Documentation Technique

1 - Partie Raspberry

Cette partie explique l'architecture du programme implémenté pour les Raspberry Pi. Il est écrit en Python et se compose de cinq fichiers : server.py RpiRunner.py proto.py avt.py et ultra.py .

1.1 - ultra.py :

Ce fichier permet d'utiliser les modules à ultrasons sur les Raspberry. Il contient une unique classe Ultra. Pour obtenir une distance, il faut d'abord créer un objet Ultra puis simplement appeléer la fonction distance().
La vitesse du son est défini par défaut à 340,29 m/s mais une fonction de calibration est disponible. Pour l'utiliser, il faut d'abord placer le micro face a un obstacle et mesurer manuellement la distance entre les deux avec précision.
Il faut ensuite appeler la fonction calibration(int) en lui transmettant la vitesse mesurée.
Une fois les mesures terminées, il est important d'appeler la fonction terminate() qui réinitialisera les ports GPIOs utilisés.

1.2 - avt.py :

Ce fichier contient le code de la classe Avt.
Celle-ci implémente l'algorithme Adaptative Value Tracking permettant de rechercher une valeur par approximations successives.
Un objet Avt doit être initialiser en lui transmettant trois valeurs : le minimum et maximum du champ de recherche ainsi que le seuil de tolérance.
Les mises à jours s'effectues en appelant la méthode update(float) avec comme argument la valeur approximative reçue.
La valeur courante est stockée dans la variable currentVal et le nombre d'itérations effectuées est stocker dans la variable it.

1.3 - proto.py

Ce fichier représente le protocole d'échange de message.
Il contient le dictionnaire des performative des messages, la classe message, la classe client, ainsi que deux fonction d'encodage et de décodage de float vers des bits.

Le dictionnaire nommé TYPES contient les codes des performatives des messages ainsi que d'autre constantes telle que l'identifiant du serveur ou la taille des message.

La classe client est un simple conteneur représentant un client connecté au serveur. Il possède un identifiant, une adresse ip, un port, un type et une socket.

La classe message représente un message reçu ou prêt à être envoyé en fonction du constructeur qui à été appelé.
Elle contient trois champs : le destinataire du message, la performative (ty) du message, et le message en soit. La fonction .str() permet d'obtenir la suite de bits correspondante au message tandis que .toString() renvoie une chaine de character lisible représentant le message.
Les message envoyer sur le réseau sont de la forme suivante :
1 octet pour le destinataire, 1 octet pour la performative et 62 octets pour le contenu du message, pour un total de 64 octets.

1.4 - server.py

Ce fichier contient le code du serveur ainsi qu'un script permettant de le lancer.
Le code du serveur se décompose en trois morceau représentés par les classes suivantes :

- Console :

    La classe Console sert d'interface homme machine, elle gère tout les affichage dans la console et permet à l'utilisateur de rentrer des commande pour surveiller l'état du serveur.
    Une instance unique est créer au lancement du script.

    La liste des commande disponible est la suivante :

    - exit : quite le programme.
    - list_m : affiche la liste des identifiants des mobiles connecté au serveur.
    - list_a : affiche la liste des identifiants des ancres connecté au serveur.
    - log : permet d'afficher ou de sauvegarder la liste des log d'un mobile.

- Serveur :

  La classe serveur gère la création du serveur et la boucle d'acceptation des clients. Comme la Console, elle est instanciée une seule fois dans le script. A son lancement, le serveur est créer sur le port et l'adresse donné en argument, les arguments par défaut étant « localhost » et le port 4000. Si la création échoue, le programme termine.
  Ensuite, la boucle d'acceptation est lancée, chaque nouvelle connexion sera gérée par un thread_client différent.

- Thread Client :

  La classe thread_client gère une connexion avec un client, une instance en est créer à chaque nouvelle connexion avec un client.
  Deux argument sont demander, une instance de l'objet client et un boolean qui indiquera si l'on souhaite afficher un message lorsque le serveur effectue une retransmission.

  Le fonctionnement est le même que celui qui est décrit dans la partie Protocole.
  La première étape consiste à donner son identifiant au client puis à lui demander son type, après quoi, une boucle de communication est lancée.

1.5 - RpiRunner.py

Ce fichier contient le code des clients, ancre et mobile ainsi qu'un script permettant de lancer un ou plusieurs d'entre eux. Le code du serveur se décompose en trois morceau représentés par les classes suivantes :

- Console :
  Son rôle est exactement le même que celui de la console du serveur, elle ne propose cependant qu'une seule commande, exit, qui met fin au programme.

- RpiRunner :

  Chaque client est représenter par une instance de cette classe.

