Projet A.N.D.R.O.I.D.E

2015-2016

Positionnement relatif en Wi-Fi

Étudiants :

Renaud Adequin
Pablo Petit

Encadrants :

Assia Belbachir (IPSA)
Cédric Herpson (UPMC)

1. Introduction

Ce projet à été réalisé dans le cadre de l'UE Projet ANDROIDE du Master 1 d'Informatique spécialité ANDROIDE à l' Université Pierre et Marie Curie.

L'objectif de ce projet est d'octroyer à des entités disposant de faibles capacités de calcul et ressources énergétiques un moyen efficace, précis et robuste, de se positionner relativement dans l'espace en mettant en place un réseau de communication ad-hoc. Les entités en question sont des Arduino[1] et des Raspberry Pi[2], le but à terme étant de déployer une solution opérationnelle sur une flotte d'entités mobiles hétérogènes . Les Arduino ne possèdent que 96 kB de RAM et un procésseur mono-coeur d'une fréquence de 84 Mhz. Sur de telle machines, la localisation absolue est généralement trop coûteuse ou chère en ressources, en plus de ne pas toujours être disponible (en intérieur notamment).

Dans le cadre de ce projet, nous étudierons la solution décrite dans l'article « Robust and Efficient Self-Adaptive Position Tracking in Wireless Embedded Systems[3] » disponible en annexe. Ce document propose une solution innovante permettant de résoudre le problème de l'auto-localisation en utilisant une technique de recherche appelée Adaptive Value Tracking (AVT[4]).

Nous devons donc en premier lieu mettre en place un réseau ad-hoc décentralisé permettant à toutes les entités de se connecter entre elles, ainsi qu'un protocole de communication qui leur permettra de se transmettre les informations nécessaires à leur positionnement. Puis en second lieu, implémenter et déployer l'algorithme en question sur le matériel donné pour pouvoir procéder au test de celui-ci dans un contexte statique et dynamique. Enfin, une analyse des résultats sera effectuée afin d'obtenir une évaluation de ses performances et de sa robustesse ainsi que ses défauts, permettant d'établir son cadre applicatif et enfin, de mettre en lumière des possibilités d'amélioration de cet algorithme.

2. Principe de l'algorithme

L'article qui guide ce projet se nomme « Robust and Efficient Self-Adaptive Position Tracking in Wireless Embedded Systems[3] » il a été écrit par Ramil Agliamzanov, Assia Belbachir et Kasım Sinan Yıldırım.

Cet article tente de répondre au problème de l'auto-localisation en environnement dynamique.
Son objectif est de fournir à des entités mobiles un moyen efficace, robuste au bruit et peu couteux pour s'auto-localiser. La méthode employée se veut novatrice dans la manière dont elle aborde le problème. Celui-ci est entièrement traité comme un problème de recherche. Les valeurs exactes de la position du mobile, obtenues par trilatération, ainsi que celles des distances entre les entitées, sont recherchées par itérations successives, sans jamais avoir accès à leurs valeurs exactes.

Pour obtenir et affiner les estimations faites de ces valeurs, l'algorithme utilisé est l'Adaptative Value Tracking (AVT[4]). L'AVT est une méthode de recherche qui se distingue des autres méthodes d'agrégation de valeurs par la faible quantité de variables qu'elle doit garder en mémoire, et du peu de calculs qu'elle nécessite. Elle correspond donc parfaitement au cadre donné.

Le principe général consiste à disposer de plusieurs entitées jouant le rôle de “mobiles” et “d’ancres”. Les ancres sont supposées connaitre leur position, et les mobiles vont tâcher de se repérer par rapport à ces dernieres. Ces différentes entités peuvent communiquer et doivent disposer de systèmes de mesure de distance (ultrason, laser, ..). A titre d’exemple, dans le cadre d’une flotte de drones, les plus puissants (ou évoluant en extérieur) peuvent disposer d’une liaison GPS. Les plus limités (ou évoluant en intérieur) en sont dépourvus et utilisent les premiers comme référence.
L’algorithme est composé de deux plus petits algorithmes. Le premier sert à estimer les distances séparant ancre et mobile et le second estime la position du mobile par trilatération à partir des résultats du premier, appliqué à au moins trois ancres. Les deux algorithmes font appel à des AVTs comme technique de recherche.

L’algorithme AVT fonctionne de la manière suivante. Il est initialisé avec quatre valeurs : le maximum et minimum du champs de recherche et le maximum et minimum du pas. Il stocke trois autres valeurs qui seront mises à jour à chaque itération :

- ➔ Le sens : celui-ci indique si la valeur courante, obtenue lors de la dernière itération de calcul, sur-évaluait ou sous-évaluait la valeur reçue.

- ➔ Le pas : il indique de combien va progresser la valeur courante lors de la prochaine mise à jours. Si le sens reste constant, sa valeur est multipliée par deux. A l’inverse, si le celui-ci change entre deux itérations, la valeur du pas est divisée par trois. Il reste cependant borné entre le maximum et le minimum, valeurs transmises à l’initialisation.

- ➔ La valeur courante : celle-ci est mise à jour en fonction du sens et du pas.

Lors de chaque itération, l’AVT recoit une valeur et un sens avec lesquels il mettra à jour son pas, puis sa valeur courante. Il y additionnera ou soustraira le pas courant en fonction du sens reçu.

Deux expérimentations visant à illustrer le comportement de l’algorithme ont été réalisées dans l’article. La premiere vise à déterminer les performances de l’AVT. Une ancre et un mobile sont placés à une distance de 100 cm qui est ensuite ramené à 50 cm. Le résultat montre que l’AVT converge sur la valeur initiale en 15 itérations, puis se replace correctement, après le changement de position, en autant d’itérations. Le second test met en scène trois ancres et un mobile. Les ancres sont placées dans trois coins d’un carré de 100 cm de coté, le mobile est initialement placé dans le quatrième coin, puis ramené au centre du carré. Les résultats obtenus dans l’article sont les suivants :

1/ La marge d’erreur de la localisation en appliquant l’AVT aux mesures de distance et aux résultats de la trilatération est de 2 à 5 cm. Tandis que ceux obtenus par pure trilatération sont de 10 à 15 cm.

2/ Le système d’auto-localisation implementé est robuste face au bruit induit par les mesures ultrasoniques et ses estimations sont stables par rapport à celles obtenues par pure trilatération.

3/ L’ensemble du système est léger en terme de ressources computationelles utilisées.

4/ Le système s’adapte relativement rapidement aux changements de position.

3. Infrastructure et Protocoles

Ce projet nécessite qu'un certain nombre d'entités hétérogènes communiquent entre elles. Dans le cadre de ce travail, la communication est réalisée en WI-FI. Nous présenterons dans cette section l'architecture réseau mise en place ainsi que le protocole de communication élaboré à partir de celui proposé dans l'article.

Pour répondre aux exigences du projet, une première solution en Ad-Hoc a été étudiée. Celle-ci devait permettre la création d'un réseau de communication entre les différents noeuds qui serait robuste face aux entrées et sorties de ces derniers. Cette solution aurait eu de plus la particularité de s'affranchir d'un serveur central, lui offrant d'autant plus de robustesse et lui permettant de s'établir sur un périmètre plus grand. La réalisation d'un tel réseau avec le matériel proposé implique une conception divisée en deux parties, une première sur Raspberry Pi et une seconde destinée à l'Arduino et à son module de communication (module ESP[5]).

Le déploiement d'un réseau Ad-Hoc sur Raspberry est relativement simple. Il faut tout d'abord connecter un dongle Wi-Fi compatible avec le mode Ad-Hoc à la Raspberry Pi. Ensuite, il suffit d'activer le mode associé [1][6] au Ad-Hoc.

La mise en place du réseau Ad-Hoc sur le couple Arduino et module ESP8266 est par contre plus complexe. Pour pouvoir y arriver, il faut d'abord réussir à faire communiquer correctement l'Arduino avec l'ESP et ensuite faire en sorte que l'ESP se connecte au réseau. Toute communication entre l'Arduino et le module ESP passe par l'envoi et la réception de commandes textuelles[6]. Ce mode de communication est très contraignant car il oblige à gérer de multiples erreurs et délais. Pour le simplifier, et pour le rendre plus rapide, nous avons mis en place une interface entre les deux appareils. Celle-ci est basée sur la librairie WeeESP8266[8] trouvée en ligne. Cette librairie étant implémentée pour fonctionner avec un autre modèle d'Arduino, nous l'avons simplement adaptée au nôtre. Une fois l'intégration de l'ESP correctement effectuée, il faut faire en sorte que celui-ci se connecte au réseau Ad-Hoc. Le module ESP propose trois modes de fonctionnement, le mode « station », le mode « access point » et enfin le mode « both » qui permet d'activer les deux premiers modes simultanément. Le protocole Ad-Hoc n'est donc pas pris en charge nativement. Deux solutions sont alors possibles:

- Modifier l'ESP8266[9] pour lui permettre de rejoindre un réseau Ad-hoc déjà existant, et en créer un dans le cas contraire.

- Simuler un réseau ad-hoc en utilisant les modes de communication déjà existant de l'ESP8266.

Bien qu'étant la piste la plus intéressante, la première solution s'est rapidement montrée inenvisageable. Le projet Meshish [10] confirme que l'ESP n'a pas la capacité de re-transmettre des messages entre différent client qui seraient connectés à ce dernier.

À cause de la difficulté et du temps nécéssaire à la réalisation d'un réseau Ad-Hoc couvrant à la fois les Raspberry et les Arduino, et dans l'optique d'assurer la réalisation des tests en respectant les délais imposés par le projet, le réseau Ad-Hoc n'a finalement pas été retenu. À la place, une architecture Client - Serveur classique à été adoptée. Bien qu'étant, comme exprimé plus haut, moins robuste et plus

[1] Il faut pour cela modifier directement le fichier '***/etc/network/interfaces***' pour indiquer le passage au mode Ad-Hoc, affecter une adresse IP statique et nommer le réseau.

contraignante à utiliser, elle possède l'avantage d'être simple et facile à mettre en place. Une telle architecture nous autorise donc à consacrer plus de temps à la réalisation des autres objectifs du projet.

