Track&Roll

Outil pour le suivi d'activité physique de sportifs de haut niveau

Conception Cervo

14/01/2018

Porteur du Projet
Geoffroy Tijou

Référent Pédagogique
Sébastien Aubin

Chef de Projet
François d'Hotelans

Equipe
Marc de Bentzmann
Benoit Ladrange
Guillaume Muret
Antoine de Pouilly
Angéla Randolph

Table des matières

Introduction

Ce document a pour objectif d'expliquer l'architecture du code qui a été implémentée pour l'objet Cervo. Le code sera compilé puis exécuté sur la carte mère (carte BeagleBone). Ce document traitera, dans un premier temps, de l'architecture du projet avec l'explication des différents packages ainsi que des classes qui les composes, puis, dans un second temps, de la mécanique de traitement des événements qui a été implémentée. Cette conception se terminera par une explication de la communication du système avec les différents objets.

I. Architecture du projet

Le projet s'ouvre avec IntelliJ IDEA et s'articule comme suit :

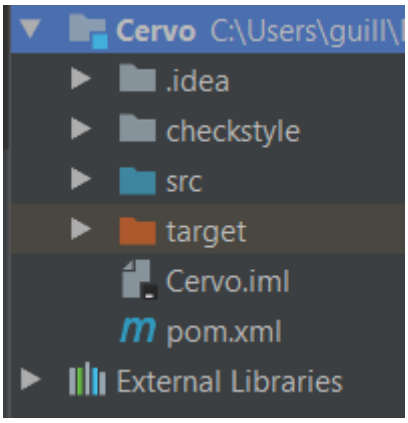

Figure 1: architecture générale du projet

On constate le premier répertoire « .idea » qui est le package de définition du projet interprétable par IntelliJ. Le second répertoire « checkstyle » est utilisé avec le plugin « checkstyle-IDEA » et exprime les différentes règles d'implémentation du code. Lors de la configuration du plugin, il faudra spécifier que ce fichier doit vérifier si le style d'écriture du code respecte les règles préalablement définies. Le répertoire source contient tout notre code source. Le package « target » contient toutes les classes compilées ainsi que l'exécutable « Cervo.jar » qui sera exécuté sur la carte mère. Le fichier « Cervo.iml » défini quelle version de java sera utilisée et avec quelles dépendances sera généré le .jar exécutable. Le fichier « pom.xml » est le fichier de configuration pour utiliser Maven.

A. Architecture du dossier « .idea »

Le dossier .idea se présente comme suit :

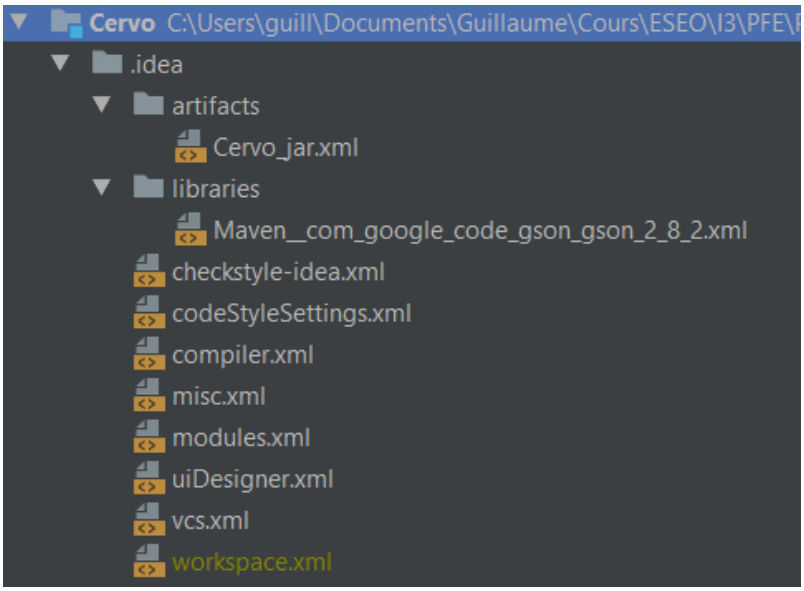

Figure 2: architecture du dossier ".idea"

Le dossier « .idea » contient différents sous dossiers. Le dossier « artifacts » contient le fichier de configuration de l'exécutable « .jar » qui sera exécuté sur la carte mère.
Le dossier « libraries » contient les fichiers générés par Maven des différentes librairies qui sont importées par celui-ci. Dans notre cas, ce fichier xml contient la configuration de la librairie permettant d'utiliser Gson (mapper permettant de convertir une chaine Json en un model Java et inversement).
Concernant les autres fichiers, vous pouvez consulter la documentation sur l'architecture par défaut d'un projet sous IntelliJ.

B. Architecture du dossier « src »

Ce dossier contient l'ensemble du code source qui sera compilé et exécuté sur la carte mère.

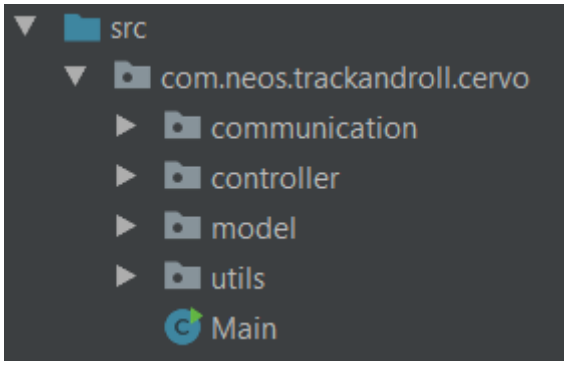

Figure 3: architecture du dossier "src"

On constate un package racine appelé « com.neos.trackandroll.cervo » qui contient différents sous packages ainsi qu'une classe « Main ». Cette classe « Main » possède la méthode « public static void main » qui est appelée en première pour exécuter le code. Cette classe est appelée le « launcher » du projet. Les packages sont expliqués dans les parties suivantes :

1. Package « model »

Le package model contient tout le model utilisé dans le projet et se présente comme suit :

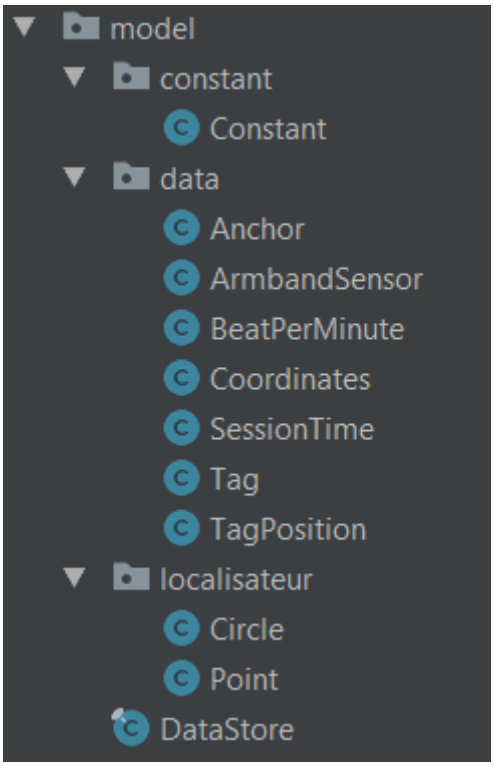

Figure 4: architecture du package "model"

On constate une classe « DataStore » qui est un singleton et qui permet de stocker des variables globales, notamment les données des points reçus par l'algorithme de trilatération, les données des pulsations cardiaques sur une plage de 10 secondes, les tags et capteurs de pulsation cardiaque présents sur le réseau et le temps d'une session.

Le package « constant » contient la classe « Constant » qui, elle, contient toutes les constantes du système et donc ne possède aucune fonction.

▪ Package « data »

Le package « data » contient les objets qui seront utilisés tout au long d'une session ou bien utilisés pour calibrer le système. Ainsi, la classe :

- « Anchor » contient les données associées à une antenne ainsi que les données pour calibrer les antennes dans l'espace.
- « ArmbandSensor » contient les données associées au capteur de pulsation cardiaque.
- « BeatPerMinute » contient les données associées au nombre de pulsations cardiaques reçu sur une plage de 10 secondes et multiplié par 6 pour avoir la donnée en bpm.
- « Coordinates » contient toutes les données associées aux coordonnées d'une antenne dans l'espace. Cette classe est utilisée pour calibrer les antennes avec notamment la méthode « setPos » du système OpenRTLS.
- « SessionTime » contient deux données : l'heure du début d'une session et l'heure de fin d'une session pour connaitre la durée totale de celle-ci.
- « Tag » contient toutes les données associées à un tag sur le réseau lors d'une réception de message.
- « TagPosition » est la donnée utilisée pour stocker le point reçu avec l'algorithme de trilatération.

Ces classes peuvent aussi être utilisées pour convertir les messages reçus lors des différentes communications.

▪ Package « localisateur »

Ce package contient toutes les données associées à l'algorithme de trilatération. On considère les classes :

- « Circle » qui contient les différents cercles construits, dont le centre est défini lors de la calibration des antennes et le rayon correspond à celui reçu par la communication, lorsqu'un tag a envoyé une position. L'antenne n'envoie qu'un simple cercle passant par le tag et ayant pour centre ladite antenne.
- « Point » correspond aux points utilisés par l'algorithme de trilatération.

2. Package « controller »

Ce package permet de gérer l'ensemble des données du système. Un pattern commande a été implémenté pour permettre la gestion des événements reçus par les communications. Le package « controller » se présente comme suit :

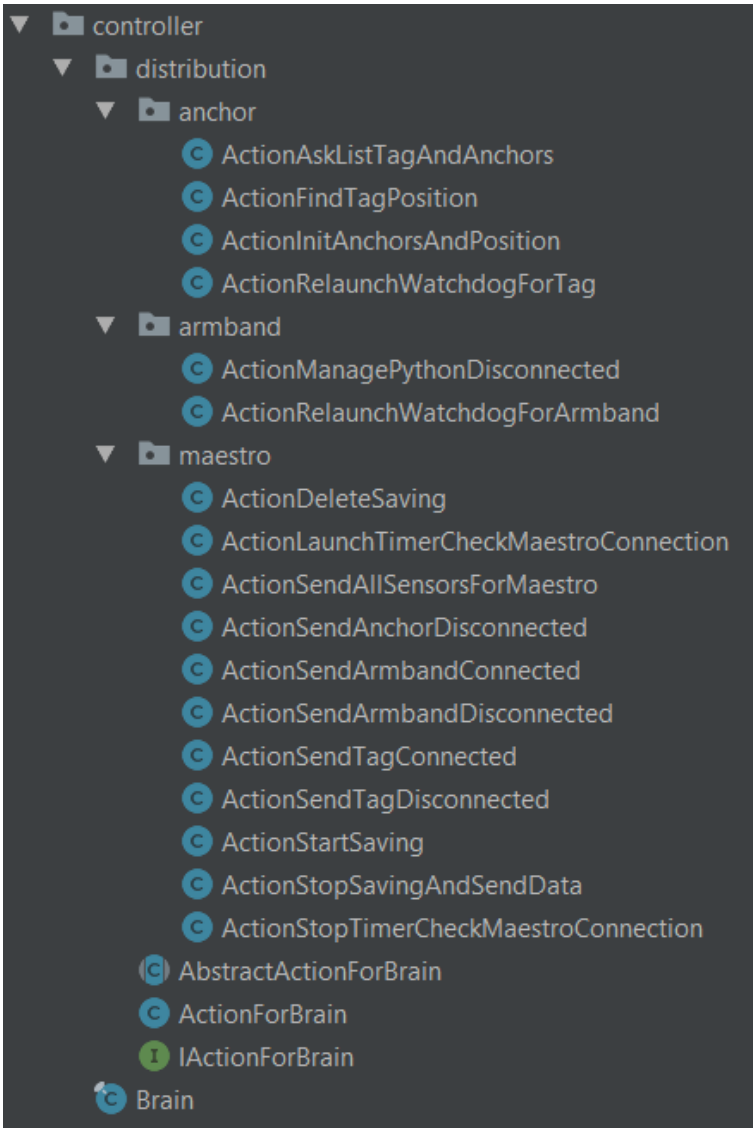

Figure 5: architecture du package "controller"

Ainsi, ce package « controller » contient un objet « Brain » qui est un singleton et qui est utilisé par le système pour gérer l'ensemble des données. L'explication de cet objet et de son architecture se fera en partie II.

