)
Accélération Y	Accélération (translation) sur l'axe Y (mg - accélération)
Accélération Z	Accélération (translation) sur l'axe Z (mg - accélération)
Rotation X	Vitesse de rotation sur l'axe X (millirad /sec)
Rotation Y	Vitesse de rotation sur l'axe Y (millirad /sec)
Rotation Z	Vitesse de rotation sur l'axe Z (millirad /sec)
Cap X	Orientation magnétique du capteur (sens du drone) sur l'axe X par rapport au Nord magnétique (milli tesla).
Cap Y	Orientation magnétique du capteur (sens du drone) sur l'axe Y par rapport au Nord magnétique (milli tesla).
Cap Z	Orientation magnétique du capteur (sens du drone) sur l'axe Z par rapport au Nord magnétique (milli tesla).
Pression	Pression Absolue mesurée (hectopascal)
Mesure 1 (température)	Température mesurée en degré Celsius (identique au fichier d'entrée)
Latitude	Position transversale du drone par rapport au référentiel WSG84 (identique au fichier d'entrée)
Longitude	Position longitudinale du drone par rapport au référentiel WSG84 (Identique au fichier d'entrée)
Altitude	Altitude du drone en mètre par rapport au niveau de la mer (Identique au fichier d'entrée)
Cap réel	Cap magnétique du drone (orientation) (identique au fichier d'entrée)

Tableau 3 - UC 2 : fichier d'entré


Testeur
Virtualizer
FileManager
Parser
VirtualizerEntry
NetworkSender
init(filePath, speed, simulationName, host, port)
Récapitulatif
start()
Stream<VirtualizerEntry> = openReVirtualizedEntry(filePath)
VirtualizerEntry = parseVirtualizer(line)
init(timestamp, GPSLat, GPSLon, GPSAlt, xacc, yacc, zacc, xgyro, ygyro, zgyro, xmag, ymag, zmag, pressure, temperature)
init(speed, host, port)
boucle
Tant que le fichier n'est pas totalement parcourus
add(VirtualizerEntry)
GPSMAVLinkMessage = getGPSMessage()
IMUMAVLinkMessage = getIMUMessage()
TemperatureMAVLinkMessage = getTemperatureMessage()
PressureMAVLinkMessage = getPressureMessage()
Wait(speed)
send(MAVLinkMessage)
Fin de la simulation
duration = getDuration()

3.1.3. Simulation : Vérifier la cohérence des données relatives au positionnement

Identifiant : 3

Type : Primaire

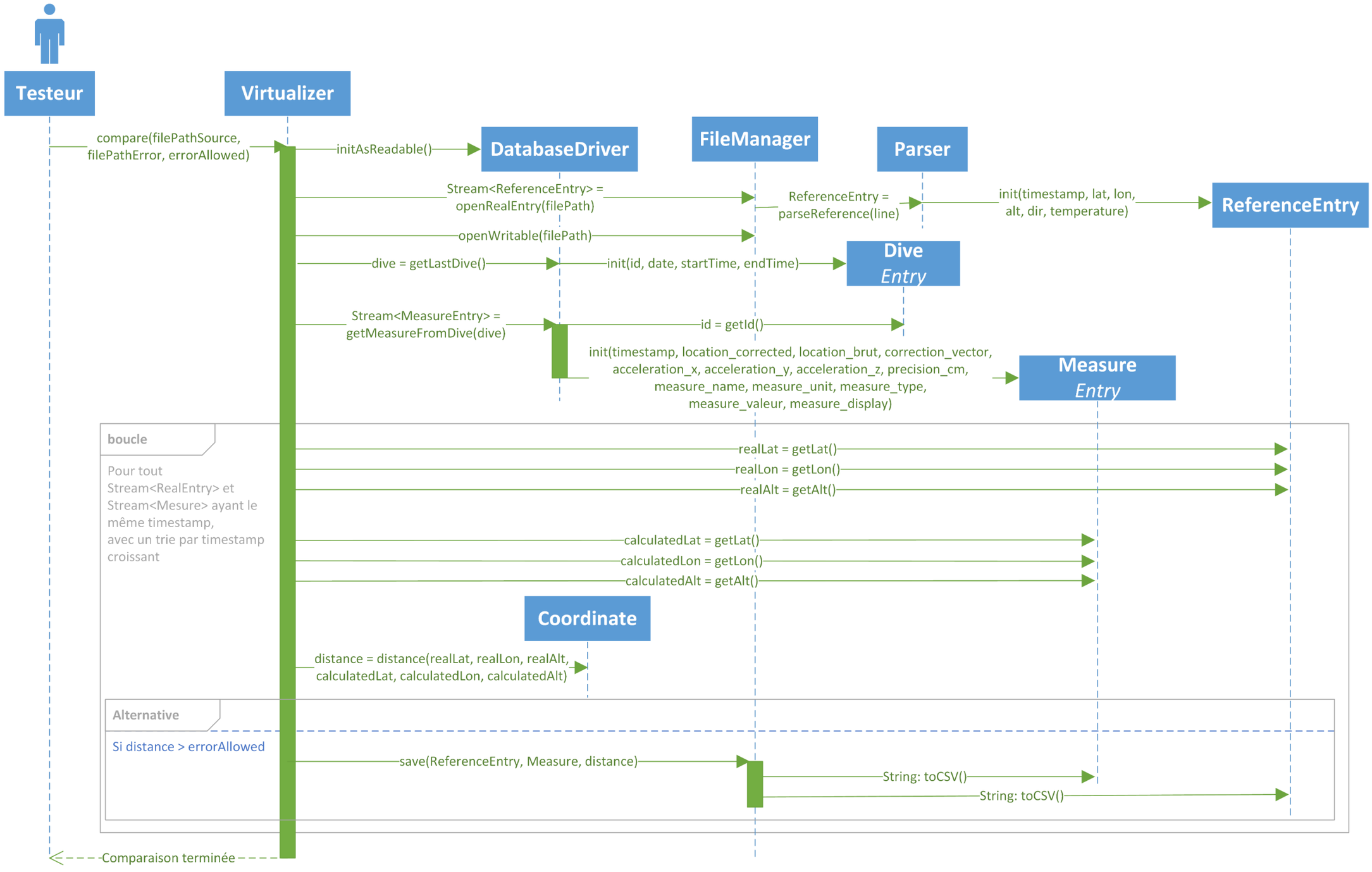
Acteurs :

- **Principal** : Testeur

Résumé : Après la simulation d'une sortie et le traitement du flux simulé par l'application qui calcule les positions, le simulateur vérifie la cohérence des résultats et récapitule son analyse dans un fichier.

Précondition : Un jeu de données pour le simulateur doit avoir été généré (cf. UC-1 - Simulation : Générer les entrées et les fichiers de simulation), et une simulation de déplacement du drone utilisant ces données doit avoir été effectuée (cf. UC 1 – 3.1.2 Simulation : Effectuer une simulation).

Post condition : Le testeur dispose d'un fichier récapitulant le jeu de données concerné, l'algorithme de calcul utilisé, le seuil d'erreur autorisé et différents indicateurs statistiques.


Testeur
Virtualizer
DatabaseDriver
FileManager
Parser
ReferenceEntry
Dive
Entry
Measure
Entry
Coordinate
compare(filePathSource, filePathError, errorAllowed)
initAsReadable()
Stream<ReferenceEntry> = openRealEntry(filePath)
ReferenceEntry = parseReference(line)
init(timestamp, lat, lon, alt, dir, temperature)
openWritable(filePath)
dive = getLastDive()
init(id, date, startTime, endTime)
Stream<MeasureEntry> = getMeasureFromDive(dive)
id = getId()
init(timestamp, location_corrected, location_brut, correction_vector, acceleration_x, acceleration_y, acceleration_z, precision_cm, measure_name, measure_unit, measure_type, measure_valeur, measure_display)
boucle
Pour tout Stream<RealEntry> et Stream<Mesure> ayant le même timestamp, avec un trie par timestamp croissant
realLat = getLat()
realLon = getLon()
realAlt = getAlt()
calculatedLat = getLat()
calculatedLon = getLon()
calculatedAlt = getAlt()
distance = distance(realLat, realLon, realAlt, calculatedLat, calculatedLon, calculatedAlt)
Alternative
Si distance > errorAllowed
save(ReferenceEntry, Measure, distance)
String: toCSV()
String: toCSV()
Comparaison terminée

3.1.4. Drone : Réaliser des mesures (sur QGis)

Identifiant : 4

Type : primaire

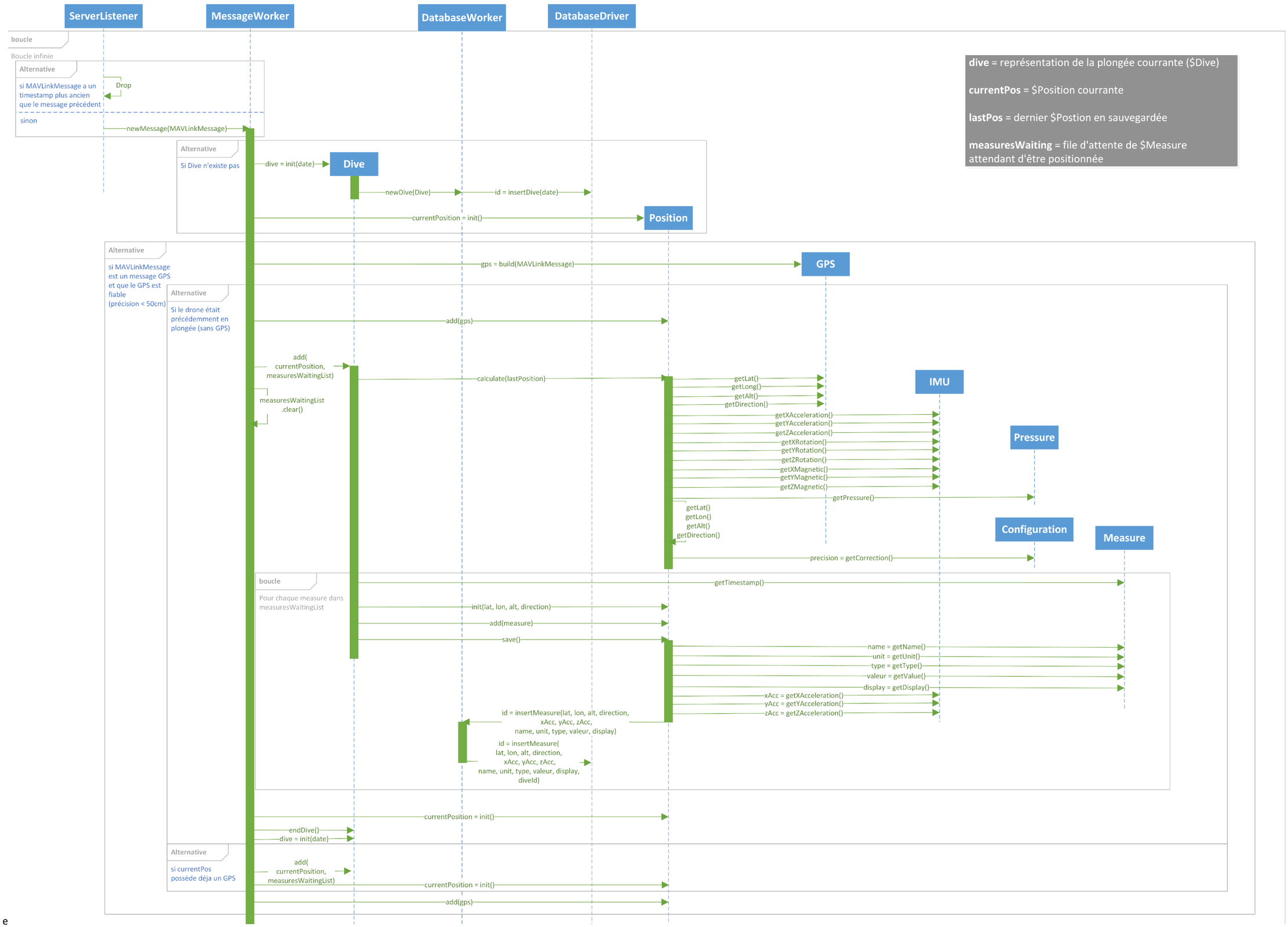
Acteurs :

- **Principal :** Analyste
- **Concerné :** Tous les capteurs

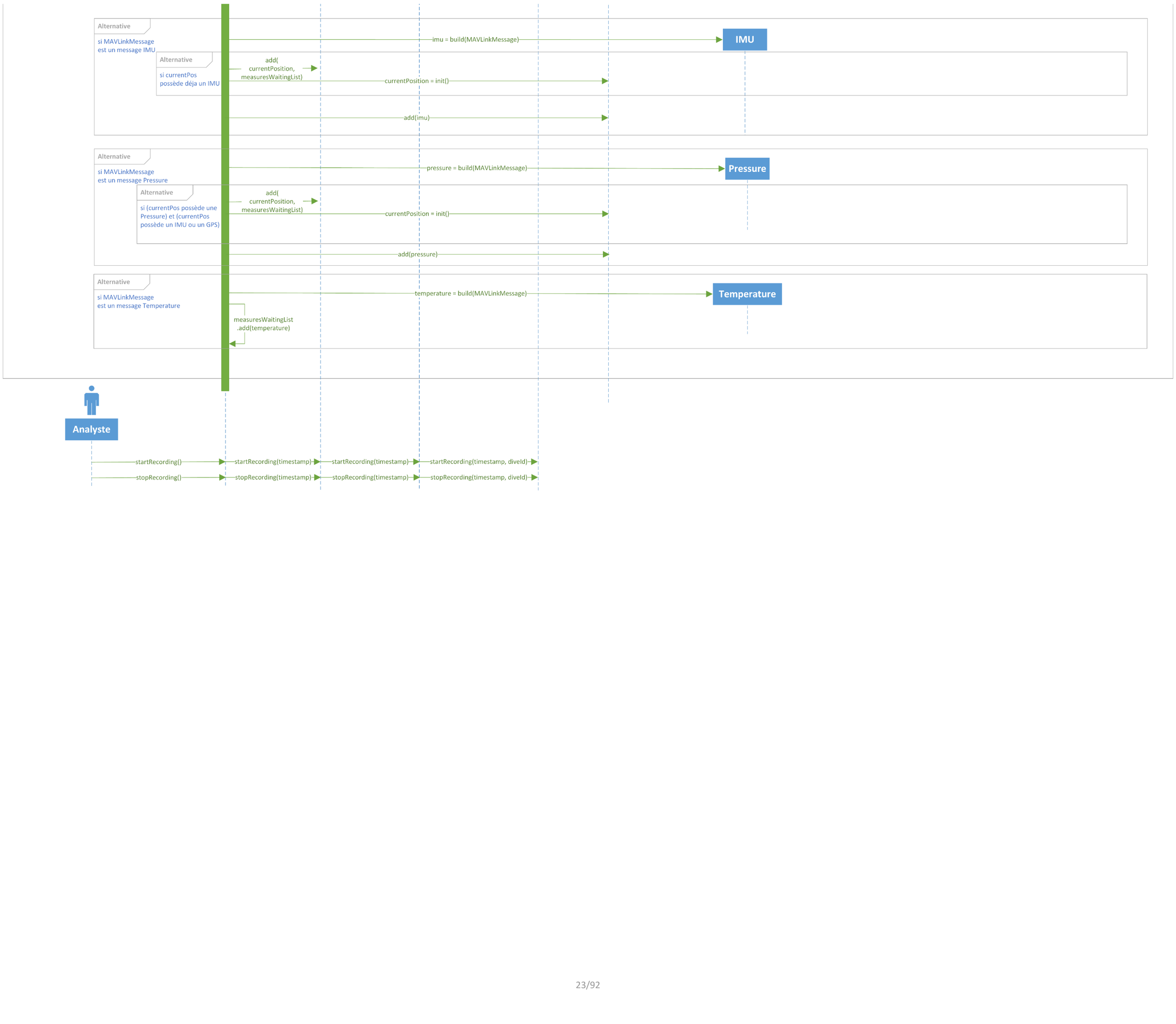
Résumé : Action permettant de démarrer une série de mesures sur QGis.

Préconditions : Le drone doit être démarré.

Post-condition : Les données des différents capteurs sont affichées sur la canevas. Les valeurs des mesures sont stockées dans la BDD PostGIS. Plusieurs champs y sont indiqués (type de données, valeur, position, géométrie). Un fichier de logs sera créé et permettra de voir les actions réalisées par l'utilisateur et le drone.


ServerListener
MessageWorker
DatabaseWorker
DatabaseDriver
boucle
Boucle infinie
Alternative
si MAVLinkMessage a un timestamp plus ancien que le message précédent
Drop
sinon
newMessage(MAVLinkMessage)
Alternative
Si Dive n'existe pas
dive = init(date)
Dive
newDive(Dive)
id = insertDive(date)
currentPosition = init()
Position
dive = $Dive courante
currentPos = $Position courrante
lastPos = dernier $Postion en sauvegardée
measuresWaiting = file d'attente de $Measure attendant d'être positionnée
Alternative
si MAVLinkMessage est un message GPS et que le GPS est fiable (précision < 50cm)
gps = build(MAVLinkMessage)
GPS
Alternative
Si le drone était précédemment en plongée (sans GPS)
add(gps)
add(currentPosition, measuresWaitingList)
calculate(lastPosition)
measuresWaitingList .clear()
getLat()
getLong()
getAlt()
getDirection()
IMU
getXAcceleration()
getYAcceleration()
getZAcceleration()
getXRotation()
getYRotation()
getZRotation()
getXMagnetic()
getYMagnetic()
getZMagnetic()
Pressure
getPressure()
getLat()
getLon()
getAlt()
getDirection()
Configuration
Measure
precision = getCorrection()
boucle
Pour chaque measure dans measuresWaitingList
getTimestamp()
init(lat, lon, alt, direction)
add(measure)
save()
name = getName()
unit = getUnit()
type = getType()
valeur = getValue()
display = getDisplay()
xAcc = getXAcceleration()
yAcc = getYAcceleration()
zAcc = getZAcceleration()
id = insertMeasure(lat, lon, alt, direction, xAcc, yAcc, zAcc, name, unit, type, valeur, display)
id = insertMeasure(lat, lon, alt, direction, xAcc, yAcc, zAcc, name, unit, type, valeur, display, diveId)
currentPosition = init()
endDive()
dive = init(date)
Alternative
si currentPos possède déja un GPS
add(currentPosition, measuresWaitingList)
currentPosition = init()
add(gps)

e


Alternative
si MAVLinkMessage
est un message IMU
imu = build(MAVLinkMessage)
IMU
Alternative
si currentPos
possède déja un IMU
add(
currentPosition,
measuresWaitingList)
currentPosition = init()
add(imu)
Alternative
si MAVLinkMessage
est un message Pressure
pressure = build(MAVLinkMessage)
Pressure
Alternative
si (currentPos possède une
Pressure) et (currentPos
possède un IMU ou un GPS)
add(
currentPosition,
measuresWaitingList)
currentPosition = init()
add(pressure)
Alternative
si MAVLinkMessage
est un message Temperature
temperature = build(MAVLinkMessage)
Temperature
measuresWaitingList
.add(temperature)
Analyste
startRecording()
startRecording(timestamp)
startRecording(timestamp)
startRecording(timestamp, diveId)
stopRecording()
stopRecording(timestamp)
stopRecording(timestamp)
stopRecording(timestamp, diveId)

3.1.5. Drone : Recalibrer les positions (sauvegardées) du drone

Identifiant : 5

Type : primaire

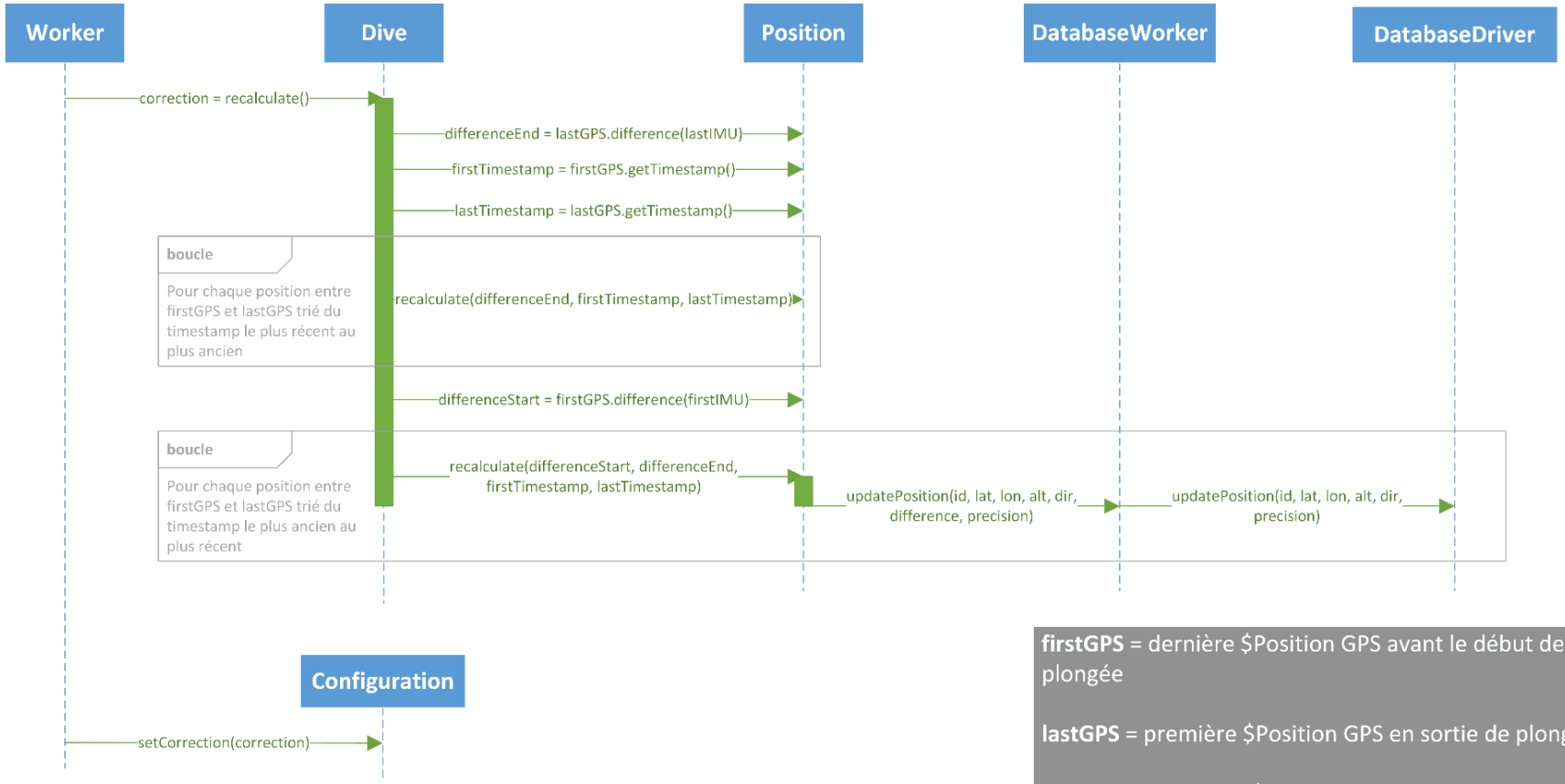
Acteurs :

- **Principal :** Pilote
- **Concerné :** Capteur GPS/GNSS/RTK et Base de données

Résumé : Action permettant au drone de corriger les données de localisation calculées au cours d'une plongée, en utilisant les données GPS acquises lors d'une immersion. Les positions calculées par les capteurs (autre que le GPS) sont recalibrées pour prendre en compte le courant de l'eau.