Pour permettre aux différentes entités maintenant connectés au même réseau de communiquer et de se transmettre les informations nécéssaires, le protocole mis en place doit permettre d'identifier les différent rôles présents sur le réseau, et permettre à chaque noeud d'obtenir les informations nécessaires. Cette section décrira le protocole en question en détaillant chacun des rôles présents.

Il existe trois différent rôles : le serveur, les ancres, et les mobiles. Chaque noeud du réseau peut tenir simultanément un ou plusieurs d'entre eux, la seule contrainte étant que le serveur doit être unique. La communication s'effectue de manière centralisée en passant par le noeud qui tient le rôle de serveur et en utilisant un format de message spécifique, détaillé dans la documentation technique (Annexe 7.4 page 24). Nous allons maintenant présenter en détail ce que font chacun des acteurs du réseau.

Le Serveur :

Le serveur est le noeud central du réseau, il a pour objectif de permettre la communication entre les autres noeuds. Pour cela, il doit donc attribuer à chaque noeud un identifiant unique, permettre la retransmission des messages et effectuer un recensement dynamique des noeuds connectés ainsi que de leurs rôles respectifs. De plus, le serveur garde en mémoire les logs envoyés par chaque mobile et permet leur sauvegarde. À son lancement, le serveur démarre une boucle d'acceptation des clients et attend une nouvelle connexion. Chaque nouveau client est géré dans un fil d'exécution différent de la manière suivante :

1/ Le client est informé de son identifiant propre, le serveur envoie donc un message au client et attend une confirmation de réception de celui-ci.

2/ Une fois que le client connait son identifiant, le serveur lui demande son type, c'est à dire les rôles qu'il rempli, puis, après réception de la réponse, il met à jour ses annuaires de mobiles et ancres.

Ces deux étapes sont répétées un nombre arbitraire de fois. Si tous les essais se soldent par un échec, la communication est interrompue. Dans le cas contraire, la boucle de communication est lancée. Le serveur va attendre que le client lui envoie un message, et selon s'il en est ou non le destinataire, il le traitera ou le transmettra à son destinataire. Les clients peuvent donc de cette manière communiquer directement avec le serveur. Cela leur donne la possibilité de demander au serveur la liste des ancres connectées, de lui envoyer un log ou bien d'annoncer leur sortie du réseau.

L'Ancre :

Une ancre est un noeud qui connait sa position et qui est capable de fournir à un mobile une évaluation de la distance qui les séparent. À son lancement, l'ancre va se connecter au serveur et attendre de recevoir son identifiant. Ensuite, elle attendra que le serveur lui demande son rôle et répondra. Une fois la boucle de communication lancée avec le serveur, l'ancre enclenchera une veille d'écoute jusqu'à ce qu'un message lui parvienne d'un mobile. Les ancres ne traitent qu'un seul type de message, les demandes d'état. Elles y répondent en envoyant un message qui contient leur position ainsi qu'une évaluation de distance[2].

Le Mobile :

Le mobile est le noeud qui cherche à estimer sa position. Il va pour cela utiliser le serveur afin d'obtenir les identifiants des ancres alentour, puis leur demandera d'évaluer la distance qui les séparent en continu, afin d'estimer au mieux sa position. Le mobile se comporte à son lancement comme une ancre : il se connecte au serveur, récupère son identifiant et informe le serveur de son type. Il demande ensuite au

[2] À l'aide d'un Module Ultrason HC-SR04[11]

serveur de lui transmettre la liste des ancres jusqu'a en obtenir un nombre qu'il juge suffisant (3 dans le cadre de nos expérimentations). Une fois la liste obtenue, le mobile va demander à chaque ancre de lui transmettre sa position.

Il initialise alors un AVT pour chacune des ancres ainsi que deux autres, pour chacune des composantes de sa position (nous rappelons que le mobile évolue en deux dimensions). L'initialisation terminée, le mobile va ensuite itérer la démarche qui lui permettra de se localiser, elle est divisée en trois étapes. Il demande d'abord successivement à chacune des ancres de lui transmettre une évaluation de la distance qui les séparent et met à jour à chaque fois les valeurs de leurs AVTs respectifs.

Ensuite, il effectue une opération de trilatération pour obtenir une nouvelle approximation de sa position, et enfin, utilise ce résultat pour mettre à jour les AVTs correspondant à sa position. Dans le cas ou une ancre quitte le réseau ou ne répond plus, le mobile redemande la liste des ancres au serveur jusqu'a en avoir à nouveau un nombre satisfaisant. Il arrête à ce moment là de calculer sa position mais continue à mettre à jour les AVTs des ancres encore disponibles pour pouvoir se re-positionner ensuite au plus vite. Pour finir, à chaque itération du processus, le mobile envoie un message « log » au serveur qui contient les valeurs courante des AVTs, de sa position et des distances avec les ancres qu'il interroge.

Le détail de l'implémentation de l'algorithme et les diagrammes de classe associés sont présentés dans l'annexe 7.4 page 25 et suivantes. L'intégralité du code source est quand à lui disponible sur GitHub[13]. Une fois le réseau et le protocole opérationnels, il est possible de réaliser les experimentations nécessaires à l'évaluation des performances de l'algorithme. C'est l'objet de la section suivante.

4. Expérimentations

4.1. Cadre expérimental

Nous reproduisons dans un premier temps les expériences de l'article, qui correspondent à un environnement statique, avant d'aborder la mobilité. Les tests ont été réalisés dans des conditions les plus fidèles possible à celle de l'article, ce, afin de permettre une comparaison non biaisée. Cependant, certaines différences notables sont à signaler.

Premièrement la prise des mesures pour l'évaluation de distance faites par les ancres est différente. Dans l'article, la méthode utilisée est la différence du temps d'arrivé entre un signal WI-FI et un signal ultrason. Cela implique une synchronisation préalable des nœuds, inutile dans notre configuration. À la place, nous mesurons le temps nécessaire au son pour faire un aller-retour entre le capteur ultra-son et le mobile. Il est possible de voir alors apparaître une différence de précision dans les mesures.

Deuxièmement, l'article n'indique pas la façon dont le pas de l'AVT est mis à jour, les seules informations indiquées sont les bornes utilisées. La vitesse de convergence de l'AVT dépendant fortement de la manière dont son pas est mis à jour, une différence du nombre d'itérations nécessaires pour converger est possible. Dans notre implémentation, la valeur initiale donnée à la valeur courante est celle de la première itération, celle du pas est sa borne inférieure et le sens prend la valeur neutre.

Enfin à la différence des expérimentations de l'article, dans lequel les ancres étaient directement pointées vers le mobile, nous avons remplacé celui-ci par un cylindre en carton de six centimètres de diamètre.En effet, la faible surface de réflexion du mobile (un arduino non intégré à un robot dans le cas présent) ne permettait pas de le rendre visible simultanéement par toutes les ancres.

Le matériel utilisé pendant le développement du projet et les expérimentations est le suivant :

- 3 Raspberry Pi 2 + Dongle Wi-Fi
- 1 Raspberry Pi 1 + Dongle Wi-Fi
- 1 Arduino Due
- 1 Module ESP8622
- 4 Modules Ultrason HC-SR04[11]

Environnement de test. 4 ancres (Raspberry Pi + capteur ultrason) sont placés aux positions fixes (0, 0), (0, 100), (100, 0) et (0, 50). Le mobile quand à lui est représenté pas le cylindre en carton.

4.2. Reproduction des expérimentations présentées dans l'article

4.2.1. Évaluation de la distance

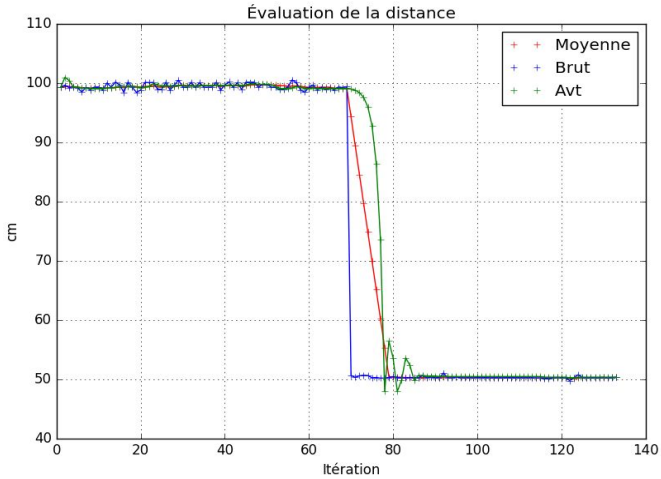
But :

Évaluer le comportement de l'algorithme AVT sur l'évaluation de distance faite par une ancre face à un "mobile" fixe, puis sa réaction lors d'un changement brusque de position.

Configuration :

Un seul mobile et une seule ancre sont utilisés.
Dans un premier temps le mobile est placé à 100 cm de l'ancre et prend des mesures toutes les 5 secondes pendant 5 minutes. Ensuite, le mobile est déplacé à 50 cm de l'ancre entre deux itérations et prend à nouveau des mesures toutes les 5 secondes pendant 5 minutes.

Résultat :



Éstimation de la distance relative entre un mobile et une ancre. La ligne bleue montre les estimations brutes. La ligne verte montre les estimations obtenues avec l'algorithme AVT. La ligne rouge montre les estimations obtenues avec la moyenne des 10 dernières valeurs. La distance initiale est de 100 cm puis après 70 itérations le mobile est déplacé à une distance de 50 cm.

Les résultats des tests d'évaluation de distance montrent que les valeurs de l'algorithme AVT (vert) et de la moyenne (rouge) corrigent les valeurs bruitées des données brutes (bleu). Lors du déplacement du mobile à la 60^{e} itération l'utilisation de la moyenne permet de converger plus rapidement que l'algorithme AVT vers la nouvelle distance (50 cm).