▪ Package « distribution »

Le package « distribution » contient trois types de classes différents :

- « AbstractActionForBrain » qui est une classe abstraite représentant le modèle pour les différentes actions qui seront effectuées par le « Brain ».
- « ActionForBrain » est une classe contenant l'ensemble des identifiants des actions à effectuer par le « Brain ». Une « ActionForBrain » contient les données relatives à cette action (identifiant et paramètres).
- « IActionForBrain » est une interface qui est implémentée par la classe « AbstractActionForBrain » et donc implémentée par toutes les actions.

Ces actions énoncées précédemment se trouvent dans les sous packages « anchor », « armband » et « maestro ». Les actions relatives aux antennes se trouvent dans le package « anchor », les actions relatives aux capteurs de pulsation cardiaque se trouve dans les sous package « armband » et les actions relatives à la communication avec le système Android se trouve dans le package « maestro ».

3. Package « communication »

Le package « communication » contient toutes les classes permettant d'implémenter les différentes communications. Le package communication se présente comme suit :

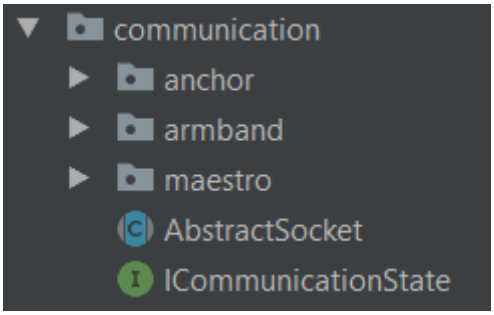

Figure 6: architecture du package "communication"

Ce package permet de rassembler les différentes communications en fonction du type d'objet avec lequel communiquer. L'interface « ICommunicationState », implémentée par le « Brain » et possédée par les instances de communication des différents objets permet de notifier le « Brain » lors d'événements majeurs liés à la communication (connexion et déconnexion). La classe « AbstractSocket » est utilisée pour les communications devant avoir un socket de connexion et un socket de communication.
Dans l'ensemble des sous package « anchor », « armband » et « maestro », l'architecture des communications respecte un pattern générique et contient différents objets de même nature. Ainsi, une instance appelée « XXXCommunication » est l'instance principale gérant un « XXXPostman », un « XXXDistributor » et un « XXXProxy ». Le « XXXPostman » gère les différents sockets et s'occupe de lire et d'écrire sur le socket de communication. Le « XXXDistributor » possède une liste de différents « ProcessXXX » convertissant les messages reçus en modèle (avec Gson) pour être interprétés. L'interprétation de ces messages permet de retourner des actions afin qu'elles soient exécutées par le « Brain ».

- **Package « maestro »**

Le package « maestro » est le plus fourni des packages de communication. En effet, celui-ci se présente comme suit :

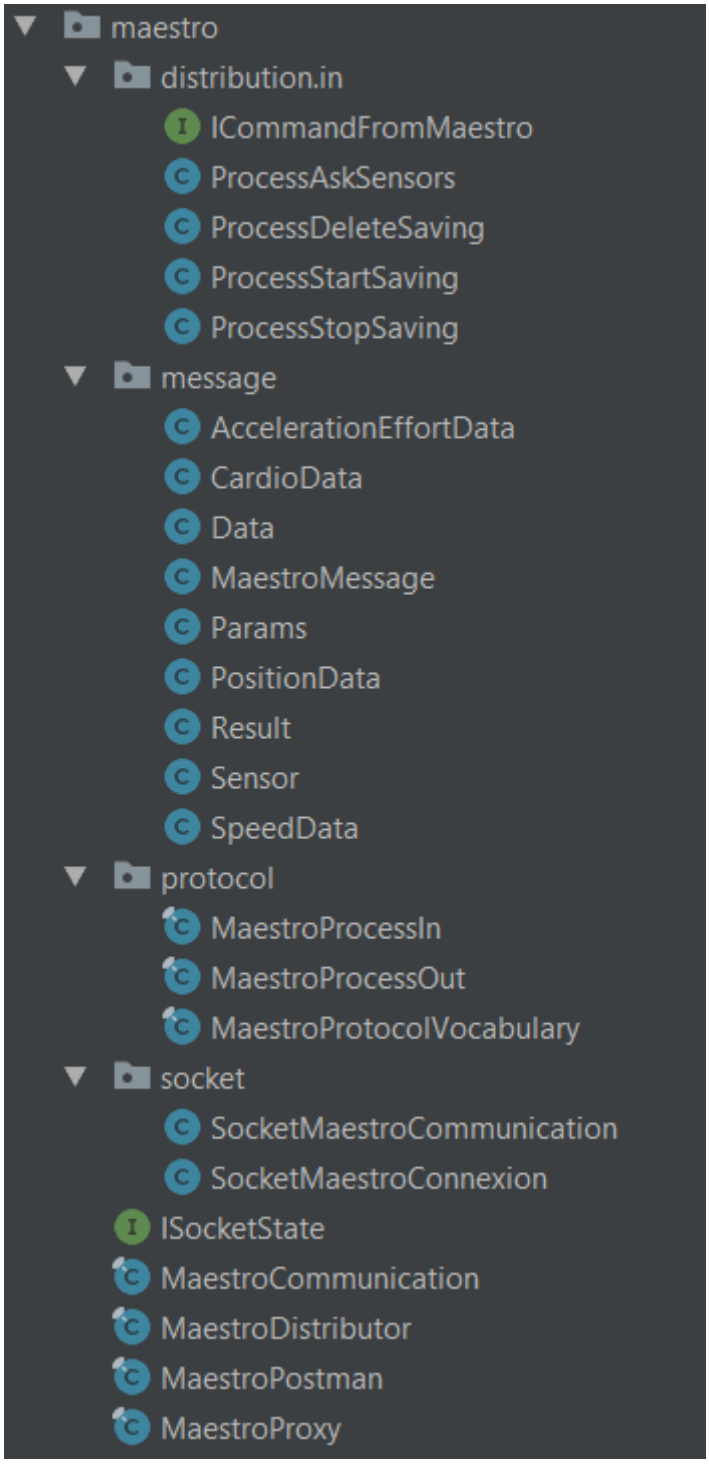

Figure 7: architecture du package "maestro"

Ce package contient les différentes classes génériques au pattern de communication :

- « ISocketState » est une interface permettant de notifier le « MaestroCommunication » lors d'une connexion ou déconnexion du système.
- « MaestroCommunication » est un singleton qui gère les différents objets de la communication (expliqué précédemment).

- « MaestroDistributor » est un singleton permettant d'appeler les différents convertisseurs pour les messages circulant de maestro vers cervo.
- « MaestroPostman » est un singleton s'occupant de gérer la communication sur les différents sockets se trouvant dans le sous package « socket ».
- « MaestroProxy » est un singleton gérant la conversion des messages allant de cervo vers maestro.

Sous package « distribution.in »

Ce sous package contient les différents convertisseurs des messages allant de maestro vers cervo. Ces convertisseurs implémentent l'interface « ICommandFromMaestro » pour permettre au distributor de n'avoir qu'une simple méthode à appeler (dispatch) pour la conversion des messages. Tous les convertisseurs retournent différentes actions qui seront ensuite envoyées au « Brain » pour être interprétées et exécutées. Voici les différentes actions :

- « ProcessAskSensor » permettant de connaitre l'état des capteurs sur le réseau.
- « ProcessDeleteSaving » notifiant le Cervo que la session en cours ne doit pas être sauvegardée.
- « ProcessStartSaving » notifiant le Cervo que la session vient de commencer et qu'il faut enregistrer les données.
- « ProcessStopSaving » notifiant le Cervo que la session vient de se terminer et qu'il faut envoyer les données à la tablette.

Sous package « message »

Ce sous package contient l'ensemble des types de messages pouvant transiter entre cervo et maestro. Ces différents messages ont une racine de type « MaestroMessage » qui est le message générique de l'ensemble des messages. Dans l'ensemble de ces classes, les attributs possèdent une annotation appelée « @SerializedName » permettant de spécifier la clé qu'aura la donnée dans le Json. La valeur associée à cette clé sera la valeur de l'attribut. Pour plus d'information concernant la conversion Json en modèle ou de modèle à Json, la documentation sur Gson sera fortement utile.

Sous package « protocol »

Ce sous package contient l'ensemble des chaines de caractères utilisées dans cette communication. En effet, ces 3 classes ne contiennent que des constantes :

- « MaestroProcessIn » ne contient que les noms des commandes allant de maestro vers cervo.
- « MaestroProcessOut » ne contient que les noms des commandes transitant de cervo vers maestro.
- « MaestroProtocolVocabulary » ne contient que les clés qui sont utilisées par la communication lors d'envoi ou de réception de messages.

