Rapport final

Challenge UTAC CERAM

PFE n°2026

2021/02/05

Résumé : Gestion d'un platoon de véhicules autonomes et connectés en milieu urbain

Contexte/Enjeux : Participation au challenge UTAC CERAM

Equipe: Margot IRLINGER, David OLIVARES, Emma PALFI, Jean PROUVOST, Alexandre SEGERAL, Jimmy VUONG
Mentors: Naila BOUCHEMAL, Jae Yun JUN KIM

Historique des modifications

Name	Date	Changes
Toute l'équipe	05/02/2021	Premier rendu complet

Sommaire

Glossaire

Term	Definition
C	Contrôleur
P	Platoon
Vi	Véhicule d'insertion
Vd	Véhicule de désinsertion
Vt	Véhicule de tête
Vk	Véhicules suiveurs
HF	High Frequency
LF	Low Frequency
ITS	Intelligent Transport Systems
C-ITS	Cooperative Intelligent Transport Systems
ROS	Robot Operating System
V2V	Vehicle-to-Vehicle
CAM	Cooperative Awareness Message
DENM	Decentralized Event Notification Message
SPAT	Signal Phase And Timing
Lidar	Light Detection and Ranging
NHTSA	National Highway Traffic Safety Administration
OICA	Organisation Internationale des Constructeurs automobiles
UNECE	Commission économique pour l'Europe des Nations unies
SAE	Society of Automotive Engineers
ONU	Organisation des Nations Unies

1 Introduction

Ce projet a été proposé par Mme Bouchemal et M. Jun Kim dans le but de participer au challenge de l'UTAC CERAM dans la catégorie libre. Il s'articule autour de l'**amélioration de la gestion d'un convoi de véhicules autonomes**.

Le but de ce projet est d'apporter une innovation dans le domaine de la gestion de convoi de véhicules autonomes. Mme Bouchemal et M. Jun Kim ont choisi de nous faire travailler sur la conception d'une solution permettant la gestion, dont l'insertion et la désinsertion d'un véhicule, d'un convoi **en milieu urbain**.

En effet, d'après l'*Association prévention routière*[1], **90% des accidents** de la route sont imputables à des **erreurs humaines**. De plus, d'après l'*Agence internationale de l'énergie*[1], les transports sont responsables de **24% des émissions directes de C02** et les véhicules représentent le ¾ de ces émissions. L'application de convois autonomes en zone urbaine permettrait donc d'**améliorer la sécurité routière,** de **fluidifier le trafic** et de **réduire la consommation d'énergie**.

Notre projet est à la fois un projet de recherche et de développement. Pour le réaliser, nous avons travaillé sur un environnement de **simulation** en nous inspirant de technologies déjà existantes et utilisées dans le secteur des véhicules autonomes.

2 Contexte du projet

2.1 Concours

L'**UTAC CERAM** (Union Technique de l'Automobile, du motocycle et du Cycle; Centre d'Essais et de Recherche Appliqué à la Mobilité)[2] organise son premier **challenge** dédié aux étudiants. Le challenge a pour objectif de développer le domaine des véhicules autonomes en France et sera répété chaque année. Plusieurs catégories existent dans le concours, dans le cas de ce projet nous participerons à la catégorie **épreuve libre**.

Le principe de la catégorie épreuve libre est d'être une épreuve sans objectif défini, pouvant accueillir toutes formes de projet sans nécessité de véhicule roulant contrairement au reste des épreuves. L'UTAC CERAM donne pour exemple du software ou des capteurs, notre projet entrant dans la première catégorie (avec si cela est possible un prototype de démonstration de ce software).

2.2 Problématique

Nous réalisons ce projet dans le but de participer au challenge de l'UTAC CERAM dans la catégorie libre. Pour pouvoir remporter cette catégorie, nous devons apporter une innovation dans le domaine du véhicule autonome.

Dans le cadre de notre projet, l'innovation apportée est la conception d'une solution permettant la gestion d'un convoi autonome de véhicules, incluant l'insertion et la désinsertion d'un véhicule, en environnement urbain. En effet, comme nous le verrons dans l'état de l'Art, des projets constitués de convois de camions autonomes ont déjà été développés sur autoroute mais pas en ville. Ce qui nous amène à la problématique suivante :

Comment développer une infrastructure **connectée**, **embarquée**, **débarquée** et **adaptée** à la **gestion** d'un **convoi autonome** de **niveau 4** [3] en **milieu urbain** grâce à la technologie **V2X** ?

3 Fonctions et contribution des acteurs

3.1 Acteurs

Les coordinateurs de ce projet sont **Madame Bouchemal** et **Monsieur Jun Kim**. Madame Bouchemal agit en qualité de mentor et d'expert en **communication**, tandis que monsieur Jun Kim nous assiste en tant qu'expert en **robotique** et intelligence artificielle.

D'autre part, le projet est réalisé par l'équipe de PFE, composée de 6 membres. Parmi les membres, Emma Palfi, Alexandre Ségéral, David Olivares et Margot Irlinger sont en majeure Systèmes Embarqués. D'autre part, Jean Prouvost est en majeure BDA et Jimmy Vuong en majeure OCRES.

Finalement, plusieurs acteurs extérieurs sont impliqués dans ce projet. L'**UTAC CERAM**, en tant qu'organisateur du concours et en ayant travaillé conjointement avec Madame Bouchemal et M. Jun Kim à l'élaboration de ce projet.

3.1.1 Répartition du travail

Pour plus d'efficacité, le groupe de PFE a été réparti en **deux sous-groupes**, un groupe **robotique** et un groupe **connectivité**. Le groupe de connectivité est composé d'Emma Palfi, Jimmy Vuong et Margot Irlinger. Le groupe robotique est composé d'Alexandre Ségéral, Jean Prouvost et David Olivares. De plus, pour réaliser la bonne réalisation de chaque tâche, un **coordinateur** pour chaque tâche est désigné.

Dans le groupe connectivité, Jimmy Vuong s'est assuré de la bonne résolution des recherches sur les messages DENM/CAM/SPAT (message V2X), Margot Irlinger a mené les recherches sur l'ASN.1 et Emma Palfi l'implémentation des messages sur les robots en collaboration avec Alexandre Ségéral. Margot Irlinger dirige le développement des messages ECE en qualité de chef de projet. L'équipe connectivité a ensuite participé activement au développement sous ROS en C++ des entités du projet, à savoir le contrôleur, les véhicules et le feu de circulation.