4.2.2. Évaluation de la position

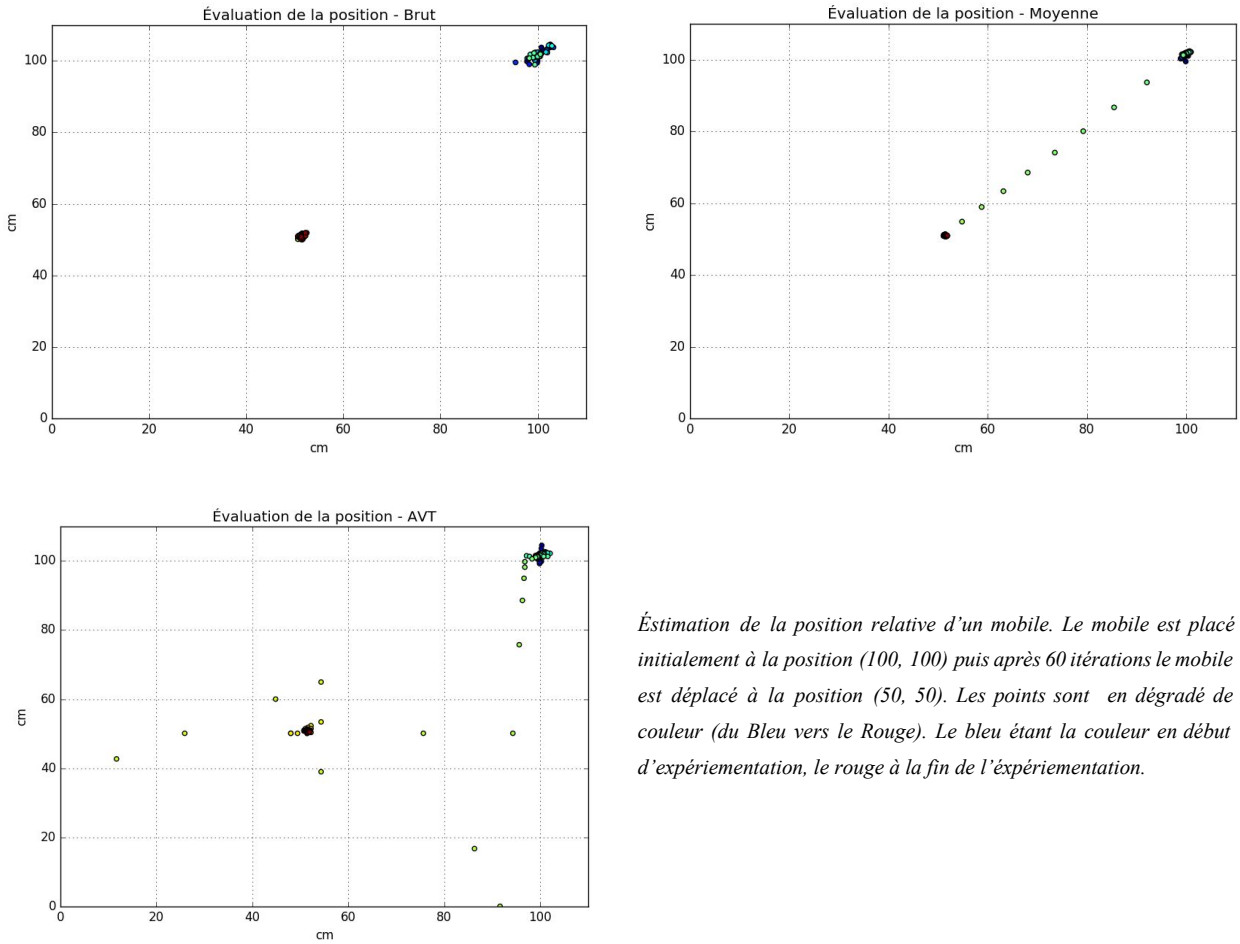
But :

Évaluer le comportement de l'algorithme AVT sur l'évaluation de la position d'un mobile en se basant sur les distances estimées par les ancres ainsi que sa réaction lors d'un changement brusque de position.

Configuration :

Les trois ancres utilisées sont placées aux positions (0, 0), (100, 0) et (0, 100).
Dans un premier temps le mobile est placé à la position (100, 100) et prend des mesures toutes les 5 secondes pendant 5 minutes. Ensuite, le mobile est déplacé à la position (50,50) entre deux itérations et prend à nouveau des mesures toutes les 5 seconde pendant 5 minutes.

Résultat :



Éstimation de la position relative d'un mobile. Le mobile est placé initialement à la position (100, 100) puis après 60 itérations le mobile est déplacé à la position (50, 50). Les points sont en dégradé de couleur (du Bleu vers le Rouge). Le bleu étant la couleur en début d'expériementation, le rouge à la fin de l'éxpériementation.

Pour l'évaluation de la position, les tests montrent que l'algorithme AVT de même que la moyenne permettent de corriger les valeurs brutes lors de l'estimation de la position. À la 60^{e} itération lors du déplacement du mobile à la position (50, 50), l'utilisation de la moyenne permet de converger vers la nouvelle position en huit itérations, soit plus rapidement que l'AVT qui converge lui en onze itérations.

La reproduction des tests de l'article nous a donné des résultats différents. L'implémentation de l'algorithme AVT nécessite une moyenne de cinq itérations de plus que les résultats présentés dans l'article pour converger. Cette différence peut être expliquée par l'absence de la méthode de mise à jour du pas de l'algorithme AVT. Cependant, les résultats obtenus nous permettent d'expérimenter plus en détails les performances l'AVT.

4.3. Essais en environnement dynamique

Les tests supplémentaires que nous avons effectués ont un double objectif. Le premier d'entre eux est d'évaluer les performances de l'algorithme dans des contextes dynamiques plus complexes que ceux vus dans l'article. En l'occurrence, un mouvement rectiligne uniforme, puis la perte de contact avec une ancre. Le deuxième est de comparer les résultats obtenus avec ceux qui l'auraient été en remplaçant l'AVT par une moyenne ou même simplement, par les résultats bruts. Cette comparaison se fera au niveau des performances, de la réactivité, de la rapidité et de la robustesse et a pour objectif d'évaluer l'intérêt de l'algorithme AVT dans le cadre donné.

4.3.1. Vitesse d'exécution

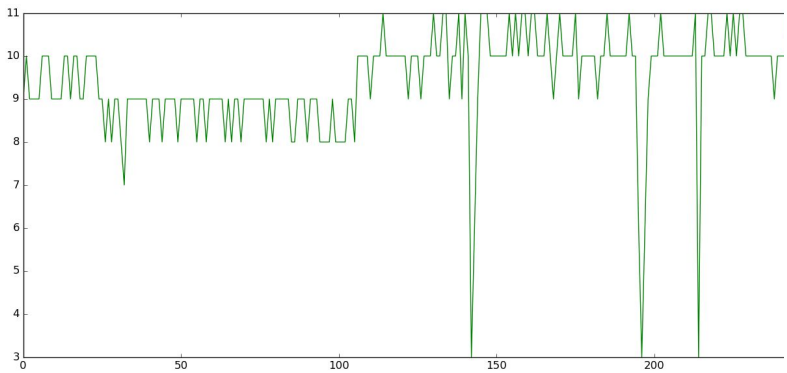
But :

Le but de ce test est d'évaluer la vitesse d'exécution de l'algorithme et d'obtenir ainsi le nombre moyen d'itération que le système peut fournir par seconde. Un tel résultat permettra dans la suite de simuler des déplacements du mobile à une vitesse donnée.

Configuration :

Les trois ancres utilisées sont placées aux positions (0, 0), (100, 0) et (0, 100). Le mobile est placé à la position (100, 100). Il est programmé pour compter le nombre d'itérations de l'algorithme qu'il effectue par seconde puis les afficher.

Résultat :



Nombre d'itération par seconde. La ligne verte indique le nombre d'itération de l'algorithme AVT qu'éffectue l'arduino Due.

La moyenne du nombre d'itérations par seconde est de 9.45. Pour des raisons de simplicité et afin de simplifier les tests suivants, nous considérons dans les tests suivants que l'Arduino est capable de produire 10 itérations par seconde.
Ce faible nombre d'itération moyen par seconde peut être expliqué par le nombre d'étape d'intermédiaire lors de la communication entre les différents nœuds. Le mobile Arduino doit interagir avec le module Esp8266, l'Esp8266 communique avec le serveur qui re-distribue le message à la Raspberry pi concerné. En effet, car même si l'Arduino serai capable d'effectuer un nombre d'itération par seconde plus important, la communication avec les ancres limite le nombre d'itération possible. Ce point sera abordé plus en détail dans la section suivante.

