SR05 - Etude

Application répartie embarquée pour réseaux dynamiques

2 décembre 2019

Étudiants : Hugo Malet - Louis Caron
Quentin Penon - Simon Bazin
Simon Duval - Thibaud De Filippis
William Lenoir - Thomas Paita

Enseignant UTC : M. Ducourthial

Juin 2019

Table des matières

1 Introduction

L'importance de la robotique au sein de nos sociétés a grandement augmenté durant les dernières années.

L'utilisation massive de drones dans de nombreux domaines est un bon exemple de cette évolution technologique.

Un robot est en effet très intéressant pour la réalisation de nombreuses tâches jusqu'ici réservées aux humains. Ils sont plus rapides, moins chers et plus disponibles que des employés humains. De la même façon que la révolution industrielle vis l'apparition de nombreuses machines remplaçant certaines tâches faites par des humains, l'utilisation massive de drones dans certains domaines semble inévitable.

Un domaine très prometteur pour l'utilisation de ces robots est le *mapping* d'une zone inconnue.

Explorer une zone peut en effet s'avérer long, fastidieux et même parfois dangereux pour un humain, tandis que des robots pourraient parfaitement réaliser cette tâche et fournir une cartographie fiable de la zone étudiée.

L'utilisation de drones pour surveiller ou explorer une zone est déjà relativement répandue et bien étudiée académiquement. Cependant, l'utilisation d'un seul robot n'est pas satisfaisante pour explorer une large zone, car celui-ci peut alors mettre un certain temps à accomplir sa tâche.

En distribuant la tâche à plusieurs drones travaillant de façon coordonnée, il est possible de réduire le temps d'exploration et même dans certains cas d'obtenir de meilleurs résultats finaux.

L'utilisation de plusieurs robots ainsi que leur coordination pose cependant de nouveaux problèmes, tels que la répartition des tâches et la distribution des calculs à réaliser par exemple. Il faut ainsi s'assurer que les robots communiquent leurs informations locales entre eux, et qu'ils explorent efficacement la zone de sorte à ne pas avoir trop de redondance entre eux.

Dans cette étude, nous nous intéresserons à l'exploration d'une zone inconnue par un essaim de robots coordonnés et décentralisés (pas d'ordinateur central pour réaliser la computation des informations). Nous commencerons par nous intéresser aux différentes informations dont notre robot a besoin pour fonctionner et se coordonner. Nous étudierons ensuite différents algorithmes permettant à chaque robot de prendre une décision et la façon dont ils communiquent entre eux. Enfin, Nous aborderons les problématiques auxquels nos robots font faces, ainsi que les solutions possibles pour y faire face.

2 Cartographie des zones

Le but de l'exploration d'une zone est d'en obtenir une carte fiable, une *map* globale. Dans notre cas, l'intérêt est d'utiliser plusieurs robots afin d'obtenir cette *map* globale plus rapidement en répartissant le travail d'exploration parmi les robots.

Idéalement, chaque robot devrait se répartir parfaitement les différents endroits à explorer de sorte à minimiser le temps d'exploration total.

Cela implique cependant de recevoir différentes informations de différents robots, et d'être capable de fusionner ces informations afin de créer la *map* globale.

2.1 Merging de map

Les informations envoyées par les robots doivent être mises en commun et fusionnées afin de générer une *map* globale. Cette *map* globale doit aussi pouvoir être continuellement mise à jour lorsque de nouvelles informations arrivent.

On pourrait traiter la *map* globale comme un variable partagée et gérer les accès concurrents à celle-ci avec des *lock*, ou encore via un algorithme de file d'attente répartie, mais ces options ne sont pas les plus performantes dans notre cas.

En effet, si l'on considère que chaque sous partie de la zone à explorer peut être soit un *obstacle*, soit *vide* ou soit *inconnue*, alors on retrouve des informations fusionnables et commutables.

Notre *map* globale est en un *CRDT* (1), plus précisément un *state-based CRDT* (Conflict-Free Replicated Data Type).

Les informations reçues des robots feront passer des zones du statut *inconnu* au statut *obstacle* ou *vide*. Une fois une zone connue, celle-ci ne peut plus repasser au statut *inconnu*. Cela permet de simplifier le transfert d'information, mais nécessite que l'environnement ne change pas et soit immuable.

La propriété de commutativité permet de ne pas se soucier de l'ordre de réception des informations. Chaque robot peut envoyer ses informations dans n'importe quelle ordre, et la fusion des informations reçues n'impactera aucunement le résultat final de la *map* globale une fois toutes les informations reçues. Cela permet de simplifier grandement le processus d'envoi d'informations, car il n'y a alors plus besoin d'algorithme réparti lourd du type file d'attente répartie pour envoyer les informations.

Les robots ont juste à envoyer leurs informations de l'exploration sans se soucier des envois des autres robots.

Concernant la propriété de fusionnabilité, elle n'est en réalité que partiellement correcte dans notre cas. Bien qu'elle soit valide lors de la fusion entre une zone *inconnue* et une zone *vide* ou *obstacle*, les zones *vide* et *obstacle* ne sont pas fusionnables.

Dans ce cas, on pourrait choisir de garder seulement la dernière information et d'écraser l'autre. Cependant, on perdrait alors la propriété de commutativité et si notre essaim possède des réplications des *map* globales, celles-ci pourraient ne pas être consistantes.

Si l'on suppose que nos robots envoient des informations "parfaites" (comprendre sans erreurs ni décalage par rapport au monde réel) et que notre monde est statique, alors ce problème ne devrait

même pas se poser. En effet, il est théoriquement impossible qu'une zone soit à la fois *obstacle* et *vide*. On peut donc le négliger si l'on considère ce cas.

Dans le cas où l'environnement pourrait évoluer, on perdrait notre hypothèse de commutativité et il faudrait constamment écraser les anciennes informations par des plus récentes. Nous nous intéresserons ici à l'exploration d'une zone immuable afin de simplifier le problème.

Même dans ces conditions, des fusions *obstacle* et *vide* peuvent avoir lieu à cause d'informations imparfaites, de décalage sur les position des robots...etc. Ce problème sera adressé plus bas dans cette section.

2.2 Différents positionnements relatifs

Un point important dans la fusion des *map* est la capacité à bien positionner les différentes informations reçues les unes par rapport aux autres. En effet, même si un robot envoi des informations sur une zone, il faut pouvoir placer cette zone sur la *map* globale ou par rapport aux autres informations.

Il existe plusieurs moyens pour positionner les robots les uns par rapport aux autres.

Le premier moyen est de simplement d'utiliser des coordonnées (GPS ou autres) afin de se positionner de façon globale. Cela permet de placer facilement les informations de chaque robot sur la *map* globale. On peut prendre l'exemple du GPS, qui permet de facilement situer tous les robots présents de façon globale.

Cependant, il existe aussi d'autres moyens d'utiliser des coordonnées. Les robots peuvent en effet avoir leurs propre système de coordonnées, et se placer eux-même sur celui-ci. Cela implique cependant de connaître sa position par rapport à ces coordonnées au moins une fois, afin de pouvoir se situer dessus.

Une fois sa position connue, un robot peut connaître sa localisation en calculant ses déplacements depuis des coordonnées connues. C'est le cas des robots sur lesquels nous avons travaillé. Ils commencent l'exploration en connaissant leur position relative, et peuvent par la suite déduire leurs coordonnées en prenant en compte les déplacements effectués.

2.3 Mitigation des erreurs des robots

Peu importe le système de localisation choisi, les erreurs restent toujours possibles. En effet, les informations envoyées par différents robots peuvent parfois être contradictoires et incohérentes. C'est notamment le cas lorsque plusieurs robots ont un léger décalage dans leurs coordonnées locales.

Un robot peut par exemple trouver un obstacle à certaines coordonnées, tandis qu'un autre y trouvera du vide, ce qui ne devrait théoriquement pas être possible selon notre hypothèse d'environnement immuable. En réalité, cela est simplement dû à une erreur de positionnement : les capteurs ayant des marges d'erreurs, le positionnement d'un robot peut être décalé par rapport à ce qu'il devrait être réellement.

Puisque qu'il est impossible de garantir une précision parfaite des équipements utilisés, il faut s'attendre à ce que des erreurs apparaissent.

Pour y faire face, des chercheurs américains (2) et mexicains (4) ont utilisé une approche probabiliste sur le statut d'une zone explorée.

Lorsqu'un robot explore un endroit, celui-ci ne vas pas donner directement le statut de la zone, mais plutôt une probabilité de présence d'obstacle à cet endroit. Cette probabilité est calculée à partir des données des capteurs des robots. Pour l'obtenir, il faudrait idéalement trouver l'estimateur de maximum de vraisemblance pour la *map* locale du robot en connaissant les données obtenues par ses capteurs. Le but est de déterminer une *map* qui s'approcherait de l'estimateur de maximum de vraisemblance en utilisant une méthode de *hill-climbing*. Cela permettrait alors d'avoir une *map* qui serait la plus consistante possible avec les données obtenues par le robot.

Cette estimation locale vise à trouver la *map* ayant la plus haute vraisemblance et minimisant les potentielles erreurs. Cette estimation locale est réalisée avec les données des capteurs du robot.

Afin d'améliorer encore plus la *map* globale, celle-ci va aussi combiner les données reçues des robots de sorte à minimiser les erreurs de localisation et à maximiser la vraisemblance de la *map* globale. Cela permet d'obtenir une représentation probabiliste fiable de la zone à explorer, et permet de bien tenir compte des potentielles erreurs de précision liées aux instruments utilisés.

3 Coordination des robots

Les robots sont capables d'évoluer seuls ou en essaim, et d'explorer un environnement en connaissant leur propre position dans celui-ci. Il est donc nécessaire que l'essaim ait accès à une carte contenant les informations découvertes par chaque robot sur l'environnement. Cette information pourra servir à mieux organiser les déplacements et la synergie des robots afin de minimiser le temps d'exploration, mais permettra aussi d'éviter les collisions entre robots.

3.1 Le cas centralisé

Dans de nombreux travaux traitant de l'exploration par des robots (2), la carte globale contenant l'information de tous les robots est contenue dans une entité centrale connectée à tous les robots. Les robots envoient régulièrement les informations qu'ils découvrent à l'entité centrale. Celle-ci s'occupe ensuite de fusionner les *map* et d'obtenir une carte cohérente de la zone explorée. Les robots connaissent éventuellement leur propres positions sur leurs *map* locales, mais ne connaissent pas nécessairement les positions des autres robots. Les robots n'ont alors aucune information sur les autres robots (ni même la connaissance de leur existence).