Préconditions : Le drone est à la surface. Il a obtenu un signal GPS/RTK fiable.

Post-condition : L'ensemble des positions de la plongée courante ont été recalibrées.



firstGPS = dernière $Position GPS avant le début de la plongée

lastGPS = première $Position GPS en sortie de plongée

firstIMU = première $Position après le début de plongée (peut être obtenu en même temps que lastGPS)

lastIMU = dernière $Position avant la sortie de plongée (peut être obtenu en même temps que lastGPS)

difference = difference de position entre la position calculée et la position originelle

precision = taux d'erreur engendré par la correction (correction forte = taux d'erreur fort = valeur élevée)

3.1.6. Drone : Calibrer le décalage / marge d'erreur

Identifiant : 6

Type : Secondaire

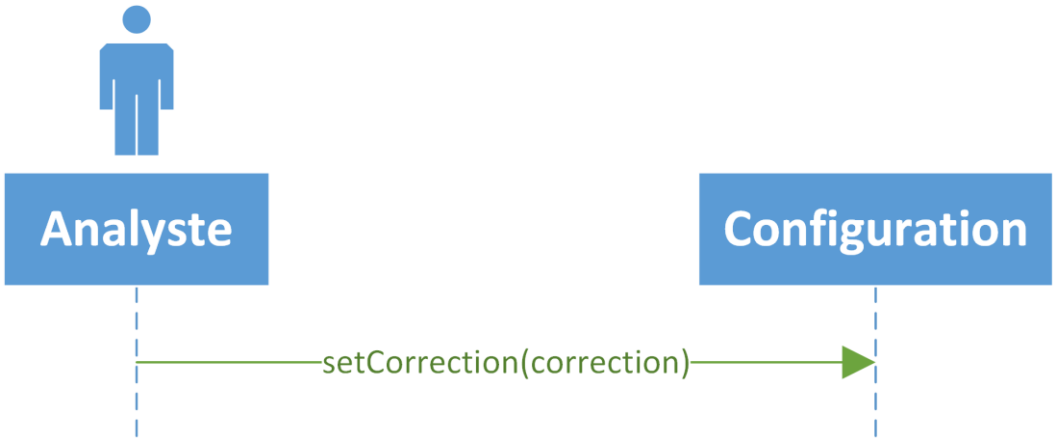
Acteurs :

- **Principal :** Analyste (Agent)
- **Concerné :** Capteur GPS/GNSS/RTK ; Accéléromètre/Gyroscope/Compas

Résumé : L'analyste accède à la console de configuration du décalage. Il définit les deux paramètres demandés, c'est-à-dire la puissance des courants verticaux et horizontaux, qui serviront à calculer la marge d'erreur et son évolution au cours du temps. Elle peut être calculée et actualisée lors de la correction de mesures réalisée (UC4).

Précondition : Le drone et la station de supervision doivent être connectés.

Post condition : L'algorithme de positionnement connait le décalage de position à prendre en compte.



3.1.7. Drone : Tester l'algorithme de positionnement

Identifiant : 7

Type : primaire

Acteurs :

- **Principal :** Testeur
- **Concerné :** Capteur GPS/GNSS/RTK, Base de données

Résumé : Action permettant de vérifier que l'algorithme de positionnement correspond au trajet effectué.

Préconditions : Pour vérifier l'algorithme, nous serons obligés d'effectuer ce test hors de l'eau pour conserver le signal GPS tout le long du test. Le drone doit donc obtenir un signal GPS/RTK fiable (précision à 5cm selon les caractéristiques du GPS/RTK Emlid – composant du drone).

Post-condition : Des mesures sont disponibles pour pouvoir réaliser des simulations et/ou des essais d'algorithme de positionnement.

Fichier de sortie (CSV) :

Champ	Description
Timestamp	Identifiant dans le temps. Dans le cadre de la simulation, le champ est incrémenté de 1 à chaque ligne. (Identique au fichier d'entrée)
GPS longitude	Position longitudinale fournie par le GPS (en surface – début et fin de plongée)
GPS latitude	Position transversale fournie par le GPS (en surface – début et fin de plongée)
GPS altitude	Altitude fournie par le GPS (en surface – début et fin de plongée)
Accélération X	Accélération (translation) sur l'axe X (mg - accélération)
Accélération Y	Accélération (translation) sur l'axe Y (mg - accélération)
Accélération Z	Accélération (translation) sur l'axe Z (mg - accélération)
Rotation X	Vitesse de rotation sur l'axe X (millirad /sec)
Rotation Y	Vitesse de rotation sur l'axe Y (millirad /sec)
Rotation Z	Vitesse de rotation sur l'axe Z (millirad /sec)
Cap X	Orientation magnétique du capteur (sens du drone) sur l'axe X par rapport au Nord magnétique (milli tesla).
Cap Y	Orientation magnétique du capteur (sens du drone) sur l'axe Y par rapport au Nord magnétique (milli tesla).
Cap Z	Orientation magnétique du capteur (sens du drone) sur l'axe Z par rapport au Nord magnétique (milli tesla).
Pression	Pression Absolue mesurée (hectopascal)
Mesure 1 (température)	Température mesurée en degré Celsius (identique au fichier d'entrée)
Latitude	Position transversale du drone par rapport au référentiel WSG84 (identique au fichier d'entrée)
Longitude	Position longitudinale du drone par rapport au référentiel WSG84 (Identique au fichier d'entrée)
Altitude	Altitude du drone en mètre par rapport au niveau de la mer (Identique au fichier d'entrée)
Cap réel	Cap magnétique du drone (orientation) (identique au fichier d'entrée)

Tableau 4 - UC 7 : fichier de sortie

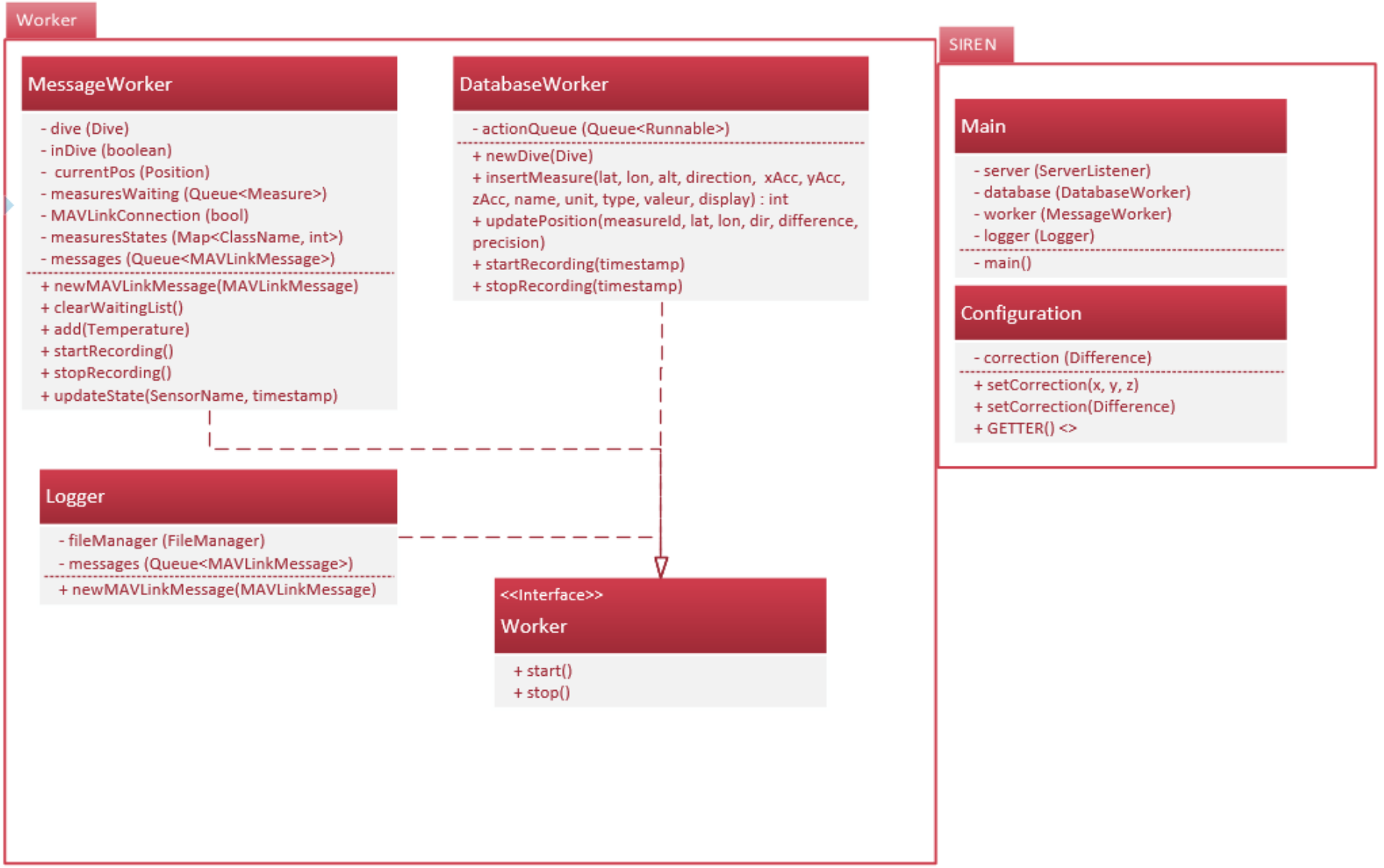
3.2. Diagrammes de classe

L'application devant être développée possède deux points d'entrées :

- La simulation (package Virtuatizer)
- L'application utilisateur (package SIREN).

L'application a été découpée en différents package. Les packages principaux sont :

- Virtualizer : point d'entrée de la simulation, contient également le coeur du simulateur
- SIREN : point d'entrée de l'application utilisateur
- Worker : contient les classes de traitement relatives au coeur de SIREN
- Database : gestionnaire de toute la partie stockage en base
- File : gestionnaire de toute la partie stockage en fichier
- Network : gestionnaire de la partie réseau
- MAVLink : représentation des paquets MAVLink
- Geo : contient les différentes représentations des coordonnées géographiques
- Sensor : contient la représentation des différents capteurs
- Drone : contient la librairie cœur du positionnement et des mesures réalisées par le drone.


Worker
MessageWorker
- dive (Dive)
- inDive (boolean)
- currentPos (Position)
- measuresWaiting (Queue<Measure>)
- MAVLinkConnection (bool)
- measuresStates (Map<ClassName, int>)
- messages (Queue<MAVLinkMessage>)
+ newMAVLinkMessage(MAVLinkMessage)
+ clearWaitingList()
+ add(Temperature)
+ startRecording()
+ stopRecording()
+ updateState(SensorName, timestamp)
DatabaseWorker
- actionQueue (Queue<Runnable>)
+ newDive(Dive)
+ insertMeasure(lat, lon, alt, direction, xAcc, yAcc, zAcc, name, unit, type, valeur, display) : int
+ updatePosition(measureId, lat, lon, dir, difference, precision)
+ startRecording(timestamp)
+ stopRecording(timestamp)
Logger
- fileManager (FileManager)
- messages (Queue<MAVLinkMessage>)
+ newMAVLinkMessage(MAVLinkMessage)
<<Interface>>
Worker
+ start()
+ stop()
SIREN
Main
- server (ServerListener)
- database (DatabaseWorker)
- worker (MessageWorker)
- logger (Logger)
- main()
Configuration
- correction (Difference)
+ setCorrection(x, y, z)
+ setCorrection(Difference)
+ GETTER() <>

3.2.1. SIREN

Ce package est le cœur de l'application. C'est dans celui-ci que se trouve le point d'entrée de l'application pour l'utilisateur Analyste et/ou Pilote.

La classe Main est le point d'entrée de l'application. C'est cette classe qui a la responsabilité de l'interaction avec l'utilisateur. Cette classe est également le centre des workers. Chaque classe de traitement (worker) est gérée par le Main. C'est donc dans le Main que le lancement et l'arrêt des worker est géré.

Le package contient également une classe ayant la responsabilité de la Configuration. La classe Configuration se charge du stockage de la correction (une différence sur les axes x, y et z).

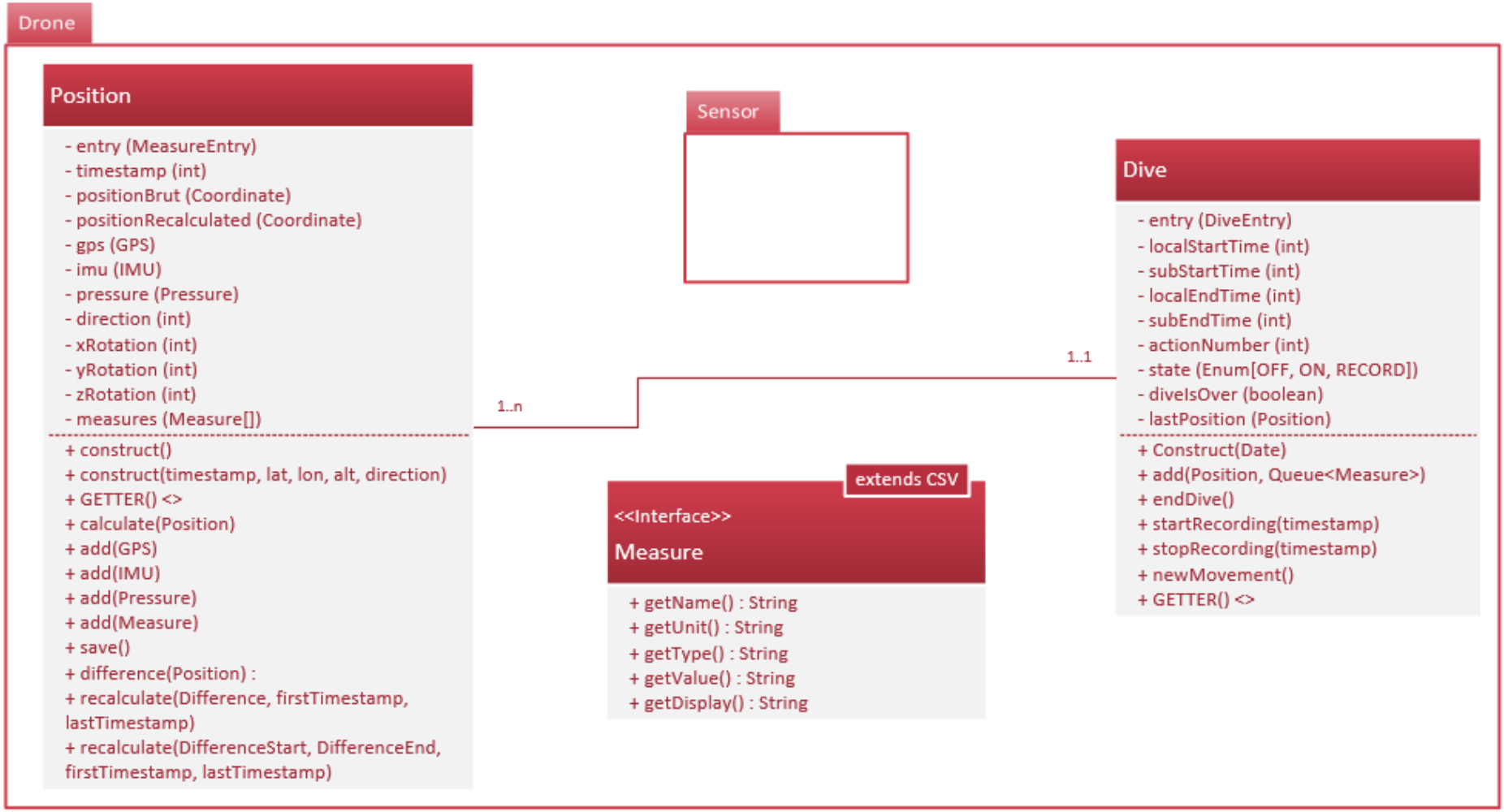
3.2.2. Worker

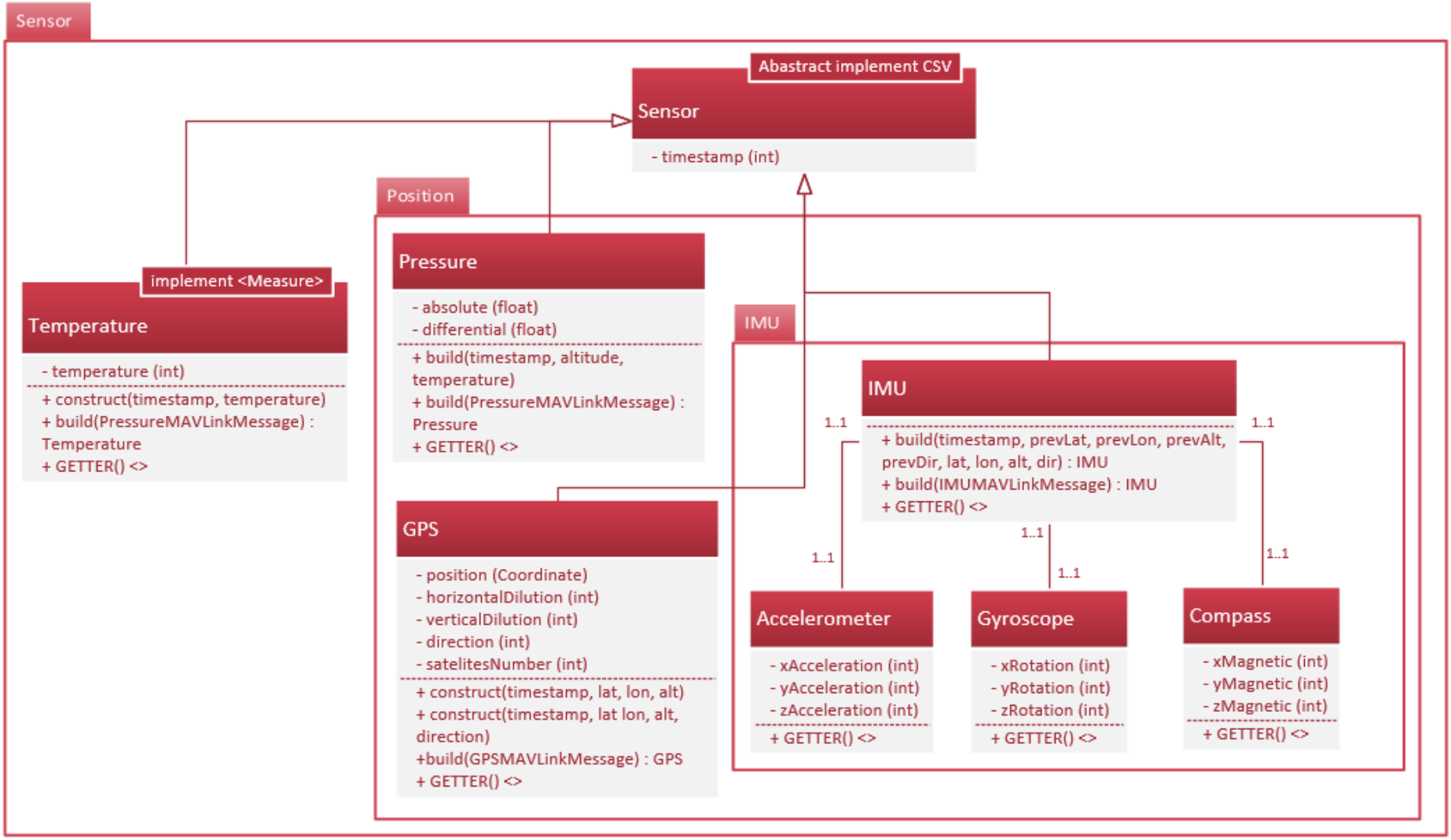
Le package Worker stocke toutes les classes ayant des traitements lourds à réaliser. Ces traitements lourds sont réalisés dans des threads. Ces workers tournent en tâches de fond.

Le worker principal est le MessageWorker. Cette classe se charge de calculer les différentes positions du drone. Ce worker interagit avec toute la représentation d'une plongée. Toutes les données relatives à une plongée sont stockées en mémoire. Ces données sont uniquement connues du Worker, d'où son rôle : calculer les positions. De plus, un stockage des messages MAVLink est réalisé dans ce Worker. En effet, le temps de traitement des messages peut être plus long que le temps d'arrivée.

Le second Worker est le DatabaseWorker. Ce worker a la responsabilité des traitements réalisés en base de données. Comme ces traitements peuvent ralentir le MessageWorker, un système de traitement en différé est réalisé. Lorsque le worker doit insérer (ou mettre à jour) plusieurs entrées, le DatabaseWorker se "réveille" alors pour réaliser plusieurs opérations.

Une requête SQL est traitée en 3 étapes : verrouillage de la table ; réalisation de l'action ; validation des modifications. Dans notre cas, nous allons garder ce fonctionnement, mais nous allons réaliser différentes actions sur la table entre le verrouillage et la validation. Cela permet d'alléger les processus de la base de données, et ainsi d'optimiser les temps de traitement.