Dans le groupe robotique, Jean Prouvost assure la bonne réalisation de l'environnement de simulation, David Olivares a assuré le bon suivi de trajectoire et comme indiqué plus tôt Alexandre Ségéral s'occupe de la bonne implémentation des messages ECE sur les robots. Ensuite, Jean Prouvost et David Olivares s'occupent de l'asservissement des robots lié aux messages sous ROS en Python. Finalement, Alexandre Ségéral dirige le développement de l'insertion et de la désinsertion de façon à valider les cas d'usage.

3.1.2 Outils de communication

Nous utilisons la plateforme Discord pour communiquer au sein du PFE car elle nous permet de créer des **salons d'études** selon les sujets abordés et d'organiser nos messages pour mieux s'y retrouver ensuite. Cette plateforme est aussi très pratique pour partager des ressources temporaires.

Pour rédiger et rassembler tous les documents nécessaires au projet, nous utilisons **Google Drive**. Nos mentors Mme Bouchemal et M. Jun Kim y ont accès afin de pouvoir consulter les rendus rapidement et facilement.

Enfin, nous utilisons **Github** pour développer et fusionner nos parties. Nous allons aussi partager le repository Git à nos mentors afin qu'ils puissent voir l'avancement du projet.

3.2 Planning

Concernant le planning initialement prévu dans le cahier des charges (voir ci-dessous), nous avons été **retardés** dans les deux équipes par d'**importantes difficultés** rencontrées **fin décembre/début janvier**. Ce n'est que très tard, après avoir testé beaucoup de solutions, que nous avons pu obtenir une **base solide** et développer les fonctionnalités importantes du projet. En effet, démarrer ce projet **depuis zéro** nous a

demandé beaucoup plus de temps que nous le pensions. Finalement, le fait d'avoir tout de même réussi à implémenter une **base de développement stable** nous a permis, vers la **fin du mois**, d'avancer **très vite** et d'implémenter **facilement** les fonctionnalités restantes.

Les tâches que nous n'avons pas pu réaliser par manque de temps et de moyens sont la détection d'obstacles et le suivi de trajectoire par capteurs mais ne sont pas essentielles au PFE.

Concernant la **méthodologie** de développement, nous avons gardé celle prévue à la base qui consiste à commencer par des **tâches simples** puis à les **complexifier** plus tard afin de pouvoir assurer l'obtention d'une **solution fonctionnelle** à la fin du mois malgré les difficultés rencontrées. De plus, à chaque nouvelle fonctionnalité développée, nous la **testions** afin d'être sûrs de son bon fonctionnement, que ce soit pour l'équipe robotique, l'équipe connectivité ou les deux.

Phases	Tâches	Sept		Octobre				Novembre				Décembre					Janvier				
		39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	1	2	3	4	5
Conception	SPRINT 1																				
	EDA + SCINAN																				
	Contenu CAM / DENM / SPAT																				
	Conception messages ECE																				
	Installer ROS																				
	Définir les cas d'usage																				
	CDC version 1																				
	Recherches sur ASN1																				
	Recherches interfaçage ANS1																				
	CDC version finale																				
Dév. ASN.1	Développement messages ECE																				
	Implémenter simu CAM / DENM / SPAT																				
	Tests messages CAM / DENM / SPAT																				
	Intégrer messages ECE à ROS																				
	Tests messages ECE																				
	Simuler envoi messages contrôleur																				
	Tests contrôleur																				
Dév. ROS	Cartographie virtuelle (template)																				
	Implémenter suivi de trajectoire																				
	Tests suivi de trajectoire																				
	Implémenter plusieurs robots																				
	Implémenter suivi des robots suiveurs																				
	Tests suivi des robots																				
	Implémenter asservissement messages																				
	Tests asservissement messages																				
	Implémenter détection d'obstacles																				
	Tests détection d'obstacles																				
ASN.1 & ROS	Tests de chaque cas d'usage et validation																				

4 Etat de l'art

4.1 FURBOT

Le projet FURBOT (Freight Urban Robotic Vehicle) est un véhicule spécialisé dans les livraisons en zone urbaine. Il n'est pas autonome mais possède une assistance à la conduite (freinage d'urgence, parking assisté, etc). Au niveau robotique, le véhicule possède un bras qui peut sortir pour prendre le colis et le ressortir ensuite grâce à des palettes.

Ce projet peut nous être utile concernant son assistance à la conduite.

4.2 Cargo-ANTS

L'objectif du projet Cargo-ANTS est la manutention des marchandises par les systèmes de transport automatisés de nouvelle génération pour les ports et les terminaux et vise à créer des véhicules guidés automatisés (AGV) et des camions hautement automatisés (HAT) intelligents qui peuvent coopérer dans des espaces de travail partagés pour un transport de marchandises efficace et sûr dans les principaux ports et pour le fret des terminaux.

Plusieurs articles ont été publiés dont :

- *« Robust and real-time detection and tracking of moving objects with minimum 2D LiDAR information to advance autonomous cargo handling in ports »*
  Cet article parle d'une solution à moindre coût afin de tracker les mouvements d'objets mouvants (système désigné DATMO) dans le cadre du cargo automatique en grid-less (sans système extérieur au véhicule pour gérer leurs déplacements). La solution n'est pas cher car elle repose exclusivement sur des 2D LRF (Laser rangefinder). La partie tracking de l'algorithme est assurée par un MHT (multi-hypothesis tracking). Le MHT sert à déterminer plusieurs trajectoires possibles et leurs probabilités.

  C'est un article intéressant dans notre cas, normalement nous aurons besoin d'implémenter une technique plus ou moins proche (pour le suivi des véhicules).

- *« Potential information fields for mobile robot exploration »*

  L'article est intéressant puisqu'il a pour thème l'exploration (en combinant les « champs » couverts par les différents capteurs des robots et l'entropie de l'information qui leurs sont liés, détermination des trajectoire optimal pour explorer le terrain).

- *« Terrain classification in complex 3D outdoor environments »*

  L'application décrite dans cet article sert à la détection d'obstacles et à son évitement, elle est donc utile pour notre projet.

- *« Chromatic Shadows Detection and Tracking for Moving Foreground Segmentation »*

  Cela peut être intéressant, il s'agit ici de réaliser la détection des objets mouvants, cette fois en utilisant leurs ombres. Contrairement aux projets qui utilisent l'ombre « penumbra » (seulement les zones en partie atteinte par les sources de lumière), cet article indique que la méthode développée peut utiliser les ombres « penumbra » et « umbra » (ou la lumière n'atteint pas la zone). La détection des ombres est basée sur les gradients « edge partitioning » et la répartition statistique des couleurs.

4.3 CityMobil2

Le projet CityMobil2 consistait au déploiement de deux flottes de 6 véhicules autonomes à dix passagers en Europe. Les véhicules ont de l'électronique embarqué pour aider à prendre des décisions de mouvement mais les décisions de flotte étaient prises au niveau du système central.