Si l'envoi d'informations est infaillible (pas de pertes de messages), alors il n'est même pas nécessaire pour les robots d'avoir une *map* locale incomplète puisqu'il suffit d'envoyer chaque nouvelle information à l'entité centrale pour que celle-ci possède toutes les informations nécessaires. L'entité centrale peut ensuite décider quelle chemin le robot doit prendre, et celui-ci peut s'exécuter sans avoir d'informations supplémentaires. Cela nécessite cependant l'envoi de nombreux messages, dont certains peuvent être redondants. Si un robot n'as pas de *map* locale, il n'a en effet aucun moyen de savoir quelle information est "nouvelle" ou pas. Il doit alors envoyer très régulièrement les informations afin de ne pas en perdre.

Les *map* locales permettent de palier à ce problème en conservant les informations du robot. En effet, il est alors possible d'envoyer moins de messages sur le réseau. Le robot peut envoyer un message de mise à jour à intervalles réguliers pour maintenir l'entité centrale au courant de sa position, ou alors il peut juste envoyer un message lorsqu'il découvre une "nouvelle" zone non explorée localement.

Même dans ce cas, les robots ont tous des informations incomplètes sur leurs environnement, et ils sont donc incapables de se coordonner entre eux sans passer par l'entité centrale. Celle-ci agit comme un cerveau, recevant toutes les informations des robots et s'occupant de répartir les différentes tâches entre les robots(2).

On peut par la suite facilement imaginer des variantes permettant de limiter les envois de messages, telle que faire envoyer par l'entité centrale une liste d'informations voulues par exemple. Cependant, bien que l'approche centrale soit intéressante, elle ne correspond pas à notre cas d'exploration distribuée.

3.2 Le cas décentralisé

Afin de permettre aux robots d'être indépendant et de décentraliser les informations, chaque robot devra posséder une *map* locale, mais contrairement au cas centralisé, aucune entité centrale ne sera présente pour centraliser les données.

Cette centralisation doit tout de même avoir lieu afin de déterminer les zones à explorer, la carte globale de la zone, et que chaque robot puisse prévoir ses futurs déplacements efficacement.

Une solution serait d'utiliser un algorithme d'élection afin de choisir un robot qui aurait pour rôle de centraliser les *map* locales de tous les robots. On pourrait ainsi retrouver un cas centralisé à partir d'un essaim de robots.

Cependant, cette approche n'est pas souhaitable. La centralisation représente en effet un point de blocage pour les performances, puisqu'il faut que l'entité centrale fusionne les *map* et calcule les prochains mouvements de tous les robots, ce qui peut demander beaucoup de puissance et de ressources. Lorsque la centralisation est bien prévue et que l'entité centrale possède une puissance adéquate, cette option est envisageable. Cependant, nos robots possèdent une puissance de calcul relativement faible individuellement, ce qui rend cette option inintéressante.

Bien que nos robots soient relativement peu performants individuellement, leur puissance de calcul combinée représente un atout non négligeable. Il serait intéressant de distribuer les tâches réalisées par l'entité centrale entre les différents robots. Pour cela, chaque robot doit posséder une *map* locale comme précédemment, mais doit aussi intégrer les informations obtenues par les autres robots. Ainsi, chaque robot pourra prévoir ses déplacements en ayant les informations globales à sa disposition (3).

Cette approche distribuées nécessite que chaque robot envoi ses données à tous les autres, et que chacun des robots fusionne sa *map* avec les informations qu'il reçoit. Cette fusion ne devrait pas poser de problème car la fusion d'information est supposée additive. Un robot devra ensuite faire ses propres choix de déplacement selon sa position et sa *map* globale.

La mise à jour de l'information globale peut se faire en envoyant des informations complètes ou partielles (les "nouvelles" informations seulement) comme dit précédemment. Il faut cependant prendre en compte que tous les robots doivent recevoir les informations de tous les autres ! Un algorithme de diffusion est donc nécessaire.

Le nombre de messages en transition pourrait aussi être important.

De plus, les robots sont incapables de se coordonner efficacement car ils n'ont pas de connaissances sur les futurs trajets des autres robots. En effet, l'entité centrale pouvait optimiser l'exploration en répartissant efficacement les trajets entre les robots. Dans le cas distribué, chaque robot doit faire ses propres choix en connaissant la *map* globale, mais pas les choix fait par ses voisins. Il est donc probable que certains trajets ne soient pas optimisés et que certaines informations obtenues soient redondantes.

La coordination efficace des robots peut tout de même avoir lieu. Afin de calculer le mouvement optimal selon une certaine heuristique de la même façon que l'entité centrale, les robots peuvent s'échanger leurs intentions de trajets par message et ensuite adapter leurs déplacements selon les choix de leurs voisins.

Encore une fois, cela augmente le nombre de message étant transmis à chaque décision.

4 Topologie dynamique

4.1 perte de messages

Dans notre cas d'exploration distribuée, chaque robot doit avoir une copie de la *map*. Lorsqu'un robot découvre une nouvelle région, il envoi les informations aux autres robots qui mettent à jour leurs propres *map* locales. Afin de pouvoir communiquer ces nouvelles informations, il faut que chaque robot puisse envoyer un message à tous les autres. Cela nécessite au minimum que les robots soient reliés entre eux selon un graphe connexe si l'on utilise un algorithme de diffusion.

Les robots ne reçoivent cependant pas les messages instantanément, et la topologie peut changer durant l'envoi d'un message. Certains messages peuvent ainsi potentiellement ne pas être reçus si la topologie change alors qu'un message est en train d'être diffusé (voir figure 1). La topologie dynamique rend la simple diffusion de message insuffisante pour assurer le transfert d'informations.

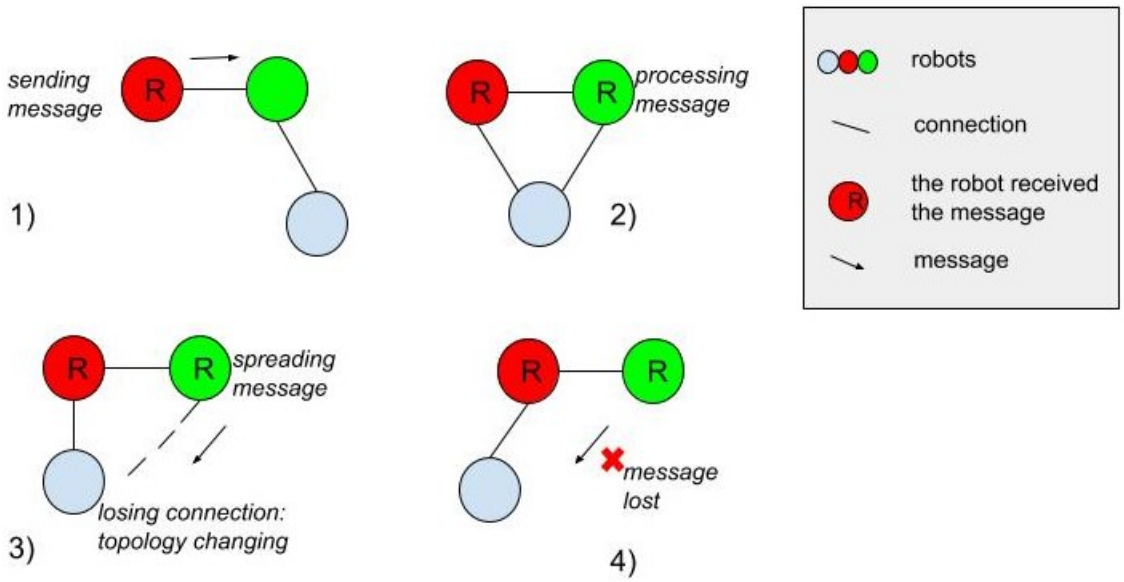

FIGURE 1 – Représentation de la perte d'un message lors d'un changement de topologie avec un algorithme de diffusion classique.

De plus, il est aussi envisageable que des robots perdent momentanément leurs connexions. Un robot pourrait se retrouver isolé en passant derrière un obstacle qui bloquerait les connexions, ou alors juste trop s'éloigner du reste de l'essaim.

Pour ces raisons, il faut prendre en compte que certains messages seront perdus et adapter les communications inter-robots adéquatement.
Puisqu'on ne peut pas garantir la consistance des *map* locales à chaque instant sans perdre en performance, nous devrons garantir leur *éventuelle consistance* au sens des *CRDT* (1).
il peut y avoir un laps de temps dans lequel les robots ont des *map* locales différentes (certains auront des *map* incomplètes). Cela ne pose pas de problème tant que les robots sont certains de recevoir l'information plus tard, car les *map* seront alors *éventuellement consistantes* au sens des *CRDT*.

4.2 Solutions potentielles

Chaque robot pourrait envoyer régulièrement toute sa *map* afin de tenir les autres robots au courant. Cela permet d'assurer que les *maps* sont éventuellement consistantes car les *map* agiraient alors comme des *State-based CRDT* (1) et cela permet aussi d'éviter le besoin de rediffusion des messages car les toutes informations se propageraient à chaque envoi de message, ce qui est simple à implémenter. Cependant, ce n'est pas applicable en pratique car une *map* entière peut vite représenter beaucoup de données et les messages deviendraient rapidement trop lourds pour le réseau.

Une autre option est d'envoyer les informations récentes seulement (ou dans un certain rayon autour de la position du robot). Si un robot est déconnecté du réseau pendant un court moment, il recevra quand même les informations manquantes dans le message suivant. Cela implique que le robot ne soit pas déconnecté trop longtemps et que chaque robot reçoive les messages des autres (diffusion des informations).

Si le robot reste trop longtemps déconnecté des autres robots, il faudrait avoir une méthode permettant au robot d'envoyer un message aux autres une fois revenu, afin que les robots échangent leur carte et mettent à jour les cartes internes.

Ces approches essaient de s'adapter à la perte de message et d'être capables de l'ignorer sans perdre d'informations. Une autre approche consiste à s'assurer que chaque robot reçoive bien tous les messages (qui lui sont destinés au moins). On peut alors juste envoyer les informations récentes de chaque robot une fois et laisser le protocole s'occuper de l'amener au bon endroit.