Drone
Position
- entry (MeasureEntry)
- timestamp (int)
- positionBrut (Coordinate)
- positionRecalculated (Coordinate)
- gps (GPS)
- imu (IMU)
- pressure (Pressure)
- direction (int)
- xRotation (int)
- yRotation (int)
- zRotation (int)
- measures (Measure[])
+ construct()
+ construct(timestamp, lat, lon, alt, direction)
+ GETTER() <>
+ calculate(Position)
+ add(GPS)
+ add(IMU)
+ add(Pressure)
+ add(Measure)
+ save()
+ difference(Position) :
+ recalculate(Difference, firstTimestamp, lastTimestamp)
+ recalculate(DifferenceStart, DifferenceEnd, firstTimestamp, lastTimestamp)
Sensor
1..n
1..1
Dive
- entry (DiveEntry)
- localStartTime (int)
- subStartTime (int)
- localEndTime (int)
- subEndTime (int)
- actionNumber (int)
- state (Enum[OFF, ON, RECORD])
- diveIsOver (boolean)
- lastPosition (Position)
+ Construct(Date)
+ add(Position, Queue<Measure>)
+ endDive()
+ startRecording(timestamp)
+ stopRecording(timestamp)
+ newMovement()
+ GETTER() <>
extends CSV
<<Interface>>
Measure
+ getName() : String
+ getUnit() : String
+ getType() : String
+ getValue() : String
+ getDisplay() : String


Sensor
Abastract implement CSV
Sensor
- timestamp (int)
Position
Pressure
- absolute (float)
- differential (float)
+ build(timestamp, altitude, temperature)
+ build(PressureMAVLinkMessage) : Pressure
+ GETTER() <>
implement <Measure>
Temperature
- temperature (int)
+ construct(timestamp, temperature)
+ build(PressureMAVLinkMessage) : Temperature
+ GETTER() <>
IMU
IMU
+ build(timestamp, prevLat, prevLon, prevAlt, prevDir, lat, lon, alt, dir) : IMU
+ build(IMUMAVLinkMessage) : IMU
+ GETTER() <>
1..1
1..1
1..1
1..1
1..1
1..1
GPS
- position (Coordinate)
- horizontalDilution (int)
- verticalDilution (int)
- direction (int)
- satelitesNumber (int)
+ construct(timestamp, lat, lon, alt)
+ construct(timestamp, lat lon, alt, direction)
+build(GPSMAVLinkMessage) : GPS
+ GETTER() <>
Accelerometer
- xAcceleration (int)
- yAcceleration (int)
- zAcceleration (int)
+ GETTER() <>
Gyroscope
- xRotation (int)
- yRotation (int)
- zRotation (int)
+ GETTER() <>
Compass
- xMagnetic (int)
- yMagnetic (int)
- zMagnetic (int)
+ GETTER() <>

3.2.3. Drone

Le package drone contient les classes relatives à l'utilisation du drone. Nous avons dans ce package une interface "Measure" qui a pour but de récupérer les données mesurées pour les traiter. Nous avons également dans ce package la classe "Position" qui caractérise la position courante du drone. Elle contient également les objets associés à chaque capteur (GPS, IMU…). La seconde classe de ce package est la classe "Dive". Cette classe possède entre une et plusieurs "Position". Cette classe "Dive" concerne une plongée unique et est caractérisée par différentes informations, comme par exemple la durée de plongée.

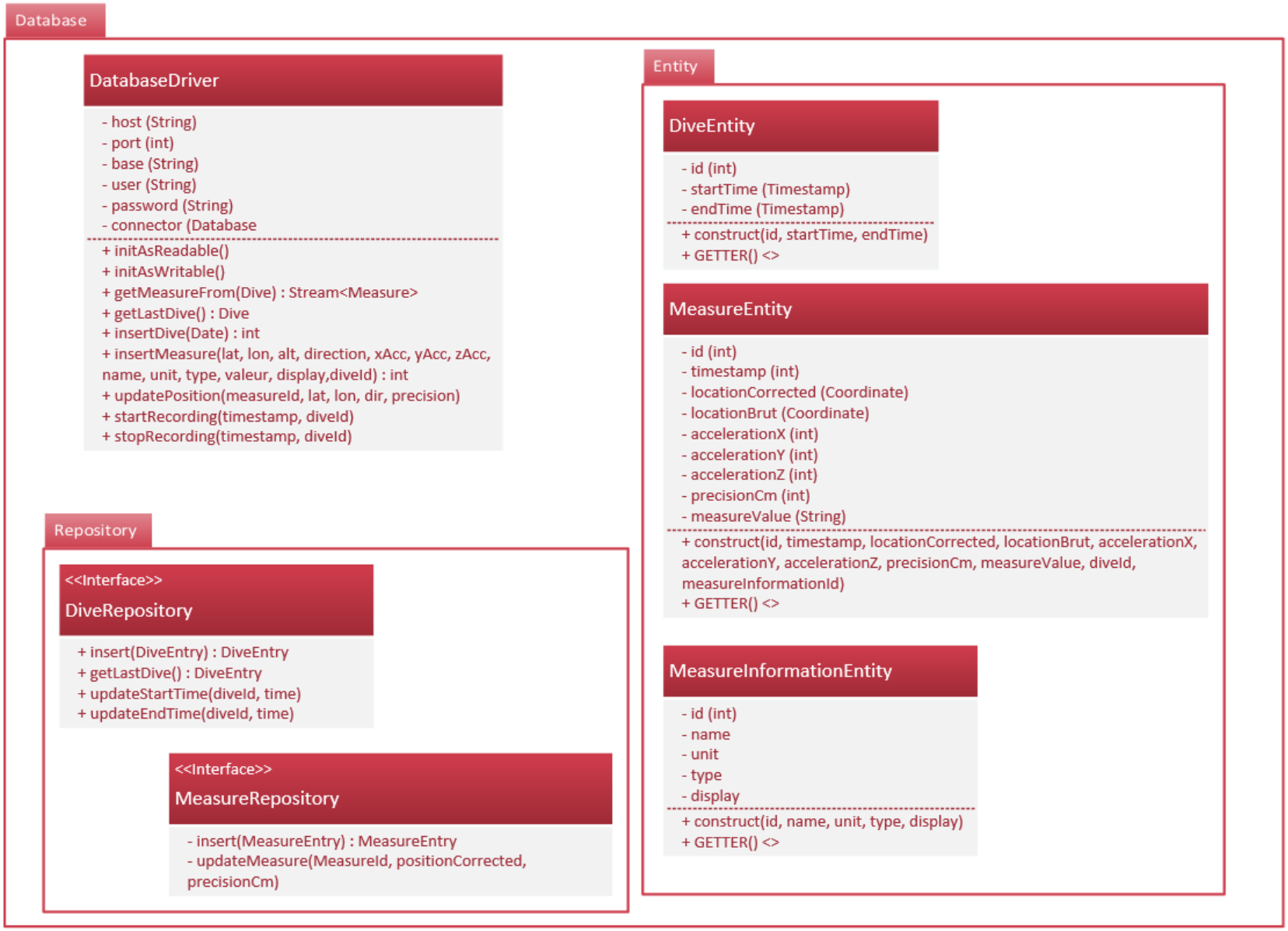
Il contient également le package Sensor détaillé dans la partie suivante.

3.2.4. Sensor

La package Sensor contient toutes les données relatives aux capteurs intégrés au drone.

Ce package est composé de :

- Une classe Temperature qui permet de récupérer les données relatives au capteur de température
- Un sous-package Position contenant :
  - Une classe Pressure permettant de traiter les données du capteur de pression pour définir la profondeur actuelle du drone
  - Un sous-package IMU contenant une classe IMU regroupant toutes les données de l'accéléromètre, du gyroscope et du compas
  - Une classe GPS décrivant la position actuelle du drone transmise par le capteur GPS/RTK

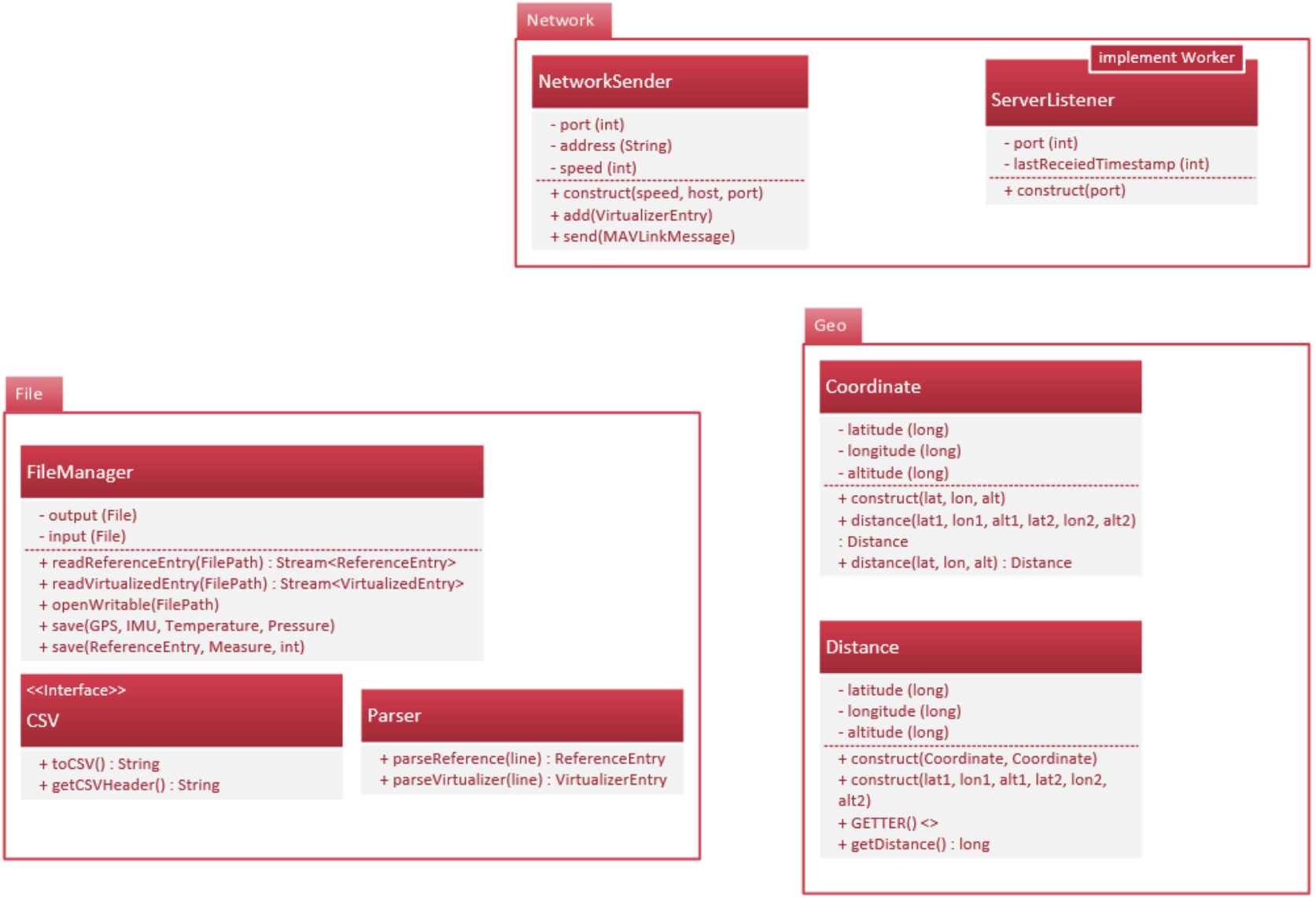

Database
DatabaseDriver
- host (String)
- port (int)
- base (String)
- user (String)
- password (String)
- connector (Database
+ initAsReadable()
+ initAsWritable()
+ getMeasureFrom(Dive) : Stream<Measure>
+ getLastDive() : Dive
+ insertDive(Date) : int
+ insertMeasure(lat, lon, alt, direction, xAcc, yAcc, zAcc, name, unit, type, valeur, display,diveId) : int
+ updatePosition(measureId, lat, lon, dir, precision)
+ startRecording(timestamp, diveId)
+ stopRecording(timestamp, diveId)
Repository
<<Interface>>
DiveRepository
+ insert(DiveEntry) : DiveEntry
+ getLastDive() : DiveEntry
+ updateStartTime(diveId, time)
+ updateEndTime(diveId, time)
<<Interface>>
MeasureRepository
- insert(MeasureEntry) : MeasureEntry
- updateMeasure(MeasureId, positionCorrected, precisionCm)
Entity
DiveEntity
- id (int)
- startTime (Timestamp)
- endTime (Timestamp)
+ construct(id, startTime, endTime)
+ GETTER() <>
MeasureEntity
- id (int)
- timestamp (int)
- locationCorrected (Coordinate)
- locationBrut (Coordinate)
- accelerationX (int)
- accelerationY (int)
- accelerationZ (int)
- precisionCm (int)
- measureValue (String)
+ construct(id, timestamp, locationCorrected, locationBrut, accelerationX, accelerationY, accelerationZ, precisionCm, measureValue, diveId, measureInformationId)
+ GETTER() <>
MeasureInformationEntity
- id (int)
- name
- unit
- type
- display
+ construct(id, name, unit, type, display)
+ GETTER() <>

3.2.5. Database

Le package Database représente la connexion et les accès en lecture et en écriture aux bases de données. Ce package contient tout d'abord une classe DatabaseDriver permettant d'établir la connexion à la base de données grâce à diverses informations comme l'host, le port ou encore le nom de l'utilisateur.

Ce package contient également deux sous-packages :

- Repository : contenant deux interfaces DiveRepository et MeasureRepository permettant respectivement d'implémenter les requêtes à la base de données concernant la plongée en cours et les mesures réalisées.
- Entity : contenant deux classes DiveEntity et MeasureEntity associées respectivement à DiveRepository et MeasureRepository. Les deux classes contiennent les données à écrire ou à lire dans la base de données.


Network
NetworkSender
- port (int)
- address (String)
- speed (int)
+ construct(speed, host, port)
+ add(VirtualizerEntry)
+ send(MAVLinkMessage)
implement Worker
ServerListener
- port (int)
- lastReceiedTimestamp (int)
+ construct(port)
Geo
Coordinate
- latitude (long)
- longitude (long)
- altitude (long)
+ construct(lat, lon, alt)
+ distance(lat1, lon1, alt1, lat2, lon2, alt2) : Distance
+ distance(lat, lon, alt) : Distance
Distance
- latitude (long)
- longitude (long)
- altitude (long)
+ construct(Coordinate, Coordinate)
+ construct(lat1, lon1, alt1, lat2, lon2, alt2)
+ GETTER() <>
+ getDistance() : long
File
FileManager
- output (File)
- input (File)
+ readReferenceEntry(FilePath) : Stream<ReferenceEntry>
+ readVirtualizedEntry(FilePath) : Stream<VirtualizedEntry>
+ openWritable(FilePath)
+ save(GPS, IMU, Temperature, Pressure)
+ save(ReferenceEntry, Measure, int)
<<Interface>>
CSV
+ toCSV() : String
+ getCSVHeader() : String
Parser
+ parseReference(line) : ReferenceEntry
+ parseVirtualizer(line) : VirtualizerEntry

3.2.6. File

Le package File représente toute la gestion des fichiers.

La classe en charge des entrées/sorties sur le disque est FileManager. Celle-ci se charge de toute la partie lecture/écriture des données. Chaque lecture est formatée afin que la donnée puisse être directement utilisable.

Le formatage des données est réalisé par la classe Parser. Celle-ci se charge de produire des Entry (en provenance du package Virtualizer, ce dernier étant décrit plus loin dans le document). Le découpage est réalisé sur des fichiers CSV.

Les données devant être sauvegardées dans les fichiers (seules les données pour la simulation sont concernées) doivent implémenter l'interface CSV. Cette interface décrit uniquement les méthodes de sauvegarde d'un objet en CSV (méthode toCSV). Celle-ci permet également de connaitre les en-têtes relatifs à ces données (getCSVHeader).

3.2.7. Network

Le package Network incorpore le nécessaire pour la gestion des fonctionnalités réseau. Il est composé de deux classes.

La première est la classe Network Sender. Elle permet d'envoyer des messages MAVLink à une adresse et un port définis.

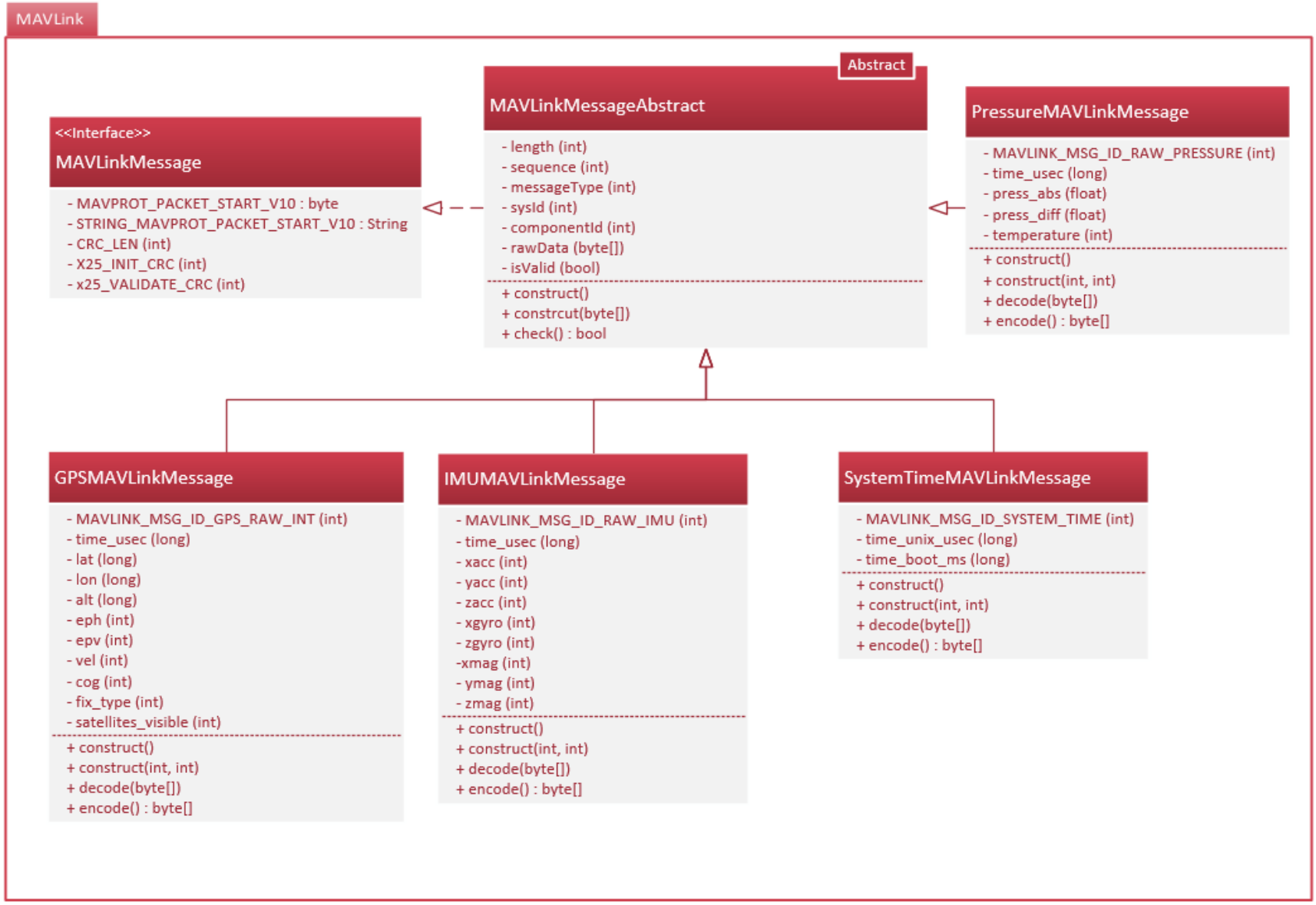
La seconde est la classe ServerListener. Elle est composée d'un thread en écoute sur un port défini dans l'attente d'un message MAVLink.

3.2.8. Geo

Ce package va être utilisé pour définir et représenter des coordonnées et distances.

Pour cela nous utilisons deux classes, Coordinates et Distance. La première classe représente un point situé à une coordonnée précise. Celui-là sera représenté par une valeur en latitude, longitude et altitude. Sur ces coordonnées, il sera possible d'appliquer une fonction, la distance. Cette fonction, aussi représentée dans la classe "Distance", permet de prendre deux coordonnées et de calculer la distance les séparant.

Cette méthode sera particulièrement utile lors du calcul de la correction de la position. En effet, il sera ainsi facile d'obtenir le delta entre les valeurs estimées et les valeurs réelles issues des plongées.

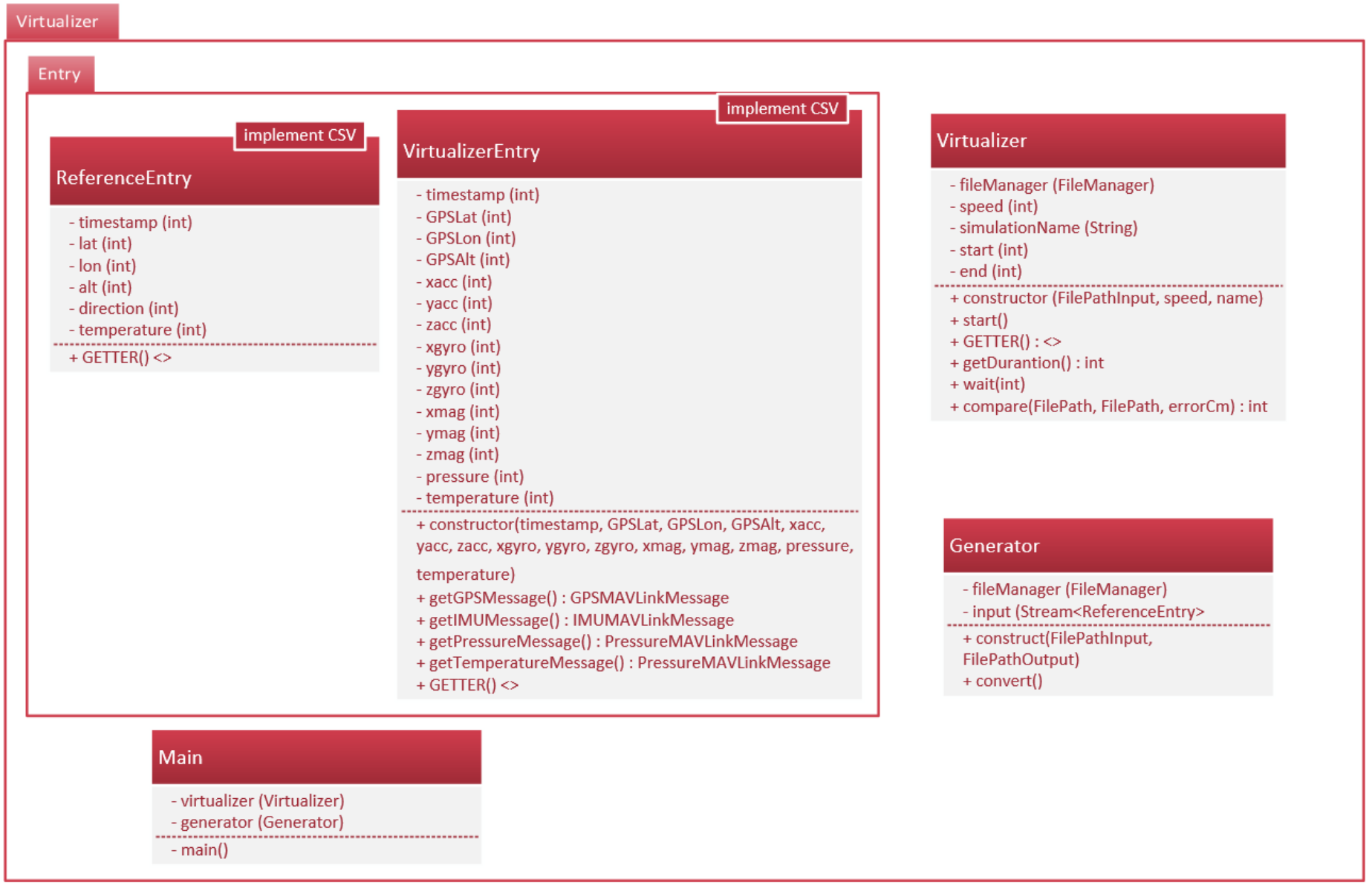

MAVLink
Abstract
MAVLinkMessageAbstract
- length (int)
- sequence (int)
- messageType (int)
- sysId (int)
- componentId (int)
- rawData (byte[])
- isValid (bool)
+ construct()
+ constrcut(byte[])
+ check() : bool
<<Interface>>
MAVLinkMessage
- MAVPROT_PACKET_START_V10 : byte
- STRING_MAVPROT_PACKET_START_V10 : String
- CRC_LEN (int)
- X25_INIT_CRC (int)
- x25_VALIDATE_CRC (int)
PressureMAVLinkMessage
- MAVLINK_MSG_ID_RAW_PRESSURE (int)
- time_usec (long)
- press_abs (float)
- press_diff (float)
- temperature (int)
+ construct()
+ construct(int, int)
+ decode(byte[])
+ encode() : byte[]
GPSMAVLinkMessage
- MAVLINK_MSG_ID_GPS_RAW_INT (int)
- time_usec (long)
- lat (long)
- lon (long)
- alt (long)
- eph (int)
- epv (int)
- vel (int)
- cog (int)
- fix_type (int)
- satellites_visible (int)
+ construct()
+ construct(int, int)
+ decode(byte[])
+ encode() : byte[]
IMUMAVLinkMessage
- MAVLINK_MSG_ID_RAW_IMU (int)
- time_usec (long)
- xacc (int)
- yacc (int)
- zacc (int)
- xgyro (int)
- zgyro (int)
-xmag (int)
- ymag (int)
- zmag (int)
+ construct()
+ construct(int, int)
+ decode(byte[])
+ encode() : byte[]
SystemTimeMAVLinkMessage
- MAVLINK_MSG_ID_SYSTEM_TIME (int)
- time_unix_usec (long)
- time_boot_ms (long)
+ construct()
+ construct(int, int)
+ decode(byte[])
+ encode() : byte[]

3.2.9. MAVLink

Le package MAVLink est une représentation de la librairie MAVLinkJava. Le package contient uniquement les classes relatives aux objets utilisé dans le projet. Les classes concernées sont les suivantes :

- PressureMAVLInkMessage (pour la pression et la température)
- SystemTimeMAVLinkMessage (pour la synchronisation du temps)
- IMUMAVLinkMessage (pour l'IMU)
- GPSMAVLinkMessage (pour le GPS).