4.4 MOSARIM

Le but du projet MOSARIM était d'établir des normes, spécifications et précautions au sujet des interférences mutuelles provoquées par les radars des véhicules. Les systèmes de radar à ondes millimétriques sont utilisés pour des applications automobiles, principalement dans des fonctions de confort et parfois dans des fonctions de sécurité. Jusqu'à présent aucune mesure où précaution n'a été prise pour réduire ou éviter les interférences mutuelles. Or ces interférences peuvent être dangereuses si les radars sont utilisés pour la sécurité comme pour la détection d'obstacles par exemple.

Ils ont effectué des tests sur 8 cas d'usage :

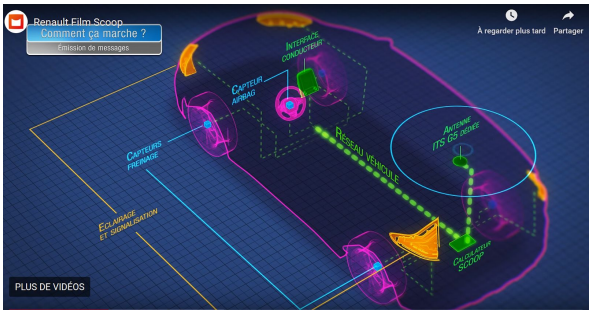
- ACC - Adaptive Cruise Control
- CWS - Collision Warning System
- CMS - Collision Mitigation System
- VUD - Vulnerable road User Detection
- BSD - Blind Spot Detection
- LCA - Lane Change Assist
- RCA - Rear Cross-traffic Alert
- BPA - Back-up Parking Assist

Ce projet nous a aidé à comprendre comment définir des cas d'usage et pourra nous servir si nous utilisons des radars comme capteurs.

4.5 Scoop

Le projet Scoop est un projet de déploiement de transports automobiles autonomes capables d'échanger des informations avec la route. Pour cela, chaque véhicule sera muni d'unités embarquées et communiquera via la 5G. Ainsi, ce projet se base sur une communication V2V (Vehicle to Vehicle) mais possède également une communication V2I (Vehicle to Interface) avec des bornes situées aux bords des routes. Ces bornes transmettent des informations à un centre de traitement de données Scoop qui examine en temps réel l'état des routes et le comportement des voitures.

Comme nous pouvons le voir dans la photo ci-dessous, chaque véhicule transmet et reçoit les informations à travers une antenne ITS G5 qui permet de détecter son environnement sur une portée située entre 500 et 1000m.



Ce projet est soutenu par le ministère français du transport et a pour objectif d'améliorer la sécurité routière et réguler le trafic dans des zones denses. Pour effectuer ses tests et montrer que ce projet est viable, le projet vise à mettre en état de circulation environ 3000 véhicules répartis sur 5 zones différentes et de densités de circulation différentes en France.

Avant la mise en circulation, plusieurs cas d'usages ont été mis en avant:

- L'Optimisation du Trafic
- L'information sur les travaux sur la route
- Le Freinage d'urgence et les pannes

Ainsi, ce projet est très similaire au nôtre et nous a aidé à nous projeter sur les différents problèmes et cas d'usages que nous pourrions rencontrer lors de la réalisation.

4.6 Companion

Le projet Companion consiste à gérer un peloton de camions de 38 tonnes. Ses objectifs sont d'augmenter la sécurité, réduire la consommation et optimiser les voyages. Le schéma ci-dessous résume les différentes fonctionnalités technologiques que Companion utilise.

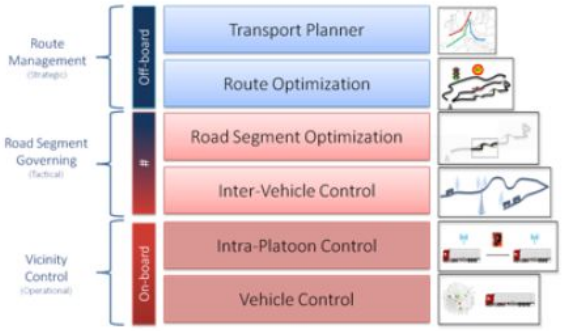


Figure 13 Les briques d'utilisation technologiques et opérationnelles de COMPANION

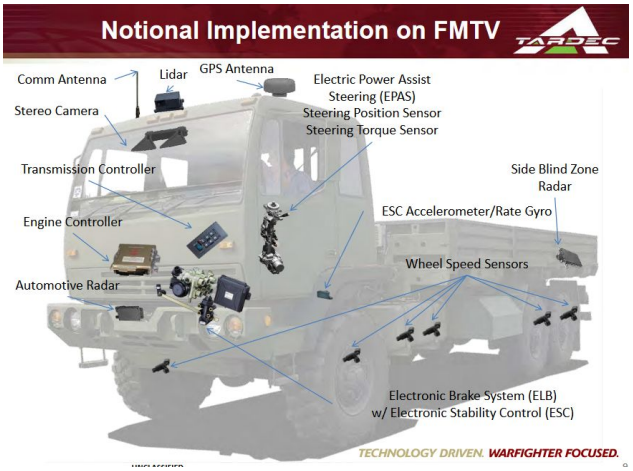
Parmi les technologies que ce projet utilise, nous notons la présence d'une communication V2V avec des échanges de messages CAM et DENM entre les véhicules. De plus, les données C-ITS sont couplées avec des informations en utilisant des données météorologiques pour une meilleure localisation.

En résumé, ce projet a été pour nous une introduction au fonctionnement des signaux CAM et DENM pour la gestion du platoon. Il nous a également montré tous les résultats aussi bien positifs que négatifs sur ce projet et des pistes d'amélioration des cas d'usage.

4.7 AMAS

Le projet AMAS (Autonomous mobility Applique System) est un peu différent de notre projet mais a retenu notre attention par rapport aux diverses technologies utilisées. Il ne s'agit pas, en effet, d'un véhicule autonome mais de l'intégration de plusieurs modules sur un véhicule permettant ainsi de le rendre semi-automatique. Le véhicule possède une assistance sur les collisions, une aide pour maintenir une position dans un peloton et d'autres fonctionnalités.

Ce projet n'est pour le moment utilisé que par l'armée. Contrairement à d'autres projets, celui-ci ne se sert que de capteurs et d'un Lidar. Le Lidar permet la détection par laser pour mesurer la distance entre les différents véhicules et de capteurs tout autour du véhicule afin d'éviter les collisions. Le schéma ci-dessous montre l'implémentation des différents modules.



Ce projet est donc intéressant pour nous notamment sur le choix des capteurs à utiliser pour nos véhicules.

4.8 AdaptIVe

AdaptIVe développe, teste et évalue des applications de conduite automatisée pour les voitures particulières et les camions dans la circulation quotidienne.