Chaque robot peut avoir un numéro de message local, *nMsg*, qui s'incrémente lors de l'envoi d'un message. Les robots possèdent aussi une sorte d'horloge vectorielle qui contient le dernier *nMsg* reçu pour chaque robot. Les robots gardent aussi en mémoire les X derniers messages qu'ils ont envoyés. A la réception d'un message, le robot vérifie que le *nMsg* reçu est égal au *nMsg* stocké plus 1. Si ce n'est pas le cas, alors le robot sait qu'il lui manque des messages. Il envoi alors une demande de message manquant à un robot (avec un timeout pour renvoyer la demande). Les autres robots vont recevoir ce message et le diffuser. Le robot émetteur du message manquant va alors aller regarder dans ses X derniers messages et renvoyer celui demandé. Avec cet algorithme, on est sûr que tous les robots auront les informations. Cela nécessite cependant de connaître ses X derniers messages, ou bien de savoir l'ordre dans lequel on envoie nos informations, ce qui peut poser problème du côté de la mémoire. Il est possible d'envoyer des informations dans un certain rayon autour de soi au lieu de garder des messages en mémoire. Cela rend les messages plus lourds, mais permet de ne rien garder en mémoire (mais quitte à choisir, il faut mieux avoir des messages légers et perdre un peu de mémoire).

Une autre possibilité est de s'inspirer du TCP et que chaque robot envoyant un message attende un accusé réception. Si un robot envoie un message, il stocke ce message et attend des accusés réceptions pour ce message particulier. Quand un robot reçoit le message, il renvoie un accusé réception à son émetteur, et diffuse le message par la suite. L'émetteur reçoit les accusés, et supprime le message de sa mémoire. Si l'émetteur ne reçoit pas le bon nombre d'accusés dans un certain temps (timeout), alors il renvoie le message initial. Cela permet de s'assurer que le destinataire recevra bien le message même si celui-ci est momentanément déconnecté. Puisque les informations envoyées sont commutatives et fusionnables (pour la *map*), les recevoir plusieurs fois n'est pas un problème.

Ainsi, si un robot ne reçoit pas les bons accusés, il peut juste envoyer un message avec la même information que le message précédent au destinataire manquant. Les autres robots ignorent ce 2ème message et le robot manquant l'obtient, puis renvoie son accusé réception.

Un accusé peut aussi se perdre, mais cela n'est pas un problème car le robot émetteur renverra le message et le robot manquant renverra alors l'accusé.
Cette méthode est plus "propre" et sécurisée, mais nécessite de connaître le nombre de robots à l'avance (pour le nombre d'accusés à attendre et les $nMsg$ correspondant), ainsi que leurs ID. Les envois d'accusés et les renvois dûs aux timeouts peuvent alourdir le réseau, mais les messages restent légers individuellement donc cela ne devrait pas trop influer sur le réseau.

5 La notion de frontière

Les algorithmes que nous allons étudier reposent sur la notion de frontière. Dans le cas d'un environnement découpé grâce à un quadrillage, une frontière peut être définie comme un ensemble de cellules contiguës qui se trouvent à la limite entre les zones explorées et les zones inexplorées (5). Les cellules *obstacles* ne peuvent pas être considérées comme frontières.

L'utilisation des frontières permet de simplifier les problèmes d'exploration en règle générale. Reconnaître une frontière revient en effet à désigner une destination potentielle pour nos robots, et permet de grandement réduire le nombre de cellules à analyser pour déterminer la prochaine destination d'un robot.
Comme le montre la figure 2, choisir les frontières revient à sélectionner les segments de frontières les plus avantageux, et à trouver une ou plusieurs cellules sur ces segments pour devenir les régions frontières. Il existe différents algorithmes pour choisir les segments de frontières puis les cellules pour devenir des régions, nous en parlerons plus tard dans la suite de ce document.

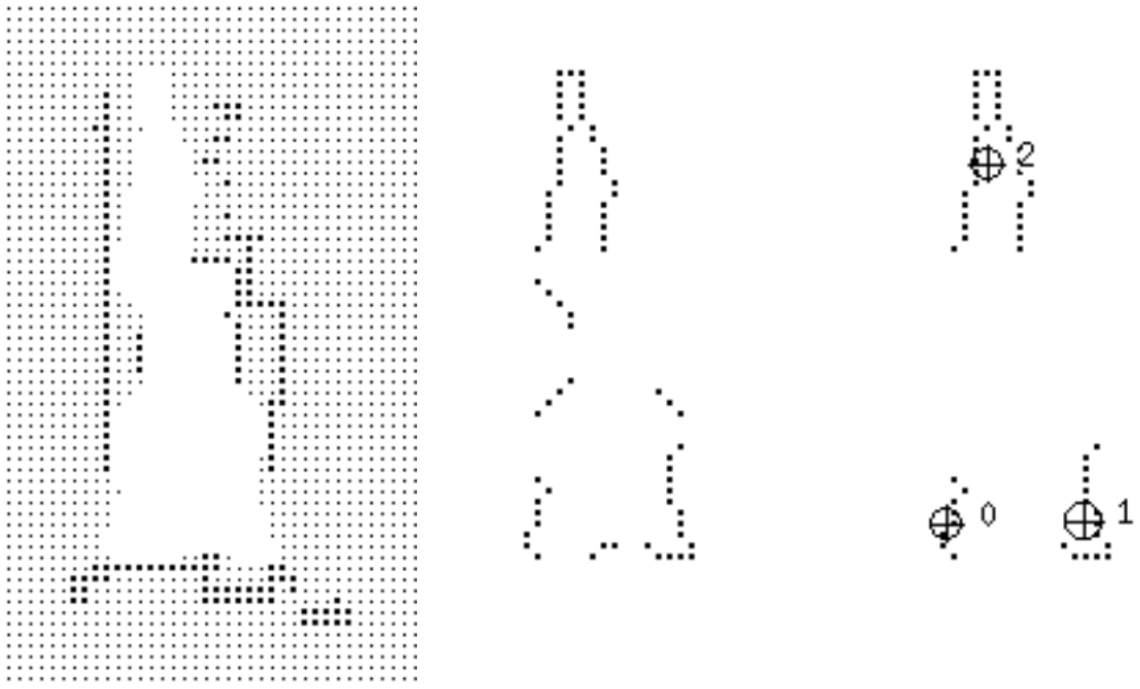

FIGURE 2 – Détection des frontières : *map* globale (gauche), segments de frontières (centre), régions frontières (droite) (5)

D'un point de vue pratique, il est possible d'avoir recours à un graphe d'exploration de frontières pour représenter les robots et leurs affectations aux différentes frontières via des arcs. Dans ce cas, on définit un nœud sur la frontière, le robot se dirigeant vers ce nœud en suivant un arc.

6 Stratégies d'exploration

6.1 Définition du problème

L'objectif est la construction d'une carte métrique grâce à des robots qui vont pouvoir explorer leur environnement.

Le choix de la frontière à explorer pour chaque robot peut en soit être réduit à un problème distribué.

Une frontière se définit comme étant la limite entre les zones explorées et les zones inexplorées. Le problème revient donc à déterminer correctement les frontières, et à affecter l'exploration de chacune de ces frontières aux différents robots.

Pour établir une carte, il est nécessaire de résoudre trois grands problèmes :

— obtenir la position des robots afin de pouvoir les localiser

— récupérer et analyser les informations issues de l'exploration des robots afin de pouvoir construire une représentation correcte de l'environnement, c'est-à-dire une carte

— définir une stratégie d'exploration permettant de découvrir n'importe quel environnement inconnu

Dans cette section nous nous concentrons uniquement sur ce dernier point.

En plus de répondre à cette problématique initiale, il est possible de rechercher à atteindre certains objectifs qualitatifs visant à réduire au maximum la durée d'exploration d'un environnement inconnu. Dans cette optique, la programmation distribuée nous permet de répartir correctement les ordres d'exploration entre les différents robots afin de maximiser les parcelles de l'environnement explorées simultanément.

Dans cette section, nous feront une synthèse des principaux algorithmes d'exploration et nous nous concentrerons plus particulièrement sur l'algorithme des positions minimales qui présente de nombreux avantages. Dans cet algorithme, un robot est affecté à la frontière pour laquelle il est en meilleure position. Un robot est en meilleure position pour une frontière s'il possède le plus petit nombre de voisins à proximité de cette frontière.

Commençons par définir les notations qui seront utilisées par la suite.

Tout d'abord il est nécessaire de faire la distinction entre l'environnement effectif dans lequel évoluent les robots et l'environnement exploré, c'est-à-dire l'environnement qu'il serait possible de reconstruire en faisant converger les connaissances individuelles de chaque robots acquises tout au long de l'exploration.

— $\mathcal{E}$ est défini comme étant l'environnement dans lequel évoluent les robots

— $\mathcal{E}_{exp}$ est défini comme étant l'environnement exploré

Il est possible à partir de ces deux définitions de définir l'environnement inexploré :

— $\mathcal{E}_{\text{inexp}} = \mathcal{E} - \mathcal{E}_{exp}$

La découverte totale de l'environnement par les robots peut être considérée comme une condition d'arrêt.

La découverte de l'environnement est terminée lorsque :

— $\mathcal{E}_{exp} = \mathcal{E}$ et $\mathcal{E}_{\text{inexp}} = \varnothing$

Les robots sont représentés par l'ensemble $\mathcal{R}$ et l'on considérera qu'ils possèdent tous un numéro i. Ainsi, pour n robots à notre disposition, l'ensemble des robots sera $\mathcal{R}_1, \ldots, \mathcal{R}_n$.

$\mathcal{E}_{exp}$ se construit au fil du temps en centralisant les différentes observation des robots. Si l'on note $\mathcal{X}_i(t)$ la configuration du robot $\mathcal{R}_i$ à l'instant t, nous pouvons depuis cette configuration en déduire l'espace qu'il sera possible d'observer à partir de celle-ci : $\mathcal{E}_{obs}\left(X_i(t)\right)$.

Ainsi, à l'instant T, l'espace exploré est défini formellement par :

$$\mathcal{E}_{exp} = \bigcup_{i=0}^{n} \bigcup_{t=0}^{T} \mathcal{E}_{obs}\left(X_i(t)\right)$$

La définition de la découverte totale de l'environnement peut également être réécrite. Cette situation survient lorsqu'il n'existe plus de configuration dans laquelle un robot pourrait observer une partie de l'environnement qui ne serait déjà pas incluse dans l'environnement exploré.

$$\nexists X_i | \mathcal{E}_{obs}\left(X_i\right) \cap \mathcal{E}_{inexp} \neq \varnothing$$

L'ensemble des frontières sera noté $\mathcal{F}$.