Ces classes implémentent l'interface MAVLinkMessage. Tous les messages peuvent donc être manipulés par l'interface parente.


Virtualizer
Entry
implement CSV
ReferenceEntry
- timestamp (int)
- lat (int)
- lon (int)
- alt (int)
- direction (int)
- temperature (int)
+ GETTER() <>
implement CSV
VirtualizerEntry
- timestamp (int)
- GPSLat (int)
- GPSLon (int)
- GPSAlt (int)
- xacc (int)
- yacc (int)
- zacc (int)
- xgyro (int)
- ygyro (int)
- zgyro (int)
- xmag (int)
- ymag (int)
- zmag (int)
- pressure (int)
- temperature (int)
+ constructor(timestamp, GPSLat, GPSLon, GPSAlt, xacc, yacc, zacc, xgyro, ygyro, zgyro, xmag, ymag, zmag, pressure, temperature)
+ getGPSMessage() : GPSMAVLinkMessage
+ getIMUMessage() : IMUMAVLinkMessage
+ getPressureMessage() : PressureMAVLinkMessage
+ getTemperatureMessage() : PressureMAVLinkMessage
+ GETTER() <>
Virtualizer
- fileManager (FileManager)
- speed (int)
- simulationName (String)
- start (int)
- end (int)
+ constructor (FilePathInput, speed, name)
+ start()
+ GETTER() : <>
+ getDurantion() : int
+ wait(int)
+ compare(FilePath, FilePath, errorCm) : int
Generator
- fileManager (FileManager)
- input (Stream<ReferenceEntry>
+ construct(FilePathInput, FilePathOutput)
+ convert()
Main
- virtualizer (Virtualizer)
- generator (Generator)
- main()

3.2.10. Virtualizer

Le package Virtualizer est le coeur du simulateur. C'est dans ce package que sont réalisées les différentes actions relatives à la simulation :

- Génération d'un fichier de simulation (via la classe Generator)
- Exécution d'une simulation (Virtualizer).

Ces différentes classes sont instanciées dans le Main. Cette dernière classe est le point d'entrée en ligne de commande du simulateur. C'est dans ce Main, que le Testeur va choisir s'il souhaite générer une simulation, ou s'il souhaite en exécuter une.

La simulation se base sur la représentation de certaines données. Cette représentation est réalisée dans le package Entry. Ce package sert uniquement à la représentation de données stockées. Ces Entry (ReferenceEntry et VirtualizerEntry) ne sont utilisés que dans le cadre de la simulation.

La classe ReferenceEntry représente les positions de références. Cette classe est utilisée pour la génération d'une simulation et pour réaliser la comparaison des positions. En effet, les coordonnées stockées dans cette classe sont la référence. Lorsque SIREN a calculé les positions (via les capteurs), une comparaison à lieu entre les positions calculées et celles stockées dans la ReferenceEntry.

La classe VirtualizerEntry représente les données que devront envoyer les capteurs du drone. Les données présentes et stockées dans la classe proviennent de la génération de la simulation.

3.3. Gestion de la mémoire

Stocker toutes les positions en mémoire pour optimiser le temps de traitement impose une charge en mémoire. Pour ce faire, l'équipe s'est assurée que le poste client pourra supporter l'application.

Classe	IMU	GPS	Coordinate	Pressure	Temperature	Position
Taille (Octet)	96	40	64	24	16	40
Fréquence d'arrivée (Hz)	400	50	400	3,3	3,3	400
Total généré par seconde (Octet)	38 400	2 000	25 600	72	48	16 000

Durée	Total (Ko)	Total (Mo
1	9 506	9
2	14 201	14
3	18 895	18
4	23 590	23
5	28 284	28
6	32 979	32
7	37 673	37
8	42 368	41
9	47 063	46
10	51 757	51
11	56 452	55
12	61 146	60
13	65 841	64
14	70 535	69
15	75 230	73
16	79 924	78
17	84 619	83
18	89 313	87
19	94 008	92
20	98 702	96

Figure 4 - Tableau d'utilisation de la mémoire

Le GPS n'est stocké que lorsque le drone est en surface. Pour l'analyse, nous partons du principe que le drone reste en surface (capte donc un signal GPS) pendant 1 minute (60 secondes). Une charge de 120 000 octets (120 Ko) est donc présente dans l'analyse ci-après.

Comme le montre le tableau ci-contre, même avec 20 minutes de plongée, il n'y a que 96 Mo de données. Comme un traitement ne peut que doubler notre volume de données dans le pire des cas, nous aurions moins de 200 Mo de données par plongée, ce qui est loin de la capacité de la RAM qui se situe souvent aux alentours de 8 Go.
Il nous est donc possible d'exploiter celle-ci pour réaliser nos calculs et conserver nos valeurs, d'où une deuxième raison d'utiliser la mémoire plutôt que la base de données.

Pour contenir ces informations, nous avons notre classe Dive, qui représente une plongée. Cette plongée correspond à une série de mesures entre la perte d'un signal GPS et la ré-acquisition de celui-ci.

La classe a la responsabilité de stocker les couples mesures/positions suivants :

- Pré-enregistrement : Toutes les mesures obtenues mais inutiles pour l'analyste
- Enregistrement : Tous les éléments que l'analyste souhaite garder
- Post-enregistrement : Toutes les informations récupérées après l'enregistrement.

Ces différents enregistrements sont nécessaires au calcul de la position. En effet, seule la classe Dive connait l'historique des positions. C'est donc de sa responsabilité de fournir à la classe Position toutes les informations nécessaires afin que celle-ci puisse calculer la position courante du drone.

3.4. Base de données

La base de données à un simple objectif : stocker les positions et les mesures. Malgré le rôle en apparence simple, ce stockage est un élément clé du projet. En effet, cet élément est la passerelle entre SIREN (calcule des positions) et le module QGIS (affichage). SIREN a la responsabilité d'insérer les données. Le module QGIS, quant à lui, se charge de récupérer les données.

3.4.1. Structure

Le choix de l'architecture de la base de données a nécessité la réalisation de différents tests. Mais avant de mettre en avant les différentes possibilités, il est important de comprendre la contrainte majeure présente : la vitesse d'insertion. Dans le pire des cas (c'est-à-dire lorsqu'aucune perte de paquet n'a lieu) 400 insertions (sur le drone, la fréquence de captures et d'envoie des données de l'IMU est de 400Hz) de position ont lieu. Il est donc important que le temps d'insertion en base de données soit le plus minime possible pour ne pas ralentir les temps de traitement de SIREN. Si l'application accumule du retard, c'est l'affiche (via le plugin QGIS) qui subit : le temps réel (1 seconde de latence) peut rapidement disparaitre (la latence peut rapidement tendre vers 30 secondes).

Nous avons donc essayé deux architectures : une architecture « normalisée » et une architecture « plate ». L'architecture normalisée consiste à mettre en place une base de données respectant les normes ACID (Atomicity, Consistency, Isolation, Durability – Atomicité, Consistance, Isolation et Durabilité). L'architecture « plate » reprend le principe des bases de données NoSQL. On met en place des doublons pour éviter des jointures inutiles et/ou gourmandes (en temps et/ou en performance). Ces deuxièmes architectures proposent donc de rompre la Consistance de la base de données.

La base de données « normalisée » est modélisée de la manière suivante :

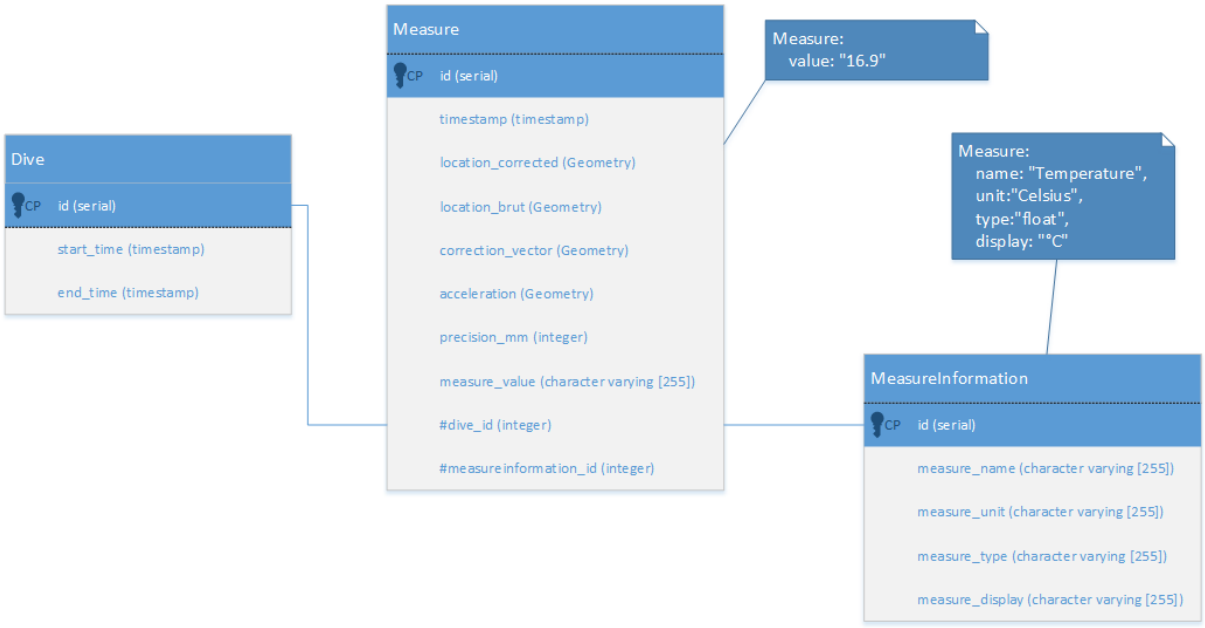


Figure 5 - Base de Données « normalisée »

La base de données « plate » est modélisée de la manière suivante :

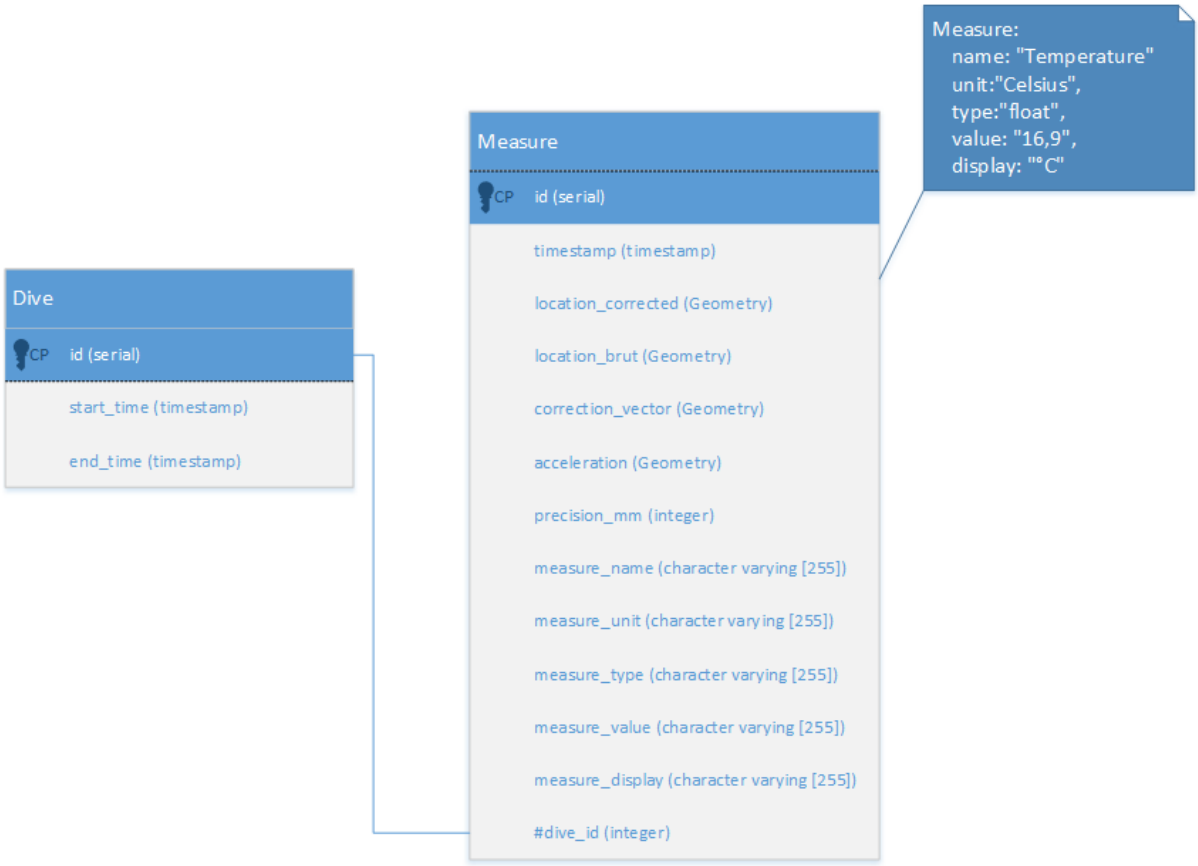


Figure 6 - Base de Données « plate »

La base de données « normalisée » intègre la table « MeasureInformation » (cf. modélisation normalisée). Les données relatives à cette table sont donc présentes dans chaque entrée de la table correspondant à un même type de mesure.

Afin de réaliser une simulation la plus proche de la réalité, nous avons développé un simulateur pour réaliser les mesures. Le simulateur est découpé en deux threads :

- Un thread représente le traitement logique de SIREN. Cette représentation est réalisée au travers de l'instanciation d'objet (type Object) à une fréquence de 400Hz. Celui-ci est ensuite stocké dans un tableau qui est réinitialisé toutes les 2 secondes.
- Le second thread insert des entrées relatives à la table Measure (de la représentation « plate » et « normalisée »).

Les tests ont été réalisés sur différentes quantités d'insertions. Chaque test a été réalisé 10 fois. Une extraction de ses 10 itérations a été réalisée : le temps moyen, et les temps extrêmes (minimum et maximum).

Les résultats sont les suivants :

Inserts	Normalized (avg)	Flat (avg)	Diff F-N (avg)	Normalized (unit)	Flat (unit)	Diff F-N (unit)
10	30	19	**-11**	3,00	1,90	**- 1,10**
50	97	50	**-47**	1,94	1,00	**- 0,94**
100	146	93	**-53**	1,46	0,93	**- 0,53**
200	223	175	**-48**	1,12	0,88	**- 0,24**
300	294	262	**-32**	0,98	0,87	**- 0,11**
400	359	337	**-22**	0,90	0,84	**- 0,05**
500	473	418	**-55**	0,95	0,84	**- 0,11**
600	521	499	**-22**	0,87	0,83	**- 0,04**
700	866	581	**-285**	1,24	0,83	**- 0,41**
800	1189	659	**-530**	1,49	0,82	**- 0,66**
900	756	745	**-11**	0,84	0,83	**- 0,01**
1000	813	820	**7**	0,81	0,82	**0,01**
1500	1279	1273	**-6**	0,85	0,85	**- 0,00**
2000	1616	1672	**56**	0,81	0,84	**0,03**
3000	2507	2524	**17**	0,84	0,84	**0,01**
4000	3383	3412	**29**	0,85	0,85	**0,01**
5000	4034	4115	**81**	0,81	0,82	**0,02**
			Résumé	**1,16**	**0,92**	**- 0,24**

Figure 7 - Résultat des tests

Les temps présents sont exprimés en milliseconde (rappel : 1s = 1 000ms).

Comme on peut l'observer la structure « plate » (Flat) est moyennement plus avantageuse (de 0,24 ms). Cependant, cette avantage, inférieur à une milliseconde est négligeable au vu du cout que pourrait engendrer un changement de données présents sur des millions de lignes.

De plus, cette avantage (de la structure « plate » sur la structure « normalisée ») est tellement infime qu'il est parfaitement inutile de briser les normes des bases de données. Par conséquent, **le projet va mettre en place la base de données normalisée présentée précédemment.**

La base de données est donc découpée en trois tables :

- Dive
- Measure
- MeasureInformation.

La première table correspond à la représentation d'une plongée (Dive). Cette plongée permet de différencier les différentes mesures entre elles. La table est caractérisée de la manière suivante :

- Id : identifiant unique de la plongée (utile pour le lien Measure-Dive)
- Date : jour de la réalisation de la plongée
- Start_time : heure de début de la plongée
- End_time : heure de fin de la plongée.

La seconde table, la plus importante, est chargée de représenter les mesures. La table Measure stocke toutes les mesures avec leur position. Dans cette table, seule les positions avec mesures sont stockées. L'affichage des positions sans mesure sur QGIS est inutile. La table est caractérisée par :

- Id : identifiant unique de la mesure
- Location_corrected : position géométrique (type Geometry – x,y,z) représentant la position de la mesure mais recalculée en fonction de l'algorithme de correction (cf. 3.2.5 Recalcul des positions).
- Location_brut : position géométrique (type Geometry – x,y,z) représentant la position de la mesure brute (sans recalcul).
- Correction_vector : vecteur de correction de la position brute pour obtenir la position corrigée (type Geometry – x,y,z).
- Acceleration_x, acceleration_y et acceleration_z : informations représentant l'accélération du drone (translation) sur les axes x, y et z.
- Precision_cm : incertitude pour la position (en cm)
- Measure_value : valeur de la mesure
- #dive_id : clé étrangère représentant la plongée (Dive) associée à la mesure courante.

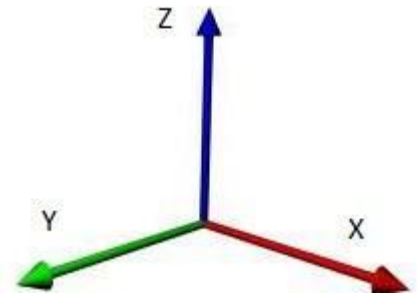


Figure 8 - Représentation des axes des capteurs

Description des axes pour la position relative au drone (pour les capteurs) :

- X : Direction du drone
- Y : Translation latérale du drone
- Z : Translation verticale du drone

La dernière table consiste à décrire une mesure. Cette description permet de réaliser une représentation de ladite mesure dans l'affichage QGIS. La table est composée des champs suivants :

- Measure_name : nom de la mesure
- Mesasure_unit : unité de la mesure
- Measure_display : texte utilisé pour l'affichage

3.4.2. Notification

Pour éviter de devoir interroger régulièrement la base de données pour connaitre les nouvelles mesures, le système de notification (dans PostgreSQL) est utilisé. Ce système a l'avantage d'être intégré de base dans le moteur de base de données, et de ne consommer que de faibles performances (https://www.postgresql.org/docs/9.3/static/sql-notify.html).

En revanche, le module QGIS doit régulièrement requêter la base de données pour être informé de la présence ou non de notifications. L'équipe projet a jugé que toutes les secondes est une fréquence adaptée au besoin. En effet, un affichage temps réel (à la seconde près) est impossible. Un décalage de 2 secondes environs (rafraichissement [1s] + temps de réception et traitement des données par SIREN [1s]) est totalement satisfaisant.

La notification est déclenchée par l'ajout de données via un INSERT suivi d'un NOTIFY.

```
LISTEN virtual; Ecoute de la notification
NOTIFY virtual; Envoie d'une notification
Asynchronous notification "virtual" received from server process with PID 8448.
```

Figure 9 – LISTEN/NOTIFY sur PostgreSQL

3.5. Algorithme de positionnement

3.5.1. Position GPS/RTK

Concernant la géolocalisation, plusieurs points sont à prendre en considération :

- Le système de coordonnées utilisé par le GPS,
- Le système de coordonnées standard utilisé en France,

Tout d'abord il faut garder en tête qu'un système GPS va fournir des informations de géolocalisation utilisant le système WGS84. C'est un système de coordonnées mondialement reconnu pour représenter le globe terrestre dans son intégralité.

Cependant, sachant que l'ONEMA est un organisme Français, il est important d'utiliser le meilleur système de coordonnées en France. Dans notre pays, c'est la projection Lambert-93 qui est officiellement reconnue.

Ainsi, sachant que nous obtiendrons des informations en WGS84 via le GPS/RTK, il sera nécessaire d'effectuer ce que l'on appelle une "re-projection" afin de transformer ces données dans un format plus compatible avec le système Français.

Cette méthode de transformation est applicable grâce à des fonctions disponibles dans la bibliothèque de PostGIS.

```
geometry ST_Transform(geometry g1, integer srid);
```

Cette méthode va nous permettre de transformer une géométrie dans le système spécifié. Dans notre cas, le code EPSG associé au Lambert-93 est le 2154.