- **Package « anchor »**

Ce package contient toutes les classes de communication avec l'antenne mère. Il se présente comme suit :

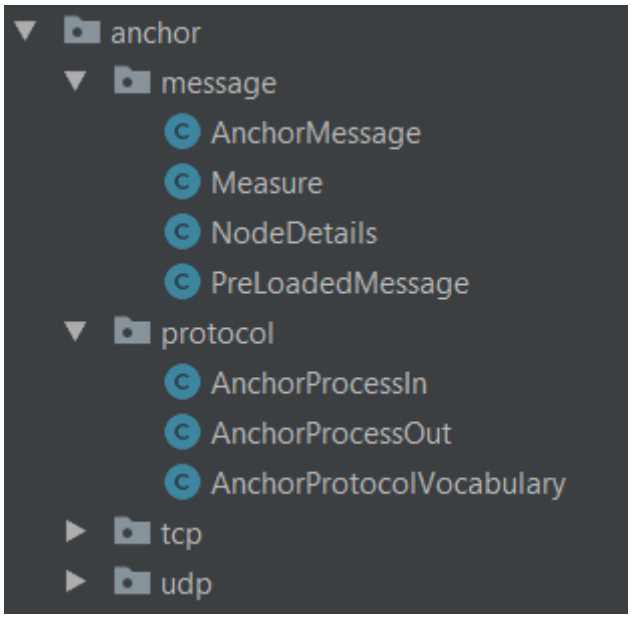

Figure 8: architecture du package "anchor"

On constate que les mêmes packages que le protocole précédent sont présents dans celui-ci, notamment le package « message » et le package « protocol » qui est de même nature qu'expliqué précédemment.
Concernant la classe « PreLoadedMessage », celle-ci contient des messages tout fait destinés à être envoyés à l'antenne mère.

Les deux sous packages « tcp » et « udp » ont une hiérarchie qui respecte le pattern de communication évoqué précédemment.

Sous package « tcp »

Ce sous package contient les instances de communication, distributor, postman et proxy.
Cette communication se présente comme suit :

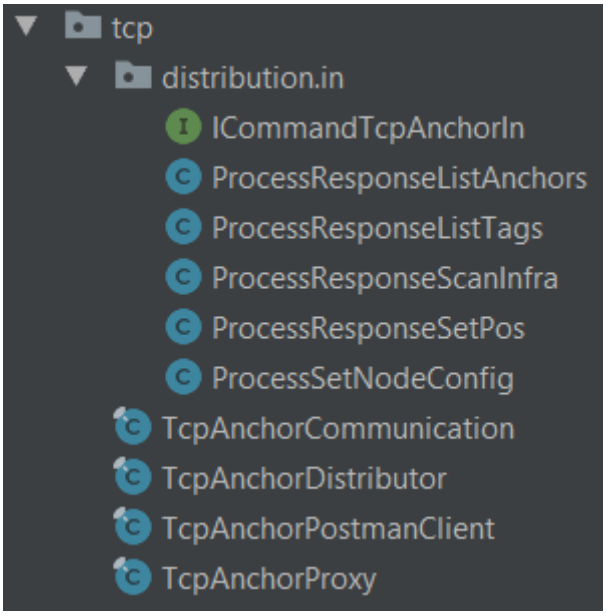

Figure 9: architecture du package "tcp"

Cette communication ne s'occupe que d'envoyer les différentes commandes à l'antenne mère et permet de recevoir les réponses aux commandes pour vérifier si elles ont bien été exécutées.

Sous package « udp »

Ce sous package contient la classe convertissant les messages de l'antenne mère vers le cervo. Cette communication ne gère donc que les messages des tags pour en relever leur position. Voici le package « udp »

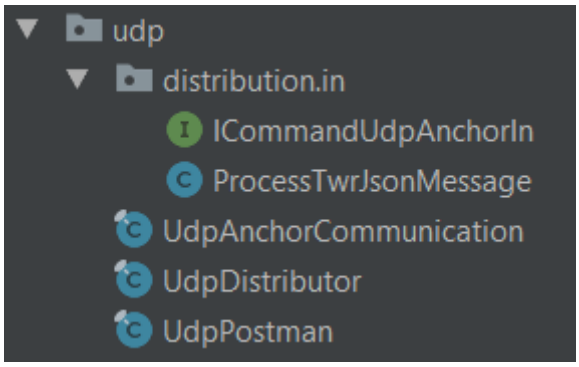

Figure 10: architecture du package "udp"

4. Package « utils »

Ce package contient les différentes classes permettant d'aider dans l'implémentation des différents algorithmes. L'ensemble des classes de ce package

possèdent des méthodes statiques qui peuvent être appelées à n'importe quel endroit du code. Ce package se présente comme suit :

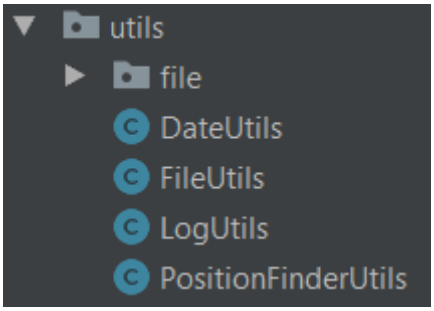

Figure 11: architecture du package "utils"

Dans ce package, on constate :

- « DateUtils » est utilisé pour convertir une date en chaine de caractère ou inversement ainsi que d'autres fonctions associées aux dates.
- « FileUtils » est utilisé pour générer ou lire des fichiers.
- « LogUtils » est utilisé pour écrire des messages de debug ou d'erreur sur la console.
- « PositionFinderUtils » contient toutes les fonctions en rapport avec l'algorithme de trilatération.

▪ Explication de l'algorithme de trilatération

Les données envoyées par les antennes au système pour déterminer la position d'un joueur à un instant donné correspondent à un ensemble de cercles.

Un cercle est défini par un centre : [coordonnée en x, coordonnée en y] et un rayon (toutes les valeurs étant en mètres). Chacun de ces cercles correspond à une mesure de la position du joueur réalisée par une antenne. Le centre du cercle correspond à la position de l'antenne qui a pris la mesure, et le rayon, à la distance du joueur par rapport à cette antenne.

A partir d'un certain nombre de cercles (3 au minimum), on peut être en mesure de déterminer la position du joueur grâce à un algorithme de trilatération. Les distances étant imprécises (à cause des erreurs de mesures et du risque de perturbation des signaux par des obstacles), il est utile d'avoir un maximum de cercles à disposition pour que la mesure soit la plus précise possible. Ainsi, plus il y a d'antennes disposées autour du terrain, plus la position du joueur sera précise.

Pour expliquer le fonctionnement de l'algorithme de trilatération, on utilisera un exemple de données telles qu'elles sont envoyées par les antennes au système. Le jeu de données utilisé pour l'exemple est le suivant (ce jeu de données a été obtenu dans la salle Fermi de l'ESEO) :

- C1 : [0, 0] à 8,929m
- C2 : [-6, -1] à 12,528m
- C3 : [8, -1] à 10,344m
- C4 : [-6, 11] à 7,002m

- C5 : [8, 11] à 5,029m

L'antenne mère (C1) se trouve au niveau du bureau du professeur, en (0,0). Les 4 autres antennes (de C2 à C5) sont disposées autour de la salle (respectivement en (-6, -1), (8, -1), (-6, 11) et (8, 11)).

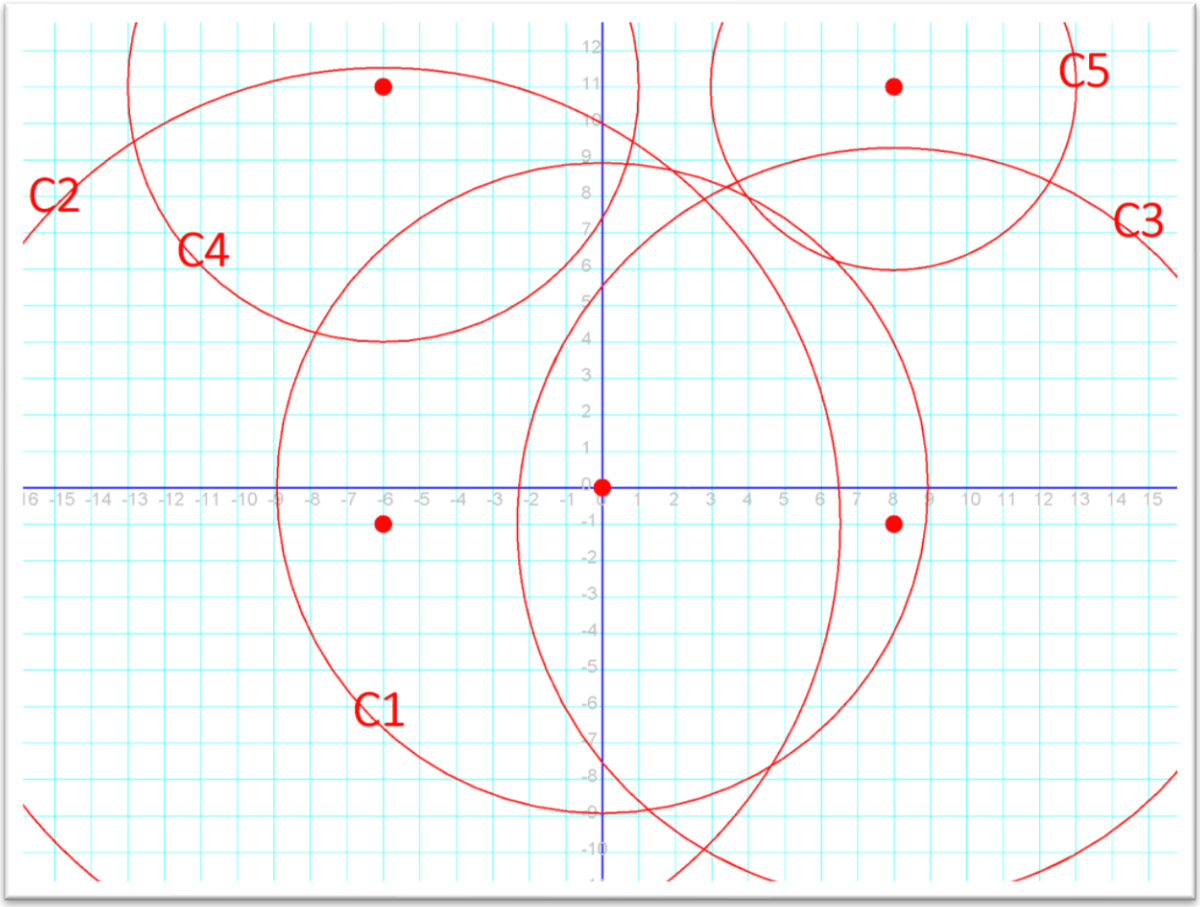

Figure 12: Graphique représentant le jeu de données exemple provenant des antennes

Comme on peut l'observer sur le graphique ci-dessus, les cercles ne se recoupent pas parfaitement en un point unique. L'algorithme de trilatération doit donc essayer d'approximer la position du joueur.

Pour cela, l'algorithme va dans un premier temps prendre chaque couple possible de cercles et déterminer leurs points d'intersections. Avec l'exemple, cela donne donc :

- C1-C2
- C1-C3
- C1-C4
- C1-C5
- C2-C3
- C2-C4
- C2-C5
- C3-C4
- C3-C5

- C4-C5

Il y a 3 types d'intersection possible entre deux cercles.

1) Soit les deux cercles sont éloignés et ne se coupent pas. Dans ce cas, on prendra le point situé à égale distance des deux cercles. Le point sera considéré comme solitaire et sera affiché en jaune dans les représentations graphiques.

Pour prendre en charge le cas où les deux cercles n'ont pas le même rayon, on ne peut pas se contenter de prendre le point se trouvant à égale distance des deux centres.