Ce projet se divise en trois scénarios :

1) Courte distance :

   Vitesses basses jusqu'à 30km/h : correspondant manœuvres ou stationnements. Ces manœuvres et le contrôle du véhicule sont tout d'abord concentrés sur la présence d'obstacles proches.

   Elle se décline en plusieurs niveaux d'automatisation :

   - Partielle : Aide au stationnement
   - Conditionnelle : Stationnement sans conducteur dans le véhicule. Cela peut prendre certaines formes : stationnement dans un garage, stationnement automatique dans un parking (trouver une place libre et s'y stationner).
   - Totale : Si le conducteur ne réagit pas à une demande de prise de contrôle, le système ralentit et arrête le véhicule en toute sécurité.

2) Environnement Urbain :

   Voici les niveaux d'automatisation proposés :

   - Conduite assistée : vitesse de croisière en ville, supervision des objets et signaux, suivi de ligne, suivi de véhicule, évitement d'obstacles et des usagers vulnérables (VRU).
   - Automatisation conditionnelle : changement de voie automatique, gestion des giratoires et intersections.
   - Totale : Arrêt d'urgence du véhicule sans autorisation du conducteur.

3) Voies rapides : AdaptiVe considère des vitesses jusqu'à 130 km/h sur des autoroutes ou infrastructures routières similaires.

   Niveaux d'automatisation :

   - Partielle : Insertion et désinsertion sur voie rapide et réponses coopérative aux véhicules de secours et d'entretien (V2V).
   - Conditionnelle : Adaptation de la vitesse et des interdistances, changement de voie et insertion dans une voie, élargit la connaissance du système sur les situations qui ne sont pas dans le champ de vision du véhicule, gestion des situations d'embouteillage (start-stop, suivi du véhicule précédent).
   - Totale : Arrêt d'urgence sans que le conducteur n'ait à intervenir.

4.9 SARTRE

Le projet SARTRE (Safe Road Trains for the Environment) définit le système de platoon et le fonde sur des améliorations de sécurité dues aux systèmes de contrôle autonomes à utiliser dans les véhicules de tête (camions) et les véhicules suivants (camions, voitures, SUV).

Si nous considérons que 87% des accidents mortels sur la route sont causés par des conducteurs, les systèmes automatisés s'avèrent être des moyens plus sûrs et efficaces. Ceci est possible notamment par la réduction du coefficient de traînée aérodynamique : une diminution moyenne de 20% a été constatée pour 4 véhicules avec un écart de longueur de 0,2 véhicule.

4.10 Autonet 2030

Le projet Autonet 2030 est un projet européen qui vise à coupler le domaine des véhicules autonomes avec celui des systèmes de transport intelligent. La problématique principale est de comprendre comment faire coopérer un parc de véhicules autonomes et classiques. Dans cette optique, l'équipe chargée du projet explore les différentes possibilités au niveau du format des messages ou encore de l'architecture à adopter (centralisée ou décentralisée).

Le but de ce projet est de proposer un standard pour les futurs projets européens portant sur le sujet. Les auteurs prennent comme base de travail un futur parc automobile qui supporte le mode automatique. Les véhicules qui n'en sont pas pourvus sont appelés "legacy".

On peut les diviser en quatre catégories :

- Les "cooperative vehicles", qui peuvent être conduits manuellement et donnent des indications aux conducteurs.
- Les "cooperative automated vehicles", qui supportent le mode coopératif automatique.
- Les "cooperative autonomous vehicles", qui représentent une sous classe dérivant des véhicules coopératifs.
- Enfin la dernière catégorie, regroupent les précédentes à la différence près qu'elles sont conduites de manière exclusivement manuelle. Nous les appelons des manually driven vehicles.

Au cours de l'étude, Autonet 2030 met en avant un système hybride centralisé et décentralisé. Cette manière de faire permet d'assurer un contrôle très précis sur l'échange de données. Cela permettra aux véhicules de rouler de manière très précise et aisément. Dans le même temps, ce système pourra coordonner et gérer le comportement d'un ensemble d'entités qui correspond à la route.

Dans cette optique, elle est scindée en zone coopérative globale ou locale. Elles sont administrées par une entité maître appelée master, en charge des interactions entre les autres zones de coopération locale. De plus, c'est elle qui coordonne les échanges d'information en son sein. Les différents groupes de véhicules qu'elles administrent sont regroupés dans une zone de coopération dynamique. Elle peut comprendre des pelotons ou des convois.

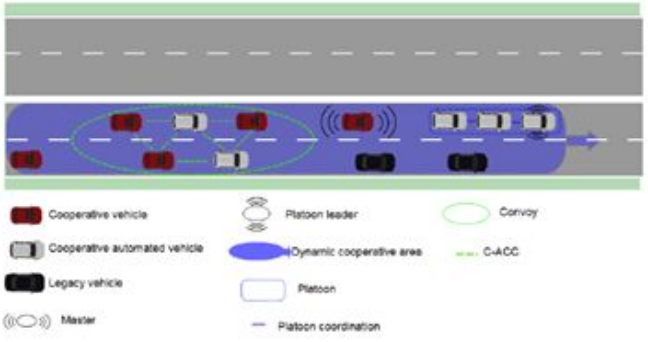


Schéma d'une zone coopérative locale

Nous pourrions nous inspirer de l'architecture présentée ici pour mieux comprendre la dynamique de notre peloton dans un environnement complexe.

4.11 Conclusion

L'ensemble des projets présentés nous donne de nombreuses informations sur la gestion ainsi que sur la conception d'un réseau de voiture autonome. Néanmoins, aucun ne traite de manière spécifique l'insertion et la désinsertion d'un véhicule au sein d'un peloton en **environnement urbain**. Nous avons ainsi développé des cas d'usage spécifiques pour étudier la question.

5. Biomimétisme

Les **fourmis**, par leurs **structures sociales complexes**, sont un **organisme modèle** couramment utilisé. De nombreuses études ont été menées sur celles-ci, dont la découverte de leur **capacité à voir les ultraviolets**[4] ou la **sélection de parentèle**[5]. De plus, l'étude des fourmis a été poursuivie aussi en **informatique et en robotique**, ou elle a permis le **développement des robots fourmis**[6] et des **algorithmes de colonie de fourmis**[7].

Dans le cas de notre projet, nous nous sommes donc intéressés à la **piste des fourmis**, mais d'une **nouvelle manière que celle présentée dans les algorithmes de fourmis**. En effet, nous avions dans le cahier des charges de notre projet la **communication des véhicules au travers du contrôleur**. Cette communication **vise à l'insertion de nouveaux véhicules**, au **maintien du suivi de trajectoire** dans le platoon et finalement à la **désinsertion d'un véhicule du platoon** une fois à destination.