La proximité avec une frontière pour un robot est définie par la distance entre ces deux objets.

Attribuer une frontière à un robot est un problème NP-Difficile. Le nombre d'arrangements possible étant :

$$\frac{m!}{(m-n)!}$$

avec n, le nombre de robots et m le nombre de frontières

La démultiplication des combinaisons possibles lorsque m et n augmentent, rend rapidement l'étude individuelle de ces combinaisons inexploitable.

Pour évaluer correctement les coûts de déplacement vers une frontière pour chaque robot, il est possible d'utiliser une matrice de coûts. Ainsi $\mathcal{C}_{ij}$ donne le coût pour le robot $\mathcal{R}_i$ d'un déplacement vers la frontière $\mathcal{F}_j$

Un robot est affecté à une seule frontière à la fois. Une matrice d'assignation $\mathcal{A}$ permet d'identifier ces assignations avec $\alpha_{i,j} \in [0, 1]$ tel que :

$$\alpha_{i,j} = \begin{cases} 1 & \text{si le robot } R_i \text{ est assigné à la frontière } F_j \\ 0 & \text{sinon} \end{cases}$$

Nous avons donc une matrice $\mathcal{A}$ de taille (n, m) qui vérifie tout au long de l'algorithme la contrainte :

$$\forall i \sum_{j=1}^{m} \alpha_{ij} = 1$$

Bien sûr, il serait possible d'affecter plusieurs frontières à un seul robot. Dans ce cas, il faudrait se ramener au problème du voyageur de commerce, puisqu'il serait nécessaire de trouver un ordre de déplacement optimal entre ces destinations pour minimiser les coûts de déplacement. Ainsi, affecter plusieurs frontières à un robot alourdi le problème, cette approche ne sera donc pas considérée par la suite.

Donnons un exemple d'utilisation des matrices de coûts et d'affectation. Si l'on considère 4 frontières et trois robots, nous pouvons en premier lieu établir la matrice des coûts en prenant en compte la distances des robots par rapport à celles-ci :

$$\mathcal{C} = \begin{pmatrix} 20 & 20 & 15 & 19 \\ 25 & 15 & 10 & 16 \\ 24 & 19 & 11 & 15 \end{pmatrix}$$

Si pour l'affectation on parcourt chaque frontière en lui affectant le robot avec le coût le plus faible ou aucun robot si le robot avec le coût le plus faible est déjà affecté, on obtient la matrice d'affectation suivante :

$$\mathcal{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A partir de ces deux matrices, il est possible de quantifier le coût total des déplacements des robots pour cette répartition. Il suffit de faire la somme des coûts associés aux frontières explorées en fonction des robots affectés.

$$C(\mathcal{A}) = \sum_{i=1}^{n} \sum_{j=1}^{m} \alpha_{ij} \mathcal{C}_{ij}$$

Dans notre exemple, nous avons un coût total de : $C(\mathcal{A}) = 20 + 15 + 15 = 50$

Plusieurs approches sont possibles pour assigner des frontières aux robots. Les stratégies les plus simples reposent sur une exploration aléatoire ou sur le suivi d'un mur.

6.2 Critères d'évaluation des différents algorithmes d'exploration

Il est nécessaire de définir et d'avoir recours à diverses métriques afin de pouvoir juger de la qualité d'un algorithme d'exploration.

En premier lieu, nous pouvons utiliser le calcul de la complexité. Cet indicateur nous permettra de différencier certains algorithmes qui seront décrits ci-dessous, comme par exemple l'algorithme

glouton.

Néanmoins, mise à part la complexité, une métrique que nous pouvons utiliser est le temps d'exploration. Le temps d'exploration est défini comme la durée entre le début d'une exploration et la fin de l'exploration. Par fin de l'exploration, on entend la découverte intégrale de toutes les cellules qui constituent l'environnement de la carte.

Il est également possible de considérer la fréquence d'affectation des frontières aux robots. En règle générale, plus la fréquence d'affectation est élevée et plus la carte se construit rapidement, ce qui est révélateur d'une bonne coopération entre les robots.

Effectivement, plus tôt le robot effectuera une observation suivie d'un calcul d'affectation, plus tôt il fera demi-tour pour explorer une autre partie de l'environnement.

Nous devons toutefois citer certaines situations problématiques où une fréquence d'affectation élevée peut également être révélatrice d'une situation de blocage, ce serait par exemple le cas si un robot demeure en oscillation permanente entre deux frontières.

Il également possible d'effectuer une évaluation qualitative en générant un environnement spécifique et en étudiant les différentes fonctions attribuées aux robots en fonction des différents algorithmes. Dans ce rapport, nous utiliserons à de multiples reprises des schémas représentant les différentes possibilités d'affectations selon les méthodes d'exploration retenues. Intuitivement, pour chacune de ces situations, nous avons une idée de la répartition idéale qu'il serait souhaitable d'atteindre. Certaines situations de tests, comme la simulation en T, en couloir, en labyrinthe, etc, permettent de rapidement comparer la répartition donnée par un algorithme par rapport à la répartition idéale espérée. Par ailleurs, ces simulations permettent d'analyser les comportements aux limites du système.

Enfin, il est important de préciser que l'évaluation de la qualité d'une exploration est relative au contexte dans lequel évolue cette application. Les objectifs d'exploration peuvent varier. Ainsi, si le but de l'exploration est de retrouver des personnes disparues, dans le cas d'une application visant à secourir des personnes dans un environnement hostile, il sera préférable de privilégier les algorithmes assurant une couverture maximale de la carte tout en acceptant une précision plus faible à petite échelle, certaines cellules isolées pouvant rester inexplorées. Effectivement, avec une telle approche, on augmente la probabilité de trouver les survivants dans les meilleurs délais.

En revanche, dans certains cas, le temps d'exploration n'a aucune importance mais l'exploration de l'environnement doit être la plus exhaustive possible.

D'autres situations nécessitent également que l'énergie utilisée par les robots soit minimisée. Par exemple, dans des contextes où les robots travaillent en totale autonomie, avec des batteries leur accordant un stock d'énergie limité et ne pouvant pas être rechargées par des opérateurs, les algorithmes d'exploration doivent assurer que l'environnement sera découvert avec le moins de déplacement possible.

6.3 Frontière la plus proche

Cet algorithme a été développé par Yamauchi en 1998. Les robots peuvent modifier leur représentation locale de leur environnement et partager ces informations avec les autres robots. De cette manière, les robots possèdent une carte similaire et il est possible d'identifier les mêmes frontières d'un robot à un autre.

Après avoir établi les frontières, chaque robot se dirige vers la frontière la plus proche de lui. Dès que le robot a atteint une frontière, il l'explore et diffuse à tous les autres robots les résultats de son exploration.

Dans cet algorithme, les robots n'ont pas besoin de se concerter avant d'effectuer une nouvelle exploration. Les seules communications servent à échanger de nouvelles informations sur la carte. Il s'agit d'un algorithme asynchrone. A noter que cette méthode offre une tolérance aux pannes importante, puisque la perte d'un robot n'influe en aucune manière sur le comportement des autres robots.

Cet algorithme assure d'obtenir à chaque nouveau déplacement vers une frontière le coût minimal de déplacement, puisque la frontière avec la plus petite distance est toujours favorisée.

En revanche, cette stratégie ne garantit absolument pas l'équilibre de la répartition des robots sur les frontières. Effectivement, si plusieurs robots sont situés à proximité d'une même frontière, ils se dirigeront tous vers celle-ci, réduisant ainsi considérablement l'intérêt d'un système distribué puisque le temps d'exploration ne pourra pas être minimisé en profitant de l'analyse simultanée de plusieurs zones distinctes.

A noter que la complexité de cet algorithme est en $O(m)$ puisqu'il est nécessaire, pour chaque site, d'analyser chaque frontière.

Algorithm 1 Frontière la plus proche (6)

Entrées C_i un vecteur de coût de chemin à chaque frontière
Sorties A_i l'assignation du robot i
Début
$\alpha_{ij} = 1$ tel que $j = \underset{\forall \mathcal{F} j \in \mathcal{F}}{\operatorname{argmin}} \mathcal{C}_{ij}$
Fin

6.4 Algorithme glouton

Cet algorithme vise à corriger en premier lieu le défaut de l'algorithme de la frontière la plus proche. Dans l'algorithme glouton, les robots doivent se concerter afin de se mettre d'accord sur chaque couple (robot/frontière) avant d'effectuer leurs explorations. De cette manière, il est impossible qu'une frontière soit explorée par plus d'un robot à la fois. Ainsi, il est désormais possible de garantir l'équilibre de la répartition des robots sur les frontières tout en conservant un coût minimal pour chaque exploration, le facteur de distance minimale étant pris en compte lors de la constitution des couples.

La complexité de cet algorithme est en revanche en $O(m * n^2)$.

Algorithm 2 Glouton (6)

Entrées C Matrice des coûts
Sorties α_{aj} l'assignation du robot $\mathcal{R}_a$ à la frontière $\mathcal{F}_j$
Début
$\mathcal{F}_{init} = \mathcal{F}$
Tant que $\mathcal{R}_a$ n'est pas assigné faire :
Trouver $\mathrm{i,j} = \underset{i,j|\mathcal{R}_i \in \mathcal{R}, \mathcal{F}j \in \mathcal{F}}{\operatorname{argmin}} \mathcal{C}_{ij}$
$\alpha_{ij} = 1$
$\mathcal{R} = \mathcal{R} \backslash \mathcal{R}_i$
$\mathcal{F} = \mathcal{F} \backslash \mathcal{F}_j$
Si $\mathcal{F} = \varnothing$ alors $\mathcal{F} = |\mathcal{F}_{init}$
Fin Tant que
Fin

6.5 Algorithme glouton sur l'utilité des frontières

En 2005, Burgard propose une amélioration de l'algorithme glouton en introduisant le critère d'utilité de frontière qui se traduit par le rapport entre le coût d'exploration et une estimation du gain d'information dans l'hypothèse où la frontière serait explorée. (6)

Le gain d'information est estimé à partir d'une taille supposée de la zone qui pourrait être découverte suite à l'exploration d'une frontière.