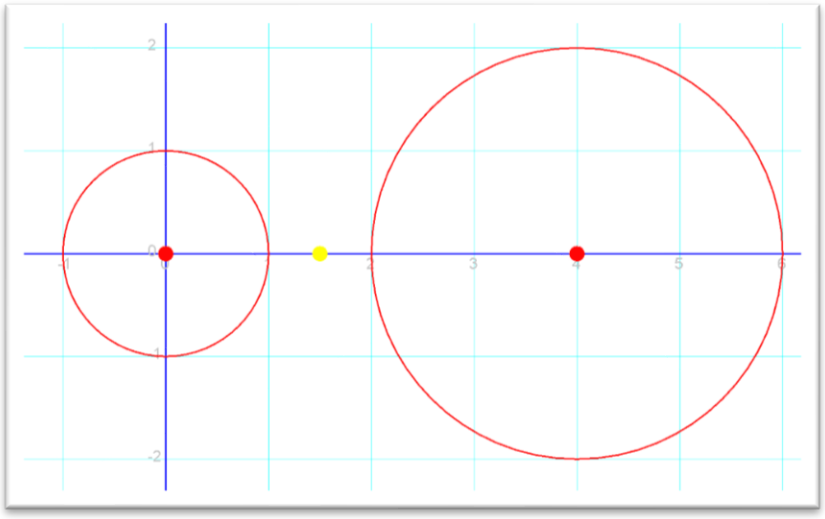

Figure 13: Intersection pour deux cercles ne se touchant pas

Pour déterminer l'emplacement de ce point (le point jaune dans l'exemple ci-dessus), l'algorithme va commencer par déterminer l'équation de la droite passant par les centres des deux cercles (ici, l'axe des abscisses).

Puis, pour chacun des deux cercles, on va chercher les points vérifiant le système d'équations suivant :

$$\begin{cases} Le\ point\ appartient\ au\ cercle \\ Le\ point\ appartient\ \text{à}\ la\ droite \end{cases}$$

On obtient pour chacun des deux cercles deux points :

- En (-1, 0) et en (1, 0) pour le cercle de gauche
- En (2, 0) et en (6, 0) pour le cercle de droite

Pour chacun des deux cercles, on va déterminer quel est le point le plus proche du centre de l'autre cercle. On obtient :

- (1, 0) pour le cercle de gauche
- (2, 0) pour le cercle de droite

Ensuite, on détermine quel est le point se situant au milieu de ces deux points. On obtient finalement le point en jaune se trouvant en (1.5, 0).

2) Soit les deux cercles se touchent en un point. Le point sera considéré comme solitaire et sera affiché en jaune dans les représentations graphiques :

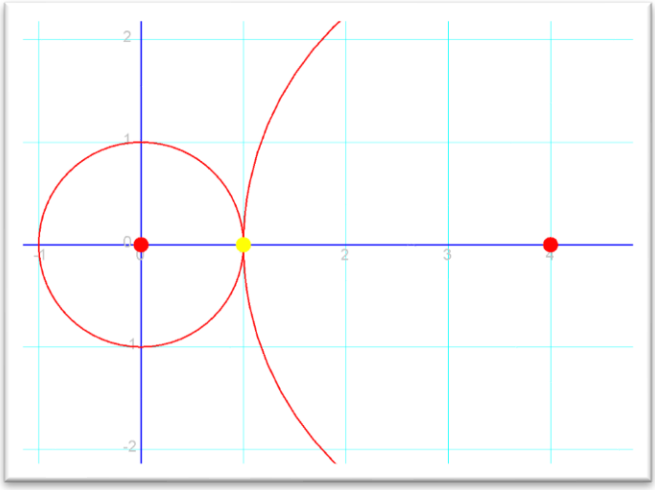

Figure 14: Intersection pour deux cercles se coupant en un point

Dans ce cas, le point doit juste vérifier le système d'équations suivant :

$$\begin{cases} Le\ point\ appartient\ au\ 1\text{er}\ cercle \\ Le\ point\ appartient\ au\ 2\text{ème}\ cercle \end{cases}$$

On obtient finalement le point en jaune se trouvant en (1, 0).

3) Soit les deux cercles se coupent en deux points. Ils seront considérés comme faisant partie de la même famille (un couple) et seront affichés en rose dans les représentations graphiques :

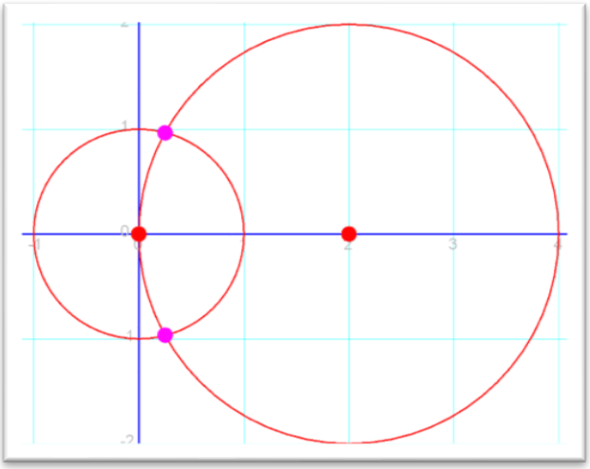

Figure 15: Intersection pour deux cercles se coupant en deux points

Dans ce cas, le point doit juste vérifier le système d'équations suivant :

$$\begin{cases} Les\ points\ appartiennent\ au\ 1\text{er}\ cercle \\ Les\ points\ appartiennent\ au\ 2\text{er}\ cercle \end{cases}$$

On obtient finalement les deux points en rose.

En appliquant ce principe à l'ensemble des cercles deux à deux, on obtient un nuage de points comme présenté sur la figure suivante :

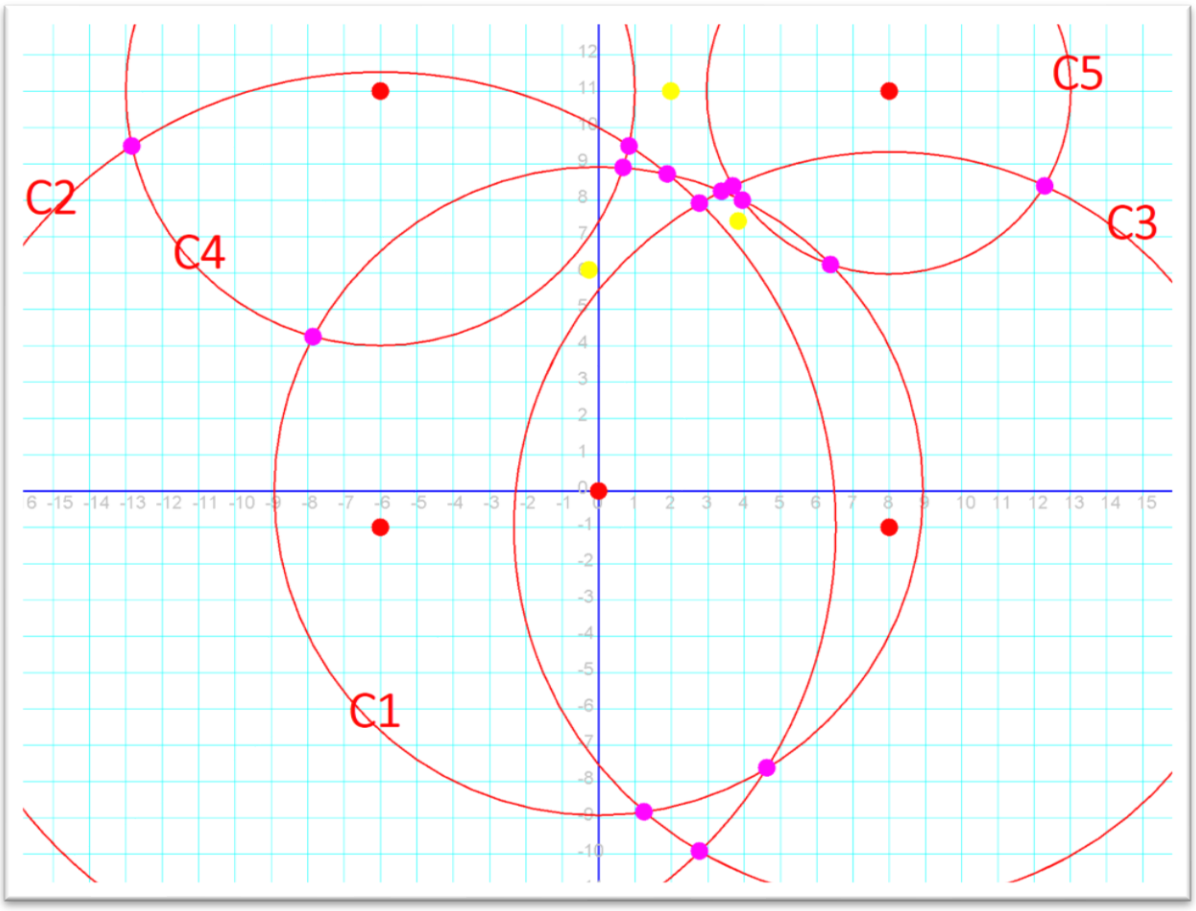

Figure 16: Graphique des points d'intersections des cercles du jeu de données exemple

On peut voir un certain nombre de points solitaires en jaune, ainsi qu'un certain nombre de points en couples en rose.

L'objectif va donc être de déterminer pour chaque couple quel est le point à supprimer et quel est le point à conserver.

Deux cas se présentent alors :

1° Le nuage de points dispose de point(s) « solitaire(s) ».
2° Le nuage de points est constitué exclusivement de points doubles.

Pour le cas 1°, il suffit simplement de trouver le barycentre des points solitaires et déterminer quel point du couple en est le plus proche. Ce sera ce point qu'il faudra conserver.