Un élément essentiel de **suivi de piste des fourmis**, qui est **exploité dans les algorithmes de colonie de fourmis**, est le **dépôt de phéromones tout au long de la piste** afin d'indiquer celle-ci et surtout **où la piste commence et s'arrête**. Ce **phénomène** ayant déjà été utilisé en partie dans le **suivi de trajectoire** pour des projets passés, nous avons décidé de conserver **l'utilisation des messages comme équivalent des phéromones**, déposant une **piste sous forme de position à suivre pour les véhicules à la suite**. De plus, notre **valeur ajoutée** est axée autour du **nouveau type de messages** qui indiquent aux véhicules à quel **endroit ils doivent s'insérer ou se désinsérer,** s'inspirant de **l'aspect temporaire des pistes des fourmis** (dû à la volatilité des phéromones).

6. Méthodologie de la partie connectivité

6.1 La technologie V2X

Pour communiquer les informations entre nos différents véhicules, nous devions utiliser la **technologie LTE (Long Term Evolution)** qui désigne le réseau 4G. Le LTE_V est donc une version spécifique dédiée aux véhicules. Cependant, nous avons finalement réalisé notre solution sous forme de simulation à cause de la crise sanitaire. Nous avons malgré tout instauré une **communication V2X (Vehicle-to-Everything)** qui prend en compte plusieurs types de communication (V2V, V2I, etc.). A travers cette communication, nous avons envoyé **différents types de message** : des CAM, des DENM et des SPAT que nous avons re-créés et implémentés dans la simulation.

6.1.1 Les messages CAM

Les messages CAM ont pour objectif d'assurer la **coopération des véhicules autonomes** entre eux et la **détection de leur environnement**. Les messages CAM sont **périodiques** et se déclenchent en général lors d'un des 3 cas suivants:

- un changement de position de plus de 4m
- un changement de vitesse de plus de 0.5m/s
- un changement de direction de plus de 4°

La génération de ces messages périodiques s'effectue en moins de 50 ms mais le temps de communication du message CAM doit être compris entre 100 et 1000ms. Comme nous le spécifions au début, ces messages sont présents pour gérer le comportement du platoon, par conséquent ce sont ces messages qui contiennent les informations sur la position des voitures, l'interdistance entre celles-ci, la vitesse et d'autres types d'informations.

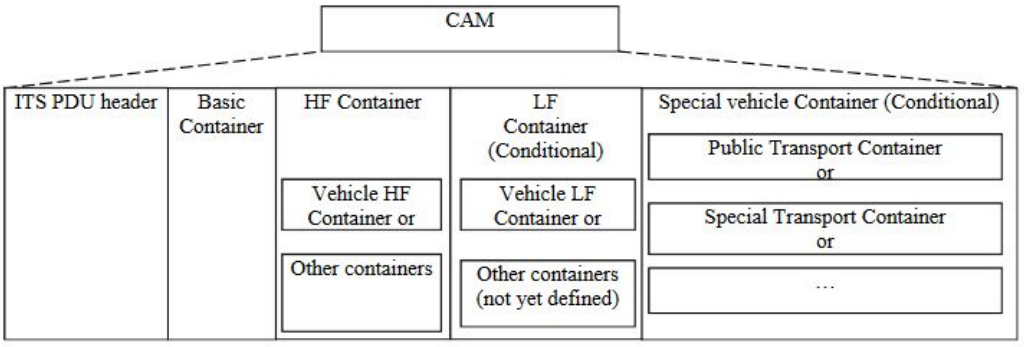


Figure 4: General structure of a CAM

Les messages CAM sont structurés comme dans le schéma ci-dessus où chaque bloc contient un type d'information précis. Nous avons donc regardé quel type d'information se trouvait dans chacun de ces blocs.

- Tout d'abord, l'entête d'un CAM, c'est-à-dire le **PDU Header,** est une en-tête commune aux DENM et aux SPAT, qui comporte la version du protocole, le type de message à transmettre et l'ID.

- Le **Basic Container** quant à lui contient toutes les informations basiques provenant de l'ITSS c'est-à-dire de l'antenne Marben dans notre cas. Nous retrouverons donc les informations liées à la localisation de la voiture et le type d'ITS-S.

- Le **High Frequency Container** contient toutes les informations qui nécessitent d'être transmises rapidement entre les différents véhicules comme la vitesse, l'accélération, etc.

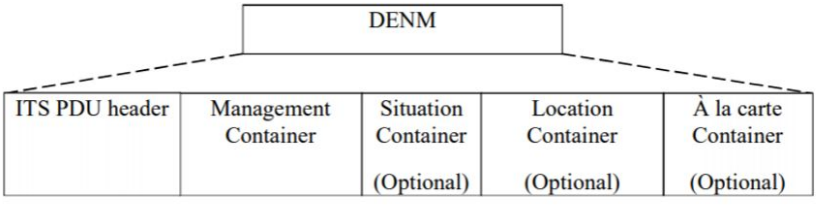
- Le **Low Frequency Container** contient toutes les informations qui sont de plus faible priorité et qui n'ont pas besoin d'être contrôlées à chaque instant comme l'état des phares, la température des sièges, la clim, etc.

- Le **Special vehicle container** contient les informations spécifiques à chaque type de véhicule (exemple: gyrophare pour les ambulances, les pompiers ou les policiers).

6.1.2 Les messages DENM

Les **DENM** sont, à l'inverse des CAM, des messages **événementiels**. Ils se déclenchent lorsqu'un obstacle ou un danger survient sur la route du véhicule. Une fois l'obstacle ou le danger détecté, le message DENM est émis à l'attention de tous les autres véhicules autonomes à proximité pour les prévenir du danger. Le message se répète jusqu'à ce que le véhicule émetteur reçoive ou émette un message d'annulation. Nous parlons de ***cancellation*** pour les messages provenant de l'ITS d'origine, Intelligent Transport Systems, et de ***termination*** pour ceux provenant de ses voisins. Parallèlement, le DENM peut aussi être interrompu si la "*duration time"* est dépassée.

De par sa nature, ce type de message est prioritaire. Son objectif est de fournir un maximum d'informations sur l'évènement détecté.



Structure général des messages DENM

Afin de remplir leur mission, les DENM sont structurés comme indiqué ci-dessus. Nous pouvons le scinder en deux conteneurs bien distincts. Le premier est identique aux CAM c'est l'***ITS PDU header***. Il y remplit les mêmes fonctionnalités. Le second est appelé ***DENM Payload***, il contient l'ensemble des données sur l'incident.