Ce nouveau facteur permet d'éviter que les robots soient attirés vers des frontières proches (car un phénomène de convergence dans l'exploration implique nécessairement une réduction progressive de la taille des parcelles pouvant être explorées). Ainsi, cette stratégie garantit une meilleure couverture de l'environnement.

La complexité est la même que celle de l'algorithme glouton, à savoir $O(m * n^2)$.

6.6 Système d'enchères

Après avoir calculé, par exemple avec l'algorithme glouton, le gain et le coût de l'exploration de chaque frontière, il est possible d'utiliser un système d'enchère afin de décider de la répartition. L'enchère peut être centralisée ou bien distribuée. Dans ce cas, on peut imaginer que lorsqu'un robot découvre une nouvelle frontière, il devient commissaire priseur de celle-ci et la propose aux autre robots, lui même pouvant participer à l'enchère.

Dans les deux cas, le robot proposant la meilleure enchère gagne le droit d'exploration de la frontière.

6.7 Algorithme des positions minimales

L'algorithme glouton offre de bons résultats mais sa complexité demeure bien trop problématique. D'où la nécessité de trouver un autre algorithme permettant de réduire la complexité de l'algorithme glouton tout en garantissant une bonne dispersion des robots sur les frontières.

Une autre approche pour la résolution du problème de répartition des robots consiste à calculer pour chaque couple (robot/frontière), le nombre de robots plus proches de la frontière considérée.

Cet indicateur correspond à la « position du robot sur la frontière ».

Lors de l'affectation des frontières, c'est le robot qui présentera la meilleure position (c'est-à-dire la plus petite valeur), qui remportera la frontière. Et lorsqu'un robot présente la même position pour plusieurs frontières, il se dirigera par défaut vers la frontière la plus proche.

Cette approche permet de résoudre le désavantage de l'utilisation des distances minimales. Effectivement, lorsque la répartition des robots s'effectuait via le principe de la plus courte distance à la frontière, on pouvait le plus souvent obtenir des situations où les robots se déplaçaient dans la même direction.

En utilisant l'indicateur de position, les robots se répartissent mieux dans l'espace, puisque même si une frontière est très éloignée, elle pourra être affectée au robot qui en est le plus proche.

Les positions des robots pour chacune des frontières peuvent être recensées dans la matrice des positions $\mathcal{P}$ tel que $\mathcal{P}_{ij}$ donne le numéro de la position du robot $\mathcal{R}_i$ pour la frontière $\mathcal{F}_j$. Plus formellement, pour déterminer $\mathcal{P}_{ij}$, nous allons construire l'ensemble $\tilde{\mathcal{R}}$ qui comprend tous les robots qui sont en meilleur position pour la frontière j, c'est-à-dire tout robot $\mathcal{R}_k$ avec $\mathcal{C}_{kj} < \mathcal{C}_{ij}$. Ensuite, il suffit d'utiliser le cardinal de l'ensemble $\mathcal{R}_k$.

Soit :

$$\mathcal{P}_{ij} = \text{Card}(\tilde{\mathcal{R}}) \text{ avec } \tilde{\mathcal{R}} = \{\forall \mathcal{R}_k \in \mathcal{R}, \mathcal{C}_{kj} < \mathcal{C}_{ij}\}$$

Algorithm 3 Positions minimales (6)

Entrées $\mathcal{F}$ l'ensemble des frontières, $\mathcal{R}$ l'ensemble des robots, $\mathcal{C}$ la matrice des coûts
Sorties α_{ij} l'assignation du robot $\mathcal{R}_i$
Début
Pour chaque $\mathcal{F}_j \in \mathcal{F}$ **faire**
$\mathcal{P}_{ij} = \text{Card}(\tilde{\mathcal{R}})$ avec $\tilde{\mathcal{R}} = \{\forall \mathcal{R}_k \in \mathcal{R} | \mathcal{C}_{kj} < \mathcal{C}_{ij}\}$
Fin Pour Chaque
$\alpha_{ij} = 1$ tel que $j = \underset{j|\mathcal{F}j \in \mathcal{F}}{\text{argmin}} \mathcal{P}_{ij}$
S'il y a une égalité on choisit la frontière avec le coût minimum parmi les minima de $\mathcal{P}_{ij}$
Fin

La complexité de position minimale est de $O(n * m)$.

Nous allons désormais étudier quelques situations qui vont nous permettre de juger de l'intérêt de l'algorithme des positions minimales en comparaison avec ses homologues.

Le cas le plus simple est un couloir contenant des robots qui, globalement, sont plus proches d'une extrémité ; ici la gauche. Avec l'algorithme des distances minimales, l'intégralité des robots se déplacent vers la frontière gauche, laissant l'autre frontière inexplorée.

En revanche, avec l'algorithme des positions, les robots se répartissent de manière homogène entre les deux frontières.

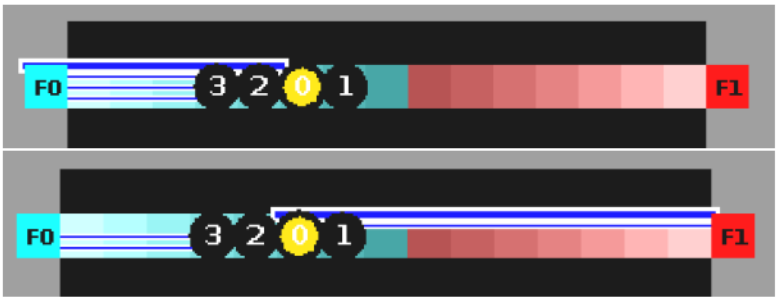

FIGURE 3 – Répartition des frontières. En haut, l'algorithme des frontières les plus proches ; en bas, l'algorithme des positions minimales (6)

L'algorithme des positions minimales profite également des évolutions dynamiques des positions des robots pour corriger leurs répartitions. Prenons l'exemple d'un environnement en T, avec une dissymétrie au niveau de la barre horizontale du T. Si deux robots débutent leurs explorations dans la partie sud de cet environnement, ils vont tout deux se diriger vers l'extrémité droite du T, mais dès que le premier robot entame son virage vers cette frontière, le second robot qui le suit de peu, devient dès lors en meilleure position pour la frontière opposée, qu'il pourra dès lors explorer. La figure suivante synthétise cette situation.

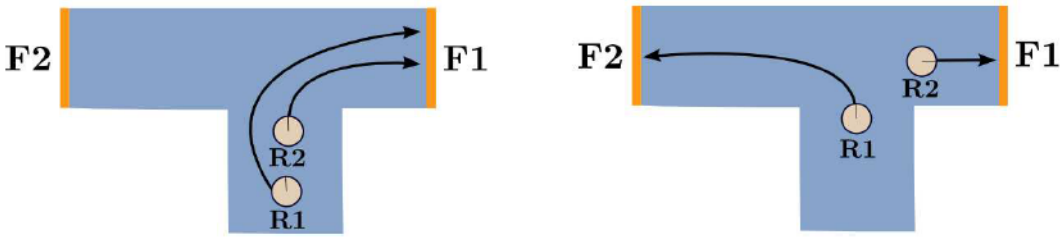

FIGURE 4 – Répartition des frontières dans un environnement en T (6)

Nous avons vu que l'algorithme glouton, en calculant l'ensemble des couples (robot/frontière) possibles et en les ordonnant en fonction des distances, peut les affecter de manière à répartir les robots entre les frontières.

Bien qu'il permette de répartir les robots sur des frontières distinctes, l'algorithme glouton ne permet pas toujours de répartir les robots dans des directions distinctes. Dans cette dernière situation, nous pouvons voir que l'algorithme des positions minimales permet une meilleure répartition des robots dans l'environnement.

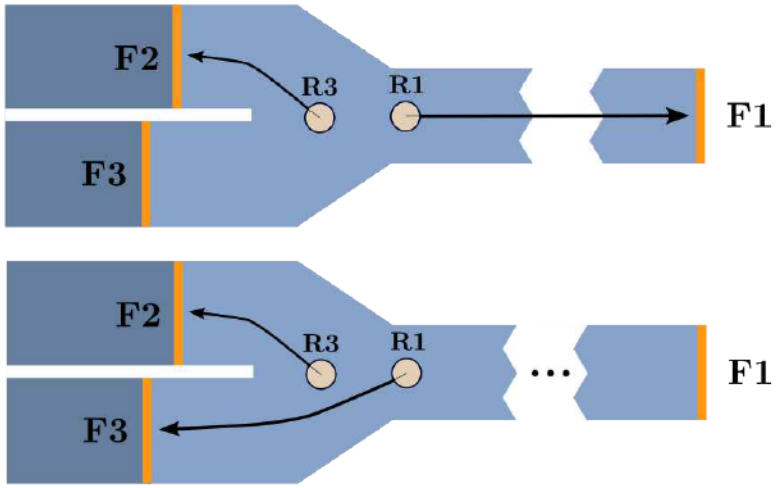

FIGURE 5 – Répartition des frontières. En haut, l'algorithme des positions minimales ; en bas, l'algorithme glouton (6)

A noter toutefois que l'algorithme glouton offre dans certains cas une meilleure répartition que l'algorithme des positions minimales. Effectivement, dans le cas où une direction présente plus de frontières que de robots, il est possible que certaines frontières soient orphelines avec l'algorithme des positions minimales alors que l'algorithme glouton aurait pu définir une répartition optimale. C'est le cas dans la situation représentée sur la figure 6.