  Le RpiRunner est le bloc d'instructions qui vas réaliser la partie commune aux ancres et mobiles, c'est à dire la connexion au serveur, la récupération de l'identifiant et la transmission du type au serveur. Une fois ces étapes terminée, le RpiRunner lancera les fils d'exécution nécessaire selon qu'il soit une ancre ou un mobile.

Le constructeur reçoit en argument l'ip qu'il doit utiliser, le port, son type, à savoir, ancre, mobile ou les deux, et des paramètres optionnels nécessaire à l'ancre. Ils seront détaillés dans a section dédié à l'ancre.

- Ancre :

  Cette classe gère la fonction d'ancre au sein d'un client.
  Son constructeur reçoit cinq arguments :

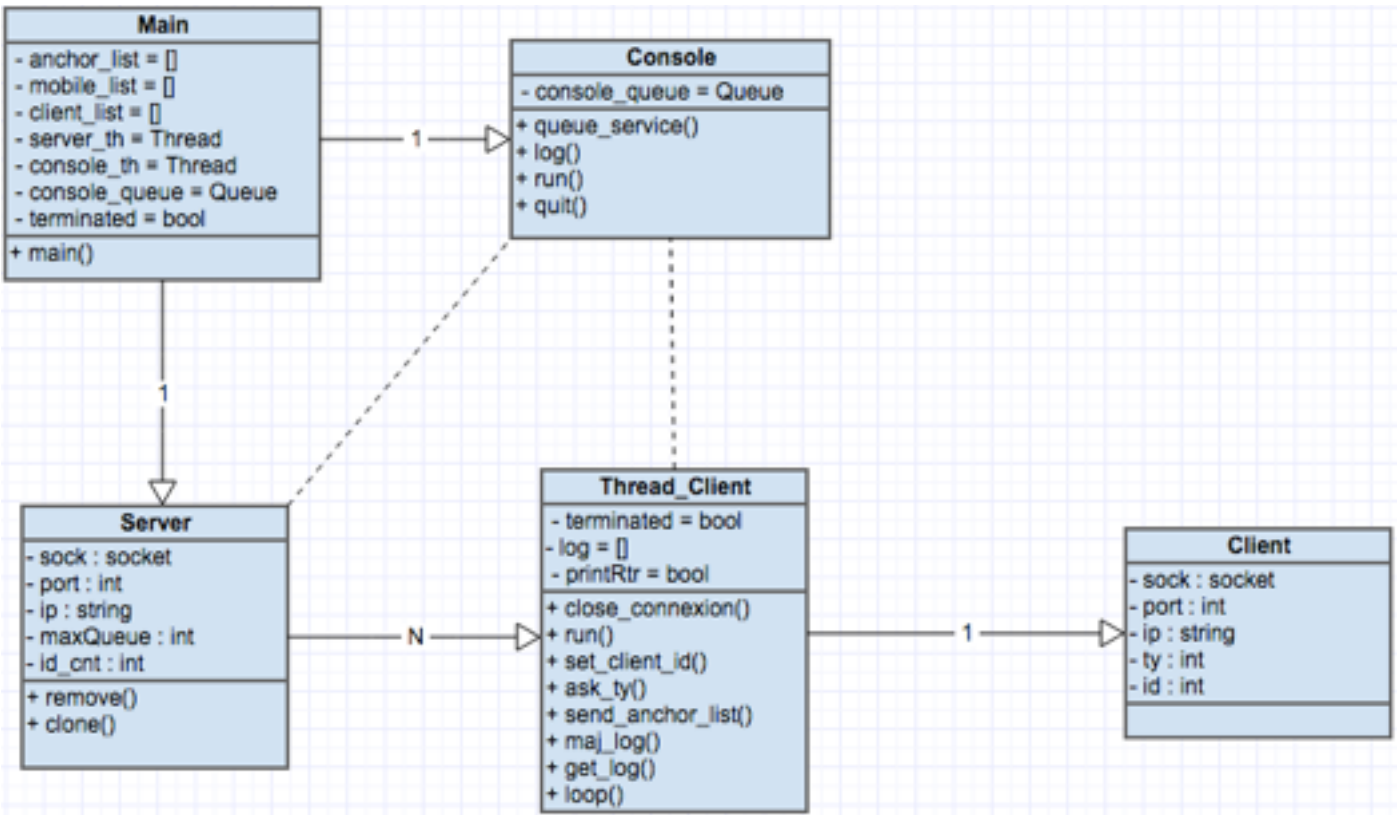
  - Le RpiRunner dont elle est issue
  - Les coordonnées x et y de sa position
  - Un boolean Ultra qui lui indique si elle doit utilisé le module Ultra
  - Un valeur de distance.