- Le ***Management Container*** est le seul qui ne soit pas optionnel. En effet, il rassemble les informations de gestion du DENM (ID, *duration time*, *reference time, transmission interval* ...).
- Le ***Situation Container*** nous donne des précisions sur le type d'événement détecté ainsi que sur sa cause.
- Le ***location container*** complète la compréhension de l'événement en nous indiquant sa localisation ou encore sa vitesse.
- Enfin, le dernier bloc ***A la carte container*** apporte les informations spécifiques sur le cas d'usage qui nécessite l'envoi d'informations supplémentaires qui ne sont pas présentes dans les précédents containers.

6.1.3 Les messages SPAT

Un SPAT (Signal Phase and Timing) est un message utilisé pour **transmettre l'état** de plusieurs **intersections signalisées**. Grâce à certaines informations, le receveur du message peut déterminer l'état du signal et prédire quand le feu tricolore passera à la prochaine étape. Le message SPAT envoie des valeurs sur l'état actuel (quand l'état a commencé, quand il va potentiellement finir, etc).

SPAT							
PDU Header	LVL 0: SPAT	LVL 1: Intersection StateList	LVL 2: Movement List	LVL 3: Movement EventList	LVL 4: Movement EventList (time)	LVL 5: Movement EventList (Speed)	LVL 6: Maneuver AssistListe

Un message SPAT se décompose comme dans le schéma ci-dessus :

- PDU Header : Contient la version du protocole, le type de message à transmettre et l'ID
- LVL 0 : Contient les informations du SPAT soit le nom du signal, l'état de l'intersection et le temps où il a été envoyé.
- LVL 1 : Contient la liste de toutes les intersections
- LVL 2 : Contient la liste des manoeuvres à envoyer aux véhicules
- LVL 3 : Contient la liste de tous les états
- LVL 4 : Contient la liste des temps de chaque état
- LVL 5 : Contient la vitesse des véhicules pour chaque état
- LVL 6 : Contient les informations qui permettent de gérer les manœuvres des véhicules en fonction de l'état de l'intersection, des piétons, ...

6.2 Le contrôleur

Afin de pourvoir gérer l'insertion et la désinsertion d'un véhicule, nous avons imaginé une entité **omnisciente**, que nous appelons "contrôleur", chargée de **superviser l'ensemble des véhicules** et des **convois** d'une ville. Nous avions défini son rôle dans le cahier des charges et nous l'avons mis à jour dans ce rapport selon les objectifs que nous avons atteints.

6.2.1 Le rôle du contrôleur

- Le contrôleur peut **former un platoon** et indiquer la voiture de tête et la position des autres véhicules dans le platoon.
- Il connaît les destinations et les positions des voitures et des platoons et le nombre de voitures de chaque platoon, il peut donc indiquer à une voiture de **rejoindre un platoon** ou de le **quitter**.
- Il permet de **gérer les feux tricolores** en déterminant si le platoon a le temps de passer au vert. Si le platoon n'est pas trop proche, le feu peut passer au rouge. Si le platoon est déjà engagé dans la traversée du feu ou trop proche pour freiner sans piler, le contrôleur laissera le feu au vert le temps que le platoon puisse passer. Cela permet d'éviter que le platoon ne soit séparé en deux.

6.2.2 Définition des phases de platooning

Pour pouvoir faire la conception des messages, nous avons rédigé les **cas d'usages** qui concernent notre valeur ajoutée, c'est-à-dire la phase d'initialisation, l'insertion et la désinsertion. Ces phases ont été définies dans le cahier des charges.

6.2.2.1 Phase Initialisation

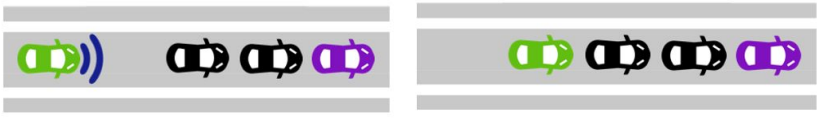
User Story

Les quatre véhicules sont proches. Le contrôleur choisit en fonction des positions et des destinations des voitures quelle sera la voiture de tête et les positions des voitures suiveuses. Le contrôleur indique la formation du platoon au platoon (trajectoire, positions, inter-distance). La voiture de tête démarre avec le suivi de trajectoire, l'interdistance et la vitesse donnés par le contrôleur.

Description fonctionnelle

- Les voitures indiquent au contrôleur leur destination
- Le contrôleur choisit la tête de platoon (véhicule autonome ou bien avec conducteur certifié)
- Téléchargement des informations nécessaires à la gestion du platoon : trajectoire, mise à jour logicielle afin de préparer le véhicule à une conduite adaptée à son environnement...
- Définition des paramètres du platoon : (nombre de véhicules, vitesse, interdistance, répartition des ID)

6.2.2.2 Phase d'insertion



Insertion en queue du platoon

User Story

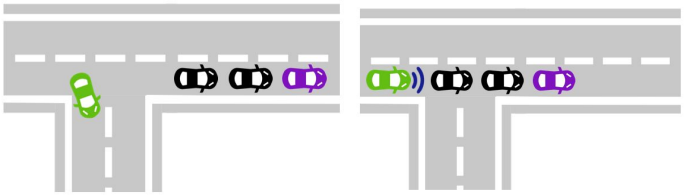
Un véhicule souhaite s'insérer **derrière** le platoon. Il prévient la voiture de tête qui va lui donner la permission. Le véhicule de tête ajoute le véhicule d'insertion au réseau du platoon. Ensuite, il se rapproche du platoon et s'insère. Il adapte sa conduite aux informations de la voiture de tête. Il prévient la voiture de tête et le contrôleur de son insertion réussie.

Description fonctionnelle

- Définition d'un point d'adhérence au platoon par le contrôleur
- Envoi des informations du véhicule à la tête du convoi (la position)
- Si la distance <= 2 m (contrainte issue du standard), alors le véhicule commence à communiquer en V2V avec le platoon

- Le processus de régulation de **vitesse et inter-distance** est activé (CACC processus défini dans le standard ETSI) (utilisation de messages CAM)
- Informer la tête de platoon de la fusion
- Le contrôleur détermine la position du véhicule d'insertion (forcément en queue du platoon) et détermine le nombre total de véhicules dans le platoon.
- La voiture doit connaître sa position pour les messages ensuite

6.2.2.3 Phase de désinsertion



Désinsertion d'un véhicule en queue du platoon

User story

- Si la voiture est **en queue** du platoon. Elle envoie une demande de sortie au contrôleur, freine, et sort à l'endroit indiqué par la tête de platoon. La voiture de tête supprime la voiture du réseau. Le conducteur reprend les commandes ou continue en autonome avec ses capteurs.