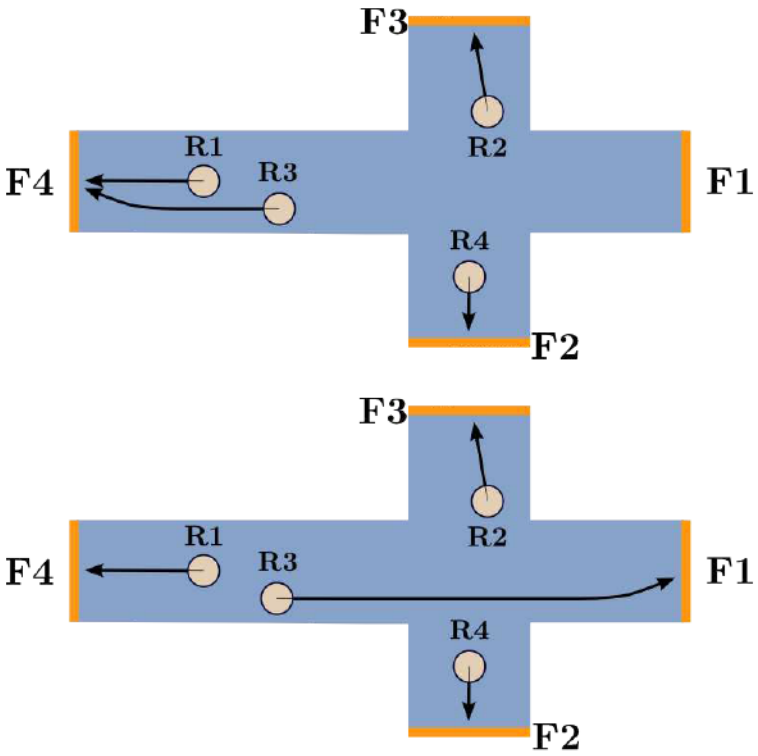

FIGURE 6 – Répartition des frontières. En haut, l'algorithme des positions minimales ; en bas, l'algorithme glouton (6)

7 Implémentation

Dans cette partie nous allons détailler l'implémentation de différents aspects qui sont l'échange des informations de déplacement des robots, les calculs de chemin et de coût ainsi que l'algorithme d'exploration des robots.

7.1 Architecture d'un robot

Un robot est composé de deux applications : le PIL (PILote) et le NET (NETwork). Le PIL se charge de toute la partie algorithmie des robots tandis que le NET s'occupe de la communication avec les autres robots, et donc les autres NET. Un, et un seul, NET est associé à un PIL. Ce dernier est en communication avec son NET ainsi qu'avec la simulation, dans le cas d'une simulation, ou avec le robot lui même.

Chaque PIL comporte une carte interne avec toutes les informations qu'il possède, provenant des autres robots ainsi que de ses actions réalisées. De plus, la carte comprend également la position des autres robots.

Lors de la simulation, les robots sont assimilés à un graphe complet pour assurer la connexité, et un filtre est réalisé sur les messages afin de savoir si la distance entre les robots qui communiquent n'est pas trop importante pour interpréter le message.

7.2 Buffer d'actions

7.2.1 Fusion des cartes

Tout d'abord, pour fusionner les cartes, nous utilisons les propriétés d'additivité et de commutativité, comme expliqué en première partie. En effet, nous considérons que les réponses obtenues par les robots sont parfaites et que deux robots ne peuvent avoir des informations concurrentes.

Pour l'envoi des cartes, nous avons décidé de ne pas envoyer toute la carte dans sa totalité, car les messages deviendraient trop lourds et nous avons donc choisi d'implémenter une méthode envoyant les dernières actions du robot. Cependant, nous n'envoyons pas seulement les dernières actions mais une série d'anciens déplacements qui peuvent être redondants. Ceci permet d'assurer une cohésion pour pallier à un éventuel problème de perte de messages.

Le buffer est envoyé par messages périodiquement du PIL au NET puis à tous les autres robots par diffusion. Pour gérer cet envoi, un thread est utilisé au sein du PIL et permet de faire des actions périodiques. Cependant, il faut bien veiller à lancer et désactiver ce thread lorsque des envois sont nécessaires.

7.2.2 Format des actions

Le buffer est donc envoyé au sein de messages, dans sa payload. Il est composé d'une liste de mots respectant une certaine mnémonique. Tout d'abord, le premier élément de la payload est *@buffer*, ce qui permet de savoir que c'est un message contenant un buffer et qu'il doit être interpreté, et pas un simple message entre les robots. Puis, le format d'un déplacement respecte le suivant :

$$estampille : action : distance\_réelle, distance\_attendue : x, y, direction$$

Ce format correspond donc à un déplacement où *action* peut être *move*, *turn* ou *init*, actions possibles du robot. Une estampille est utilisée car le buffer est redondant : il est renvoyé sans être vidé, donc il peut contenir des informations identiques d'un envoi à l'autre. De ce fait, il est nécessaire de s'assurer que le déplacement n'a pas encore été lu pour l'interpréter et l'exécuter localement. De plus, le message contient la distance réelle et la distance attendue afin de savoir si le robot émetteur est entré en collision avec un obstacle lors de son déplacement : il suffit de vérifier que les deux distances soient différentes. Enfin, le dernier élément du déplacement est la position du robot après avoir appliqué l'action. Ceci permet de filtrer les messages : il est possible de comparer le robot émetteur avec le récepteur et de calculer la distance les séparant afin de savoir si le message doit être lu. Le message ne doit pas être lu si la distance est trop importante, ce qui signifie qu'ils ne sont pas dans leurs portées respectives et qu'ils ne sont pas sensés recevoir les messages.

Le buffer est donc constitué d'une liste de déplacements au format présenté précédemment. A sa réception, il est lu intégralement et les messages non déjà lus sont appliqués sur la carte interne du robot afin de déplacer le robot en question et de mettre à jour les informations.

7.3 Recherche de chemin et calcul des coûts

Tout d'abord voyons les fonctions de coûts choisies qui sont un élément clé dans le choix d'une frontière, ainsi qu'un algorithme de *pathFinding* pour permettre au robot de se déplacer.

7.3.1 Distance de Manhattan

Pour définir le coût d'une frontière pour un robot nous utilisons dans un premier temps la distance de Manhattan donnée par la formule suivante :

$$d_M(A, B) = |x_B - x_A| + |y_B - y_A| \tag{1}$$

Cette distance, bien qu'elle ne prend pas en compte de potentiels obstacles entre le robot et la simulation, permet d'effectuer un pré-tri sur les frontières, ce que nous verrons juste après.

7.3.2 Distance Euclidienne

Pour coller au plus près de la réalité, il sera nécessaire de discrétiser davantage la carte : le robot et chaque obstacle ne peuvent représenter réellement une unique et même case. De ce fait, la distance vers laquelle on tend à utiliser sera la distance euclidienne, beaucoup plus réaliste.

$$d_E(A, B) = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2} \tag{2}$$

7.3.3 Recherche du plus court chemin : A*

Une estimation du coût plus précise sera obtenue grâce à une implémentation de l'algorithme **A***. Celui-ci permet de définir le plus court chemin entre un robot et une frontière (au sens d'une heuristique et d'un coût déterminés) en contournant les obstacles sur le chemin. La complexité de cet algorithme est en $O(n^2)$ dans le pire des cas (correspond à l'exploration complète de la carte avant de trouver un chemin), ce qui est donc raisonnable.

De plus nous avons choisi dans cette algorithme de favoriser le passage par des frontières. En effet, plutôt qu'affecter à chaque case une valeur égale à une unité, nous fixons aux frontières un coût inférieur à celui d'une cellule normal. A distance égale, l'heuristique sera plus faible sur la cellule frontière. De ce fait, la destination (une frontière le plus souvent) sera potentiellement atteinte en passant par d'autres frontières, faisant gagner du temps pour l'exploration, toutes les cellules frontières étant à explorer.
Enfin, une fois le chemin calculé, afin de réduire le nombre de messages à envoyer à la simulation et aux autres robots, nous condensons l'information lorsque les cellules se succèdent dans la même direction.

Afin d'améliorer davantage cet algorithme, nous pourrions valoriser les chemins linéaires (valoriser la recherche d'une case dans la même direction que celle précédemment ajoutée) afin de réduire encore une fois les déplacements (les déplacements en escaliers étant beaucoup plus coûteux en nombre d'actions réalisées et messages envoyés).

Une autre solution est d'autoriser les déplacements en diagonale, mais dans ce cas il faut prendre en considération la distance aux obstacles. En effet, certains mouvements peuvent être physiquement impossibles si les cas limites sont mal gérés.

7.4 Algorithme

Nous allons décrire plus en détail l'algorithme de choix de frontière. La première étape est de récupérer toutes les frontières connues sur la carte locale.

Si aucune frontière n'est trouvée c'est que l'exploration est terminée. Sinon l'algorithme se poursuit.

Algorithm 4 A* (begin, end)

```
openList ← List()
closedList ← List()
cell ← Cell()

openList.push_front(end)

while (openList is not empty) do
  cell ← openList.first()
  // Retrieving element with the lower heuristique closedList.push_front(cell)
  openList.pop_front()
  // Suppression of this first element, considered "explored"
  if (cell.x = begin.x and cell.y = begin.y) then
    return compute_pathList(cell, closedList)
    // Compute and return condensed instructions to move from cell
  else
    add_neighbours(openList, closeList, end, cell)
    // new Cells to be inserted in the openList if not already in, not in closedList.
    // end cell used to compute heuristique
  end if
end while
End of loop : no path to reach end
return NULL;
```

Algorithm 5 Choix frontière

```
frontiers ← findFrontier()
if frontiers == empty then
  return Fin
end if
frontiers ← sortFrontierByManhattan(frontiers)
robotLocal ← récupère le robot local
for front in frontier do
  for robot in robots do
    cost_{front/robot} = A*(front, robot).getCost()
  end for
  if cost_{front/robotLocal} == minCostForFrontier(cost_{front}) then
    return front
  end if
end for
return getFrontierWhereLocalRobotIsTheNearest()
```

Nous effectuons un premier tri des frontières par rapport à la distance de Manhattan entre la frontière et le robot local.

Puis nous avons implémenté l'algorithme ***MinPos*** qui parcourt les frontières précédemment triées. Pour chaque frontière nous calculons un $A*$ entre la frontière et chacun des robots présent sur la map locale.

Si le robot local est le robot le plus proche de la frontière alors il choisit d'explorer cette frontière sinon nous passons à une autre frontière.

Pour finir, s'il n'existe aucune frontière pour laquelle le robot local est le plus proche, il choisit la frontière pour laquelle il est le moins éloigné.
Cet algorithme est plutôt coûteux en temps de calcul car, dans le pire des cas, un calcul de $A*$ est effectué pour chaque couple *(robot, frontière)*. La complexité est donc :

$$O(\text{nombre de robots} * \text{nombre de frontières} * (\text{nombre de case})^2) \tag{3}$$

C'est pourquoi nous avons choisit de faire un pré-tri par rapport à la distance de Manhattan. En effet nous nous sommes rendu compte que dans la majorité des cas les frontières les plus proches en distance de Manhattan sont aussi les plus proche suite à un $A*$. Cela s'explique simplement par le fait que lorsqu'un robot découvre une frontière, les zones inexplorées qui juxtaposaient cette frontière deviennent des frontières. Ainsi les frontières générées sont collées au robot, il ne peut donc pas y avoir un obstacle entre le robot et cette nouvelle frontière.