4.3.2. Perte de contact avec une ancre

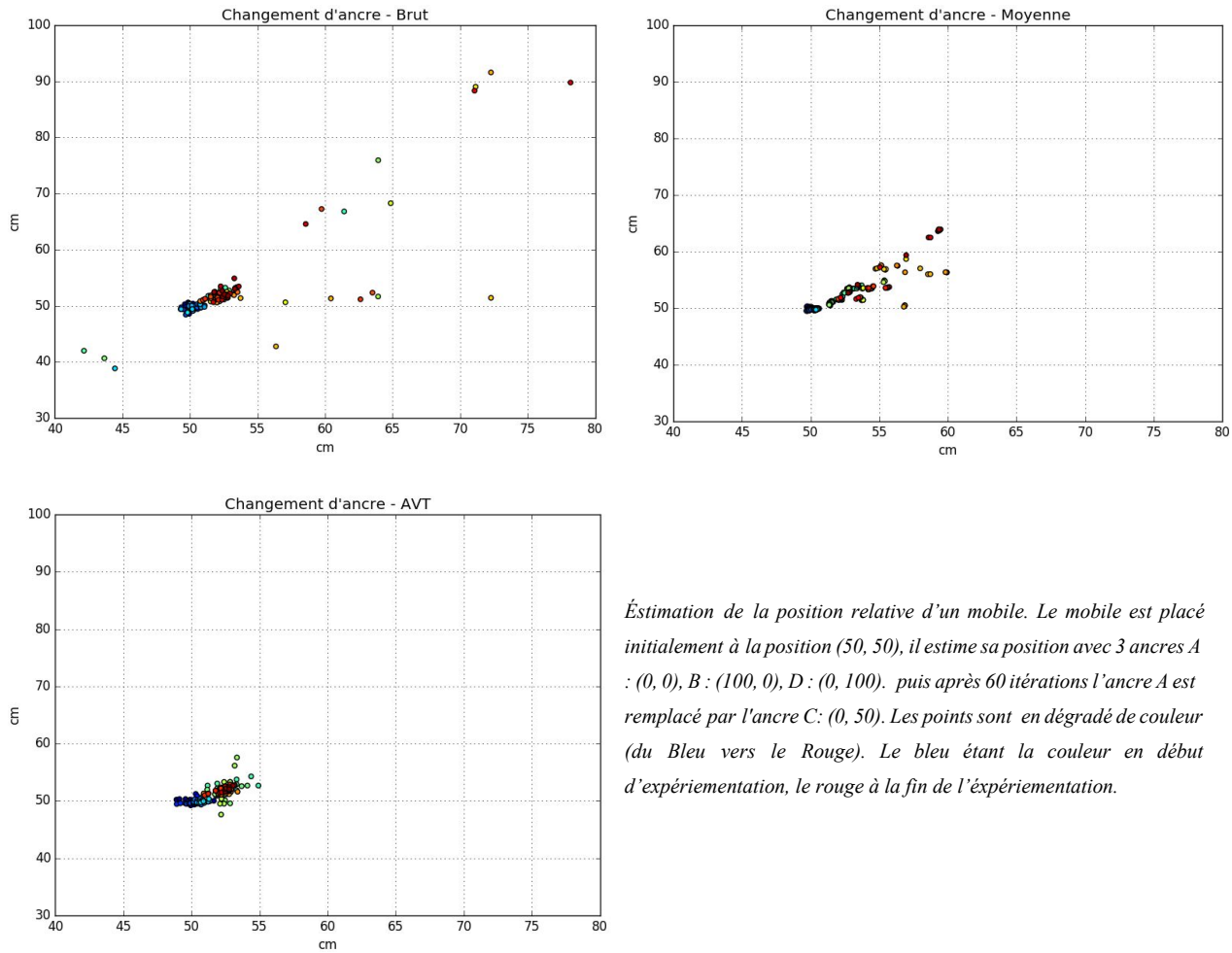
But :

Dans un environnement dynamique, il est nécessaire de pouvoir continuer à évaluer sa position après un changement d'ancre. Le but de ce test est donc d'évaluer la vitesse de re convergence de l'AVT après un changement d'ancre.

Configuration :

Les quatre ancres utilisées sont placées aux positions A : (0, 0), B : (100, 0), C : (0,50) et D : (0, 100). Le mobile est placé à la position (50, 50) et communique initialement avec les ancres A, B et D. Il va prendre des mesures toute les secondes pendant une minute, puis l'ancre A cessera d'émettre. Le mobile disposera alors de soixante nouvelles itérations pour retrouver sa position.

Résultat :



Éstimation de la position relative d'un mobile. Le mobile est placé initialement à la position (50, 50), il estime sa position avec 3 ancres A : (0, 0), B : (100, 0), D : (0, 100). puis après 60 itérations l'ancre A est remplacé par l'ancre C: (0, 50). Les points sont en dégradé de couleur (du Bleu vers le Rouge). Le bleu étant la couleur en début d'expériementation, le rouge à la fin de l'éxpériementation.

Durant les 60 premières itérations, l'algorithme AVT et l'utilisation de la moyenne permettent d'avoir une estimation de la position plus précise. Après le changement d'ancre l'utilisation de l'algorithme AVT a permis de corriger les valeurs bruitées et d'obtenir des estimations de position plus robustes que la moyenne ou les données brutes

4.3.3. Déplacement 1 km/h

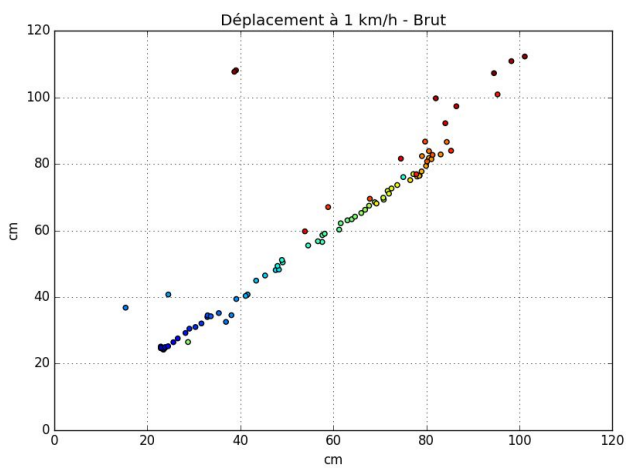
But :

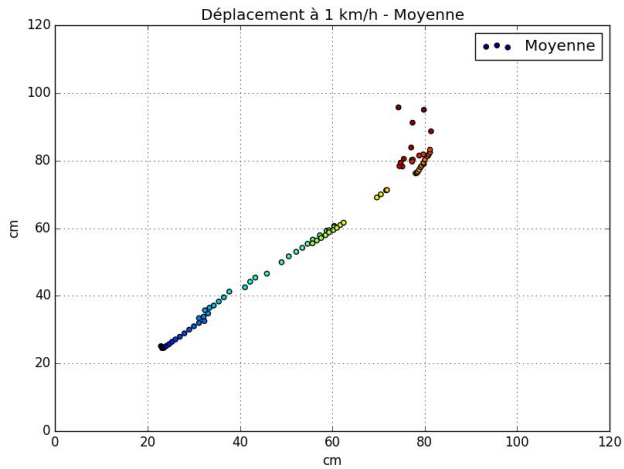
Le mobile se déplace à une vitesse de 1 km/h avec un mouvement rectiligne uniforme.Le résultat de ce test permettra de déterminer le comportement de l'algorithme AVT lors d'un déplacement continu.

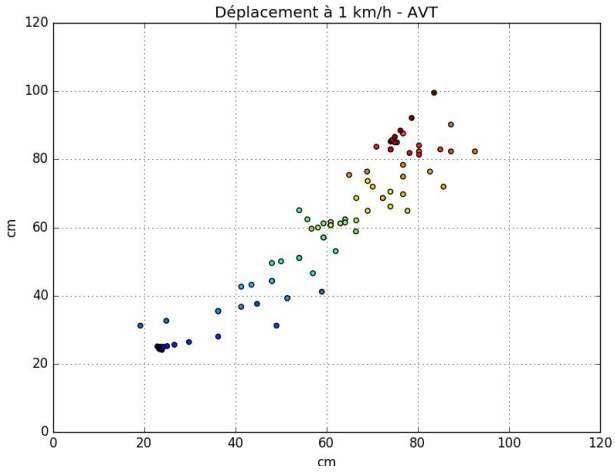
Configuration :

Pour ce test les trois ancres aux positions (0, 0), (100, 0) et (0, 100) sont utilisées, le mobile est placé à la position (24, 24) et se déplace jusqu'à atteindre la position (100, 100).
L'Arduino étant capable d'effectuer 10 itérations par seconde, le mobile est déplacé de 2,7 cm par itération.

Résultat :







Éstimation de la position relative d'un mobile lors d'un déplacement à 1 km/h. Le mobile est placé initialement à la position (24,24). Le mobile effectue une itération toute les 5 secondes. Entre chaque itération le mobile est déplacé de 2.7 cm pour arriver a la position (100, 100) à la fin de l'expérimentation. Les points sont en dégradé de couleur (du Bleu vers le Rouge). Le bleu étant la couleur en début d'expériementation, le rouge à la fin de l'éxpériementation.

Lors d'un déplacement continu, on remarque que l'utilisation de la moyenne permet au mobile d'avoir une estimation cohérente à chaque itération de sa position contrairement à l'algorithme AVT qui n'arrive pas à estimer de façon précise la position du mobile. La raison étant que lors du déplacement du mobile, les estimations de distances envoyés aux l'AVTs sont toujours plus grandes (ou plus petite) que lors de la précedente itération. Ce qui empêche l'algorithme AVT de pouvoir faire converger son pas.

4.3.4. Déplacement 2 km/h

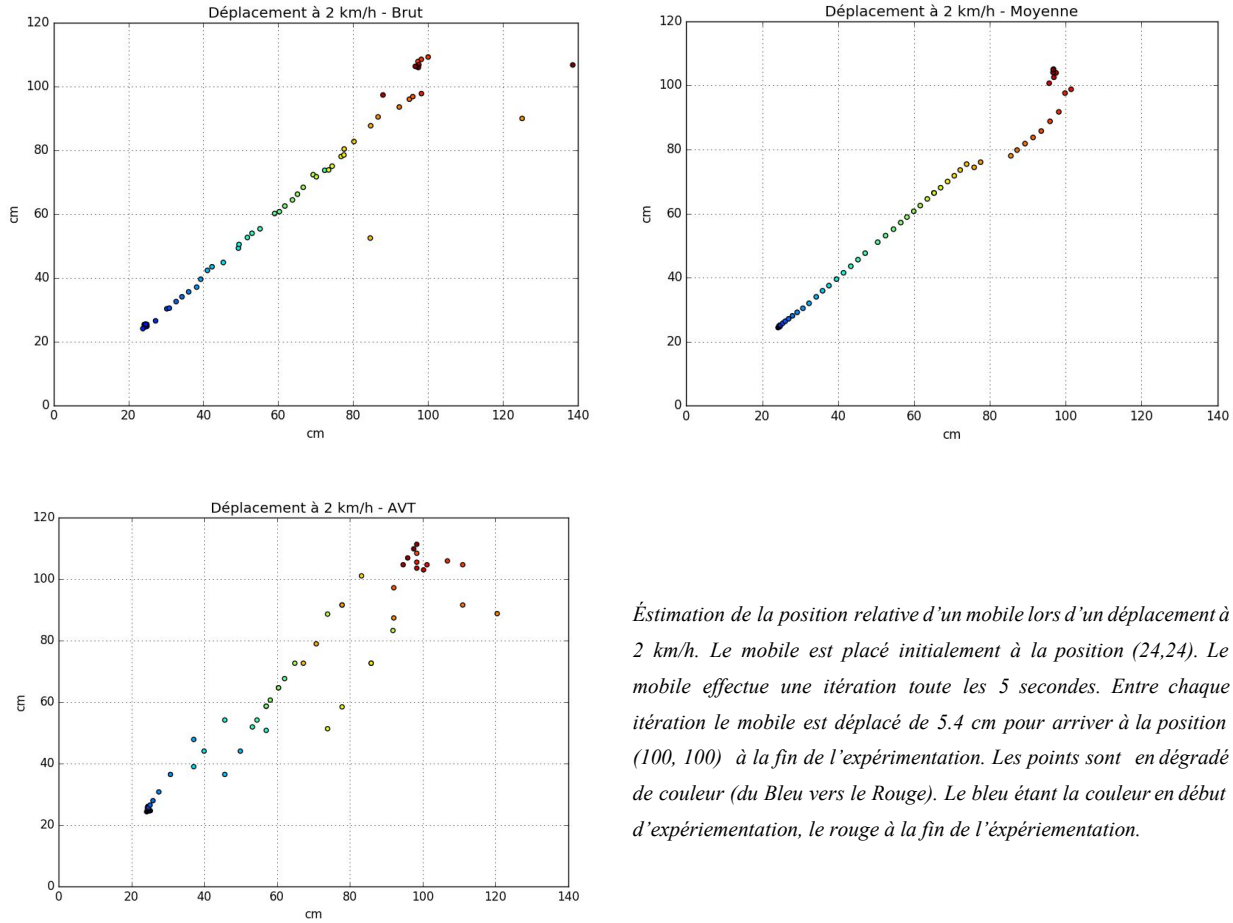
But :