Description fonctionnelle

- Le véhicule envoie une demande au contrôleur et/ ou à la tête du convoi
- Le contrôleur informe les autres véhicules du convoi
- Le contrôleur calcule la nouvelle vitesse du véhicule sortant et définit son point de sortie
- Calcul par le contrôleur et **régulation de vitesse et inter-distance** (augmenter la distance) afin de permettre au véhicule de sortir
- Le véhicule quitte le platoon et reprend sa trajectoire individuelle
- Le contrôleur informe les autres véhicules du convoi (nouvelles positions)
- Revenir à la configuration initiale du convoi

6.2.2.4 Exigences d'un platoon détaillées

6.2.2.4.1 Interdistance (Norme CACC)

User Story

La voiture adapte selon la vitesse l'interdistance avec la voiture devant elle. Car plus la vitesse est grande, plus l'interdistance de sécurité est grande afin d'avoir le temps de freiner en cas de problème.

Description fonctionnelle

- Calcul périodique (à chaque nouveau message de vitesse) de l'interdistance à respecter. Il évolue selon la vitesse.
- Si la voiture est trop proche de celle de devant (fixer une distance et marge), freiner/décélérer jusqu'à atteindre la bonne distance. Prévenir les voitures Vn-1,k.
- Si la voiture est trop loin de celle de devant (fixer une distance limite et marge), accélérer jusqu'à atteindre la bonne distance. Prévenir les voitures Vn-1,k.
- Si la voiture est à bonne distance, continuer.

6.2.2.4.2 Vitesse

User Story

Adapter sa vitesse en fonction du message reçu périodiquement de la voiture de tête.

Description fonctionnelle

- Réception du message de vitesse
- Comparaison avec la vitesse actuelle
- Fonction respect de l'interdistance (cf plus haut)
- Alignement de la vitesse avec le véhicule de tête :
    - Si la vitesse du véhicule est supérieure à celle reçue, décélérer jusqu'à atteindre la bonne vitesse.
    - Si la vitesse du véhicule est inférieure à celle reçue, accélérer jusqu'à atteindre la bonne vitesse.
    - Si la vitesse est bonne, continuer.

6.2.2.4.3 Suivi de trajectoire

User Story

A chaque fois que la voiture de tête change de trajectoire (selon certains critères), elle prévient les voitures suiveuses grâce à un message leur indiquant la nature du changement de trajectoire et les indications à suivre.

Description fonctionnelle

- Un message est envoyé selon les cas suivants :
    - un changement de position de plus de 2m
    - un changement de direction de plus de 10°

6.3 Les "messages ECE"

Pour permettre au contrôleur et aux véhicules de communiquer, nous avons créé un nouveau type de message que nous appelons **"messages ECE"** basés sur la structure des messages CAM, DENM et SPAT utilisés dans la technologie **V2X** c'est-à-dire **sous forme de containers**.

Le container **PDU Header** est commun à tous les messages ECE et permet de connaître la version du protocole, le type de message qui est envoyé ainsi que l'ID de l'émetteur. Le **basic container** est aussi commun à tous les messages ECE et permet de connaître l'expéditeur, le destinataire et la phase de platooning. Le reste du message ECE dépend du cas dans lequel nous nous trouvons au moment de l'envoi du message.

Glossaire	
C	Contrôleur
Vi	Véhicule d'insertion
Vd	Véhicule de désinsertion
Vt	Véhicule de tête
Vk	Véhicules suiveurs
HF	High frequency
LF	Low Frequency

Message ECE / Asynchrone				
Container	**Sens du flux**	**ECE message**	**Taille (octets)**	**Priorité**
ITS PDU Header	Commun à tous les échanges	Version protocole	1 octet (u)	/
		Type de message	1 octet (u)	
		ID station	4 octets (u)	
Basique	Commun à tous les échanges	ID (émetteur)	1 octet (u)	/
		ID (destinataire)	1 octet (u)	
		Phase de platooning	1 octet (u)	
Initialisation	Vk to C	Destination	20 octets	LF
		Position (coordonnées)	20 octets	
		Type de station	1 octet (u)	
Insertion	C to Vi & Vk	ID platoon	1 octet (u)	LF
		Nombre de vehicules	1 octet (u)	
		Destination	20 octets	
		Interdistance	1 octet (u)	
		Vitesse	1 octet (u)	
		Tableau de véhicules (ID et rang)	8 octets (u)	
Désinsertion	Vd to C	Demande de sortie	1 octet	HF
	C to Vd	Permission de sortie	1 octet	HF
Feux	C to Vt	Permission de passer	1 octet	HF

Les variables des "messages ECE" ne peuvent pas avoir une taille inférieure à **un octet** car c'est le fonctionnement d'un ordinateur. Un octet permet de coder **256 valeurs**. Ci-dessous les **justifications** de la taille de chaque variable:

- ITS PDU Header : il est utilisé dans les messages CAM, DENM et SPAT de la **technologie V2X**.
- ID : la simulation contient 4 véhicules et 1 contrôleur, soit **5 valeurs** donc 1 octet est suffisant pour les représenter tous (valeurs = [0, ..., 4]).
- Phase de platooning : il existe **4 phases de platooning**, l'initialisation, l'insertion, la désinsertion et les feux donc 1 octet est suffisant pour les représenter toutes (valeurs = [0, ..., 3]).
- Destination/Position : les coordonnées d'un endroit sont constituées d'une **latitude** (64 bits), d'une **longitude** (64 bits) et d'une **altitude** (32 bits). La somme de ces tailles nous donne 20 octets.
- Type de station : inspiré des messages de la **technologie V2X**. Le type que nous utilisons est "Passenger Car" pour les véhicules.
- ID platoon : il n'y a qu'**un seul platoon** nécessaire pour notre solution donc un octet est suffisant.

- Nombre de véhicules : il y a **4 véhicules maximum** dans la simulation donc 1 octet est suffisant (valeurs = [0, ..., 3]).
- Interdistance : dans la réalité, il faut 7m pour une vitesse de 100 km/h pour des véhicules sans convois, ou 4m pour pour une vitesse de 50 km/h. En convoi, les véhicules peuvent être plus proches. La valeur de l'**interdistance** étant forcément **inférieure à 4m** en ville (limite de vitesse à 50 km/h), 1 octet est suffisant pour la coder.
- Vitesse : la vitesse en ville est limitée à **50 km/h**, 1 octet est donc suffisant.
- Tableau de véhicules : ce tableau à **deux dimensions** a une taille maximale de **4*2 cases** puisqu'un convoi contient 4 véhicules au maximum. La première dimension est un ID (codé sur 1 octet) et la deuxième un rang (codé sur un octet). Il faut donc (1 + 1)*4 = 8 octets.
- Demande/Permission : ces variables sont des **booléens**, ces derniers sont codés sur 1 octet.