8 Simulation

Le but de cette partie est de présenter les caractéristiques de l'application mise en place pour simuler le comportement des robots en réaction aux algorithmes d'explorations implémentés.

8.1 Motivation

Une des consignes de cette étude était de réaliser une application destinée à simuler le comportement réel des robots. En effet avant d'implémenter notre méthode exploratoire à l'aide de la flotte de robots et de pouvoir réaliser de réelles expérimentations d'exploration, il nous faut vérifier que le comportement virtuel est bien conforme à nos attentes.

Cette application que nous avons simplement nommée *Simulation* ou *SIM* doit remplir plusieurs conditions :

— Elle doit disposer d'une interface graphique affichant la carte de l'environnement à explorer.
— Elle doit disposer de plusieurs boutons permettant de modifier la carte (taille, obstacles), ajouter des robots, en supprimer etc.
— Elle doit pouvoir recevoir des messages et en renvoyer, ainsi que les afficher dans un historique.

8.2 Représentation de l'environnement

Une des problématiques les plus importantes a été de savoir comment représenter l'espace sur notre simulation. Nous avons décidé de le représenter sous la forme d'un tableau à 2 dimensions.

Chaque case représente $1cm^2$ dans notre environnement, et un robot aura donc la dimension de $1cm^2$.

Bien entendu c'est un parti pris de représenter notre environnement en cases et surtout nos robot en cases de $1cm^2$. Cette décision était la plus optimale dans notre cas, car notre objectif était de créer un outil de simulation simple, destiné à être potentiellement amélioré par la suite.

8.3 UML

Notre Simulation est développée en language objet, plus particulièrement en Qt C++. Ce choix nous a permis de créer une architecture objet composée de plusieurs classes reliées entre elles dont le graphique UML est décrit en figure 7.

8.4 Classes

Les classes les plus importantes, à savoir la carte de l'environnement Map, le Message Manager ainsi que l'interface graphique MapGUI seront détaillées dans les parties suivantes.

8.4.1 Classe Map

La classe Map représente notre environnement, elle est composée des attributs suivants :

— Un map (liste associative) de tous les robots
— Le nombre de robots maximum
— Un tableau à deux dimensions d'états (full ou empty)
— Les dimensions de ce tableau

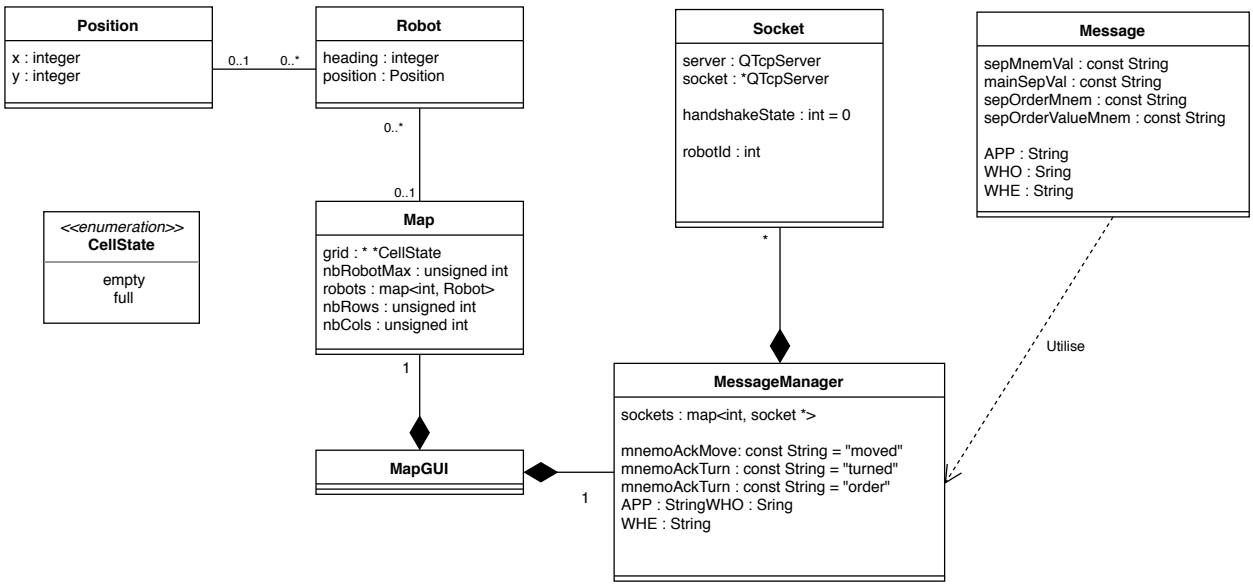

FIGURE 7 – UML de la Simulation

Le tableau est destiné à être mis à jour lors des modifications de l'environnement. La classe Map dispose donc de nombreuses méthodes (outre les getter et les setter) permettant de réaliser les déplacements de robots.

Ajout et déplacement de robots Un robot peut être créé seulement si la case visée est dans l'état empty. Chacun des robots a pour attributs une position ainsi qu'un angle appelé heading. À la suite de cette création, des méthodes de déplacement peuvent être appelées.

— Move : Qui permet de se déplacer d'un certain nombre de cases/cm dans une direction définie par heading.

— Turn : Qui permet de changer la valeur de heading

— Join : Qui permet de se déplacer vers une position donnée.

8.4.2 Message Manager

La classe MessageManager assure le lien entre l'application SIM et l'application PIL. Elle dispose donc d'un tableau de sockets permettant de communiquer individuellement avec chaque robot. Ainsi, lorsqu'un robot envoie un message vers l'application SIM, le socket associé peut émettre un signal afin de prévenir la classe MessageManager que son robot a envoyé un message. Inversement, lorsque l'application SIM désire envoyer un message à un robot (message d'acquittement par exemple), la classe messageManager lui fourni une interface permettant d'écrire dans le socket correspondant.

8.4.3 Interface Graphique MapGUI

L'interface graphique est composée de plusieurs modules permettant de faire la liaison entre les messages reçus et la map.

Affichage de l'environnement L'environnement est représenté par un canevas, un tableau à 2 dimensions qui est composé d'objets pouvant être des obstacles, des cases vides ou des robots. Les

cases peuvent être modifiées pour ajouter ou retirer un obstacle. La fenêtre affichée est donnée en figure 8.

Un robot est représenté par une couleur lui correspondant.

Sauvegarde et chargement d'environnement Un système de sauvegarde de configuration a été mis en place pour faciliter les simulations exécutées. Une configuration peut être enregistrée dans un fichier, puis pourra être chargée plus tard pour être réutilisée.

Lancement de la simulation Un bouton pour lancer la simulation est présent sur l'interface graphique. Celui-ci va initialiser un objet Map décrit précédemment avec tous les paramètres du canevas. En effet les dimensions sont passées en argument et le tableau est parcouru entièrement pour transférer l'état de toutes les cellules vers l'objet Map.

A partir de ce moment, notre simulation est prête à recevoir des robots.

Ajout de robot Une connexion est mise en place en remplissant l'adresse TCP du robot en question. Le robot sera ensuite initialisé dans notre objet Map et sera en attente d'un message d'initialisation de la part de son application PIL, qui communiquera via l'adresse renseignée.

Gestion des messages Une fenêtre d'historique est présente pour afficher les différents messages à la fois reçus par les PIL et envoyés vers ceux-ci. Lorsqu'un message est disponible sur un socket, un signal est émis vers MessageManager. Ce dernier récupère l'identifiant du robot associé et transfère le message à l'application MapGUI. Le message est alors traité en fonction du type d'ordre envoyé par le robot. Le traitement consiste à calculer la nouvelle position du robot en fonction de l'ordre, de sa position sur la carte, et des obstacles présents. Une fois le message traité, la classe MapGUI charge au MessageManager de renvoyer au robot ayant soumis un ordre un message d'acquittement, conformément au protocole des vrais robots.

Exemple :

- Réception du message "PILROBLCH/robord˜move :10" sur le socket associé au robot 1.
- Traitement du message. On fait avancer le robot de 10 cases.
- Le robot rencontre un obstacle sur la 6ème case. Le robot s'arrête donc à la 5ème case.
- Mise à jour de la position du robot sur l'interface graphique.
- Envoi du message d'acquittement "ROBPILLCH/roback˜moved :5" dans le socket du robot 1.

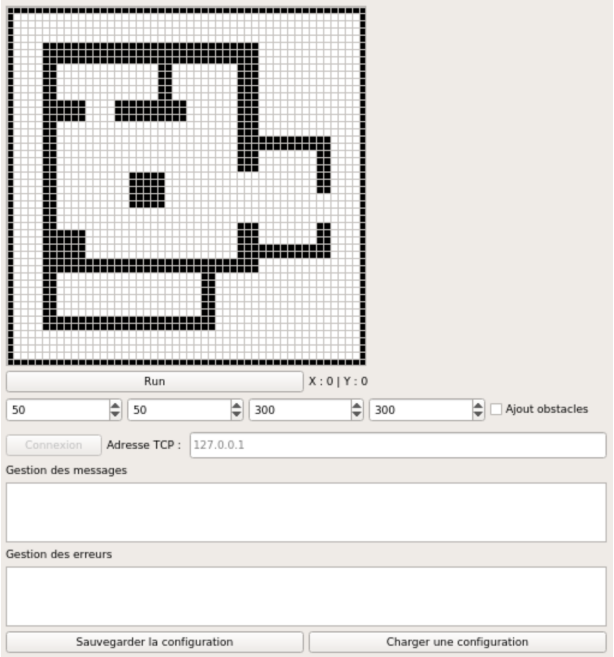

FIGURE 8 – Fenêtre de Simulation

9 L'architecture de communication

Il a fallu mettre en place un système de communication, entre les applications de contrôle et la simulation présentée ci-dessus, et entre les robots. Une vue globale de l'architecture décrite ci-après est présentée sur la figure 9.

9.1 Communication entre la simulation et les robots

Pour être au plus prêt de la réalité, et faciliter un éventuel déploiement sur les robots réels, l'application de contrôle PIL communique avec la simulation de la même manière qu'elle communiquerait avec le RMT du robot, à l'aide d'une socket TCP.