Le mobile se déplace à une vitesse de 2 km/h avec un mouvement rectiligne uniforme. Le résultat de ce test permettra de déterminer l'impact de la vitesse de déplacement sur l'estimation de la position

Configuration :

Pour ce test les trois ancres aux positions (0, 0), (100, 0) et (0, 100) sont utilisées, le mobile est placé aux positions (24, 24) et se déplace jusqu'à atteindre la position (100, 100).
L'Arduino étant capable d'effectuer 10 itération par seconde, le mobile est déplacé de 5,4 cm par itération.

Résultat :



Éstimation de la position relative d'un mobile lors d'un déplacement à 2 km/h. Le mobile est placé initialement à la position (24,24). Le mobile effectue une itération toute les 5 secondes. Entre chaque itération le mobile est déplacé de 5.4 cm pour arriver à la position (100, 100) à la fin de l'expérimentation. Les points sont en dégradé de couleur (du Bleu vers le Rouge). Le bleu étant la couleur en début d'expériementation, le rouge à la fin de l'éxpériementation.

Les résultats du tests de déplacement à 2 km/h montrent des résultats similaires au déplacement à 1 km/h. L'algorithme AVT ne parvient pas à estimer de façon cohérente la position du mobile, lors d'un déplacement continu.

Synthèse des résultats expérimentaux :

Lors des essais en statique, l'algorithme a montré des résultats conformes à nos attentes et cohérents avec ceux obtenus dans l'article. La marge d'erreur observée est inférieure à cinq centimètres et lorsqu'un mouvement brusque à lieu, l'AVT reconverge sur une estimation correcte en moins de quinze itérations. Si l'AVT est un peu plus lent que la moyenne pour converger sur une position, il s'est révélé plus efficace en cas de perdition d'une ancre, avec un maximum d'erreur de moins de sept centimètres, contre près de vingt pour la moyenne.

Par contre, une fois placé dans un environnement dynamique, l'AVT perd énormément en précision. Les résultats obtenus sur les déplacements rectilignes uniformes à un et deux kilomètres par heure montrent une erreur moyenne de respectivement 7.16 cm et 10.02 cm avec des valeurs erreurs maximales de 23.67 cm et 27.26 cm. En comparaison, la moyenne permet d'obtenir une erreur moyenne de 3.84 cm et 9.66, pour une erreur maximale de 7.14 et 15.58.

5. Discussion et perspectives

Au-delà du simple test de performance, les essais expérimentaux menés ont pour but de juger de l'utilisabilité de l'algorithme dans le contexte qui nous est fixé. Permettant ainsi de préciser son cadre applicatif et de valider l'intérêt de son utilisation dans un tel contexte.

Pour rappel, le but de ce projet est de fournir à des entités mobiles ne disposant pas de moyen de localisation absolue, un moyen efficace et robuste de s'auto-localiser relativement aux autres noeuds du système.Afin répondre à ces exigences, l'algorithme en question doit être capable de :

- Localiser son hôte de manière statique.
- Rester cohérent en cas de déplacement.
- Exécuter les deux tâches précédentes avec le peu de ressources qui lui sont allouées.

Dans l'optique d'évaluer ces trois critères, plusieurs test ont donc été réalisés. Chacun de ces essais est accompagné des valeurs brutes reçues par le mobile, servant de référence, ainsi que de la position estimée en faisant la moyenne des dix dernières valeurs brutes reçues. L'intérêt d'une telle comparaison réside dans le fait que la moyenne est une méthode d'agrégation simple, qui fut initialement écartée à cause de sa trop grande consommation de mémoire. Il est cependant possible, en limitant le nombre de valeurs retenues, d'aligner cette consommation sur celle de l'AVT, et d'obtenir ainsi une méthode d'agrégation comparative, ayant une charge de travail équivalente. En effet, lorsque les cinq AVTs sont utilisés, ils requièrent ensemble trente-cinq variables et effectuent un total de cinq additions et cinq multiplications. La moyenne, limitée à dix valeurs, demande quand à elle trente deux variables, dix additions et une division.

Nos conclusions sont donc que :

L'algorithme remplit parfaitement son rôle dans le cadre d'une localisation en statique, même avec d'éventuels changements brusques de position. Il se rattrape très bien quand une ancre est défaillante et accompli sa mission en un temps suffisamment court.
Néanmoins, lorsque sont hôte subit un déplacement continu supérieur à un kilomètre heure, l'algorithme ne parvient pas à maintenir une mesure correcte de sa trace. La précision de l'algorithme et sa robustesse face au bruit ne sont pas suffisantes pour assurer un suivi précis de la position de son hôte.

En l'état actuel, l'algorithme conviendrait donc à une entité à faible ressource, nécessitant une localisation statique, qui ne se déplacerait pas, ou seulement lentement. Il n'est pas possible de l'utiliser sans modification sur une machine qui serait susceptible de se déplacer et qui aurait besoin de connaître suffisamment précisément sa position pendant ses déplacements. En revanche, la précision obtenue semble suffisante pour pouvoir estimer la direction dans laquelle se dirige l'hôte.

Les résultats obtenus par le calcul de moyenne en dynamique sont bien plus précis et cohérents que ceux de l'AVT, ce qui fait donc de la moyenne un candidat à reconsiderer pour compléter ou remplacer l'AVT dans l'algorithme. Il est cependant nécessaire de considérer le fait qu'un mouvement rectiligne seul ne suffit pas à rendre compte de toutes les situations auxquelles peut être confronté l'hôte de l'algorithme. Il est alors difficile de conclure sur le choix le plus pertinent.

Dans l'idée d'améliorer notre implémentation, ainsi que l'algorithme en général, nous avons réfléchi à plusieurs approches. Celles-ci n'ont pas put être explorées dans les délai imposé mais voici néanmoins

les pistes que nous avons identifiées.

Le principal défaut de notre système est le nombre d'itération qu'il peut produire, en moyenne dix par seconde. Pour identifier la source d'une telle lenteur, nous avons mesuré d'une part le temps d'acheminement des messages dans notre réseau et le temps d'exécution du traitement des messages, donc des calculs de trilatération et ceux du aux AVTs. Nos résultats montrent que la durée de traitement une fois le message reçu est négligeable, en l'occurrence, elle est inférieure à une milli-seconde sur les deux plateformes. Le temps associé à une itération est donc quasi intégralement consacré à la communication. La latence observée dans les communications est dûe à deux phénomènes distincts. D'abord, notre incapacité à mettre en place un réseau Ad-Hoc nous impose d'utiliser un serveur central qui doit re-transmettre les messages. Ensuite, l'Arduino et l'ESP sont deux appareils différents qui doivent communiquer entre eux. Et bien que cette communication soit filaire, celle-ci se révèle être relativement lente. Ainsi, lorsqu'un mobile demande une mesure à une ancre, le message suit le chemin suivant : De l'Arduino à l'ESP, de l'ESP au serveur, du serveur à l'autre l'ESP, et enfin, de l'ESP au serveur. On compte donc quatre étapes pour une demande, et huit pour une question réponse. Sachant que les demandes aux trois ancres sont séquentielles et que le mobile envoi un message de log à chaque itérations, on obtient (4 + 4) * 3 + 1 soit 25 étapes par itérations. Les problèmes de précision étant en grande partie dû au nombre insuffisant d'itérations et celui-ci étant provoqué par la latence du réseau, il apparait alors clairement que la majeure amélioration qui peut être effectuée se situe au niveau du réseau. Celle-ci peut être accomplie de plusieurs manière différentes.

Nous pouvons dans un premier temps, réduire le nombre de messages échangés. Deux solutions sont possibles. Offrir de nouveaux types de messages, pour permettre par exemple à un serveur de demander aux ancres leurs distances par rapport à un mobile, le nombre de messages passerait alors de 25 à 21. Ou alors, mettre en place un réseau ad-hoc qui permettrait une communication directe entre les noeuds. Le nombre de messages serait à ce moment là réduit à seulement 19.

L'autre solution, et selon nous la meilleure, consisterait à utiliser un autre module que l'ESP pour permettre à l'Arduino de communiquer en Wi-Fi. En plus d'un gain considérable en terme de quantité d'étapes, la communication globale pourrais être très fortement accélérée par la suppression de la communication entre les deux appareils. Nous n'avons pas pu efficacement mesurer le temps de cette communication, cependant, lors nos travaux de prise en main de cette plateforme, nous nous sommes rendu compte que celle-ci pouvait être très lente.