  Dans le cas ou ou lui a indiqué d'utiliser le module ultra, l'ancre créera une instance d'ultra et l'utilisera à chaque fois qu'on lui demande une evaluation de distance. Dans le cas contraire, la valeur renvoyer sera la valeur de distance reçue en argument après application d'un bruit gaussien.

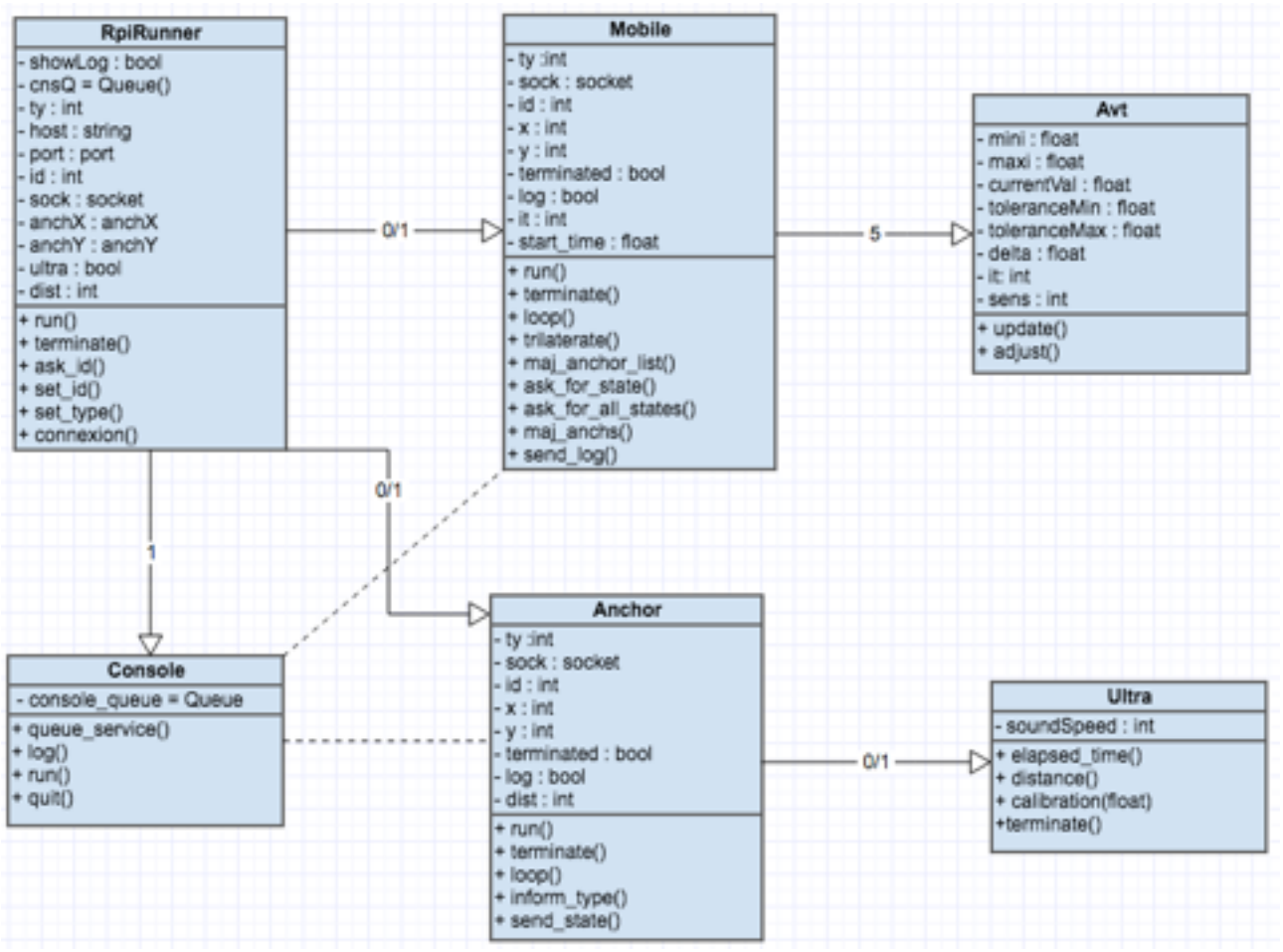
- Mobile :

  Cette classe gère la fonction de mobile au sein d'un client.
  Elle ne reçoit aucun argument et se comporte exactement comme décrit dans la section Conception - Protocole.

1.6 - Schéma de l'Architecture du Serveur Rpi



1.7 - Schéma de l'Architecture du RpiRunner



2 - Arduino

Le code pour Arduino intègre une partie mobile et une partie ancre.
Le serveur n'est pas implémenté dessus, l'Arduino ne disposant ni des ressources, ni des capacité de multi-threading nécessaires pour remplir simultanément le rôle de serveur en plus de celui d'un mobile ou d'une ancre.