Pour le cas 2°, il s'agit d'un cas particulier où l'ensemble des cercles coupe individuellement chacun des autres cercles en deux points exclusivement (comme montré dans l'exemple suivant) :

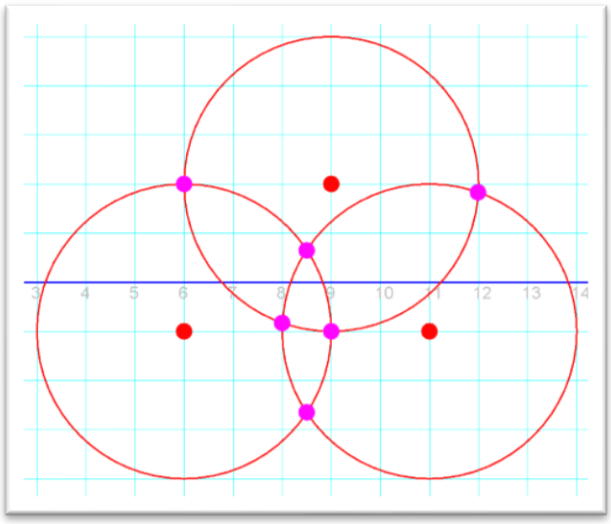

Figure 17: Exemple de cercles sans point solitaire

Dans ce cas, il suffit simplement de ne conserver que les points qui se trouvent sur les trois disques (c'est-à-dire, les points qui se trouvent, vis-à-vis du centre de chaque cercle, à une distance inférieure ou égale au rayon de ce cercle). On obtient alors les points orange suivants :

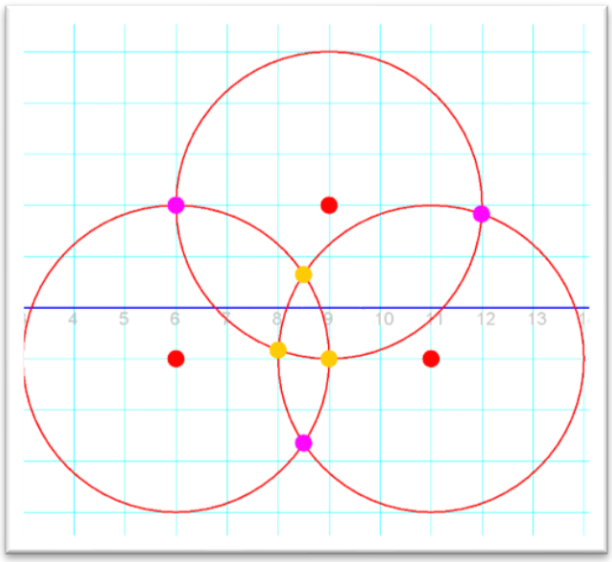

Figure 18: Exemple de détermination des points à conserver

Dans notre exemple, une fois le tri du nuage de point effectué, on ne conservera plus que les points en orange représenté dans le graphique suivant :

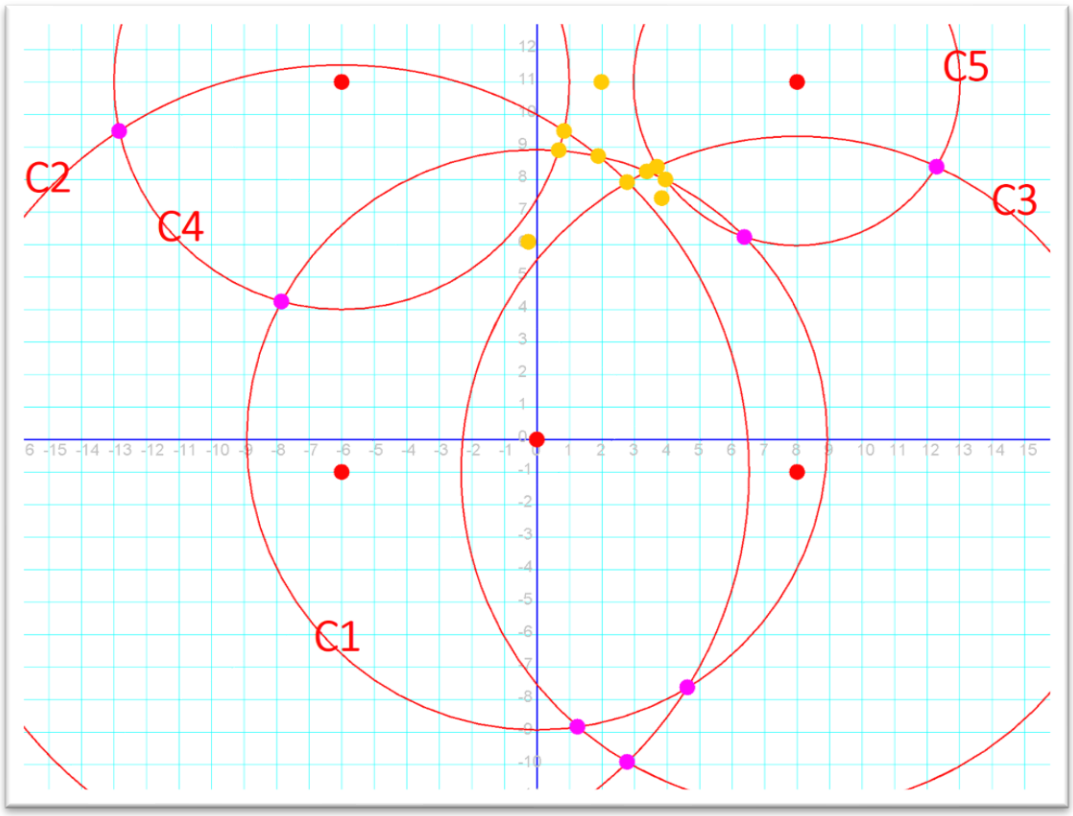

Figure 19: Graphique des points d'intersections triés du jeu de données exemple

Pour déterminer la position du joueur, il ne suffit plus qu'à déterminer le point moyen de ce nuage de point. On obtiendra alors, le point vert du graphique suivant :

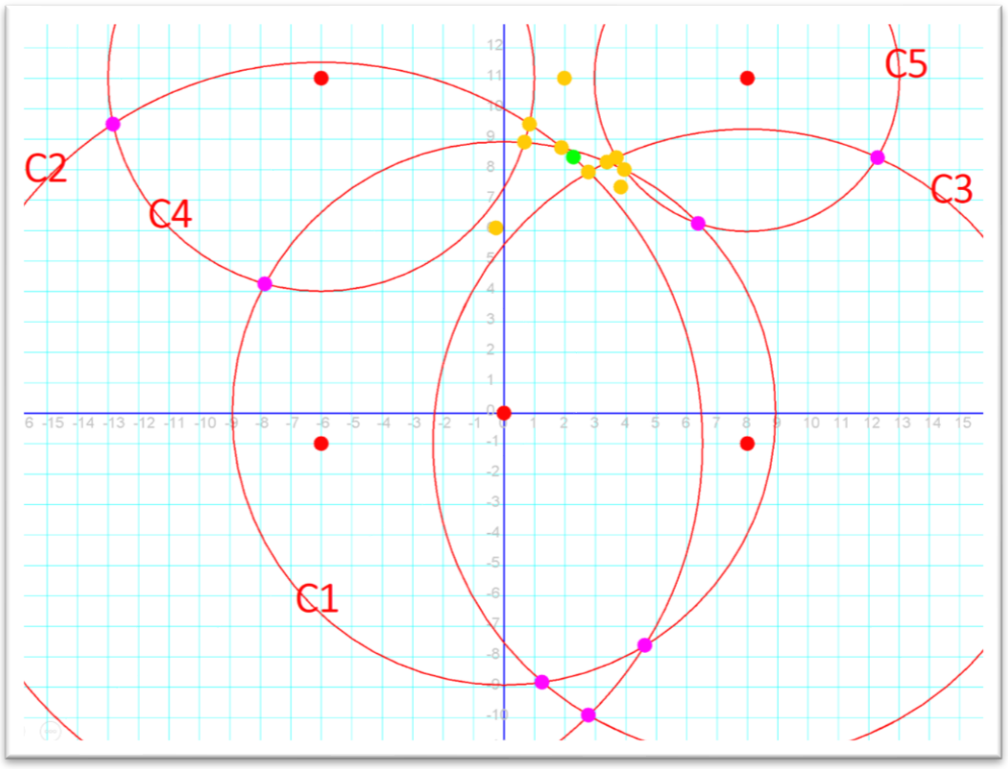

Figure 20: Graphique de la position finale du jeu de données exemple

II. Architecture du « Brain »

La gestion des différentes actions et des différents événements sont réalisés par le « Brain ». L'architecture du « Brain » est représentée comme suit (les classes de communication ne sont pas complètes pour une meilleure visibilité) :

A. Architecture générale

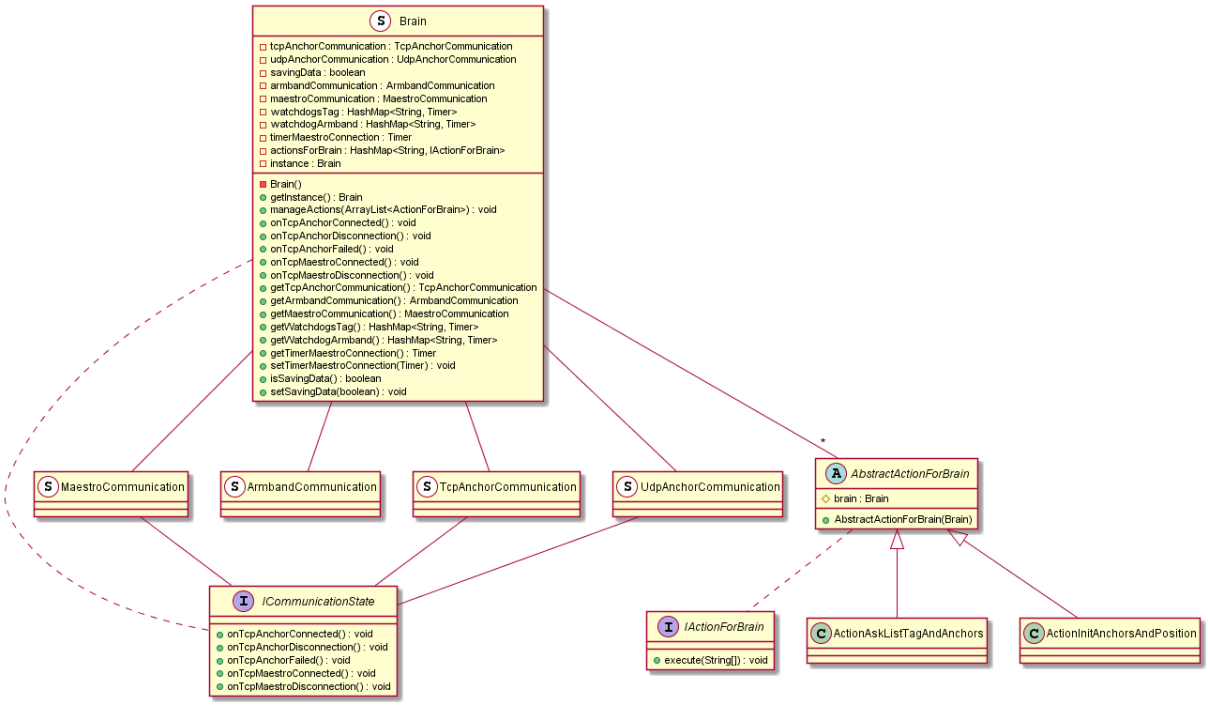

Figure 21: architecture générale du "Brain"

A la création du « Brain », les différents objets de communication sont instanciés, notamment « MaestroCommunication », « TcpAnchorCommunication », « UdpAnchorCommunication » et « ArmbandCommunication ». Le « Brain » du système implémente l'interface « ICommunicationState » qui permet de connaitre l'état des différentes connexions sur la carte mère.
Les différents attributs du « Brain » sont expliqués comme suit :

- Le « tcpAnchorCommunication » permet de gérer la communication TCP avec l'antenne mère.
- L'« udpAnchorCommunication » permet de gérer la communication UDP avec l'antenne mère.
- L' « armbandCommunication » permet de gérer la communication pour récupérer les pulsations cardiaques d'un joueur.