La deuxième piste explorée concerne les performances de l'AVT en lui-même. Sur les tests de trajectoire rectiligne les résultats de l'AVT sont décevants. L'AVT, ou plutôt, l'implémentation que nous en avons fait, ne peut suivre correctement la trace d'un mobile en mouvement. Cela peut être corrigé, ou du moins grandement amélioré en jouant sur les paramètres de celui-ci. Lors des tests cités, les bornes du pas étaient les mêmes que celles des tests de l'article [0.1,50]. Or, en prenant en compte la vitesse du mobile et les marges d'erreur de mesure, il est possible de calibrer le pas en fonction de la situation. Par exemple, lors du test de déplacement rectiligne à 2 km/h, en supposant une cadence de 10 itérations/sec, on sait que le mobile se déplace de 2.77 cm maximum par itération. En ajoutant ensuite quelques centimètres de plus pour supporter les erreurs de mesure des outils d'évaluation de distance, on peut alors poser un pas maximum bien plus réduit. Dans notre cas, 2.77cm + 10 cm (valeur de bruit maximum enregistré sur les micros ultrasons utilisés) soit environ 13 cm semble être un bon choix. En réduisant le pas, l'AVT s'éloigne moins de la valeur cible dans une situation ou il ne peut converger. Pour compléter cette méthode, ou pourrait imaginer un AVT qui auto-ajusterait les bornes de son pas en fonction de paramètres transmis par son hôte. Si un mobile connais, au moins approximativement sa vitesse de déplacement, l'AVT peut alors s'en servir.

La dernière piste explorée traite du bruit présent sur les mesures de distance. Lorsque les mesures sont obtenues avec des outils comme ceux qui étaient à notre disposition, celui-ci augmente en fonction de cette même distance. Or, plus le signal reçu est bruité, moins la localisation est précise. Étant donné que

notre système peut contenir plus d'ancres que le nombre minimal nécessaire à la trilatération, il est possible de mettre en place un système de filtrage des ancres. Le mobile choisirait alors un nombre d'ancre suffisant parmi le panel que lui propose le serveur, en utilisant une méthode d'évaluation. Cette méthode d'évaluation pourrait prendre en compte la distance avec l'ancre, son temps de réponse, le qualité de son signal etc…au risque de devenir beaucoup plus gourmande que la moyenne.

6. Bibliographie

1. Arduino - ArduinoBoardDue." 2015. 18 May. 2016 <https://www.arduino.cc/en/Main/ArduinoBoardDue>

2. "Raspberry Pi - Teach, Learn, and Make with Raspberry Pi." 2011. 18 May. 2016 <https://www.raspberrypi.org/>

3. Agliamzanov, Ramil, Assia Belbachir, and Kasim Sinan Yildirim. "Robust and Efficient Self-Adaptive Position Tracking in Wireless Embedded Systems." *2015 8th IFIP Wireless and Mobile Networking Conference (WMNC)* 5 Oct. 2015: 152-159.

4. Yildirim, KS. "Efficient Time Synchronization in a Wireless Sensor Network by Adaptive Value Tracking " 2014. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6785999>

5. "Everything ESP8266." 2014. 18 May. 2016 <http://www.esp8266.com/>

6. "Créer une réseau ad hoc - Raspberry Pi, Robots ... - Sites - Google." 2013. 16 May. 2016 <https://sites.google.com/site/blainoperso/debuter-avec-raspberry-pi/14---creer-une-reseau-ad-hoc>

7. "ESP8266 AT Command Set - pridopia.co.uk." 2015. 16 May. 2016 <http://www.pridopia.co.uk/pi-doc/ESP8266ATCommandsSet.pdf>

8. "GitHub - itead/ITEADLIB_Arduino_WeeESP8266: An easy-to-use ..." 2015. 16 May. 2016 <https://github.com/itead/ITEADLIB_Arduino_WeeESP8266>

9. "Everything ESP8266 - MESH network (any sort of?) - ESP8266 ..." 2014. 16 May. 2016 <http://www.esp8266.com/viewtopic.php?f=6&t=903>

10. "GitHub - olab-io/meshish: A mesh-like Arduino ESP8266 library for ..." 2015. 16 May. 2016 <https://github.com/olab-io/meshish>

11. "HC-SR04 User's_Manual - Google Docs." 2012. 18 May. 2016 <https://docs.google.com/document/d/1Y-yZnNhMYy7rwhAgyL_pfa39RsB-x2qR4vP8saG73rE/edit>

12. "Arduino - container for Objects like c++ vector? - Sites - Arduino." - 2010. 16 May. 2016 <http://forum.arduino.cc/index.php?topic=45626.0>

13. "GitHub - PabloPetit/ProjetPositionementWifi: Projet Androide S2." 2016. 18 May. 2016 <https://github.com/PabloPetit/ProjetPositionementWifi>

7. Annexes

7.1. Planning prévisonnel

1. Première découverte de la solution proposée par l'article et prise en main des deux plate-formes **(10 jours, soit jusqu'au 7 février)**

□2. Mise en place d'un réseau Wifi ad-hoc couvrant les Raspberry et les Arduino **(3 semaines, soit jusqu'au 29 février)**

□3. Implémentation de l'algorithme sur les plateformes **(2 semaines, soit jusqu'au 14 mars)**

□4. Test de celui-ci dans un contexte statique et élaboration des scénarios d'expérimentation en dynamique **(2 semaines, soit jusqu'au 1er avril)**

□5. Expérimentations dans un contexte dynamique à partir des scénarios retenus, propositions d'améliorations et rédaction du rapport **(1 mois, jusqu'au 1er mai)**

□6. Livraison du rapport, du code, et de la documentation technique et fonctionnelle associée: **18 mai.**

7.2. Sujet

Positionnement relatif et automatique sur systèmes embarqués hétérogènes à l'aide d'un signal WIFI
(2 étudiants)

***Mots-clés** : systèmes embarqués, agents mobiles, robotique, Wifi adhoc, raspberry, arduino*

***
Sujet : L'auto-localisation d'une entité mobile ne disposant pas d'un composant offrant un positionnement absolu et évoluant en environnement ouvert est aujourd'hui un problème non résolu. En effet, la variabilité et la complexité d'un tel environnement rendent la détermination d'une position précise complexe et coûteuse. Cette complexité n'est pas adaptée aux systèmes disposants de ressources limitées (tant en calcul qu'en énergie). Les approches basées sur la triangulation représentent une classe de solutions moins gourmandes en ressources mais très sensibles à la dynamicité de l'environnement, ce qui les rend difficilement exploitables en environnement ouvert. Les travaux de Agliamzanov *et al* [1] représentent une piste de solution susceptible de palier cette limitation.

L'objectif de ce projet est de réaliser l'implémentation, le déploiement et le test de l'algorithme proposé dans l'article avec des composants mobiles et hétérogènes organisés en réseau adhoc via Wi-Fi. En fonction de l'avancement des étudiants, des propositions d'amélioration de l'algorithme pourront être proposées et testées.

***
Le matériel à disposition comprend : 2 raspberry (B et 2) et 2 arduino équipés de dongles wifi
***

Les tâches à réaliser sont donc :
1. Analyse de la solution proposée par l'article et prise en main des deux plate-formes
2. Mise en place d'un réseau Wifi ad-hoc
3. Implémentation de l'algorithme
4. Test de celui-ci dans un contexte statique
5. Test de celui-ci dans un contexte dynamique
5. Propositions d'améliorations
6. Livraison d'un rapport, du code, et de la documentation technique et fonctionnelle associée.

***
Nombre d'étudiants : 2.
***
Prérequis : Bonnes connaissances en python/C , un intérêt pour les technologies sans fil et les systèmes embarqués.

[1] Robust and Efficient Self-Adaptive Position Traking in Tracking in Wireless Embedded Systems. Ramil Agliamzanov, Önder Gürcan, Assia Belbachir, Kasım Sinan Yıldırım; In WMNC 2015, Germany.

7.3. Manuel Utilisateur

1 - Raspberry Pi

Comme expliqué dans la documentation technique, le code sur Raspberry est composé de deux exécutables : un qui permet de créer un serveur et un second qui permet de créer une ancre ou un mobile. Nous allons voir ici comment utiliser chacun des deux.

1.1 - Serveur

Pour créer un serveur, il faut exécuter le fichier python server.py :

-$ python3 server.py

Si aucun argument n'est précisé, le serveur sera lancé avec les paramètres par défaut.

La liste des options et paramètre par défaut est la suivante :

-ip : Permet d'entrer l'adresse IP du serveur. Par défaut : « localhost »

-p : Permet d'entrer le port du serveur. Par défaut : 4000

-l : Permet d'activer le mode verbeux. Par défaut : False

-mxQ : Indique la taille de la queue d'acceptation du serveur. Par défaut : 5

Une fois le serveur lancé, l'utilisateur aura la possibilité d'entrer des commandes dans la console.

Les commandes disponibles sont les suivantes :

- exit/quit : Ferme le serveur. Il est recommandé d'utiliser cette commande pour quitter le serveur plutôt que couper brutalement l'exécution.

- list_m : Affiche la liste des mobiles connectés.

- list_a : Affiche la liste des ancres connectées.

- log : Permet de visualiser ou d'enregistrer les logs reçu d'un client

1.2 - Ancres et Mobiles :

Le fichier python RpiRunner.py permet de lancer une ancre ou un mobile selon l'argument donné à l'option -t .

Pour lancer une ancre :

-$ python3 Rpi Runner.py

Pour lancer un mobile :

-$ python3 Rpi Runner.py -t mob

Les options disponibles sont :

- Options communes au mobile et à l'ancre :

-ip : Permet d'entrer l'adresse IP du RpiRunner. Par défaut : « localhost »

-p : Permet d'entrer le port du RpiRunner. Par défaut : 4000

-l : Permet d'activer le mode verbeux. Par défaut : False

-t : Précise le type du RpiRunner, par défaut : Ancre

- Options propre à l'ancre :

-x : Précise la coordonnée x de la position de l'ancre. 1 par défaut.

-y : Précise la coordonnée y de la position de l'ancre. 1 par défaut.