Ainsi, il sera nécessaire d'effectuer cette transformation au sein de SIREN. Le fait de traiter cette modification dans ce programme nous permettra d'effectuer les calculs en RAM afin d'améliorer la vitesse des traitements.

Une fois les traitements faits, les positions transformées en Lambert-93 pourront être stockées dans la base de données PostGIS en toute sécurité.

3.5.2. Aspects mathématiques

$$A(t) = Ax(t) + Ay(t) + Az(t)$$

$Ax(t) = \frac{dVx(t)}{dt}$, $Ay(t) = \frac{dVy(t)}{dt}$ et $Az(t) = \frac{dVz}{dt}$

Donc pour la vitesse du point, on trouve :

$Vx(t) = \int_o^T Ax(t) + Vx_0 = Ax * T + Vx_0$, $Vy(t) = \int_o^T Ay(t) + Vy_0 = Ay * T + Vy_0$ et

$$Vz(t) = \int_o^T Az(t) + Vz_0 = Az * T + Vz_0$$

Avec Vx_0, Vy_0 et Vz_0 les valeurs de vitesse selon les axes X, Y et Z au début de la période sur laquelle on a mesuré l'accélération.
On en déduit les équations de position dans le référentiel GPS :

$$X(t) = \int_0^T Vx(t) + X_0 = 1/2\, Ax * T^2 + Vx_0 * T + X_0,$$
$$Y(t) = \int_0^T Vy(t) + Y_0 = 1/2\, Ay * T^2 + Vy_0 * T + Y_0,$$
$$Z(t) = \int_0^T Vz(t) + Z_0 = 1/2\, Az * T^2 + Vz_0 * T + Z_0,$$

Avec X_0, Y_0 et Z_0 les valeurs de position selon les axes X, Y et Z au début de la période sur laquelle on a effectué les calculs.

Il reste maintenant à déterminer les formules permettant d'obtenir les valeurs Ax, Ay, et Az depuis les valeurs d'accélération et de rotation mesurées par les capteurs du drone.

3.5.3. Calcul de la position

D'après la documentation du drone, la fréquence de mesure est de 25 Hz, et nous savons que T = 1/f, la période de mesure vaudra donc 0,04 s. Si l'on considère la vitesse du drone (4 nœuds) et la précision demandée (50 cm), il est pertinent de se demander s'il est réellement intéressant de calculer la position du drone à chaque mesure ou si on peut se contenter de calculer l'accélération moyenne sur un nombre de périodes défini et de ne calculer la position qu'après ce laps de temps. Nous allons donc introduire un paramètre permettant de choisir au bout de combien de mesures calculer la position et stocker le résultat en base.

L'ordinateur du pilote sera utile pour le traitement des données de plongée reçue par QGControl. Ces données sont les données de position générées par le boitier Pixhawk, transmises au Raspberry Pi, puis à QGControl et enfin à l'application SIREN installée sur le PC du pilote.

Nous aurions pu décider de traiter ces données directement sur le Raspberry et ainsi transmettre uniquement les coordonnées de la position. Cependant, nous avons décidé de faire tous ces calculs sur le poste client par soucis de performance. Le Raspberry possède une plus faible puissance comparée à un poste client (4 cœurs cadencés à 1.2 GHz et 1 Go de RAM contre 4 à 8 cœurs cadencés à 2,5GHZ et 8 Go de RAM). Il a donc été décidé que le Raspberry serait uniquement utilisé pour :

- Récupérer les données, les traiter et les envoyer en temps réel à QGControl
- Gérer le flux vidéo et l'envoyé à QGControl.

Un ordinateur portable possédera des capacités de traitement bien plus performantes, et c'est pour ces raisons que nous avons décidé de faire tous les calculs sur le poste client.

3.5.4. Algorithme de plongée

```
paramètre entier MAXITERATIONS
paramètre entier TAILLEPAQUET
globale booléen mesuresEnCours
globale booléen resynchroGPSDemandée

EffectuerUnePlongée()
Début
   entier itération = 0
   relevéMavlink[] mesures = relevéMavlink[TAILLEPAQUET]

   position premièrePosition = initialiserLaPlongée()
   position positionCourante = premièrePosition

   tant que()
         si (itération > MAXITERATIONS)
               signalerTropDIterations()
         fin si
         si ( resynchroGPSDemandée et capteSignalGPS())
               resynchroGPSDemandée= faux
               position nouvellePosition = synchroniserGPS()
               corrigerPositions(premièrePosition,
         nouvellePosition, positionCourante)
               premièrePosition = nouvellePosition
               positionCourante = nouvellePosition
               stockerEnBase(positionCourante)
               itération = 0;
         fin si

         initialiser(mesures)
         initialiser(data)
         pour (entier i allant de 1 à TAILLEPAQUET)
               mesures[i] = obtenirMesures()
               itération = itération + 1
         fin pour
         positionCourante  = calculerPosition(positionCourante,
      mesures)
         si (mesuresEnCours)
               stockerEnBase(positionCourante)
         fin si
   fin tant que
Fin
```

Notes :
Toutes les variables qu'utilisera l'algorithme une fois implémenté ne sont pas détaillées de façon exhaustive, il s'agit plutôt de l'aspect conceptuel (notamment, les positions successives seront conservées en mémoire jusqu'à ce qu'elles soient corrigées).

MAXITERATION correspond au nombre d'itérations possibles avant que l'algorithme ne perde trop de précision. Dépasser ce nombre n'arrête pas la prise de mesure mais l'utilisateur est averti.

TAILLEPAQUET correspond au nombre de mesures à effectuer avant de recalculer une position.

Le type position possède un champ timeStamp correspondant à l'instant auquel le drone était à la position correspondante.

resynchroGPSDemandée et mesuresEnCours sont des booléens modifiables par des actions de l'utilisateur.

initialiserLaPlongée doit attendre que le drone ait pu se localiser à l'aide de son GPS, les erreurs liées doivent être gérées.

relevéMavlink est un type représentant un ensemble de données correspondant à une mesures, et données est un type qui représente ce qu'il faut stocker en base de données, conformément à ce qui est évoqué dans la partie correspondante.

Le calcul de position se fait même si l'utilisateur n'a pas demandé de prendre les mesures car il est impossible de calculer la position à un instant sans connaître toutes les positions intermédiaires depuis la dernière synchronisation GPS.

3.5.4.1. Algorithme CalculerPosition

CalculDePosition(position depart, RelevéMavLink[] mesures)
Renvoie : la position finale du drone

```
Début
      entier période = ObtenirDernièreMesure(mesures).timeStamp -
      depart.timeStamp
      Point arrivee = calculDeplacement(depart, listeMesures,
      période)
      Retourner (arrivee)
Fin
```

3.5.4.2. Méthodes de calcul du déplacement

Comme expliqué précédemment, la position est toujours calculée selon le système de position GPS, tandis que les valeurs d'accélération sont définies par rapport au drone, dans son repère local.
Ainsi il est impossible d'utiliser directement les valeurs d'accélération, et plusieurs démarches sont possibles. Mais finalement tout peut se résumer à effectuer un ensemble d'opérations matricielles.

Une première méthode pourrait être de transformer les valeurs d'accélération locales dans le référentiel GPS, de déterminer la vitesse initiale à partir des positions calculées précédemment, puis d'utiliser les formules donnant la position à partir de l'accélération et de la vitesse initiale.

La seconde méthode consisterait à calculer la position du drone à la fin de son mouvement dans son repère (celui d'avant le mouvement), puis de convertir les coordonnées dans le référentiel GPS. Il faut

bien faire attention de tenir à jour la matrice de passage à chaque instant entre le référentiel local et le référentiel GPS.

Dans les deux cas, la méthode calculDéplacement se contente d'effectuer des opérations mathématiques élémentaires qu'il est important de bien définir.

3.5.5. Recalcul des positions et correction de l'erreur

Etant donné que les positions du drone sont calculées de façon successive à partir d'une position que l'on sait exacte puis en utilisant les valeurs fournies par le gyroscope et l'accéléromètre du drone, il est presque certain que lorsque le drone sort de l'eau et se localise via GPS, il existe un écart entre la position calculée et la position réelle.
Cependant, l'ensemble des données transmises via MAVLink sont conservées en mémoire entre deux positionnements par GPS, ce qui permet de corriger la trajectoire calculée.
Pour ce faire, deux idées semblent assez intuitives : corriger grâce à la marge d'erreur constatée, ou bien corriger en effectuant le trajet inverse.

3.5.5.1. Par la marge d'erreur

L'idée pour corriger dans ce cas-là serait la suivante : on observe en sortant de l'eau que la position calculée se trompe de **Δ**x, **Δ**y et **Δ**z, et que N mesures ont été effectuées.
On peut donc corriger chaque position à l'aide d'une suite de correction définie entre 0 et N.
A noter que cette méthode n'a de sens que si l'imprécision a été causée de façon uniforme au cours du déplacement.

```
CorrigerPositions(position départ, position arrivée, position courante)
Début
   position[]    positions    =    obtenirTableauPositions(départ,
courante)
   Vecteur3D delta = calculerDelta(arrivée, courante)
   pour chaque point dans positions à l'indice i
        point     =     translation(point,     homothétie(delta,
suiteCorrection(i)))
   fin pour
Fin
```

Avec suiteCorrection() une suite définie sur [0, N] ou N est le nombre de positions à corriger, avec

- suiteCorrection(0) = 0
- suiteCorrection(N) = 1

La phase de tests permettra d'affiner la suite de correction.

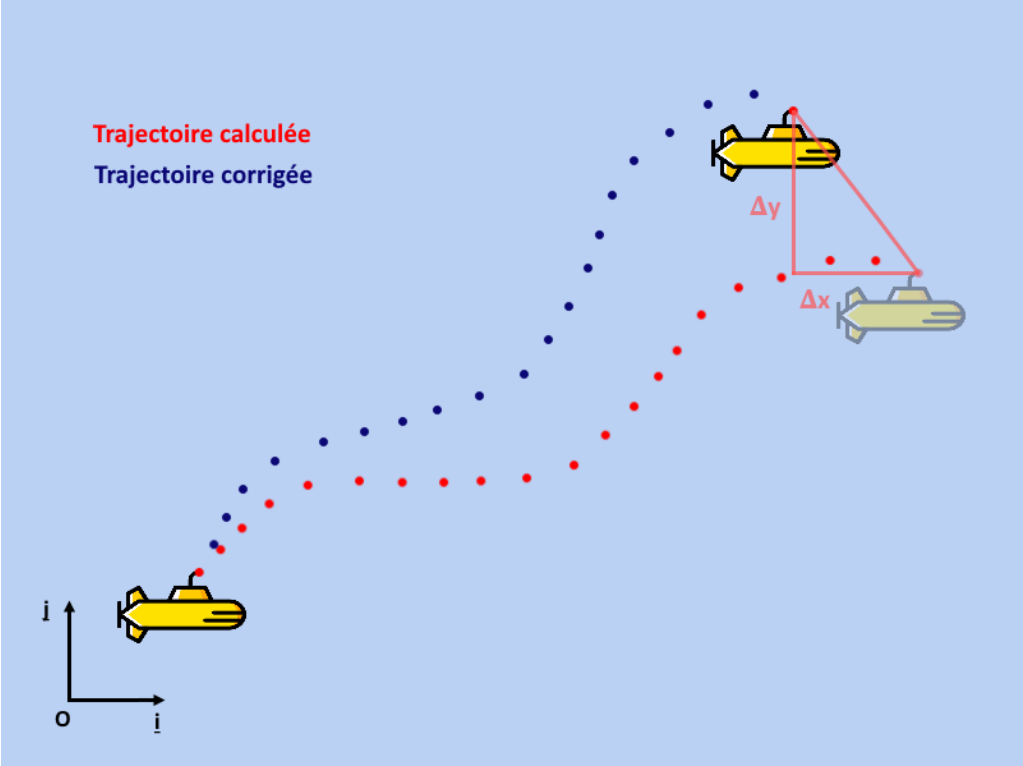


Figure 10 - Schéma représentant la correction par la marge d'erreur dans un plan (O,i,j)

3.5.5.2. Par le trajet inverse

Lorsque l'on considère comment sont calculées les positions successives, il est facile de se rendre compte que plus on s'éloigne du point de départ, plus la marge d'erreur est grande. Ainsi les premières positions calculées sont probablement précises, mais plus on s'éloigne et moins c'est le cas.
Or avec les valeurs stockées en mémoire, il est possible d'appliquer l'algorithme de calcul de position successives en partant du point de sortie, en utilisant des valeurs d'accélération inversées.
On obtiendra donc une nouvelle trajectoire qui sera plus précise aux alentours du point de sortie, et moins précise en s'approchant du point de plongée.
En se servant de ces deux trajectoires, on peut donc en déterminer une troisième qui fera office de trajectoire corrigée.

CorrigerPositions(position départ, position arrivée, position courante)

```
Début
     position[]    positions    =    obtenirTableauPositions(départ,
courante)
     position[]    inverses    =    calculerPositionsInverses(départ,
arrivée, courante)
     pour chaque point, inverse dans positions et inverses à l'indice
i
          position calcul = homothétie(point, pondération(i))
          position inv = homothétie(inverse, pondérationComplem(i)
)
          point = additionner (calcul, inv )
     fin pour
Fin
```

Avec pondération() et pondérationComplem() deux suites définies sur [0,N] avec N le nombre de positions successives telles que:

- pondération(0) = 1
- pondération(N) = 0
- pondérationComplem(0) = 0
- pondérationComplem(N) = 1
- $\forall i \in$[0,N], pondération(i) + pondérationComplem(i) = 1

La phase de test permettra de déterminer des suites répondant à ces critères et qui corrigent la trajectoire de façon satisfaisante.

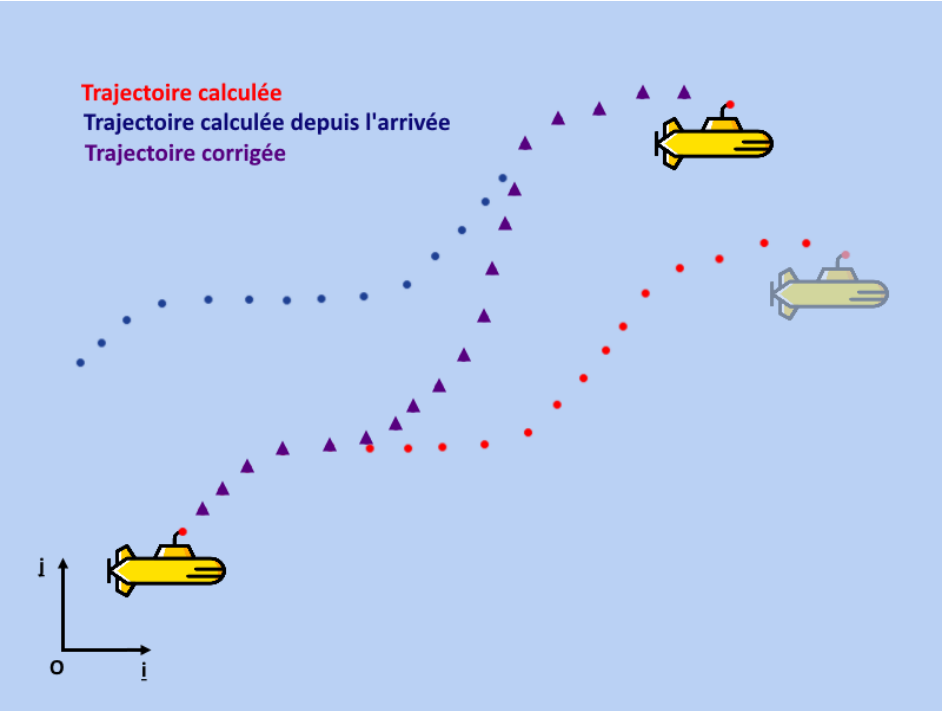


Figure 11 - Schéma représentant la correction par le trajet inverse dans un plan (O,i,j)

4. Outils

4.1. Drone

4.1.1. BlueROV

Figure 12 - Connectique BlueRov 2

Pour le projet, nous avons à réaliser un drone aquatique capable de récupérer des informations caractéristiques sur le milieu aquatique.

Il nous faut donc un drone qui peut naviguer sous l'eau. Nous avons donc choisi le BlueROV 2 de l'entreprise BlueRobotics, qui est une solution complète bien qu'il faille la monter et dont le code, qui est défini par un système d'exploitation appelé ArduSub, est OpenSource. Il est également extensible grâce à 3 pénétrateurs de câble (sortes de vis étanches en plastique avec un trou au milieu pour passer des câbles).

Il a de nombreuses fonctionnalités intéressantes mais qui ne serviront pas forcément, comme un réservoir pouvant stocker de l'eau pour réaliser des analyses (l'ONEMA n'a pas d'utilité pour le moment), ou encore des lumières pour éclairer son chemin qui seront utiles dans le cas de relevés réalisés dans des eaux où il y a peu de visibilité. Il est également possible d'y ajouter une grande quantité de capteurs pour pouvoir mesurer de nombreuses caractéristiques, ce qui est un atout pour l'évolution des besoins de l'ONEMA.

Il est également assez maniable pour que l'on puisse le diriger avec une manette de console (plus précisément, une manette XBOX).

Sur ce drone, nous aurons également divers capteurs qui interagiront avec l'extérieur, pour faire des mesures ou pour géolocaliser ces dernières. Nous pouvons les diviser en 2 groupes : les capteurs déjà implémentés ; les capteurs à implémenter.

Les capteurs implémentés sont le GPS, accéléromètre/compas/gyroscope et le capteur de pression. Il faudra néanmoins tester ces implémentations pour voir s'il n'y a pas de problème (surtout avec le GPS). Les capteurs non implémentés sont le capteur de température et tout autre capteur qui pourrait être ajouté par la suite.

Nous allons donc construire le drone nous-même, avant la période des 5 semaines (développement) à la fin de la période académique, afin de nous consacrer exclusivement au développement à réaliser.

Le temps pour monter le drone, d'après le site BlueRobotics, est compris entre 4 et 8 heures.

Voici à quoi il ressemble une fois assemblé :

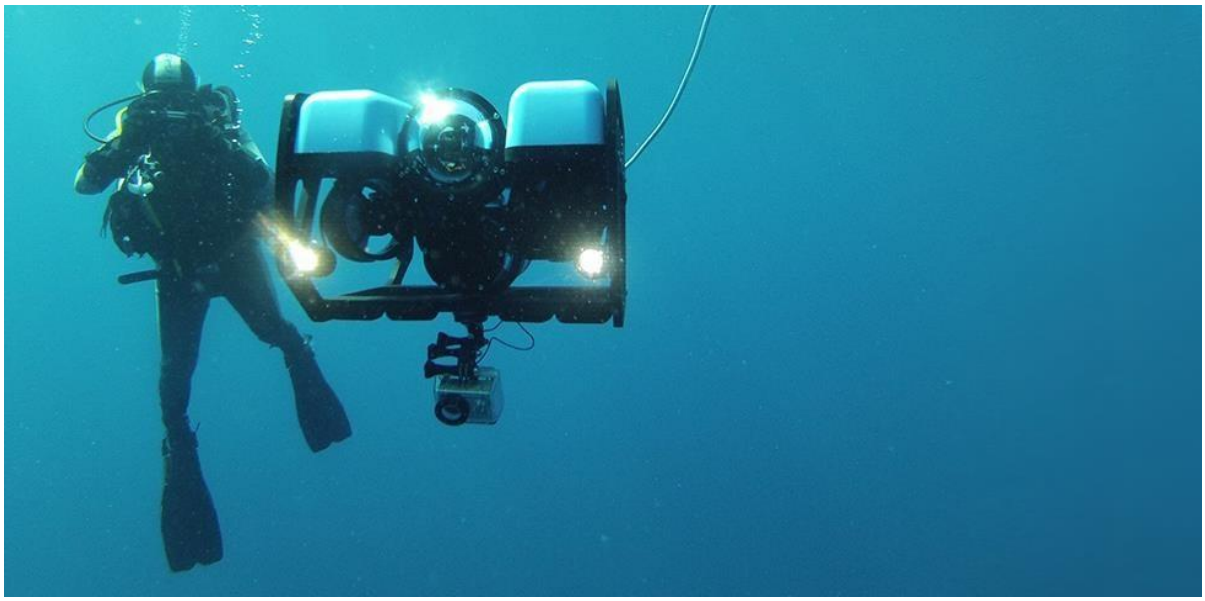


Figure 13 - BlueRov 2 Assemblé

4.1.2. Système vidéo

Nous avons un système vidéo déjà fourni avec notre drone. Mais nous avons choisi un paquetage supplémentaire contenant du matériel qui n'était pas présent dans la version de base.

Figure 14 - Caméra du BlueRov 2

A l'intérieur dudit paquetage, il y a une caméra Raspberry Pi HD, plus puissante que celle de base. En effet, la caméra de base du drone a un flux vidéo avec une latence très faible, mais c'est une caméra analogique standard, donc avec une qualité de vidéo inférieur à celle de la Raspberry Pi HD. En effet, celle-ci est performante, car elle nous permet d'avoir un flux vidéo en HD avec une résolution de 1080p et avec une latence de 200 millisecondes, ce qui reste assez raisonnable.

4.1.3. Batterie et alimentation

Dans le pack de notre drone, il n'y a pas de batterie incluse. Néanmoins, le site de Blue Robotics propose une batterie vendue sur un site externe à celui-ci.

Il nous propose d'utiliser la batterie Multistar High Capacity 4S 10000mAh Multi-Rotor Lipo Pack. Celle-ci a une capacité de 10000 milliAmpère-heure avec une dimension de 160x65x36 millimètres pour un poids de 804 grammes. Elle possède un débit constant de 10 Coulomb et un débit maximum de 20 Coulomb.
Elle est configurée en 4S1P/14.8V/4 Cell. "4S1P" signifie qu'il n'y a qu'un seul bloc (donc pas d'autres blocs en parallèle) de 4 cellules dans la batterie. "14.8V" est le voltage de la batterie et "4 Cell" fait référence à "4S1P", pour dire qu'il n'y a que 4 cellules dans la batterie.

Elle possède également deux types de prises :

- Une prise JST-XH pour la recharger
- Une prise XT90 pour la décharger

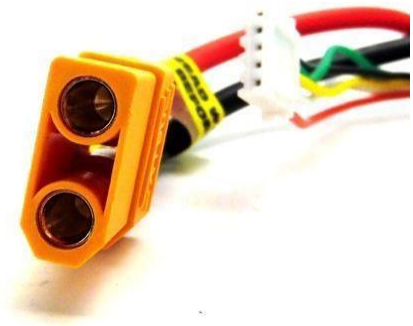


Figure 15 - Connecteur Batterie

Son autonomie permet également de faire des opérations sur le terrain pendant toute une journée, ce qui signifie que le drone ne tombera pas en panne de batterie lors d'une longue mission.

De plus, dans le cas où la batterie serait déchargée, son compartiment permet de la changer facilement. Donc le seul événement durant lequel une mission ne pourrait être menée à son terme serait que la batterie n'ait pas été rechargée avant la mission et que les agents sur le terrain n'aient pas de batterie de rechange.