- Le « maestroCommunication » permet de gérer la communication avec l'Android.
- Le boolean « savingData » permet de connaitre l'état du système : soit le système est en train de sauvegarder les données, soit le système n'est pas en train de sauvegarder.
- Le « watchdogTag » est une HashMap ayant pour clé l'identifiant d'un tag et comme valeur un timer. Dès qu'un tag est connecté au réseau, un timer est initialisé et lancé. Si aucun message de ce tag n'est reçu pendant un temps « TIME_MAX_FOR_TAG_POSITION », le tag est considéré comme déconnecté.
- Le « watchdogArmband » est une HashMap ayant pour clé l'identifiant d'un brassard (capteur physiologique) et comme valeur un timer. Dès qu'un capteur est connecté au réseau, un timer est initialisé et lancé. Si aucun message de ce brassard n'est reçu pendant un temps « TIME_MAX_FOR_ARMBAND_BPM », le capteur physiologique est considéré comme déconnecté.
- Le « timerMaestroConnection » est un timer permettant d'envoyer tous les « TIME_FOR_CHECK_MAESTRO_CONNECTION » une requête « checkConnection » pour permettre de savoir si la communication est toujours opérationnelle sur le réseau.
- L'HashMap « actionForBrain » contient l'ensemble des actions à effectuer sur l'ensemble du système en fonction des événements reçu par la communication.

B. Pattern commande

Le pattern commande implémenté pour effectuer les différentes actions se présente comme suit :

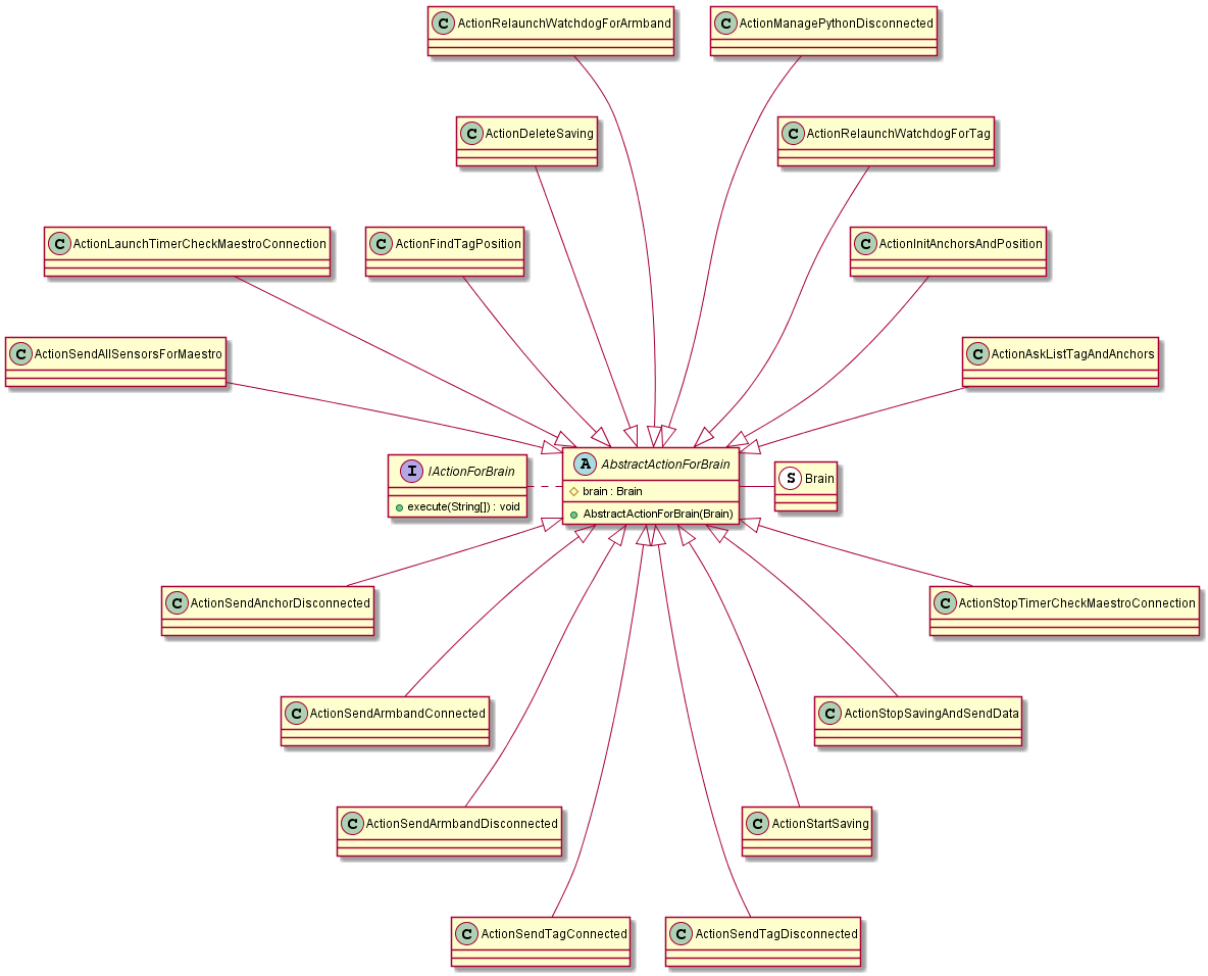

Figure 22: architecture des actions du "Brain"

Toutes les actions héritent de la classe « AbstractActionForBrain » qui possède en attribut le « brain » pour effectuer des actions sur celui-ci.
Les actions se présentent comme suit :

Actions relatives aux antennes et aux tags

Les actions relatives aux antennes et aux tags se trouvent dans le package « controller.distribution.anchor ». Ces différentes actions sont :

- « ActionAskListTagAndAchors » permettant de demander au système UWB d'envoyer l'état des tags et des antennes sur le réseau
- « ActionFindTagPosition » permettant d'appeler l'algorithme de trilatération pour trouver la position d'un tag en fonction des distances renvoyées par les antennes du système.
- « ActionInitAnchorsAndPosition » permet de spécifier la position des antennes sur le réseau, de spécifier la fréquence d'envoi des positions des tags, le scan du système et la liste des antennes et des tags.
- « ActionRelaunchWatchdogForTag » permet de demander au système de relancer le watchdog du tag ayant envoyé son signal sur le réseau.

Actions relatives au capteur de pulsation cardiaque

Les actions relatives au capteur physiologique se trouvent dans le package « controller.distribution.armband ». Ces différentes actions sont :

- « ActionManagePythonDisconnected » permettant de commander à la communication spécifique au « armband » de se relancer.
- « ActionRelaunchWatchdogForArmband » permet de demander au système de relancer le watchdog du capteur ayant envoyé son signal sur le réseau.

Actions relatives à l'Android (maestro)

Les actions relatives à l'Android se trouvent dans le package « controller.distribution.maestro ». Ces différentes actions sont :

- « ActionDeleteSaving » permettant de demander la suppression des données enregistrées
- « ActionLaucnhTimerCheckMaestroConnection » permettant d'initialiser le timer pour vérifier si la connexion avec le système Android est opérationnelle.
- « ActionSendAllSensorsForMaestro » permet d'envoyer toutes les données relatives aux capteurs pour l'Android.
- « ActionSendAnchorDisconnected » permet d'envoyer une notification au système Android lorsque la communication avec l'antenne mère ne fonctionne plus.
- « ActionSendTagConnected », « ActionSendTagDisconnected », « ActionSendArmbandConnected » et « ActionSendArmbandDisconnected » permettent d'envoyer l'état des tags et des capteurs de pulsation cardiaque à l'Android.
- « ActionStartSaving » permet de spécifier au système de sauvegarder les données reçues des capteurs.
- « ActionStopSavinAndSendData » permet de stopper la sauvegarde des données sur le réseau et d'envoyer l'ensemble des données au système Android.
- « ActionStopTimerCheckMaestroConnection » permet de stopper le timer en vérifiant l'opérationnalité de la communication entre le cervo et maestro.

III. Communication du système

La communication du système respecte un pattern générique qui a été utilisé pour l'ensemble des communications. Ce pattern générique a été expliqué en partie II.

A. Communication avec le système UWB

La communication avec le système UWB permet de communiquer avec l'antenne mère pour pouvoir récupérer les positions des tags sur le réseau. Cette communication se trouve dans le package « communication.anchor » dans le projet racine et se divise en deux parties : une communication TCP permettant de calibrer le système UWB et d'envoyer des commandes à l'antenne mère, et une communication UDP permettant de recevoir l'ensemble des données des tags et des antennes présents sur le réseau.

1. Communication TCP

Cette communication s'occupe d'envoyer des commandes à l'antenne mère pour calibrer le système ainsi que de recevoir la réponse à ces commandes. Les commandes sont : le positionnement des antennes, la fréquence d'envoi des tags et le scan du système pour connaitre l'état des différents tags et des antennes présents sur le réseau.

- **Création de la communication et envoi de message**

L'architecture de la communication avec le système UWB se décompose comme suit :