-u : Précise si l'ancre doit utiliser le capteur ultrason. Dans le cas où elle ne devrait pas le faire, elle transmettra la valeur de la variable dist bruitée. Par défaut : Vrai.

-d : Précise la valeur à transmettre dans le cas où l'ancre n'utilise pas son capteur. Par défaut : 42.

Pour arrêter l'exécution d'un RpiRunner, il faut taper la commande « exit » ou « quit » dans la console. Il est particulièrement recommandé de ne pas stopper l'exécution autrement dans le cas d'une ancre qui utilise un module ultrason, ceci empêcherai le programme de nettoyer les ports GPIOs de la Raspberry Pi.

2 – Arduino Due

2 . 1 – Configuration

Les variables correspondant à la configuration de l'Arduino se trouvent dans le fichier « ***Config.h*** ».

- SSID :
Permet de nommer le réseau, par défaut : « NETGEAR »

- PASSWORD :
Permet la saisie du mot du passe pour la connexion au réseau, par défaut : *pas de mot de passe*

- SERVER_ADDR :
Permet d'entrer l'adresse du serveur TCP, par défaut : « 192.168.0.4 »

- PORT :
Permet d'entrer le port du serveur, par défaut : « 4000 »

- NODE_TYPE :
- **ANCRE** : Permet d'indiquer que l'Arduino est une ancre. Il est alors nécessaire d'indiquer les valeurs de **POS_X** et **POS_Y**,

- **MOBILE** : Permet d'indiquer que l'Arduino est un Mobile.

- POS_X :
Permet de préciser la coordonnée X de l'ancre, par défaut : « 0,0 »

- POS_Y :
Permet de préciser la coordonnée Y de l'ancre, par défaut : « 0,0 »

2 . 2 – Exécution

Pour créer un mobile ou une ancre, il faut télécharger le code compilé sur l'Arduino.
L'Arduino exécutera le code compilé une fois le téléchargement complet et l'exécution s'arrêtera une fois l'Arduino éteinte.

7.4. 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 appeler la fonction distance(). La vitesse du son est définie 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. Ensuite, il faut 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 GPIO utilisés.

1.2 - avt.py :

Ce fichier contenant le code de la classe AVT. Celle-ci implémente un 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 performatives des messages, la classe message, la classe client
ainsi que deux fonctions d'encodage et de décodage de float vers des bits.

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

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 morceaux représentés par les classes suivantes :

- Console :

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

La liste des commandes disponibles est la suivante :

- exit : quitte le programme.
- list_m : affiche la liste des identifiants des mobiles connectés au serveur.
- list_a : affiche la liste des identifiants des ancres connectés 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 créé 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 créée à chaque nouvelle connexion avec un client.
Deux arguments sont demandés, 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.

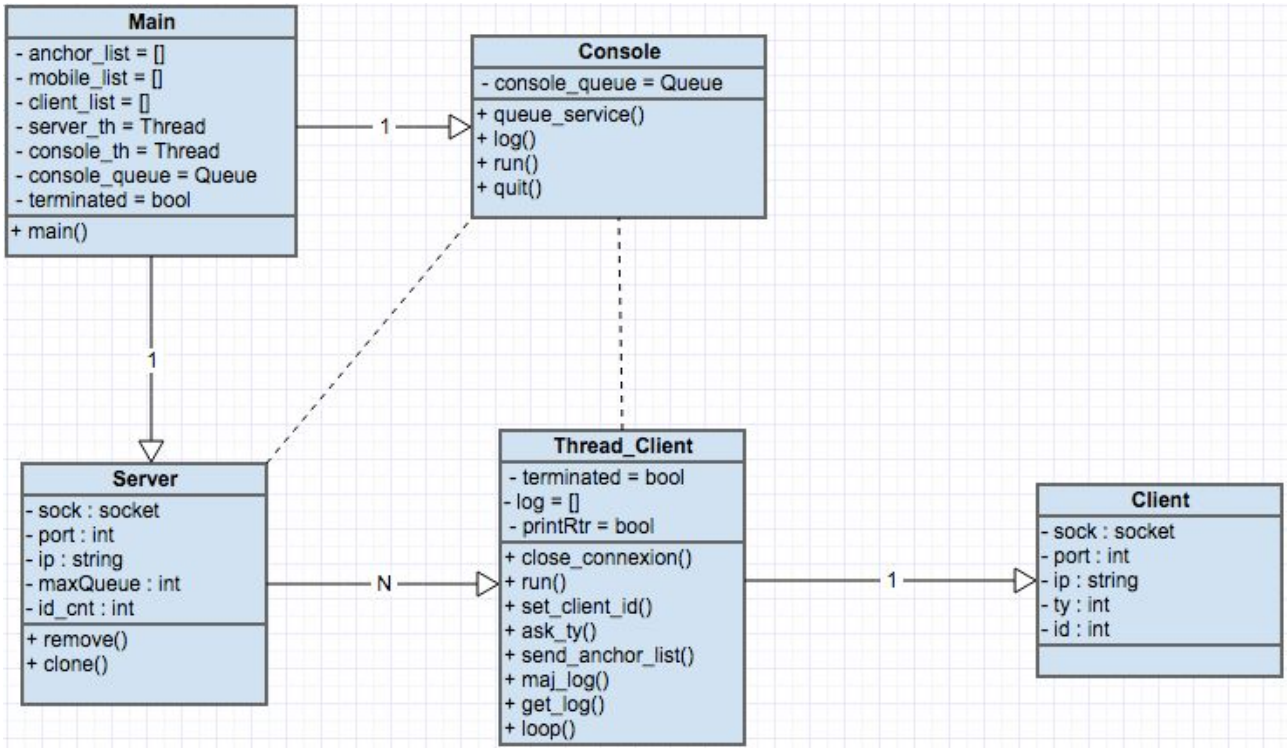


Diagramme de classe de l'implémentation du serveur sur Raspberry Pi

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 morceaux 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 représenté par une instance de cette classe.

Le RpiRunner est le bloc d'instructions qui va 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 terminées, le RpiRunner lancera les fils d'exécution nécessaires 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 nécessaires à l'ancre. Ils seront détaillés dans la section dédiée 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 utiliser le module Ultra
  - Une valeur de distance.

  Dans le cas où on 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 évaluation de distance. Dans le cas contraire, la valeur renvoyée 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