Mais, en plus du circuit principal, il nous faut alimenter le circuit du Raspberry Pi en électricité. Pour ce faire, nous avons besoin d'un BEC (Battery Eliminator Circuit) qui nous permettra de rediriger du courant depuis la boucle principale vers celle du Raspberry Pi, et ainsi n'avoir qu'une seule batterie. Il se trouve dans le même package supplémentaire que notre caméra HD.

Celui qui nous est fourni est une source de courant UBEC (Universal BEC) pour Raspberry Pi. Ce type de redirection de courant est plus efficace, plus fiable et permet de fournir plus de courant à notre composant qu'un BEC classique.

2.3.1.3. Transmission de données

Pour la transmission de données, elle est encore une fois limitée dans la version standard du drone. En effet, dans la version standard, il y a un Fathom-S, qui a un débit de transmission de 250Kbit/s, contre 80Mbit/s pour le Fathom-X inclus dans le package que nous avons pris.

Dans l'hypothèse où l'on récupère les données du capteur de température et de l'accéléromètre/gyroscope/compas, ou n'importe quelle combinaison de capteurs, nous sommes limités techniquement par les 38.4kbits/s du PixHawk. Dans ce cas, le débit nécessaire nous permet d'utiliser n'importe lequel de ces deux Fathom. Néanmoins, le flux vidéo s'ajoute à ce débit.

Dans sa plus basse qualité, à une résolution de 480p, un flux vidéo a besoin d'au minimum 500 kbits/s. Le Fathom-S, avec ses 250 kbits/s, est donc incompatible.

Voilà donc pourquoi le Fathom-S est utilisé dans le package avec une caméra analogique, donc moins gourmande en débit, alors que le Fathom-X se trouve dans le même package que notre caméra.

C'est également un avantage certain, quand on prend en compte le fait que les informations renvoyées par les capteurs doivent arriver au plus vite à la station client. De plus, ces informations sont envoyées en parallèle du flux vidéo … Il faut donc que le débit soit optimal.

Format d'image	Nombre d'image/seconde	Latence
480p	30 fps	100 ms
720p	40 fps	250 ms
1080p	15 fps	250 ms

En plus de ce dernier, il y a un câble d'une longueur allant de 1 à 300 mètres. Pour éviter que les agents de l'ONEMA ne soient limités à cause d'un câble de transmission trop court, mais qu'il n'y ait pas une trop longue distance de câble, nous avons opté pour une longueur de 100m.

Le câble est en réalité une connectique Ethernet de catégorie 5. Le câble permet donc d'atteindre des débits théoriques allant jusqu'à :

- 10 Mbit/s pour la norme 10-BASE-T
- 100 Mbit/s pour la norme 100-BASE-T.

Le Fathom-X utilise donc la norme 100-BASE-T pour pouvoir offrir les meilleures performances possibles et ainsi utiliser la totalité de la Bande passante à sa disposition (80 Mbit/s).

BlueRobotics a effectué quelques essais et mesures intéressantes :

- Bande passante maximale réelle : 80 Mbit/s
- Bande passante physique théorique : 200 Mbit/s
- Taille maximale théorique du câble Ethernet : 2 000m
- Taille maximale testée du câble Ethernet : 300m.

La configuration réseau est donc entièrement satisfaisante et permet d'assurer un temps de communication le plus optimal et réduit.

4.1.4. Motorisation

Ce moteur est le plus performant de sa catégorie en termes de rapport poussée sur masse grâce à ses 6 propulseurs T200. De plus, son système de propulsion utilise la poussée vectorielle, ce qui lui permet d'avoir une grande manœuvrabilité, quel que soit la vitesse à laquelle se déplace le drone.
Ce mode de propulsion est utilisé dans l'aéronautique, et plus précisément sur les avions à réaction.

Figure 16 - Moteur T200

Pour illustrer cette propulsion, voici un schéma de fonctionnement :

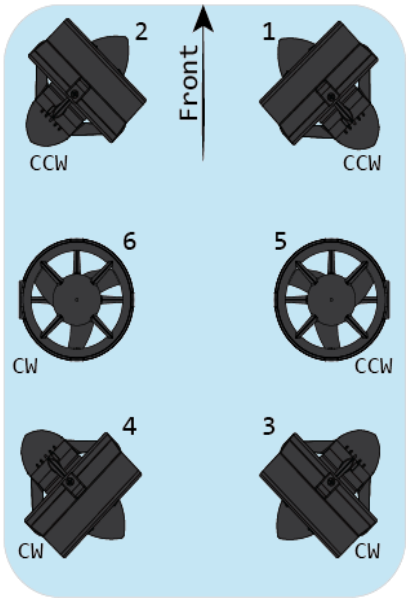


Figure 17 - Schéma de propulsion

Dans ce schéma, on peut voir 6 propulseurs. Ils sont tous présents sur le drone.

Les propulseurs 5 et 6 sont utilisés pour le déplacement vertical du drone, c'est à dire pour monter ou descendre quand le drone est droit. Les autres sont utilisés pour avancer (3 et 4) ou reculer (1 et 2).

Plus précisément, le 1 tout seul sert à reculer en tournant à droite, le 2 tout seul sert à reculer en tournant à gauche, le 3 tout seul sert à avancer vers la gauche et le 4 tout seul sert à avancer vers la droite.

Le 2 et le 3 ensemble devraient donc faire tourner le drone à gauche sur place et le 1 et le 4 devraient faire tourner le drone à droite sur place.

De plus, chaque propulseur offre la possibilité de stabiliser le drone. Le drone (via le Pixhawk) possède un algorithme de maintien en position. Ainsi, le drone se stabilise automatiquement en actionnant les différents propulseurs.

4.1.5. Pixhawk

Le Pixhawk est présent dans les deux packages proposés par BlueRobotics. Cependant, il n'est pas présent dans la version sans package du drone, et reste en option même dans les packages. Il sera lié à notre Raspberry via le Pixhawk Shelf, un support pour le Pixhawk qui permet de les assembler en un seul bloc.

Le modèle que nous avons est un Pixhawk 3DR. Ce dernier doit pouvoir héberger le contenu du framework ArduSub, la base logicielle qui nous permettra de déplacer le drone et de récupérer les informations transmises par nos capteurs. De plus, avec sa capacité mémoire de 2 Mo, il y a largement assez de place pour y intégrer notre bibliothèque ArduSub, qui ne fait même pas 1 Mo.

Figure 18 - Pixhawk

4.1.6. Capteurs

Concernant les capteurs, nous avons besoin de mesurer plusieurs paramètres pour nous repérer et faire des relevés. Nous allons en utiliser quatre types :

- Un capteur de température pour faire des relevés sous l'eau et prouver que c'est possible
- Un capteur GPS/RTK pour se repérer à la surface de l'eau
- Un capteur de pression pour obtenir la pression, mais qui calcule également l'altitude à laquelle se trouve le drone. Il sera utilisé pour connaître à quelle profondeur notre drone se situe
- Un capteur Gyroscope/Accéléromètre/compas pour pouvoir se repérer dans l'espace lorsque le GPS n'est plus fonctionnel (par exemple, sous l'eau)

Chacun d'eux est détaillé plus amplement dans la suite du document.

4.1.6.1. Capteur de température

Comme son nom l'indique, ce capteur sert à mesurer la température. C'est notre capteur test, pour vérifier que l'on est bien en mesure d'envoyer n'importe quelle donnée vers la station client. Nous utilisons l'un de ceux vendus par BlueRobotics, le Celsius Fast-Response ±0.1°C Temperature Sensor (I2C).

Ce capteur a une précision qui varie selon la température :

Température : -40 à -5 °C	**Température : -5 à +50 °C**	**Température : +50 à +125 °C**
± 0.5 °C	± 0.1 °C	± 0.5 °C

Cependant, dans la documentation du capteur, ni le temps de traitement ni la vitesse de transmission ne sont indiqués.

Figure 19 - Capteur de température

4.1.6.2. Capteur GPS/RTK

Le capteur GPS/RTK est un élément obligatoire de notre architecture. En effet, le but final étant de pouvoir récupérer des données du côté de l'agent et de géolocaliser le drone à tout moment, il nous faut un moyen de connaître sa position en temps réel.
Le rôle du GPS comme du RTK étant de donner une position géographique, il est donc normal de se tourner vers eux pour résoudre notre problématique. Comme aucun GPS ni RTK n'est intégré de base à notre drone, nous avons donc la charge de le choisir.

Le capteur que nous avons choisi et dont l'image se trouve ci-contre est le GPS Reach RTK.

En effet, il nous faut certes récupérer la position du drone en temps réel, mais cette position doit être précise à 50 cm près. Comme la plupart des GPS n'ont une précision qu'à quelques mètres, il nous fallait trouver un GPS plus précis que la moyenne.

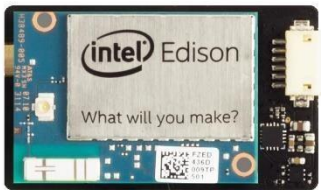


Figure 20 - GPS/RTK - Emlid Reach

Nous nous sommes donc tourné vers le RTK, qui est une technologie permettant de localiser des objets plus précisément grâce à deux balises qui se repositionnent automatiquement, avec une des deux balises sur un point fixe et l'autre sur un point mobile. Nous aurons donc l'une des deux sur notre drone et l'autre au niveau de l'agent qui pilote ce dernier.

Pour appuyer le fait que cette solution soit adaptée, il faut bien évidemment regarder ce qu'elle nous offre au niveau précision. Ce que nous cherchons à avoir est une position à 50 cm près. Ce RTK nous offre la possibilité de nous localiser à 2 cm horizontalement et 4 cm verticalement, ce qui est beaucoup plus précis, comparé à nos attentes.

Comme nous cherchons la meilleure précision possible, il ne fait aucun doute que l'objectif des 50 cm de précision est atteint, et très largement dépassé, et que le Reach RTK est une solution que nous pouvons choisir sans problème.

De plus, bien que nous ne connaissions pas la vitesse de traitement du capteur, nous savons qu'il transmet ses informations à la vitesse de 38400 bauds (ou informations par seconde), ce qui peut être considéré comme assez rapide.

Figure 21 - Connexion Pixhawk-GPS/RTK

En effet, notre drone a une vitesse maximum de 1 m/s, donc il y aura maximum 2 mesures dans une seconde. Admettons qu'une information désigne un bit. Il y a donc 19200 bits disponibles pour chaque mesure.

Une mesure ne peut avoir plus de bits qu'un message MAVLink, dans lequel elle est incluse. Or, ces messages ont une taille limitée à 263 octets, soit 2104 bits, ce qui veut dire que notre capacité de 19200 bits est largement suffisante pour envoyer une mesure.

Néanmoins, il est possible de modifier cette valeur dans la configuration du capteur.

La connexion entre les deux se fait via un port DF13 du côté Reach RTK et un connecteur Serial ⅘ pour le Pixhawk

4.1.6.3. Capteur de pression

Précédemment, nous avons parlé du GPS/RTK. Il permet de bien situer le drone dans l'espace, tant qu'il est en surface. En effet, sous l'eau, il est impossible de connaître sa position à cause des perturbations et du coefficient de réfraction entre l'eau et l'air. Il nous faut donc d'autres alternatives.

Une partie de la solution est d'utiliser notre capteur de pression, qui nous donnera notre position au niveau profondeur. En effet, la pression n'est pas la même que l'on soit à la surface, à 10m ou à 500m de profondeur. Celle-ci augmente à cause de la force exercée par l'eau sur les corps.

Notre capteur de pression sera le Bar30 Pressure Sensor, un capteur inclus avec le drone. D'après sa documentation sur le site de BlueRobotics, il fonctionne pour des pressions comprises entre 0 et 30 bar (ce qui correspond à 300m sous l'eau), mais a une valeur maximale de 50 bar.

Il possède la précision suivante :

	Température : -25 à 0 °C	**Température : 0 à +40 °C**	**Température : +40 à +85 °C**
Pression : 0 à 6 bars	± 100 mbars	± 50 mbars	± 100 mbars
Pression : 6 à 20 bars	± 200 mbars	± 100 mbars	± 200 mbars
Pression : 20 à 30 bars	± 400 mbars	± 200 mbars	± 400 mbars

De plus, grâce à la bibliothèque ArduSub, la fréquence à laquelle nous pouvons récupérer des informations du capteur est de 10 Hertz, c'est à dire 10 mesures par secondes, ce qui est intéressant compte tenu du fait qu'il nous faut une position tous les 50 cm.

En effet, pour que l'on ne puisse pas se repérer à 50cm, il faudrait que notre drone ait une vitesse supérieure à 5m/s. Or, selon la documentation de celui-ci, sa vitesse maximum en ligne droite est de 1m/s. Donc, au vu de ces chiffres, nous pouvons en conclure que le capteur devrait nous permettre d'atteindre notre objectif.

Figure 22 - Capteur de Pression

Il semblerait également, selon la bibliothèque spécialement implémentée pour le capteur, qu'un traitement soit planifié toutes les 20 millisecondes. Néanmoins, ne pouvant l'affirmer à 100%, le temps de traitement nous est encore inconnu.

4.1.6.4. Capteur gyroscope/accéléromètre/compas

Comme énoncé précédemment, le GPS et le RTK ne peuvent pas obtenir une position fiable sous l'eau. Nous avons vu le capteur de pression pour la profondeur. Il nous faut donc un/des capteur(s) pour pouvoir obtenir notre position comme avec un GPS/RTK.

C'est pour cela que nous utilisons un capteur composé d'un gyroscope, d'un accéléromètre et d'un compas. C'est ce que l'on appelle un capteur 9 axes, qui permet de détecter des mouvements sur 9 axes et de calculer ensuite une position approximative assez proche de la position réelle. Cette solution, couplée au capteur de pression, est donc une alternative plus que satisfaisante au GPS/RTK lorsque ce dernier ne peut plus assurer son rôle, c'est à dire quand nous ne récupérons pas de mesure du GPS/RTK.

Notre capteur, qui est intégré au Pixhawk, a une précision de ±1.11mg pour son accéléromètre (sachant que g est l'unité de l'accélération et vaut 9.80665 m/s^2), ±0.1 °/s pour son gyroscope et ±7 milli-gauss pour son compas.

A chaque seconde, nous devrions avoir une incertitude qui serait de :
$a*t^2/2 = a/2 = g*val_{mg}$ = 0.00111*9.80665/2 m= 1.11*9.80665/2 mm = 5.443 mm
Cette incertitude ne prend en compte que l'accéléromètre, qui agit directement sur la position du drone.

Pour le temps de traitement, aucune information n'est fournie pour les trois capteurs.

Bien que la documentation soit assez difficile à comprendre, il semblerait que l'envoi d'informations depuis le gyroscope se fasse à une fréquence de 4 à 8000 Hz, à une fréquence de 4 à 1000 Hz pour l'accéléromètre et une fréquence de 400 kHz pour le compas.

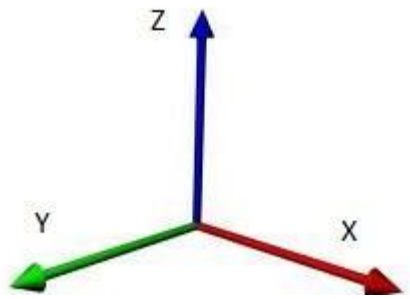


Figure 23 - Représentation des axes des capteurs

Description des axes pour la position relative au drone (pour les capteurs) :

- X : Direction du drone
- Y : Translation latérale du drone
- Z : Translation verticale du drone

4.1.7. Raspberry Pi 3

4.1.7.1. Le Raspberry

Le Raspberry Pi 3 est un micro-ordinateur doté de 4 cœurs cadencés à 1.2 GHz. Il est équipé de 6 ports USB, de 1Go de RAM et d'une connectivité Wi-Fi et Bluetooth. Notre Raspberry possédera un Raspbian comme Système d'exploitation stocké sur une carte SD de 8Go.

Figure 24 - Rapsberry Pi 3

4.1.7.2. La vidéo du Raspberry

Le Raspberry sera chargé de récupérer le flux vidéo afin de le transmettre à QGControl. Pour lancer la récupération du flux vidéo, une commande sera à taper :

```
raspivid --nopreview \
      --mode 5 \
      --bitrate 15000000 \
      --intra 1 \
      --awb auto \
      --brightness 55 \
      --saturation 10 \
      --sharpness 50 \
      --contrast 15 \
      --drc medium \
      -fl \
   --timeout 0 \    --
output - | \
   gst-launch-1.0 -v fdsrc ! h264parse ! rtph264pay
configinterval=10 pt=96 ! \
      udpsink host=192.168.2.1 port=5600
```

Les paramètres de cette commande seront probablement modifiés pour optimiser l'utilisation de la vidéo à nos besoins. Les informations sur cette caméra sont détaillées dans la partie 2.3.1.1. Système Vidéo

4.1.7.3. MAVLink

Il est nécessaire de lancer Mavproxy pour permettre l'envoi des données entre le Raspberry et QGControl. Une commande est nécessaire pour démarrer Mavproxy pour permettre au lien série d'ouvrir la connexion en sortie pour l'envoi des messages au format Mavlink. Le protocole UDP en mode broadcast est utilisé pour le transfert de ces données. La commande suivante permet d'initialiser la liaison entre le Raspberry et QGControl :

```
mavproxy.py --master /dev/ttyACM0,115200 \
            --out udpin:localhost:9000 \
            --out udpbcast:192.168.2.255:14550
```

L'entrée se faire en local sur le port 9000 et la sortie sur une adresse IP de broadcast sur le port 14450.

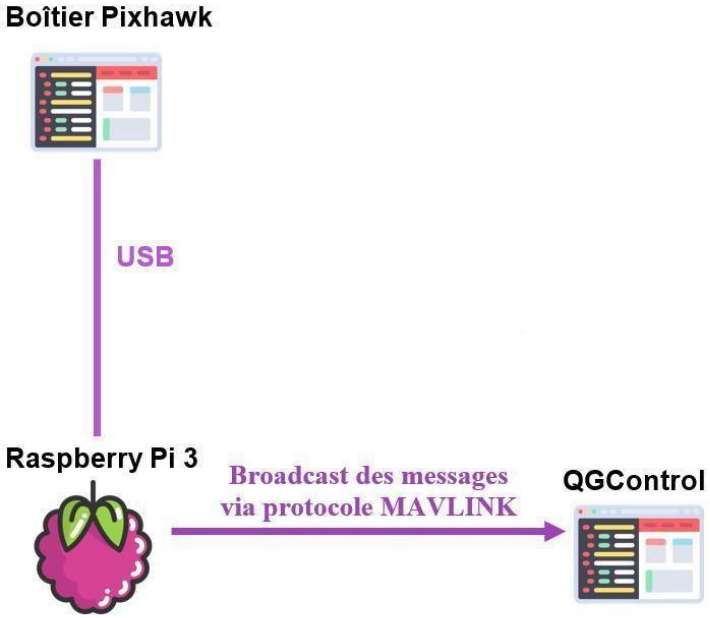


Figure 25 - Schéma simplifié de la connexion entre le drone (Pixhawk/Raspberry Pi) et les utilisateurs (QGControl)

4.1.8. Le protocole MAVLink

4.1.8.1. Contexte

Le protocole de communication réseau MAVLink permet de réaliser la communication entre la station de pilotage QGControl et le drone. Le Pixhawk est le composant principale du drone qui utilise ce protocole de communication. Le présent protocole permet de gérer dans sa totalité le drone. En effet, le protocole est le seul moyen de communication entre le pilotage / la supervision et le drone.

Parallèlement au protocole MAVLink, un autre flux de données transite entre le drone et le QGControl. Ce flux correspond à la vidéo. Celui-ci va donc occuper une bonne partie de la bande passante disponible. Les paquets MAVLink devront être le plus léger possible pour que la communication soit la plus rapide et la moins parasitée/impactée par la vidéo.

La partie capteur de « mesure » (servant uniquement pour la réalisation des mesures – la température dans notre cas), la partie « position » (GPS ; IMU – Accéléromètre, Gyroscope, Compas ; Pression), ainsi que les moteurs sont reliés au Pixhawk. Ainsi, toutes les communications relatives à l'un de ses composants sont établies au travers du protocole MAVLink.

Cependant, l'application permettant de gérer le positionnement, ainsi que le stockage du couple mesure/position, est positionnée après le QGControl. L'application de pilotage permet de réaliser une réplication des flux MAVLink. Ainsi, le contrôleur recopie le flux MAVLink reçu vers l'application. C'est donc pour cette raison que l'étude du protocole MAVLink est réalisée.

4.1.8.2. Présentation

Le protocole MAVLink est un protocole destiné au pilotage de drone. Le Micro Air Vehicle Link (ou MAVLink) est avant tout destiné à une utilisation sans-fil. En effet, le format et la méthode d'émission sont conçus pour des usages sans-fils et possédant un faible débit (due à la distance ou à la puissance d'émission).
A l'origine, le système installé sur le Pixhawk (ArduSub) a été conçu pour les drones aériens (ArduPilot). Cependant, les équipes de développement BlueRobotics ont décidé de ne pas recommencer tout le système. Ils se sont donc basés sur ArduPilot pour créer ArduSub. En reprenant ainsi la base d'ArduPilot, les équipes ayant développé le BlueRov2 ont gardé le protocole de communication existant : MAVLink. C'est donc pour cette raison que le protocole de communication aérien est utilisé dans un contexte sous-marin.

Le protocole de communication MAVLink est en service depuis 2009. Celui-ci est libre d'utilisation et de modification : il est sous licence LGPL (une des licence OpenSource les plus permissives existantes). Le protocole a été créé par Lorenz Meier. Depuis sa création, le protocole a évolué, et a rencontré un certain succès auprès des fabricants de drone.

Le protocole est utilisé dans les systèmes suivants :

- ArduPilot (ArduSub, le système du présent drone, est basé sur ce système)
- MatrixPilot UAV DevBoard
- ETH Flying Machine Arena
- ETH SenseSoar Solar Airplane Project
- ETH Skye Blimp Project
- UC Santa Cruz SLUGS
- ArduCAM OSD
- Sky-Drones - UAV Flight Control Systems
- AutoQuad - Autonomous Multirotor Vehicle controller

4.1.8.3. Configuration

Figure 26 - Schéma réseau pour le protocole MAVLink

Comme on peut le percevoir sur le schéma précédent, le Pixhawk possède le rôle de serveur pour la communication entre le QGControl et le drone. Afin de faciliter les communications et le pilotage, le Pixhawk communique en broadcast. Ainsi, tous les appareils capables de gérer le protocole MAVLink peuvent communiquer avec le drone. Il est ainsi tout à fait possible de connecter la station de pilotage sur un poste pour visualiser le flux vidéo, tout en pilotant le drone avec son smartphone.

Contrairement à la communication entre le drone et la station de contrôle, la réplication du flux MAVLink est réalisée sur une communication unidirectionnelle. QGControl n'a pas la nécessité de recevoir des informations en provenance de l'application. Cette communication est un réplica de la communication réalisée entre le Pixhawk et le QGControl. Cela permet donc à l'application de réaliser son rôle.