La simulation crée un serveur TCP par robot, qui écoute sur une IP locale sur le port 4646. Chaque robot simulé écoute donc sur le même port, sur une IP différente. Pour déployer, il suffira de remplacer les adresses locales (de la forme `127.0.0.X` avec X le numéro du robot) par les adresses des robots. L'application de contrôle se connecte au robot, répond au *handshake*, et envoie des instructions en utilisant le protocole Airplug en whatwhowhere au ROB simulé. Tout comme sur les robots, si une deuxième application de contrôle essaie de se connecter à un robot, la connexion avec l'application précédente est fermée.

Les robots simulés renvoient les mêmes messages d'acknowledge que les robots réels. Il s'agit de la grande force de cette simulation. L'interface à laquelle font face les applications PIL est exactement la même que celle offerte par les vrais robots du laboratoire de l'UTC.

9.2 Communication inter-robot

Les robots communiquent entre eux par l'intermédiaire d'une application NET, et des pipes nommés. Nous avons choisi cette architecture car nous ne savions pas exactement comment fonctionne la communication inter-robot sur le réseau, et comment récupérer les messages envoyés par

cette voie. Nous avons donc tiré partie de la présence de tous les PILs sur une seule machine pour permettre aux robots de communiquer.

Ils se transmettent les dernières actions effectuées, ce qui permet à chaque application de contrôle de construire une représentation de la carte. Comme nous l'avons expliqué à la section 4, les robots sont sensés avoir une portée de communication maximale, au delà de laquelle ils perdent leur interconnexion. Pour simuler cela, nous ignorons les messages provenant de robots trop éloignés.

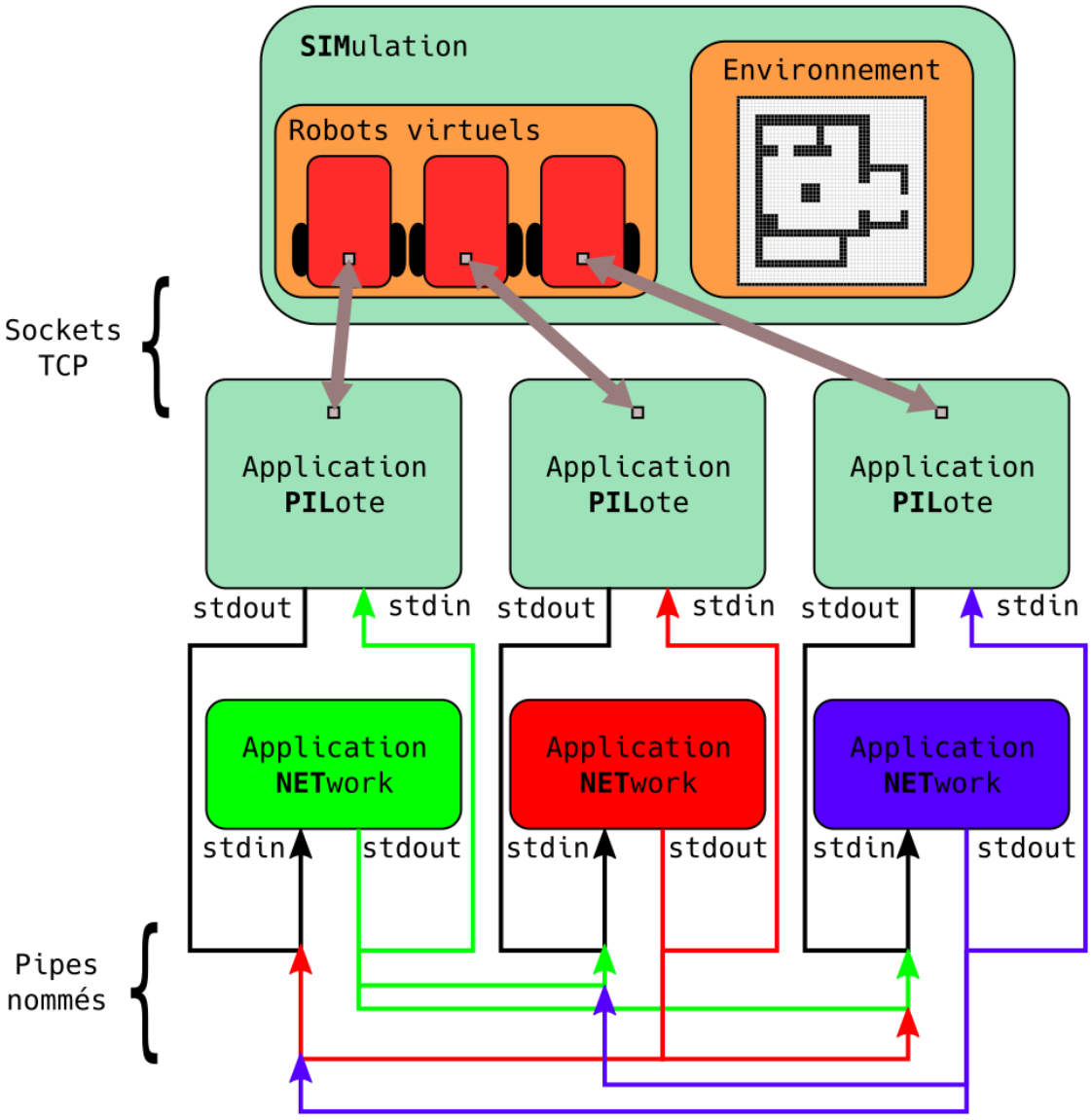

FIGURE 9 – Architecture globale des communications

10 Conclusion et pistes d'amélioration

Ce projet nous a permis de nous familiariser avec des travaux de recherche, et d'explorer un sujet intéressant et plein d'avenir. Le travail de recherche fourni nous a permis de comprendre les problématiques qui entourent l'exploration d'une zone par une flotte de robots. De la gestion de perte de connexion au choix d'une stratégie maximisant le gain d'information total, ce champ de recherche est très dynamique et prendra sûrement encore plus d'importance dans les années à venir. Nous nous sommes appropriés plusieurs algorithmes de choix de destination, notamment le MinPos, et avons pu en découvrir plusieurs autres. Nous avons aussi pu nous apercevoir de leurs limites. Nous terminerons sur quelques pistes d'améliorations que nous n'avons pas eu le temps d'implémenter.

Pistes d'amélioration : Actuellement , la gestion des distances de communication maximale entre les robots est implémentée, mais nous n'avons pas pu tester toutes les fonctions permettant d'assurer son bon fonctionnement. Ainsi, d'après notre implémentation, un robot qui perd la connexion avec tous ses voisins va calculer le chemin vers la position du robot le plus proche avant perte de connexion (grâce à l'algorithme A*). Il doit interrompre son parcours lors qu'il retrouve une connexion. Cependant, nous redoutons des oscillations entre des connexions et des déconnexions en agissant uniquement de cette manière, nous devons donc poursuivre la réflexion.

Dans notre application de simulation, les collisions entre les robots ne sont pas gérées, ils se superposent. Il est techniquement très simple de transformer un robot en un obstacle potentiel pour les autres robots. Cependant, nous devrions alors établir un nouveau comportement à adopter dans le cas d'une collision, afin de vérifier si l'obstacle est un robot. Dans ce cas, l'un des deux devra laisser passer l'autre sans pour autant changer de destination. Nous supposons que les messages d'acquittements envoyés par la simulation (ou les robots physiques) à l'application PIL ne sont pas perdu. Cela peut engendrer des blocages, car une application PIL attendra indéfiniment la réponse de ROB avant de continuer l'exploration.

Actuellement, les robots ont tendance à se diriger dans la même direction, ce qui nuit à l'efficacité de l'algorithme, et, en condition réelle, risque d'entraîner des collisions entre robots, ce qui peut nuire à la localisation de ceux ci en introduisant des dérives ... Nous pourrions adapter l'heuristique de l'algorithme de recherche de chemin pour encourager une dispersion "contrôlée" des robots. Cela peut se faire en ajoutant des coûts aux frontières dont plusieurs robots sont proches.

Enfin, la grille constituant l'environnement et la carte interne des robots a actuellement une résolution d'une largeur de robot. Ainsi, les obstacles découverts auront comme dimensions un multiple de la largeur d'un robot, ce qui peut générer des imprécisions. Il serait intéressant de diviser la grilles en de plus petites cases. Chaque robot occuperais plusieurs cases, et nous pourrions déplacer les robots plus précisément que d'une largeur de robot par déplacement.

Difficultés rencontrées : Nous avons malheureusement passé beaucoup de temps sur le développement de l'application "Simulation". Celle-ci dispose d'une architecture complète et propre mais certaines parties qui nous ont pris du temps ne sont pas nécessaires. Par exemple, nous avons développé toutes les fonctions de déplacement des robots réels, tandis que nos algorithmes n'utilisent que les fonctions "init", "join", et "move" d'Airplug. Nous aurions pu nous contenter d'une application plus simple comme un simple affichage d'un environnement déjà créé au préalable par exemple. Cela aurait permis aux personnes réalisant la simulation de venir en renfort concernant l'implémentation des algorithmes d'exploration.
L'exploration d'un environnement dans des conditions réelles avec les robots n'a malheureusement pas pu être expérimentée, bien que la simulation que nous avons développée assure en théorie une possibilité de déploiement quasi-transparent.

Références

[1] Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. *Conflict-free Replicated Data Types*. [Research Report] RR-7687, 2011, pp.18. inria-00609399v1

[2] Reid Simmons, David Apfelbaum, Wolfram Burgard, Dieter Fox, Mark Moors, Sebastian Thrun, Håkan Younes *Coordination for Multi-Robot Exploration and Mapping*. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

[3] Micah Corah and Nathan Michael *Efficient Online Multi-robot Exploration via Distributed Sequential Greedy Assignment*. The Robotics Institute, Carnegie Mellon University

[4] Abraham Sanchez L. and Alfredo Toriz P. *Coordinated multi-agent exploration*. Facultad de Ciencias de la Computacion, BUAP, 14 Sur esq. San Claudio, CP72570 Puebla, Pue., Mexico

[5] Brian Yamauchi. *Frontier-Based Exploration Using Multiple Robots*. Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Washington, DC 20375-5337, 1998

[6] Antoine Bautin. *Stratégies d'exploration multirobot fondées sur le calcul de champs de potentiels*. Université de Lorraine, 2013