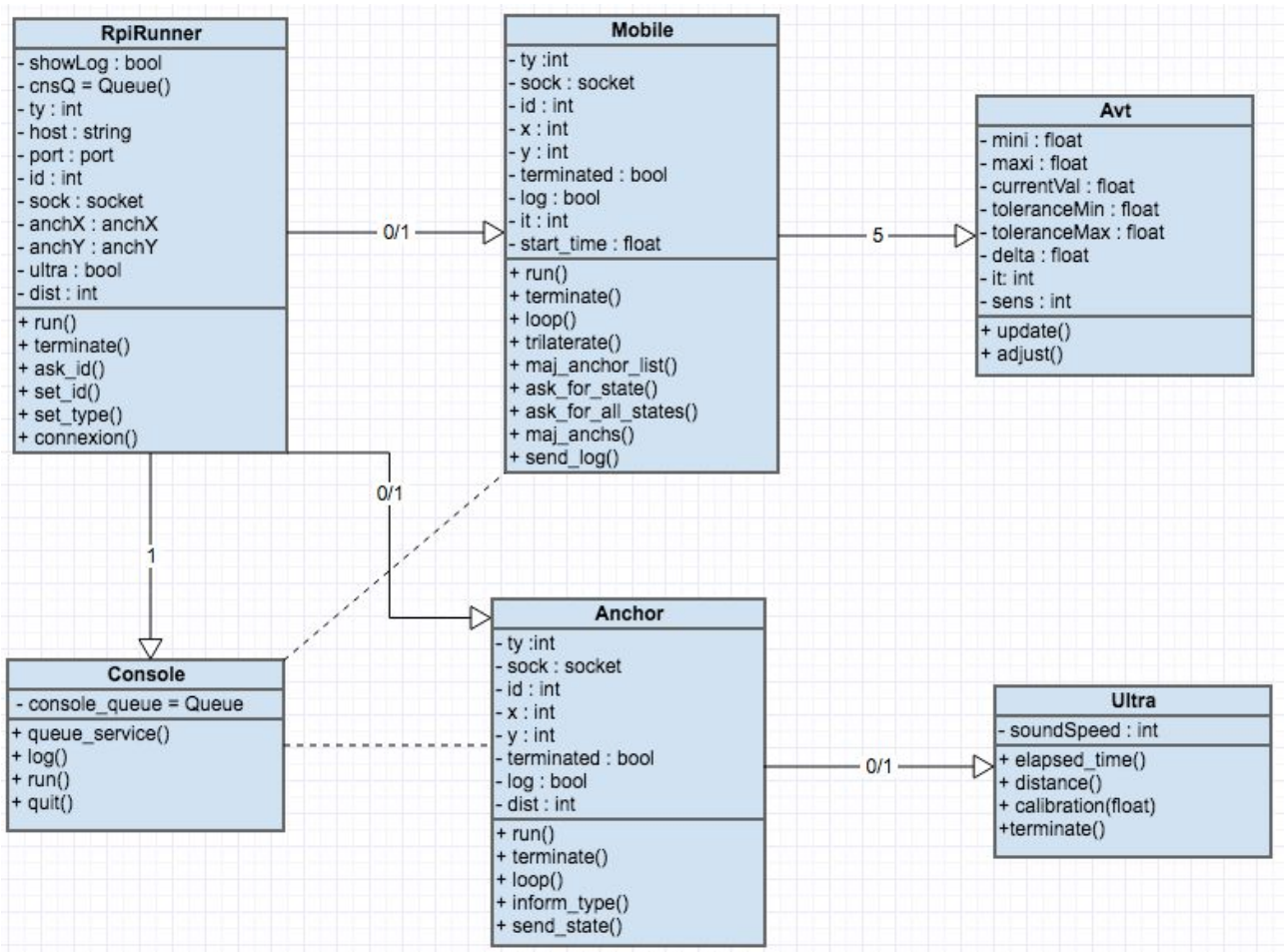


Diagramme de classe de l'implémentation des noeuds sur Raspberry

2 – Arduino

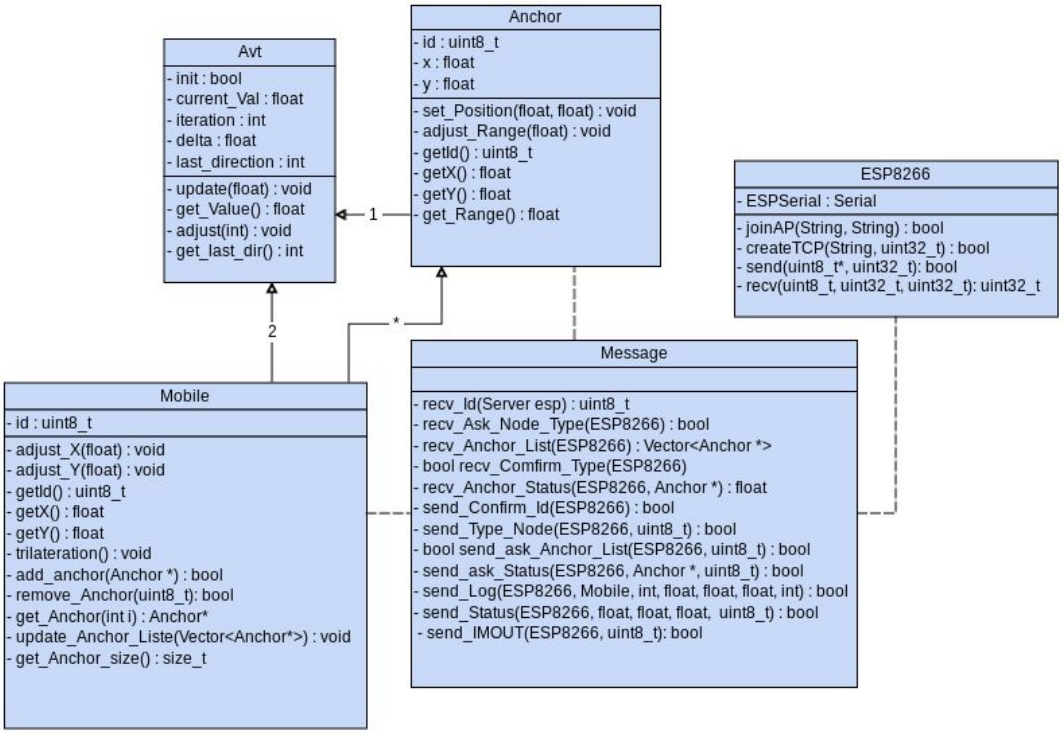


Diagram de classe de l'implémentation des noeuds sur Arduino Due

Cette partie explique l'architecture de la partie Arduino. Il est écrit dans le langage Arduino qui est basé sur le C/C++.

2.1 - Vector

La classe **Vector** (native en C++) est absente des librairies standard de l'Arduino. Une version ré-implémenté pour l'arduino a été utilisé[12].

2.2 - ESP8266

Cette classe fourni une interface de communication avec le module ESP8266[8] de plus haut niveau. Elle implémente toutes les commandes de l'ESP[2]

- bool joinAP(String ssid, String pwd)

Cette méthode permet de rejoindre le réseau où se trouve le serveur.
Elle prend en paramètre le nom du réseau (**ssid)** ainsi que le mot de passe (**pwd**).
Elle renvoie un **Booléen** correspondant au succès ou à l'échec de la connexion

- bool createTCP(String addr, uint32_t port)

Permet la connexion au serveur TCP.
Elle prend en paramètre l'adresse du serveur TCP (**addr**) et le port (**port**) où se trouve le serveur.
Le booléen retourné indique si la connexion avec le serveur TCP a réussi.

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

Permet l'envoi d'un message au serveur TCP de taille **len(32 bytes)** comme indiqué dans le protocole sous la forme d'un tableau de bytes (**const uint8_t *buffer**). Le booléen de retour indique si le message a bien été envoyé.

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

Permet de recevoir les messages venant du serveur TCP sous la forme d'un tableau de byte de taille maximum **buffer_size (32 bytes)**.
Si aucun message n'est reçu avant la fin du **timeout** la fonction renvoie 0, sinon elle retourne la taille du message reçu.

2.3 - Message

Cette classe implémente l'envoi et la réception des messages décrits par le protocole de communication mis en place. Ce protocole est composé de 15 prérogatives celle-ci étant déjà décrite précédemment.

- uint8_t recv_Id(ESP8266 esp) :

Permet de récupèrer l'id attribué par le serveur.

- bool recv_Ask_Node_Type(ESP8266 esp) :

Permet de recevoir une demande de type de la part du serveur *(Ancre ou Mobile)*
Retourne la confirmation de l'envoi.

- bool recv_Comfirm_Type(ESP8266 esp);

Permet la réception de la confirmation sur le type de nœud envoyé au serveur.
Retourne la confirmation de l'envoi.

- Vector<Anchor*> recv_Anchor_List(ESP8266 esp) :

Permet de recuperer la listes des ancres connues et actives connectées au serveur.
Renvoi la liste des ancres reçu

- float recv_Anchor_Status(ESP8266 esp, Anchor *ancre) :

Réception du statut d'une ancre. Un status correspond à la position x, y de l'ancre et de son évaluation de distance. Une fois le statut récupéré les informations de l'ancre sont mises à jour avec l'algorithme AVT.
Retourne la distance mesurée par l'ancre.

- bool send_Confirm_Id(ESP8266 esp):

Permet l'envoi de la confirmation de l'id attribué, une fois d'id reçu de la part du serveur.
Retourne la confirmation de l'envoi.

- bool send_Type_Node(ESP8266 esp, uint8_t node_type):

Permet d'envoyer au serveur le type de notre noeud (Ancre, mobile).
Retourne la confirmation de l'envoi.

- bool send_ask_Anchor_List(ESP8266 esp, uint8_t id):

Effectue une demande de la liste des ancres qui se sont enregistrées sur le serveur.
uint8_t id : Correspond à id du noeud qui effectue la demande de liste.
Retourne la confirmation de l'envoi.

- bool send_ask_Status(ESP8266 esp, Anchor *anchor, uint8_t id):

Permet l'envoi au serveur d'une demande de statut pour l'ancre **(Anchor *anchor)**.
uint8_t id : Correspond à id du noeud qui effectue la demande de statut.
Retourne la confirmation de l'envoi.

- bool send_Log(ESP8266 esp, Mobile mobile, int iteration, float d1, float d2, float d3, int memory):

Permet d'envoyer au serveur le log d'une itération.
Le log comprend les valeurs de l'AVT pour les positions x et y du mobile, les données brutes correspondant aux évaluations des trois ancres (d1, d2, d3) utilisées pendant l'itération ainsi que les valeurs de l'AVT actuelles de ces ancres suivie du numéro de l'itération.

- bool send_Status(ESP8266 esp, float x, float y, float d, uint8_t id):

Permet l'envoi du statut au serveur (uniquement si le noeud est une ancre).

- **uint8_t id** : L'id du noeud qui a effectué la demande de statut.
- **float x** : la position X de l'ancre.
- **float y** : la position Y de l'ancre.
- **float d** : estimation de la distance entre l'ancre et le mobile.

- bool send_IMOUT(ESP8266 esp, uint8_t id);

Permet l'envoi au serveur d'un message annonçant que le nœud quitte le réseau. « **uint8_t id »** id du nœud annonçant la sortie du réseau.

2.4 - Ancre

Les fichiers Ancre(.cpp/.h) fournissent deux classes **Ancre** et **Mobile**.

Ancre :

Cette classe correspond à la représentation d'une ancre vue par le mobile.
Elle possède une position X, Y « fixe », un Id permettant au serveur de l'identifier ainsi qu'un objet AVT pour l'évaluation de distance.

- void set_Position(float x, float y)

Permet de definir la position (x, y) de l'ancre.

- void adjust_Range(float r)

Cette fonction permet de mettre à jour l'AVT avec la distance reçue en paramètre.

- Les fonctions suivantes renvoient les variables de la classe **Ancre**

uint8_t getId(), float getX(), float getY(), float get_Range()

Mobile :

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

Ces fonctions permettent de mettre à jour les AVTs correspondant à la position X et Y du mobile avec les positions reçues en paramètre

- void trilateration()

Cette fonction effectue une trilatération du mobile avec les positions de trois ancres.

- void update_Anchor_Liste(Vector<Anchor*> liste)

Permet de mettre à jour la liste des ancres connu du mobile avec la « liste » des ancres envoyées par le serveur.

- Les fonctions suivantes renvoient les variables de la classe **Mobile**

uint8_t getId(), float getX(), float getY()

2.5 - AVT

La classe AVT est similaire à la classe AVT développée en python pour la Raspberry, et les fonctions y sont implémentées de façon similaire.

- void update(float value);
- float get_Value()
- void adjust(int direction)

2.6 - Node

Le fichier **Node.ino** est le fichier ***main*** pour l'Arduino. Il contient deux fonctions obligatoires pour que l'Arduino puisse fonctionner :

- setup():

"*Setup*" permet de mettre en place la configuration du noeud :

- Connexion au reseau.
- Connexion au serveur TCP
- Configuration du nœud (Ancre ou Mobile)

Si le nœud est un mobile, une demande de la liste des ancres ainsi que leurs positions est effectué.

- loop() :

Une fois la configuration terminé. La fonction “loop” est appelé en boucle jusqu’à l’arret de l’arduino. Si le nœud est un mobile,cette fonction effectue des demandes successives de distance à trois ancres sélectionnées calcul la position (x, y) du mobile par trilatération.

Si le nœud est une ancre, la boucle attend une demande d’évaluation de distance, de sa position ou son type.

2.7 - Config

Le fichier “***config.h***” contient les variables de configurations de l’Arduino.

- Le nom du réseau ainsi que le mot de passe.
- L’adresse et port du serveur TCP.
- Le type de nœud que représente l’arduino (Mobile, Ancre).
- La position du noeud, si celui-ci est une ancre.