L'ensemble des communications réalisées avec le protocole MAVLink utilise le protocole de transport (couche 4 du modèle OSI) UDP (datagramme). Ce protocole de transport a l'avantage d'être léger et rapide. En effet, le réseau entre le drone et la station de pilotage n'est pas parasité par d'autre communication.

Le protocole de transport UDP ne garantit pas le bon acheminement des datagrammes (donc des messages MAVLink). Il ne garantit pas non plus l'arrivée des paquets dans l'ordre d'envoi.

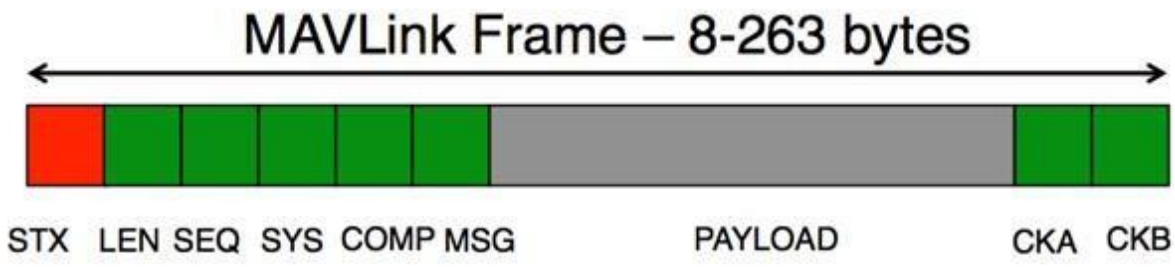


Figure 27 - Format des messages MAVLink

Les serveurs de communication sont les suivants :

	Pixhawk <=> QGControl	QGControl <=> Application
Composant	Pixhawk (via l'IP du Raspberry)	SIREN
Port/Protocole	14550/UDP	14555/UDP
Type de routage/diffusion	Broadcast	Unicast

4.1.8.4. Structure Générale

Afin de ne pas surcharger le réseau, le protocole MAVLink impose une taille maximale de paquet : 263 octets.

Un paquet est composé selon la structure suivante :

Indicateur	Titre	Valeur	Explication
STX	Indicateur de début de paquet	0xFE	Indique le début d'un nouveau paquet
LEN	Taille du Payload	0 - 255	Indique la taille des données stockées (Payload)
SEQ	Numéro de séquence Composant	0 - 255	Chaque composant possède un compteur de paquet envoyé. Ce compteur est ajouté au paquet. Celui-ci permet de détecter la perte de paquet.
SYS	Identifiant du Système	1 - 255	Identifiant du système émetteur. Cet identifiant permet à différents systèmes d'émettre sur le même réseau.
COMP	Identifiant du Composant	0 - 255	Identifiant du composant émetteur. Cette identification permet à chaque à différent composant d'un même système d'envoyer des trames (ex : IMU et pilote automatique).
MSG	Identifiant du type de message	0 - 255	Identifiant permettant de connaître le format des données stockées dans le champs Payload.

PAYLOAD	Données	(0 - 255) octets	Données utiles du paquet. Les données sont formatées, le type de formatage est indiqué dans le champs MSG.
CKA CKB	Hachage	Ce champ sur deux octets permet de calculer le hachage permettant de vérifier l'intégrité du message. L'algorithme ITU X.25/SAE AS-4 hash est utilisé. Celui-ci se base sur les bits de poids forts et faibles.	

Comme indiqué-ci dessus, les données utiles sont formatées selon un typage défini. Ce formatage est indiqué au travers d'un identifiant unique représenté par le champ MSG. Le typage est évolutif. En effet, un formatage principal est défini dans le protocole, mais il est également possible de le faire évoluer. Le protocole est basé sur un formatage défini dans un fichier XML. Il est donc tout à fait possible de rajouter de nouveaux formats de message pour ainsi communiquer de nouvelles informations. Les identifiants de message 150 à 240 sont dédiés à cet effet.

Attention, il faut s'assurer que les différents identifiants de type de message sont uniques. Par exemple, ArduSub intègre de nouveaux messages. L'intégration de nouveaux messages devra être réalisée en s'assurant que le message ID est disponible.

4.1.8.5. Messages utiles

Les messages (Payload) présentés par la suite sont les plus importants du projet. En effet, ce sont eux qui auront la charge d'envoyer les données relatives au positionnement et aux mesures.

Les différents messages sont définis dans la documentation du protocole (en Anglais) : https://Pixhawk.ethz.ch/mavlink/.

SYSTEM_TIME #2

Ce message permet d'obtenir la synchronisation temporelle avec le drone. En effet, on obtient l'heure locale du drone, ainsi que le temps écoulé depuis le démarrage de celui-ci. Ce type d'information permet de calibrer la totalité des algorithmes (positionnement et mesure).

Champs	Type	Description
time_unix_usec	Entier non-signé (8 octets)	Temps écoulé en microseconde depuis le 1er Janvier 1970. Cette durée est locale au drone.
time_boot_ms	Entier non-signé (4 octets)	Durée en milliseconde depuis le démarrage du drone. Cette information est présente dans tous les messages relatifs aux capteurs. Cette information est donc importante et permettra donc de synchroniser les temps drone avec l'heure locale.

GPS_RAW_INT #24

Message contenant les différentes informations relatives au GPS. Ces informations **sont brut.** Aucun traitement par le système n'a été réalisé.

Champs	Type	Description
time_usec	Entier non-signé (8 octets)	Référence temporelle (en milliseconde) de la mesure (Temps depuis le dernier démarrage du drone ou depuis le 01/01/1970)
fix_type	Entier non-signé (1 octet)	Se référer à GPS_FIX_TYPE
lat	Entier signé (4 octets)	Latitude (en fonction du système de coordonnés WGS84) en degré. La valeur est multipliée par 10^7.
lon	Entier signé (4 octets)	Longitude (en fonction du système de coordonnés WGS84) en degré. La valeur est multipliée par 10^7.
alt	Entier signé (4 octets)	Altitude (en mètre) relative au niveau de la mer (AMSL). La valeur est multipliée par 10^3. L'altitude en fonction du système de coordonnés WGSA n'est pas fournie (mais peut être calculée par le module GPS).
eph	Entier non-signé (2 octets)	Dilution horizontale (précision du positionnement sur le plan horizontal) de la position (aucune unité). Si la valeur est inconnue, alors UINT16_MAX (Tous les bits à 1).
epv	Entier non-signé (2 octets)	Dilution verticale (précision du positionnement sur le plan vertical) de la position (aucune unité). Si la valeur est inconnue, alors UINT16_MAX (Tous les bits à 1).
vel	Entier non-signé (2 octets)	Vitesse sol (en m/s multiplié par 10^2) calculée par le GPS. Si la valeur est inconnue, alors UINT16_MAX (Tous les bits à 1).

cog	Entier non-signé (2 octets)	Direction du mouvement (pas le sens du module GPS) en degrés. La valeur est multipliée par 10^2 (variation : 0.0 ... 359.99 degrés). Si la valeur est inconnue, alors UINT16_MAX (Tous les bits à 1).
satellites_visible	Entier non-signé (1 octet)	Nombre de satellite permettant de calculer la position. Si l'information n'est pas connue, la valeur est 255.

GPS_FIX_TYPE

Type de GPS ayant généré les messages de positionnement GPS. Dans notre cas, les identifiant GPS_FIX_TYPE_RTK_FLOAT (5) et GPS_FIX_TYPE_RTK_FIXED (6) sont les messages nous intéressant. En effet, ce sont ces messages qui permettent d'obtenir une position précise à 10 cm près (50 cm souhaité), mais uniquement en surface.

Valeur	Nom	Description
0	GPS_FIX_TYPE_NO_GPS	Aucun GPS connecté
1	GPS_FIX_TYPE_NO_FIX	Aucune réception de position, mais le GPS est connecté
2	GPS_FIX_TYPE_2D_FIX	Position 2D (latitude et longitude) obtenu par GPS
3	GPS_FIX_TYPE_3D_FIX	Position 3D (latitude, longitude et altitude) obtenu par GPS
4	GPS_FIX_TYPE_DGPS	Position GPS 3D assisté par DGPS/SBAS
5	GPS_FIX_TYPE_RTK_FLOAT	Position 3D GPS obtenu en coordination avec le RTK flottant (calcul de position RTK sans limite/minimum - flottante)
6	GPS_FIX_TYPE_RTK_FIXED	Position 3D GPS obtenu en coordination avec le RTK flottant (calcul de position RTK avec des limites de minimum pour obtenir un résultat « acceptable »)
7	GPS_FIX_TYPE_STATIC	Position GPS fixe.

SCALED_IMU #26

Ce message permet de communiquer les informations relevées par le gyroscope, accéléromètre et compas (IMU). Les informations transmises ont été converties par le système.

Les données calculées sont relatives à la dernière position du drone. C'est-à-dire que lorsque la position est envoyée, le capteur réinitialise toutes les mesures à 0. La position (inclinaison, cap …) devient alors la référence pour la mesure suivante. Il est donc impératif de connaître la position précédente pour pouvoir modéliser la nouvelle position relevée par l'IMU.

Champs	Type	Description
time_boot_ms	Entier non-signé (4 octets)	Référence temporelle (en milliseconde) de la mesure (Temps depuis le dernier démarrage du drone)
xacc	Entier signé (2 octets)	Accélération (translation) sur l'axe X (mg - accélération)
yacc	Entier signé (2 octets)	Accélération (translation) sur l'axe Y (mg - accélération)
zacc	Entier signé (2 octets)	Accélération (translation) sur l'axe Z (mg - accélération)
xgyro	Entier signé (2 octets)	Vitesse de rotation sur l'axe X (millirad /sec)
ygyro	Entier signé (2 octets)	Vitesse de rotation sur l'axe Y (millirad /sec)
zgyro	Entier signé (2 octets)	Vitesse de rotation sur l'axe Z (millirad /sec)
xmag	Entier signé (2 octets)	Orientation magnétique du capteur (sens du drone) sur l'axe X par rapport au Nord magnétique (milli tesla). **Attention à la dérive magnétique.**
ymag	Entier signé (2 octets)	Orientation magnétique du capteur (sens du drone) sur l'axe Y par rapport au Nord magnétique (milli tesla). **Attention à la dérive magnétique.**

zmag	Entier signé (2 octets)	Orientation magnétique du capteur (sens du drone) sur l'axe Z par rapport au Nord magnétique (milli tesla). **Attention à la dérive magnétique.**

SCALED_PRESSURE #29

Ce message permet de récupérer la pression et la température mesurées par le système.

Champs	Type	Description
time_boot_ms	Entier non signé (4 octets)	Référence temporelle (en milliseconde) de la mesure (Temps depuis le dernier démarrage du drone)
press_abs	Entier flottant	Pression Absolue (hectopascal)
press_diff	Entier flottant	Pression différentielle (hectopascal). Ce champ est inutile dans notre cas. Aucune sonde de vitesse (Pitot) n'est installée.
temperature	Entier signé (2 octets)	Température en degrés Celsius (La valeur est multipliée par 10^2).

4.1.9. Console - QGControl

QGControl est un logiciel disponible sur les 3 systèmes d'exploitation principaux ainsi que sur les smartphones Android et Apple. Il fournit un contrôle total du véhicule et des paramètres associés. L'objectif général de QGControl est d'apporter une facilité d'utilisation du drone pour les nouveaux utilisateurs et également des outils avancés pour les utilisateurs expérimentés.

Les fonctionnalités globales de QGControl sont :

- Un paramétrage complet de notre drone
- La planification de missions pour des plongées autonomes
- Un affichage de la carte en temps réel et des instruments du véhicule
- Un streaming de la vidéo
- Un support de pilotage pour tout véhicule capable de communiquer avec le protocole MAVLink.

Toutes les communications entre le drone et l'application sont réalisées au travers du protocole MAVLink.

Pour utiliser QGControl, il faut commencer par télécharger (sur le site de l'éditeur - qgroundcontrol.com) et installer le logiciel. Sur l'écran d'accueil, nous avons accès aux fonctionnalités principales disponibles :

Figure 28 - Console QGControl

Les boutons en haut à gauche permettent, dans l'ordre, de :

- Configurer l'application QGControl
- Configurer le véhicule
- Créer des missions autonomes
- Surveiller le véhicule pendant sa mission, vidéo incluse

De plus, l'application offre une réplication de flux de MAVLink. Cette fonctionnalité est nécessaire au projet. En effet, les différents messages MAVLink arrivant sur QGControl sont automatiquement transférés à une autre application. Dans notre cas, cette autre application n'est autre que l'application de gestion des positions/mesures pour les traiter et les afficher dans QGIS.

4.2. Langages

4.2.1. Java

La brique Java est une des pierres angulaires de notre projet, nous envisageons pour interfacer les éléments applicatifs du drone (données Pixhawk via QGControl) et ceux du poste contrôleur (QGIS) d'utiliser ce langage dans la réalisation de cette partie de notre système.

Le choix de ce langage s'est fait naturellement au regard des contraintes imposées par l'environnement, ainsi que les compétences des membres de l'équipe. Java étant l'une des technologies que nous maîtrisons le mieux, nos connaissances nous ont permis d'établir la faisabilité de cette partie en levant rapidement nos interrogations vis à vis de l'environnement, l'équipe disposant déjà d'un certain recul à ce sujet.

En effet, après étude approfondie de l'environnement du drone, nous avons pu nous rendre compte que nous pouvions récupérer les messages MAVLink émis par celui-ci.

Ces messages sont disponibles soit en passant directement sur le réseau en récupérant les trames, soit par réplication de ces dernières par l'application QGControl (déjà présente sur le poste analyste) nécessaire au contrôle et monitoring du drone aquatique.

La réplication des données par QGControl nous permet de choisir un port de destination sur lequel sera envoyé toutes les trames MAVLink, les rendant accessibles sur n'importe quelle application écoutant le port spécifié. Ces messages contenant les informations relatives aux relevés des capteurs ainsi que sur l'état du drone, il est nécessaire de pouvoir en extraire le contenu en s'affranchissant des problèmes liés au protocole MAVLink.

La découverte d'une API écrite en Java (MAVLinkJava) permettant l'analyse syntaxique (parsing) des paquets de type MAVLink nous a permis de nous conforter dans ce choix technologique. De même, nous pourrons depuis cette application traiter les données (corrections, stockage, log …) des paquets émis par le drone et ainsi les faire suivre à d'autres applications comme QGIS via le remplissage d'une base de données.

4.2.2. C++

Dans le projet, le C++ joue une part importante. En effet, bien que l'application que nous allons développer soit en Java et que le module QGIS à implémenter pour y ajouter des fonctionnalités soit en Python, toute la partie que nous ne développons pas (ou très peu) et qui sert à relier SIREN au Raspberry Pi et au Pixhawk, ainsi que les fonctionnalités intégrées dans ce dernier, sont écrites en C++.

Ce langage n'étant pas vu pendant nos années au sein de l'école, en plus du fait qu'il implémente les fonctionnalités de communication entre le système et notre application (comme dit précédemment), l'utilisation de ce langage dans le projet pose une contrainte technique assez lourde. En effet, apprendre les mécanismes d'un langage ainsi que sa syntaxe dans un temps plus que limité et en partant de la base n'est pas une chose facile.

Cependant, bien qu'il ne soit pas étudié en IR ou en IG, certains membres de l'équipe l'ont étudié avant d'arriver à l'ESIPE. Ils ont donc un certain bagage qui leur permet d'analyser le code de

l'ArduSub, dans le but d'expliquer le fonctionnement de ces "modules" aux autres membres de l'équipe.
Cela apportera à l'ensemble de l'équipe une compréhension plus globale des possibilités offertes par la partie en C++.

Au final, ce que nous devons faire, c'est un pont entre ce langage et le JAVA pour que notre application puisse communiquer avec notre système, des modifications sur l'interface QGControl qui sera utilisée par les agents et le développement des fonctionnalités utilisées par les capteurs ajoutés.

Comme dit dans la partie précédente, la bibliothèque de communication, à savoir MAVLink, qui est implémentée en C++ l'est également en JAVA. Cette partie ne devrait donc pas subir de modification en C++. De plus, les modifications de l'interface QGControl seront implémentées en JAVA, ce qui veut dire que les seules modifications à effectuer dans ce langage sont les ajouts de capteurs.

4.2.3. Python

Afin de réaliser un plugin QGIS permettant l'affichage des couches métiers, nous devons utiliser le langage Python. Ce choix nous est imposé par la nature du logiciel de SIG (QGIS) qui n'offre que cette option concernant le développement d'applications tierces.

Ce langage n'est pas réellement enseigné dans notre formation mais une partie du groupe projet (Informatique et Géomatique) a déjà développé, dans le cadre d'un enseignement géomatique, un plugin QGIS. Cependant, un autre point est le fait que Python est un langage de scripting très répandu et facile d'utilisation.

Dans le cas de QGIS une variante du langage appelé PyQGIS est utilisée. Sachant que la majeure partie du code de QGIS a été réalisé avec le framework QT (C++), PyQGIS utilise PyQT et SIP. Ces derniers permettent d'écrire facilement des modules Python permettant ensuite de créer des interfaces C++.

L'avantage avec QGIS réside dans le fait que bon nombre de fonctions prédéfinies existent, permettant ainsi de manipuler facilement l'interface. Ainsi, grâce à Python il sera possible de puiser dans la base de données PostGIS, traiter/modifier les données, les afficher sur le canvas et reprojeter dans un système de coordonnées donné.

4.2.4. PostgreSQL et PostGis

Dans le cadre de notre projet nous allons utiliser le système de base de données (SGBD) PostgreSQL. Cela est dû à son extension PostGIS qui ajoute plusieurs fonctionnalités géographiques.

Le stockage des informations se fera de manière traditionnelle. C'est-à-dire en utilisant des champs "ID", "nom du capteur", "valeur de la mesure"… Le seul changement ici sera le fait de stocker directement des "géométries".

Ces géométries autoriseront les représentations des mesures sous QGIS. Elles seront principalement sous la forme de points mais pourront parfois être des polygones si besoin est. Chaque géométrie insérée dans la base de données possède son propre système de projection, il sera ainsi important de veiller à l'uniformité des valeurs utilisées dans notre table afin de ne pas avoir d'incohérences lors de l'affichage sous QGIS.

Aussi, afin de peupler la base de données, il sera nécessaire d'utiliser les fonctions PostGIS pour créer des points avec un système de coordonnées donné. Pour illustrer cela, voici un exemple de fonction que nous pourrons utiliser lors de l'ajout d'un nouveau point de mesure dans notre base de données PostGIS.

ST_SetSRID(ST_MakePoint(X, Y, Z),2154)

La première fonction, "ST_SetSRID", permet de définir une projection à une géométrie donnée et possède le prototype suivant :

```
geometry ST_SetSRID(geometry geom, integer srid);
```

Le premier argument est une géométrie. Dans notre cas cela se traduira par l'appel d'une fonction (ST_MakePoint) qui permettra de construire un point grâce à ses coordonnées (X, Y, Z).

```
geometry ST_MakePoint(double precision x, double precision y, double precision z);
```

Puis, le deuxième argument sera un entier représentant le code EPSG d'un système de projection (2154 pour Lambert-93 dans notre cas).

4.2.5. QGIS

Pour afficher les informations issues du drone nous utiliserons QGIS. C'est une demande formulée par notre client. Cependant, nous supportons totalement ce choix. Afin de respecter une architecture OpenSource, QGIS remplit parfaitement son rôle. Son utilisation est gratuite et permet de réaliser la même chose que son concurrent payant (ArcGIS de ESRI).

QGIS va nous permettre d'afficher les informations reçues sous forme de couches. Ces-dernières contiendront les informations issues de la base de données PostGIS. Il sera possible, pour chaque couche, de consulter la table attributaire correspondante.

Afin d'afficher les informations issues de PostGIS, nous développerons un plugin Python. La gestion des plugins est une des fonctionnalités principales de QGIS. En effet, la communauté a rendu l'intégration facile et rapide en fournissant notamment une documentation complète.

4.2.6. Framework

4.2.6.1. Spring data JPA

Spring data JPA est un module de la famille de Spring data, qui permet de rendre plus facile l'accès aux données contenues dans la base de données. Cet outil nous permettra d'écrire et de lire les données renvoyées par les différents capteurs du drone. Les principaux avantages de l'utilisation de Spring Data JPA sont la facilité d'implémentation de l'outil, la facilité de prise en main et d'utilisation et le peu de code nécessaire à écrire pour obtenir les fonctionnalités nécessaires à notre besoin.

Une utilisation efficace de cet outil nécessite la compréhension de deux instruments : les repository et les entity. Chaque repository est associé à une entity via un ID de classe. Un repository est une interface Java qui définit le contrat de l'interface avec la base de données, c'est à dire les requêtes possibles sur cette base de données par le repository. Une entity est un objet Java classique qui contient des variables et des méthodes.

L'efficacité de Spring Data JPA repose dans la faible quantité de code nécessaire à l'exploitation de la base de données. Regardons le repository suivant :

```
1  package com.guitar.db.repository;
2
3  import java.util.List;
8
9  public interface LocationJpaRepository extends JpaRepository<Location, Long> {
10     List<Location> findByStateLike(String state);
11
12     List<Location> findByStateOrCountry(String value, String value2);
13
14     List<Location> findByStateAndCountry(String value, String value2);
15
16 }
```

Ce repository est associé à la classe "Location" comme défini ligne 9 dans l'implémentation de JpaRepository. La classe Location contient 2 champs : "State" et "Country".
Techniquement, ces 16 lignes de code nous permettent de faire plusieurs accès à la base de données :

- Les opérations CRUD (Create Read Update Delete) sont effectives de base grâce à l'implémentation du "JpaRepository", c'est à dire qu'il est possible de faire ces opérations simples juste en implémentant la classe "JpaRepository".
- Ligne 10 : cette ligne nous offre la possibilité de récupérer une liste de villes en passant l'état en argument.
- Ligne 12 : ce code nous permet de récupérer une liste de villes contenues dans un état ou dans un pays en passant, dans l'ordre, l'état et le pays en argument.
- Ligne 14 : cette ligne suit le même principe que le code à la ligne précédente mais cette fois en utilisant un "ET" logique.

Concrètement, cet outil facile à implémenter dans notre projet nous facilitera la tâche pour faire nos accès et nos écritures en base de données.

4.2.6.2. JavaFX

JavaFX est une bibliothèque de création d'interfaces graphiques officielle de Java permettant la confection de toutes sortes d'applications. L'utilisation de cette librairie permet de définir des vues contenant un agencement particulier de contrôles afin de représenter des actions et des informations dans notre cas.
L'utilisation d'une bibliothèque graphique permet d'améliorer l'expérience utilisateur en rendant accessible et en centralisant toutes les fonctionnalités du système proposé dans la même vue.