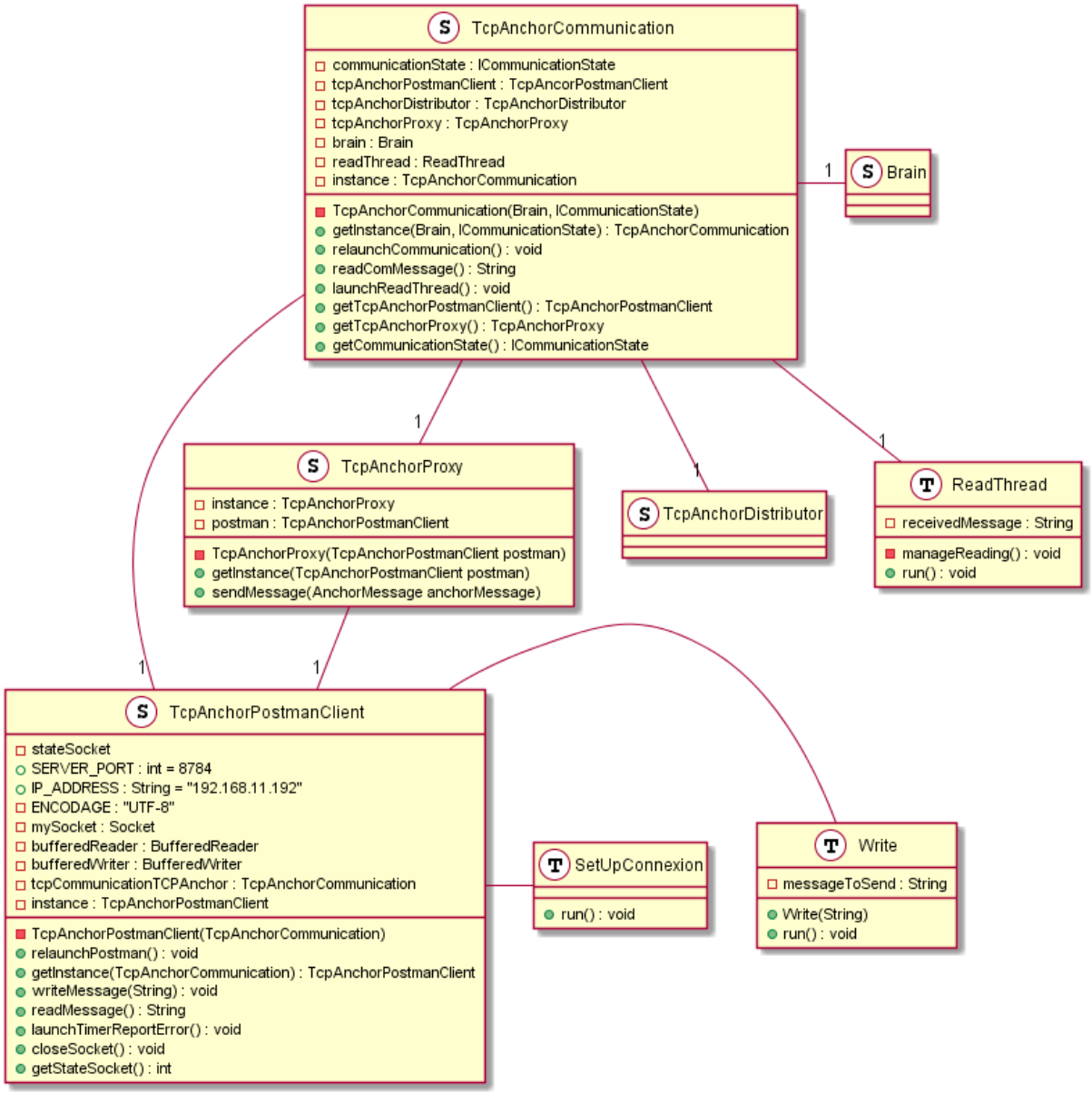

Figure 23: architecture de la communication TCP avec l'antenne mère

Sur cette figure on constate le TcpAnchorCommunication qui est une classe permettant de gérer le TcpAnchorProxy, le TcpAnchorPostmanClient et le TcpAnchorDistributor.

Comme expliqué précédemment, le « TcpAnchorProxy » s'occupe d'encoder les messages de type « AnchorMessage » en chaine Json avec l'outil Gson qui est un mapper permettant d'encoder ou de décoder les objets Json en « model » et inversement. Une fois les messages encodés, le « TcpAnchorProxy » demande au « TcpAnchorPostmanClient » d'envoyer le message sur le socket.
L'objet « TcpAnchorDistributor » s'occupe de gérer les différents messages qui seront lus sur le socket. C'est-à-dire les messages qui seront envoyés par l'antenne mère. Ce point sera expliqué dans la partie suivante.

Le thread « SetUpConnexion » permet de créer un socket client pour se connecter au socket server se trouvant sur l'antenne mère. Le thread « Write » permet au postman de pouvoir écrire sur ce socket pour envoyer des messages à l'antenne mère.

Scénario :

A la création du « TcpAnchorPostmanClient », le socket client se connecte en TCP/IP au système UWB à l'adresse 192.168.11.192. Une fois la connexion établie avec l'antenne mère, le système peut communiquer avec celle-ci L'objet « TcpAnchorCommunication » est notifié que la connexion a été établie et le thread « ReadThread » est créé, permettant ainsi de lire en scrutation les messages se trouvant sur ce socket. Ensuite, la méthode « onTcpAnchorConnected » de l'interface « ICommunicationState » implémentée par le « Brain » est appelée et permet ainsi d'envoyer une commande au « Brain » pour pouvoir initialiser le système. Ceci sera expliqué dans les parties suivantes.

Lors d'une demande d'envoi de message, le système fait simplement appel au proxy qui exécute la méthode « sendMessage » en spécifiant bien en paramètre le message à encoder et à envoyer.

- **Décodage des messages**

Lorsque la communication a été initialisée en TCP, lors de la réception d'un message sur le « ReadThread », les décodeurs du distributor sont appelés en fonction de la commande envoyée. Les décodeurs sont des classes représentées comme suit :

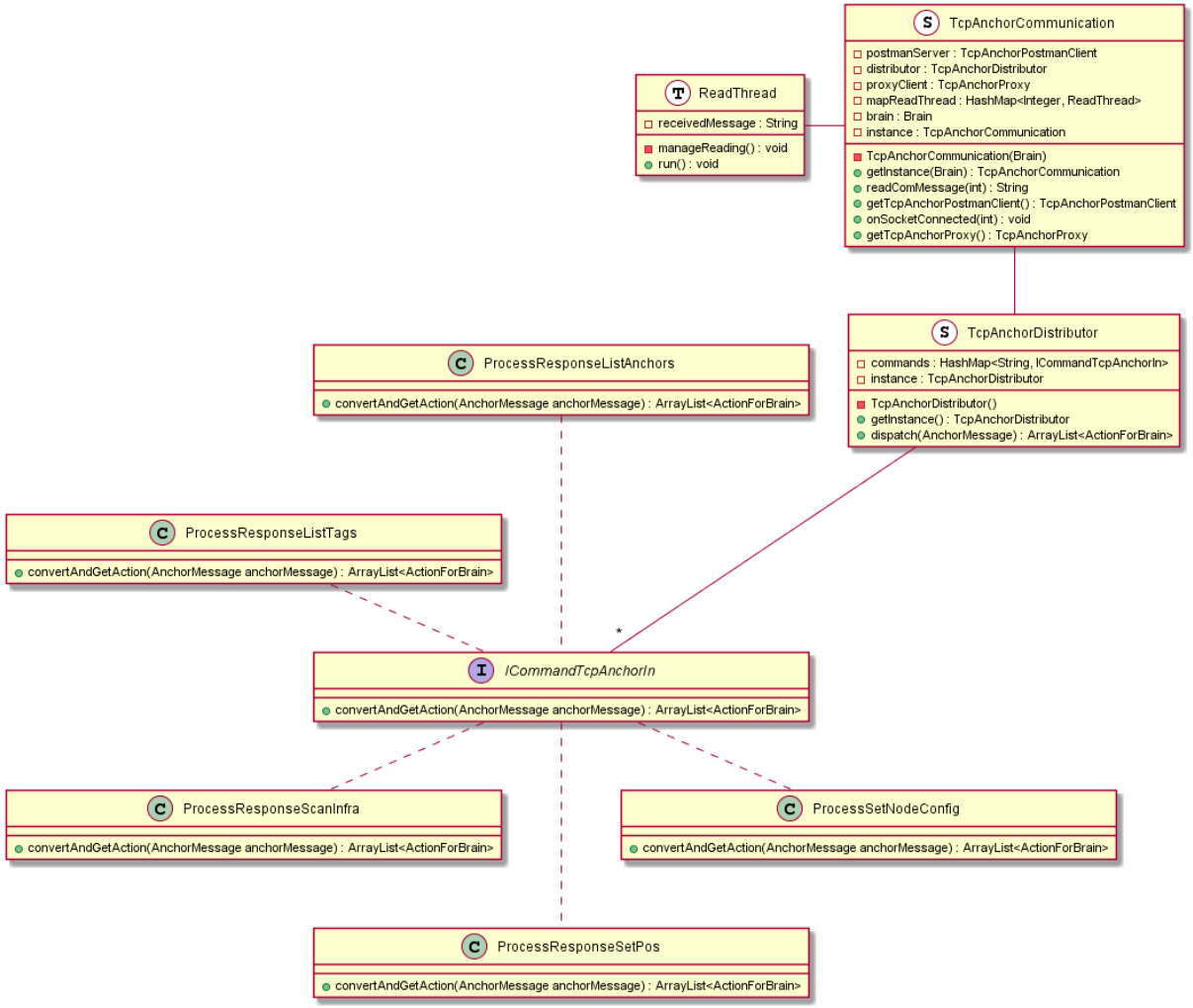

Figure 24: architecture de la conversion des commandes de l'antenne mère vers cervo

On constate sur cette figure le « TcpAnchorDistributor » qui possède la méthode « dispatch » permettant d'appeler par l'intermédiaire d'un pattern commande les différents décodeurs. Ces décodeurs implémentent l'interface « ICommandTcpAnchorIn », et n'implémentent qu'une seule méthode « convertAndGetAction(AnchorMessage) » permettant de convertir le message en model ou/et en événement pour le « Brain » par l'intermédiaire d'une liste appelée d'actions de type « ArrayList<ActionForBrain> ».

On constate de même les cinq procédures de réponses de l'antenne master après l'envoi des commandes.

2. Communication UDP

Cette communication permet de récupérer les différents messages envoyés par les antennes et les tags sur le réseau. En effet, le système permet de récupérer les différentes distances des tags par rapport aux antennes.

Cette communication respecte la même architecture que la communication précédente et se présente comme suit :