2.1 - Vector

La classe **Vector** (native en C++) est absente de la librairie standard de l'Arduino Nous avons trouver une implémentation (plus légère que l'original) qui nous permet d'éviter de devoir utiliser les **malloc** déconseillé sur l'Arduino.

2.2 - ESP8266

Cette classe nous fourni l'interface de communication avec le module ESP de plus haut niveau. Elle implémente toutes les commande de base de l'ESP², mais pour le projet nous n'utilisons que quatre de ces méthodes.

- bool joinAP(String ssid, String pwd)

Contrairement à la Raspberry l'Arduino ne possède pas d'OS donc il ne peux pas se connecter directement au réseau, c'est pourquoi nous avons besoin ce cette méthode pour rejoindre le réseau ou se trouve notre serveur.
Elle prend en paramètre le nom du réseau et le mot de passe et renvoi un Booléen correspondant au succès ou à l'échec de la connexion

- bool createTCP(String addr, uint32_t port)

La connexion au serveur se fait avec l'appel à cette méthode, on doit lui fournir l'adresse et le port ou se trouve le serveur.
Le booléen retourné nous indique si la connexion avec le serveurTCP a réussi

- bool send(const uint8_t *buffer, uint32_t len)

Cette fonction envoi un message au serveur TCP de taille **len(32 bytes)** comme indiqué dans le protocole sous la forme d'un tableau de bytes.
Le message de retour nous indique seulement que le message a bien été envoyé.

- uint32_t recv(uint8_t *buffer, uint32_t buffer_size, uint32_t timeout)

Comme pour la fonction send, cette fonction communique directement avec le serveur TCP.
Mais sert a recevoir les message venant du serveur sous la forme d'un tableau de byte de taille max **buffer_size (32 bytes)** dans la limite du Timeout passé en paramètre.
Retourne la taille réel du message reçu

2.3 - Message

Cette classe implémente directement l'envoi et la réception des messages décrit par le protocole de communication mis en place.
Notre protocole est composé de 15 prérogatives celle-ci étant déjà décrite précédemment.
[FINIR]

2.4 - Ancre

le fichier Ancre fourni deux classes indispensables pour le projet.
Ancre : Classe correspondant a a la représentation d'une ancre vu par le mobile. Elle possède une position X, Y « fixe », un Id pour que le mobile puisse communiquer avec et un objet Avt qui correspond a l'estimation de la distance

- void set_Position(float x, float y)

Permet de definir la position de l'ancre qui est utile pour la Trilateration du mobile

- void adjust_Range(float r)

Avec cette fonction on met a jour l'AVT avec la distance reçu en paramètre correspondant a la distance du mobile par rapport l'ancre

- Les fonctions suivantes renvoi les variables de la classe **Ancre**
uint8_t getId(), float getX(), float getY(), float get_Range()

2.5 - Mobile

- void adjust_X(float d) void adjust_Y(float d)

Avec cette fonction on met a jour l'AVT avec la position reçu en paramètre correspondant à la position X, Y du mobile

- void trilateration()

Cette fonction effectue la trilatération du mobile avec les positions de trois ancres et l'évaluation de distance de leurs Avt

- void update_Anchor_Liste(Vector<Anchor*> liste)

Cette fonction met a jour la liste de ancre que le mobile connaît avec la « liste » des ancre envoyé par le serveur

- Les fonctions suivantes renvoi les variables de la classe **Mobile**
uint8_t getId(), float getX(), float getY()

2.6 - AVT

La classe AVT est similaire a la classe Avt développé en python pour la Raspberry, et les fonctions y sont implémenté de la même manière.
- void update(float value);
- float get_Value()
- void adjust(int direction)

2.7 - Node

le fichier **Node.ino** est le fichier main de l'Arduino, il contient deux fonction obligatoire pour l'Arduino :

- setup()

Nous sert ici a mettre en place le réseau, se connecté au serveur et configurer le Nœud (ancre ou mobile)
si le nœud est un mobile il effectue une demande de la liste des ancres ainsi que leurs positions

- loop() :

si le nœud est mobile cette fais des demande successive de distance aux trois ancres sélectionnes et effectue la trilatération du mobile
par contre si le nœud est une ancre la boucle attend une demande du serveur ou d'un mobile et renvoi sa distance, sa position ou son type.

2.8 - Config

Ce fichier contient toutes les configurations du réseau et du nœud comme le nom du réseau où il doit se connecter, le mot de passe, l'adresse du serveur, le port mais aussi le type du nœud et sa position dans le cas où le nœud est une ancre