JavaFX utilise le langage FXML qui se rapproche du HTML et de la syntaxe XML. Ce système permet de définir à l'aide de balises la hiérarchie entre les éléments d'une vue. La définition d'un squelette permettant d'ajuster à souhait la disposition des éléments et d'associer un contrôle à des variables Java est simple. Ces éléments nous permettront de réaliser l'application Java (SIREN) regroupant les composantes liantes des briques de notre système.

4.2.6.3. GDAL API

GDAL est une librairie open-source très répandue dans le monde de la géomatique. Elle est disponible dans de très nombreux langages. Cette librairie permet principalement de faciliter la gestion des formes géométriques. Dans notre cas, nous l'utiliserons avec JAVA afin de re-projeter les coordonnées GPS au format Lambert-93 (Système officiel en France) avant de peupler la base de données.

4.2.6.4. Bibliothèque Ardusub

Pour la partie C++, ce framework définit l'ensemble du protocole d'interrogation des capteurs et de déplacement du drone. De plus, après analyse du contenu, il semble que celle-ci soit modulable et permette l'implémentation/ajout de nouveaux capteurs.

Elle inclut également une autre bibliothèque très importante qui se trouve également en java : celle du protocole de communication MAVLink, le protocole de communication entre notre système et l'application en JAVA que nous allons implémenter.

Ce protocole est détaillé plus haut, dans la partie matérielle physique, donc nous ne reviendrons pas dessus ici.

Concernant QGroundControl, l'application utilisée pour visualiser la position du drone, elle utilise une bibliothèque spécifique nommée SDL, qui est également un Framework mais en C cette fois. C'est une bibliothèque utilisée pour des développements en C d'applications graphiques en 2D, comme des émulateurs, des jeux 2D… qui reste fonctionnelle quand utilisée dans des projets en C++. QGroundControl étant une application développée avec Qt, elle est codée en C++. De plus, c'est une application graphique en 2D, donc il est normal qu'elle utilise la bibliothèque SDL.

4.2.7. API et UT (Unit Test)

Afin de mener à bien la partie Java du projet, nous aurons besoin d'utiliser des librairies diverses afin de nous servir des fonctionnalités tierces.

Parmi ces API, nous listons :

- MAVLinkJava (Github/MAVLinkJava)
    - Cette librairie nous permet, à partir d'un code Java, de manipuler les paquets MAVLink. Nous l'avons testée et sommes sûrs de son fonctionnement. Ces manipulations peuvent s'effectuer dans deux sens :
        - **En lecture** : en passant par un reader déjà implémenté, nous pouvons récupérer la nature ainsi que le contenu présent dans les messages du protocole MAVLink. Cette fonctionnalité nous permettra de traiter correctement les données qui émanent du drone.
        - **En écriture** : la librairie dispose d'une partie génératrice, nous permettant de créer des messages MAVLink. Cette fonctionnalité induira la simulation des messages venant du drone, ce qui va s'avérer indispensable afin de mettre en place l'environnement de simulation, par le biais d'une autre application étant limitée à l'envoi de simples messages.
- JUnit (version 4)
    - Cette librairie propose des fonctionnalités nous permettant de tester notre code et ainsi fiabiliser celles hébergées dans nos applications (traitement et simulation). L'écriture des classes de test devra s'effectuer lors du processus de développement avant la rédaction des composantes associées Java suivant un modèle TDD (Test Driven Development). Ce modèle permettra de valider le cas initial d'utilisation ainsi que prévenir les erreurs au cours de l'exécution.
    - L'exécution de ces fichiers de test se fera dans un premier temps sur l'environnement de développement (lors de la compilation) et dans un second temps sur le serveur d'intégration continue qui pourra transmettre ses rapports en cas d'échecs de l'exécution des tests unitaires.

4.2.8. IDE

4.2.8.1. Java

L'outil de développement en langage Java que nous utiliserons durant les phases de rush sera IntelliJ IDEA. Il s'agit d'un environnement complet nous permettant de mener à bien toute la réalisation de l'application concernée.

L'ergonomie et le panel de ses fonctionnalités en font un outil parfait pour un développement en Java version 8.

4.2.8.2. C++

Pour le code écrit en C++, nous allons utiliser l'environnement de développement CLion développé par JetBrains. Cet IDE nous rendra plus facile l'écriture de code grâce à ses fonctionnalités diverses. Premièrement, il est aisé à prendre en main pour ceux parmi nous qui ne l'auraient jamais utilisé, il possède un bon outil de génération et de refactorisation de code, il analyse à la volée les erreurs de frappes ou de syntaxe et possède tous les outils nécessaires au debug, au build et à l'intégration du code.

Nous utiliserons en plus la bibliothèque multiplateforme QT pour la création d'interfaces graphiques. Nous créerons des fenêtres qui seront superposées au logiciel QGControl. Ces fenêtres auront pour fonctionnalités d'aider les utilisateurs dans leur mission en indiquant diverses informations comme l'état des capteurs, l'état des applications ou la durée de la plongée en cours.

4.2.8.3. Python

L'environnement de développement choisi afin de pouvoir écrire du code en Python est PyCharm, il s'agit d'un autre outil du fournisseur logiciel Jetbrains (comme InteliiJ IDEA). Son choix est conditionné par notre expérience de ce type d'IDE ainsi que nos familiarités nous permettant de prendre corps avec l'environnement plus aisément. Cet outil nous aidera à réaliser la portion de notre système correspondant au plugin QGis.

Afin de pouvoir utiliser pleinement les fonctionnalités de QGis au travers du langage Python, il est d'abord nécessaire d'effectuer une configuration appropriée.

4.3. Documentation

Nous pouvons différencier plusieurs types de documentations.

La première documentation sera écrite avant le début du développement du projet. Celle-ci permettra de construire le code de notre projet. Une partie de ces documents ont commencé à être rédigé, à savoir les cahiers des charges fonctionnelle, technique ou graphique. Mais ceux-ci ne suffiront pas à réaliser le projet. En effet nous devrons détailler chaque lot à réaliser avant le début du développement afin de faciliter le fractionnement en tâche de ces lots. Bien sûr, ces documents seront sujet à modification durant les cinq semaines du projet.

Durant le projet, en plus de modifier les documents réalisés, nous devrons produire une documentation expliquant comment créer et ajouter un nouveau module dans l'interface de notre application. L'ONEMA pourra ainsi intégrer de nouveaux capteurs et surveiller ces derniers au travers notre application : SIREN. En plus du précédent document, notre code sera commenté, et la javadoc sera générée à chaque livraison de l'application.

Une fois les trois semaines de développement réalisées, nous commencerons à réaliser une dernière documentation qui sera le mode d'emploi de notre solution.

4.4. Gestion du développement

4.4.1. Convention de nommage

Nous allons définir plusieurs conventions pour pouvoir garder une homogénéité à travers l'ensemble du projet et aider chaque développeur à mieux s'y retrouver dans le projet. Ainsi l'ensemble du code sera en anglais avec la convention CamelCase. En revanche l'ensemble des commentaires sera en français. Ces commentaires nous permettront de mieux comprendre le code des autres développeurs mais aussi de pouvoir facilement effectuer des retouches. Cette convention de nommage sera également appliquée, plus généralement, à tous les documents rendus par l'équipe, en étendant la partie rédigée de ces derniers.

De plus nous appliquerons l'architecture MVC (Modèle Vue Contrôleur) pour le projet. Cette architecture permet de différencier le code en trois parties (modèles, vues, contrôleurs). De plus, la tenue d'un glossaire mis à jour au fur et à mesure de l'avancée du projet, et contenant les noms de tous nos documents, fonctions et classes, facilitera notre travail de relecture et de validation continue.

4.4.2. Branches

Afin de se repérer au mieux dans le projet, l'équipe a décidé de mettre en place un système de branches adapté à la situation. La nomenclature choisie se base sur certains principes et idées provenant de GitFlow (plus de renseignements disponibles (Anglais) : https://www.gittower.com/learn/git/ebook/en/desktop-gui/advanced-topics/git-flow).

Le projet sera découpé en deux branches majeurs : la branche master pour les versions stables de l'application ; les branches modules pour la partie développement d'un module.

La branche Master servira uniquement pour le déploiement de versions stables et pouvant être utilisées par le client. Dès qu'un module est stable (nota : plus de bug et répondant aux spécifications du module), il est intégré à cette branche. L'intégration dans la branche doit être réalisé avec précaution. En effet, il faut s'assurer que le nouveau module ne créera pas de régression. Si des bugs sont découverts sur la branche master, un patch en urgence est déployé sur celle-ci. C'est le seul cas où un développement direct est réalisé sur cette branche principale.

Les branches modules sont des branches orientées développement. C'est sur celles-ci que le développement de composants liés à l'application sont réalisés. La création de ces branches est réalisée en se basant sur la branche Master. Ainsi, le développement réalisé se base sur la version la plus récente de l'application. Une fois le développement du module terminé, la branche est fusionnée avec la branche Master. Ainsi, le développement du module est achevé. Il peut arriver que le développement d'un module reprenne. Pour ce faire, la branche du module se rebase sur la dernière version de l'application (branche Master). Il est ensuite possible de continuer le développement du module, comme si la branche venait d'être créée.

Les branches de modules peuvent être :

- Module simulateur
- Module Position
- Module Mesure
- Module Algorithmes
- Etc…

5. Lots et Livrables

5.1. Lots

La réalisation du projet se déroulera sur cinq semaines du lundi 6 février 2017 au vendredi 10 mars 2017. Les trois premières semaines seront consacrées exclusivement au développement du projet. Ces trois semaines ont été divisées en quatre grandes parties de développement.

5.1.1. Lot 1 : Simulateur et algorithmes

Ce lot est le plus important de tout le projet. En effet, celui-ci sert de base pour tout le reste de la solution que l'équipe va développer. Ce lot est composé de la réalisation du simulateur, ainsi que des algorithmes et fonctionnalités hébergés par le simulateur. Réaliser le simulateur et les fonctionnalités en parallèle permet de s'assurer que le simulateur est capable de tester le bon fonctionnement de l'application, et que l'application est capable de prouver que le simulateur est bien conçu.

Les modules développés sur le simulateur seront :

1. Simulateur de GPS/GNSS/RTK
2. Simulateur de l'Accéléromètre/Gyroscope/Compas
3. Simulateur du capteur de pression
4. Simulateur du capteur de température.

Les fonctionnalités qui seront développées en parallèle du simulateur sont :

1. Algorithme de positionnement
2. Récupération des mesures et association de celles-ci avec une position.

Par conséquent, les cas d'utilisation suivants seront développés :

- UC-1 Simulation : Générer les entrées et les fichiers de simulation
  - 2 personnes à temps plein pendant 4 jours (8 jours-hommes)
- UC-2 Simulation : Effectuer une simulation
  - 1 personnes à temps plein pendant 4 jours (4 jours-hommes)
- UC-3 Simulation : Vérifier la cohérence des données relatives au positionnement
  - 1 personnes à temps plein pendant 2 jours (2 jours-hommes)
- UC-4 Drone : Réaliser une mesure (sur QGis)
  - 2 personnes à temps plein pendant 4 jours (8 jours-hommes)
- UC-5 Drone : Recalibrer les positions (sauvegardées) du drone
  - 1 personnes à temps plein pendant 4 jours (4 jours-hommes)
- UC-6 Drone : Calibrer le décalage / marge d'erreur
  - 1 personnes à temps plein pendant 1 jours (1 jour-homme)

5.1.2. Lot 2 : Intégration des développements dans le drone

Ce lot représente toute l'intégration des développements réalisés dans le lot 1, sur le drone. A la fin de cette étape, l'ensemble des développements orientés drone devra être entièrement fonctionnel sur celui-ci. Le lot intégrera également la partie communication, c'est-à-dire que les données (positions et mesures) devront être communiquées à la station de contrôle.

L'intégration du développement réalisé en lot 1 sur le drone permet de mettre en évidences les différences entre la réalité (mesures et relevés en provenance du drone) et la simulation. En effet, la simulation ne sera pas le reflet de la réalité en fin de lot 1. Le lot 2 permettra donc d'ajuster les différents algorithmes de génération et d'exécution de la simulation en fonction de la « réalité ».

Par conséquent, les cas d'utilisation suivants seront améliorés et intégrés au drone (sauf le simulateur) :

- UC-2 Simulation : Effectuer une simulation
  - 2 personnes à temps plein pendant 4 jours (8 jours-hommes)
- UC-4 Drone : Réaliser une mesure (sur QGis)
  - 2 personnes à temps plein pendant 4 jours (8 jours-hommes)
- UC-5 Drone : Recalibrer les positions (sauvegardées) du drone
  - 2 personnes à temps plein pendant 4 jours (8 jours-hommes)

5.1.3. Lot 3 : Développement de l'interface QGIS

Le présent lot représente la partie interface des mesures. Dans ce lot, toutes les mesures réalisées, ainsi que les positions associées aux relevés, seront affichées. Toute la partie affichage est réalisée sur un fond de carte représentant au mieux l'environnement des mesures.

Le cas d'utilisation concerné par ce lot est :

- UC-4 Drone : Réaliser une mesure (sur QGis)
  - 2 personnes à temps plein pendant 4 jours (8 jours homme).

Le reste de l'équipe aura la charge de stabiliser le code de l'application, les algorithmes et le simulateur (5 personnes à temps plein pendant 4 jours - 20 jours homme)

5.1.4. Lot 4 : Développement des indicateurs sur la console de pilotage

Le dernier lot correspond à l'intégration d'alertes et d'indicateurs au sein de la console de contrôle du drone. Les différents indicateurs ne sont pas caractérisés par des cas d'utilisations. Cependant, ces informations permettent au pilote de connaître l'état de fonctionnement du drone.

Des indicateurs seront donc intégrés aux différentes fonctionnalités et aux différents modules. Ces indicateurs seront ensuite transmis en « temps réel » (moins d'une seconde) à la station de contrôle. Ainsi, toutes les personnes derrière le moniteur de pilotage seront informées du bon fonctionnement (ou non) des mesures et du drone.

Par conséquents, les cas d'utilisation suivants seront mis à jour :

- UC-7 : Drone : Tester l'algorithme de positionnement
  - 3 personnes à temps plein pendant 4 jours (12 jours-hommes).
- Ajout d'indicateur(s) de fonctionnement (précision de la position calculée …) du drone

- 4 personnes à temps plein pendant 4 jours (16 jours-hommes).

La quatrième semaine sera consacrée à la livraison de la version beta de notre produit et à son amélioration suite aux remarques de notre client. Nous finirons la dernière semaine par la livraison de notre release avec l'ensemble des documents rédigés. De plus nous en profiterons pour tester en milieu aquatique notre solution.

Afin de s'assurer du bon fonctionnement du drone et des algorithmes, des tests en environnement naturel (nota : dans un lac ou autre environnement aquatique) seront réalisés.

5.2. Applications livrées

5.2.1. Archive JAR du simulateur/générateur de plongée

Dans notre archive JAR pour le Simulateur/Générateur, il y aura l'ensemble de l'implémentation du simulateur et du générateur de données de plongée. Donc il contiendra les packages Virtualizer et File, qui sont utiles pour la simulation (Ces packages ont été détaillés dans les parties 3.2.6 File et 3.2.10 Virtualizer).

Comme c'est un JAR, il pourra être utilisé pour lancer une simulation ou générer un jeu de données test. Pour cela, il suffira de l'exécuter et de générer un fichier de simulation (UC-1 : Générer les entrées et les fichiers de simulation) à partir de données recueillies sur le terrain (UC-7 : Tester l'algoritme de positionnement), d'effectuer une simulation du fichier généré (UC-2 : Effectuer une simulation) et/ou de vérifier la cohérence des résultats de positionnement (UC-3 : Vérifier la cohérence des données relatives au positionnement) dans le cas où l'on effectue une simulation.
Son utilisation étant purement destiné à des fins de tests, il est donc impossible de s'en servir pour effectuer des sorties sur le terrain.

5.2.2. Archive JAR de l'application SIREN

Dans notre archive JAR pour l'application SIREN, nous retrouverons l'ensemble du code source nécessaire au fonctionnement de notre application. Donc il contiendra les packages SIREN, Drone (et donc Sensor), MAVLink pour les messages entre l'application et le drone, Network pour envoyer les messages MAVLink, Geo et Worker pour récupérer les données de position et les recalibrer ensuite et Database pour mettre en base les mesures et positions récupérées. Chacun de ces packages, comme pour les packages de l'archive précédente, sont détaillés plus amplement dans la partie 3.2 Diagramme de classe.

Comme c'est un JAR, il pourra être utilisé pour réaliser des plongées avec le drone. Pour cela, il suffit de l'exécuter et de lancer la récupération des mesures des capteurs (UC-4 : Réaliser des mesures (sur QGIS)), mesures qui seront associées à une position recalculée (UC-5 : Recalibrer les positions (sauvegardées) du drone), mais également entrer des valeurs de marge d'erreur prévisionnelles concernant les positions recalculées précédemment (UC-6 : Calibrer le décalage/marge d'erreur).
Pour cette dernière option, elle sera mise à jour lors du recalcul de position, comme la marge d'erreur aura été effectivement calculée sur le terrain.

A noter que l'interface associée à ce JAR est décrite dans le CDCG, partie 2 (Interface de SIREN).

5.2.3. Plugin QGIS

C'est la seule de nos applications livrées qui n'est pas une archive JAR. Il s'agit d'un module développé en Python pour QGIS. Il ne possède pas de package et n'utilise aucun de nos cas d'utilisation de manière directe. Il possède néanmoins une interface décrite dans le CDCG, partie 3 (Interface graphique du plugin QGIS).

Comme c'est un module Python pour QGIS, il faut naturellement utiliser QGIS pour s'en servir. Il apparaît dans la barre de tâches de ce dernier pour pouvoir être lancé facilement. Comme il s'agit un module permettant de visualiser les données récupérées depuis le drone, et que QGIS gère l'affichage des informations, le module n'a pas énormément de fonctionnalités. Il nous permet de choisir une base de données, de préférence une qui contient des données de relevés, puis de choisir quelles données afficher : les données brutes, les données corrigées (obtenues depuis l'UC-5 : Recalibrer les positions (sauvegardées) du drone) ou un delta de ces données. Une dernière fonctionnalité nous permet de visualiser en direct les données envoyées par le drone (pour les données corrigées, il s'agit également des données en direct, mais après recalibrage via l'UC-5).

6. Annexe

6.1. Préparer le Raspberry

Pour préparer le Raspberry pour l'utiliser avec Ardusub, il suffit d'installer une image disque sur sa carte SD. Cette image nous offre le système d'exploitation et tous les paramètres nécessaires pour lancer Ardusub. L'image disque est en téléchargement gratuit sur le site d'Ardusub. Les commandes pour l'installation sont toutes fournies dans le tutoriel. Voici la liste des commandes à entrer pour l'installation de l'image sur la carte SD du Raspberry :

- `df -h`

Cette commande permet de trouver le numéro de disque qui ressemblera à quelque chose comme "/dev/sdd1". Il faut commencer par démonter ce disque.

- `umount /dev/sddX`

Cette commande permet de démonter le disque en remplaçant le X par la valeur obtenu grâce à la commande précédente.

- `sudo dd bs=1M if=~/Downloads/ardusub-raspbian.img of=/dev/rdiskX`

Cette commande permet d'écrire l'image disque sur la carte SD. Une fois ces commandes effectuées, le Raspberry est prêt.

6.2. Installation de Ardusub via le Raspberry

Pour déployer Ardusub sur le Pixhawk, nous aurons besoin du Raspberry. Il sera utile pour flasher le Pixhawk en passant par le Raspberry pour effectuer cette opération. Le Raspberry devra cependant être préparé via les étapes décrites dans le chapitre précédent. L'avantage de cette méthode consiste à ne pas avoir à accéder directement au Pixhawk.

L'opération de flash du Pixhawk nécessite obligatoirement une connexion à internet filaire ou sans fil. Il faut commencer par vérifier que le Raspberry possède les scripts les plus récents grâce à la commande :

```
Ssh pi@192.168.2.2 "git --work-tree=/home/pi/companion --
gitdir=/home/pi/companion/.git pull origin master"
```

Puis, il faut lancer l'updater du firmware Ardusub :

```
ssh pi@192.168.2.2 "/home/pi/companion/RPI2/Raspbian/flash_px4.py -
frame=vectored"
```

Pour finir, nous aurons en sortie un message du type : `if the board does not respond, unplug and re-plug the USB connector.`

```
attempting reboot on /dev/ttyACM0...
```

La procédure de flash du Pixhawk est terminée.

6.3. Logiciels et Versions

6.4. Machine Virtuelle de référence

Extraction du descriptive de la machine de référence :

6.4.1. Configuration

La création de la VM (Virtual Machine - Machine Virtuelle) a été réalisé sur l'hyperviseur VMware.
Celle-ci fonctionne également sur VirtualBox 5.1.12.

La version de l'hyperviseur utilisée pour la création est :

- VMware Workstation 12 Pro
  - Version 12.0.0 build-2985596

La machine à l'origine de la VM est :

- OS : Windows 10 Professionnal (1511) x64
- Processeur : Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 2201 MHz, 2 cœur(s), 4 processeur(s) logique(s)
- RAM : 2 x 4 Go
- Disque : 256 Go de SSD

6.4.2. Configuration physique

- **OS** : Windows 8.1 Enterprise en Français
- **CPU** : 2vCore
- **RAM** : 4 Go
- **Disque** : 40 Go en « *thin provisionning* »
- **Réseau** : NAT sur la passerelle de l'hôte

6.4.2.1. Réseau

- **Hostname** (*Nom d'hôte*) : AquaDrone
- **hosts** (*C:\Windows\System32\drivers\etc\hosts*)
  - **localhost** : 127.0.0.1

- **aquadrone.local** : 127.0.0.1

6.4.3. Sessions Windows

- Utilisateur **Standard**
  - Nom d'utilisateur : test
  - Mot de passe : *[aucun]*
- Utilisateur **Administrateur**
  - Nom d'utilisateur : Administrateur
  - Mot de passe : admin

6.4.3.1. Logiciels

- Oracle JRE 8u112 x64 (Java 8)
- Firefox ESR 45.6.0 x64 en Français
- PostgreSQL 9.4.10 x64 par EDB (Entreprise DB)
- PostGIS (pour PostgreSQL 9.4) 2.1.8-1 x64
- pgAdmin 1.20.0 (revision REL-1_20_0)
- QGroundControl 3.0.2
- QGIS 2.18.2-1 en Français
- Python 2.7.5 x64
- Wireshark 2.2.3

6.4.3.2. Base de données

- PostgreSQL 9.4.10
- **Hostname** : localhost
- **Port** : 5432
- Utilisateur **Administrateur**
  - Nom d'utilisateur : postgres
  - Mot de passe : postgres
- Utilisateur de **Production**
  - Nom d'utilisateur : production
  - Mot de passe : production
- Utilisateur de **Test**
  - Nom d'utilisateur : test
  - Mot de passe : test
- **Bases** :
  - production (*pour la production*)
  - test (*pour les tests - ex : tests unitaires*)
- **Extensions** :
  - postgis
  - postgis_topology