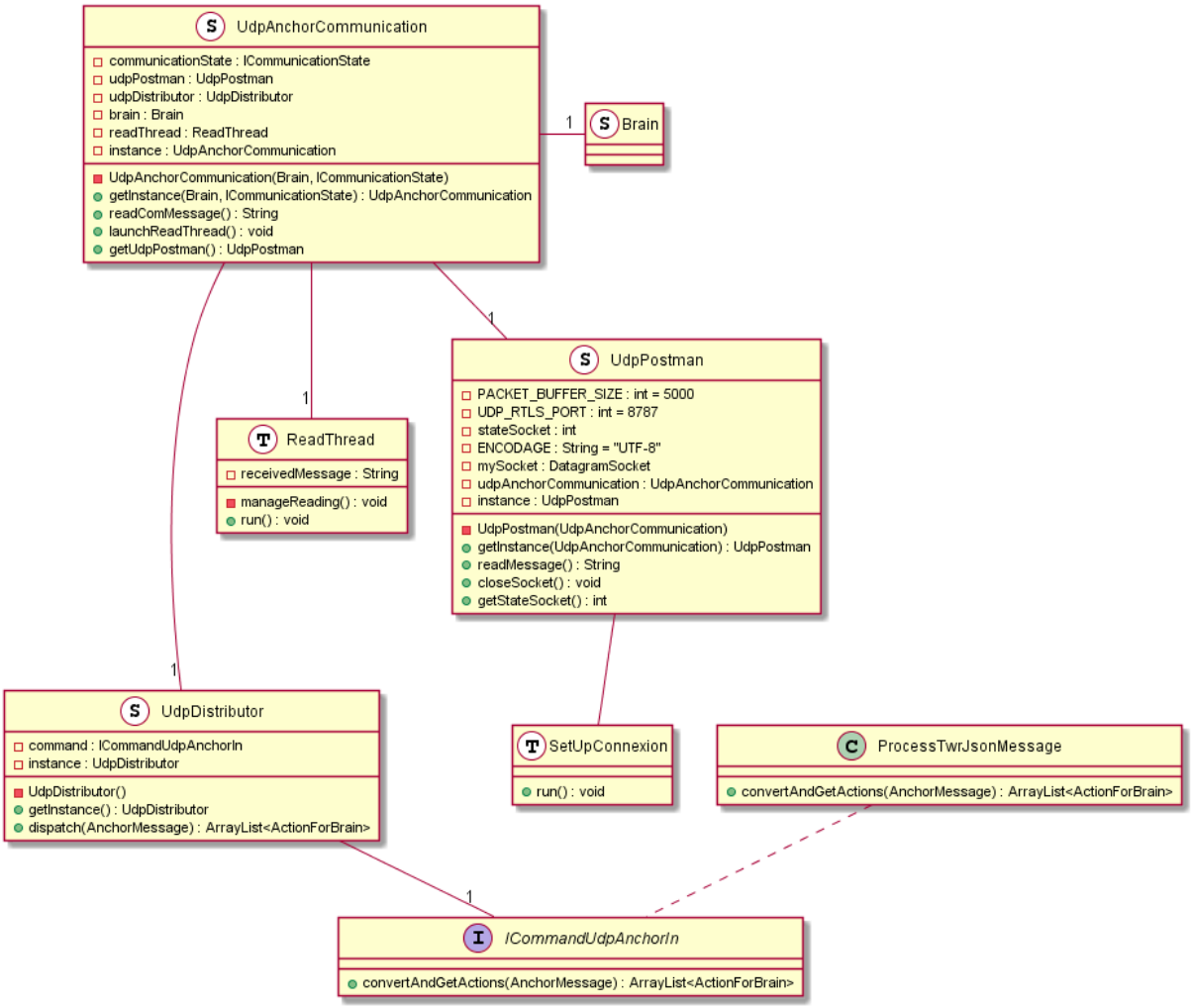

Figure 25: architecture de la communication "udp"

On constate sur cette figure le même « SetUpConnexion » et le même « ReadThread » que précédemment. On constate de même que l' « UdpPostman » ne possède ni de thread « Write » ni de proxy car aucun message n'est envoyé en udp à l'antenne mère.

La création de ce socket s'effectue sur le port 8787 imposé par le système OpenRTLS.

Lorsqu'un message est lu sur ce socket dans le « ReadThread », le message est tout de suite envoyé au distributor qui s'occupe d'envoyer ce message au seul décodeur appelé « ProcessTwrJsonMessage » par la méthode « dispatch ». Celle-ci permet de décoder le message reçu par les tags et les antennes. Un seul type de message est envoyé par cette communication. C'est dans ce décodeur que l'algorithme de trilatération est appelé pour pouvoir récupérer un unique point dans un plan.

B. Communication avec le système bluetooth

Cette communication permet de récupérer le nombre de pulsations cardiaques d'un joueur sur une plage de 10 secondes.
Cette communication possède une mécanique différente des autres communications du système. En effet, la communication avec le système bluetooth est indirectement réalisée par le Cervo. La communication bluetooth est réalisée par une carte bluetooth annexe. Cette carte bluetooth communique en série (UART) avec la carte mère. Ainsi, la récupération des messages par communication série est réalisée par un code Python implémenté dans la carte mère et fonctionnant comme un proxy. Ce code permet de récupérer les données reçues en série, d'encoder le message en Json en spécifiant le process, l'identifiant de la carte émettrice et l'identifiant du nombre de pulsations reçues sur la plage de 10 secondes. Une fois la donnée encodée, le message est envoyé sur un socket en TCP sur l'adresse local et sur le port 55000.

L'architecture de cette communication est représentée comme suit :

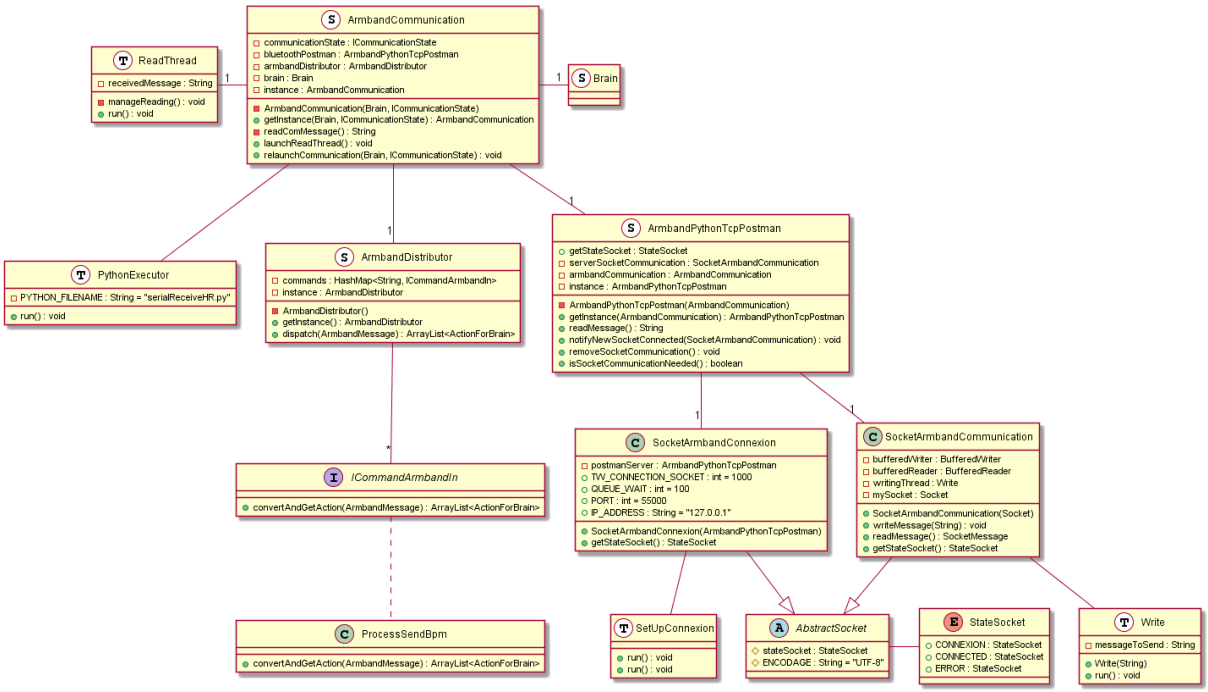

Figure 26: architecture de la communication avec les capteurs de pulsation cardiaque

Comme vu précédemment, on constate le « ReadThread » permettant de lire sur le socket TCP en scrutation. Le « ArmbandDistributor » permettant de décoder les messages reçus. Ne pouvant recevoir qu'un unique message sur ce socket appelé « sendBpm », la classe « ProcessSendBpm » permet de décoder le message reçu sur ce socket.
L'objet « ArmbandPythonTcpPostman » possède deux types de sockets :

- Un socket de connexion appelé « SocketArmbandConnexion » permettant de créer le socket de connexion pour permettre au socket client (code Python) de se connecter sur ce socket dans le thread « SetUpConnexion ».
- Un socket de communication appelé « SocketArmbandCommunication » permettant d'envoyer et de recevoir des messages.

La différence avec les autres instances de communication, le « PythonExecutor », permet d'exécuter dans un thread le code python sur la carte mère.

C. Communication avec Maestro

La communication avec le système « Maestro » est une communication serveur TCP/IP. Elle permet au système Android de se connecter pour communiquer avec la carte mère. Cette communication respecte la même architecture de communication vue précédemment et est représentée comme suit :

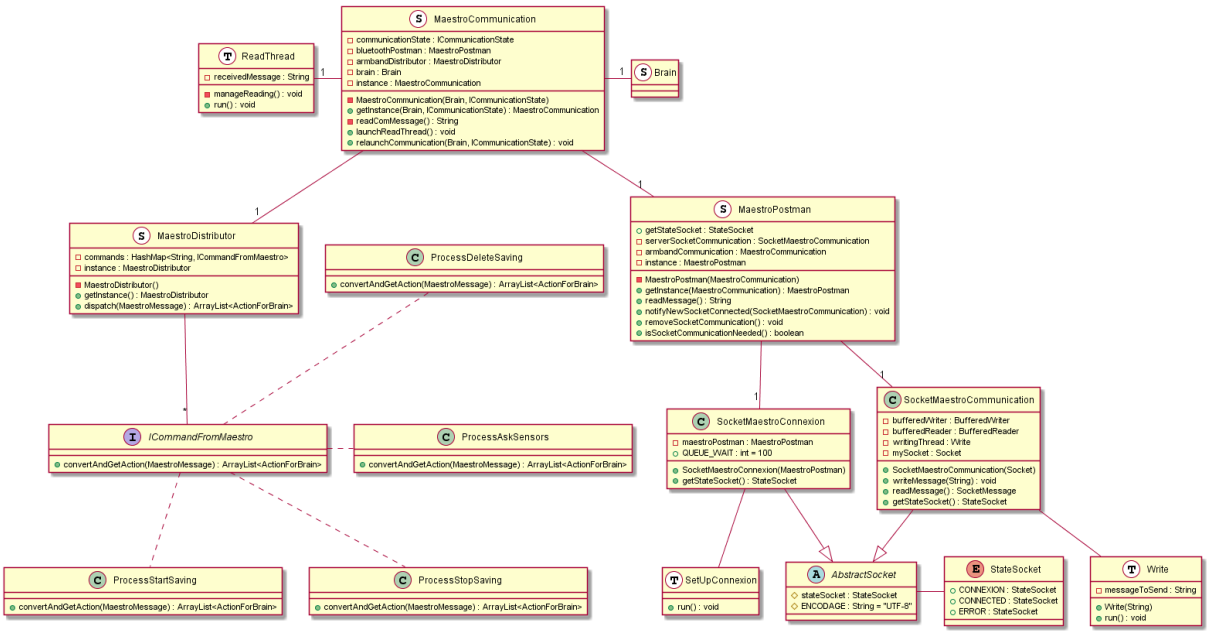

Figure 27: architecture de la communication avec maestro (Android)

Sur cette communication, le « MaestroDistributor » possède 4 types de messages. Donc 4 types de messages peuvent être reçus sur ce socket.
De plus, le « MaestroPostman » possède un socket de connexion et un socket de communication permettant d'envoyer et recevoir les messages sur ce socket.
Les différents messages pouvant être reçus sont :

- « ProcessStartSaving » : permettant de commencer l'enregistrement des données sur la carte mère.
- « ProcessAskSensors » : qui demande à la carte mère d'envoyer la liste des capteurs sur le réseau.
- « ProcessStopSaving » : qui demande à la carte mère d'arrêter l'enregistrement des données et d'envoyer la totalité des données à l'Android.

- « ProcessDeleteSaving » : qui demande à la carte mère d'arrêter l'enregistrement des données et de supprimer la session enregistrée.

La communication avec l'Android se fait sur le port 13579.

Table des illustrations