6.4 Implémentation des messages dans ROS

6.4.1 Les messages customisés

Il est possible de créer des **messages ROS customisés** grâce aux fichiers ".msg". Ils sont constitués d'une ou plusieurs variables de n'importe quel type souhaité et sont faciles à implémenter. Voici un exemple de message:

```
int ID
bool permission
```

Les messages ROS fonctionnent sous forme de **containers**, c'est à dire qu'il est possible de déclarer une variable d'un autre type de message customisé. Cette méthode a donc été **idéale** pour implémenter les messages ECE et ETSI. Par exemple, nous pouvons créer le message customisé appelé "vehicle.msg" constitué d'un identifiant et d'un rang comme suit:

```
int ID
int Rank
```

Il est alors possible de créer un tableau alloué de véhicules du type "vehicle.msg" dans un message customisé appelé "platoon.msg" comme suit:

```
int ID
int Phase
Vehicle vehicles[]
```

On obtient alors la structure de message suivante et il est possible de **rajouter** des **niveaux** :

Platoon message			
int ID	int Phase	Vehicles vehicles	
		ID	Rank
		...	...

6.4.2 Les messages ETSI

Afin d'implémenter les messages CAM, DENM et SPAT, nous étions dans l'obligation de les recréer dans l'environnement ROS. Pour cela, nous avons récupéré des messages CAM et DENM sous forme de fichiers ".msg" ROS depuis un **répertoire Github open source [8]**. Pour les SPAT, nous les avons simplifiés et créés nous-même grâce aux **messages customisés**.

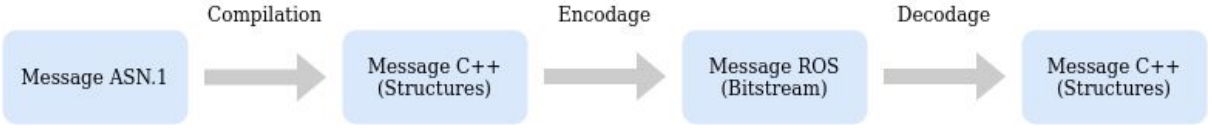
Ensuite, nous remplissons les containers des messages ETSI avec les données que nous **récupérons** des **robots**, par exemple leur position sur la carte. Nous avons implémenté les messages DENM mais ne les utilisons pas.

6.4.3 Les messages ECE et l'ASN.1

Afin d'implémenter nos messages ECE, nous devions utiliser le standard international appelé ASN.1, qui ressemble fortement au format XML. Un compilateur open source appelé **asn1c [9]** nous permettait d'obtenir des fichiers sources C compatibles en C++ à partir d'un fichier .ASN1.

Nous avons donc commencé par définir notre message ECE en ASN.1. Puis, grâce à **asn1c**, nous obtenions la structure de nos messages ECE sous forme de fichiers C. Ces fichiers étaient alors encodés pour être utilisés sous forme de **bitstream** (suite de bits) sous ROS.

En utilisant cette méthode, nous voulions pouvoir échanger nos messages ECE en les **encodant** et en les **décodant** entre chaque structure de code C++ que nous disposions.



Cependant, l'intégration avec le code ROS déjà implémenté pour effectuer les tests d'envoi et de réception de messages sous ROS n'a pas fonctionné.

Comme nous n'y arrivions pas, nous avons dû trouver une solution afin de ne pas perdre trop de temps sur l'implémentation des messages. Nous avons décidé d'utiliser les **messages ROS**, sous forme de fichiers ".msg", tout comme pour les messages CAM, DENM et SPAT. Cela nous a permis de pouvoir utiliser les messages ECE plus facilement et ainsi de généraliser la gestion de nos messages dans notre code C++. Cependant, le format de nos messages n'a pas changé, la **structure réalisée** avec le standard **ASN.1** a été **maintenue** pour l'implémentation des messages ROS ECE.

6.5 Implémentation des différentes entités

Nous avons défini **différents packages ROS** nous permettant de créer nos **entités**. Ces entités interagissent pour la plupart entre elles. Il existe donc six entités différentes pour gérer la **communication** de notre système. Les packages sont nommés ici comme ils le sont dans le code.

6.5.1 Controller

Le **contrôleur** est l'élément **le plus important** parmi toutes les entités. Il est **omniscient** et se charge de:

- Répertorier les véhicules : les véhicules envoient un message afin que le contrôleur puisse les **recenser**.

- Former les convois : une fois que les véhicules sont répertoriés, le contrôleur peut **comparer** leurs **destinations** afin de former des convois. S'il estime que les destinations des véhicules sont **assez proches**, alors il décide d'**ajouter** ces véhicules dans un **convoi**.

- Communiquer avec le feu tricolore : le contrôleur reçoit **périodiquement** des messages **SPAT** du **feu tricolore** afin d'être informé sur son **état** actuel.

- Donner la permission au convoi de passer le feu : lorsque le **véhicule de tête** du convoi arrive à **proximité** du **feu tricolore**, le contrôleur lui envoie un message pour indiquer si le convoi a le temps de **passer le feu ou non**. En fonction de ce message, le convoi **peut continuer** ou **doit s'arrêter** afin de pouvoir repartir.

- Gérer l'insertion : comme mentionné précédemment, après avoir décidé d'**ajouter** des véhicules dans un convoi en **comparant** leurs **destinations**, le contrôleur envoie un message ECE d'insertion avec les **informations du convoi**. Les véhicules appartenant au convoi reçoivent donc ces informations par la suite.

- Désinsertion : lorsqu'un véhicule souhaite se désinsérer, il envoie une **demande de désinsertion** au contrôleur au travers d'un message ECE. Le contrôleur lui envoie alors à son tour un accusé de réception pour lui indiquer qu'il a **bien pris connaissance** de sa demande. Lorsque le véhicule est correctement désinséré, il envoie à nouveau un message ECE de **confirmation de désinsertion** afin que le contrôleur ne le comptabilise plus dans le convoi de véhicules. Le contrôleur envoie à nouveau un **accusé de réception** au véhicule. Tous ces échanges permettent de mettre au mieux en place la désinsertion de véhicules du convoi.

6.5.2 Vehicles

Ce package nous permet d'implémenter et de gérer **chaque véhicule indépendamment**, notamment les quatre véhicules du **convoi**. Ces quatres véhicules possèdent tous des **destinations proches** les unes des autres afin de faciliter la simulation et d'ainsi assurer le bon fonctionnement du scénario défini plus tard.

- Réception des messages venant du contrôleur : une fonction callback est utilisée pour effectuer certaines actions à la **réception** d'un message **ECE** venant du contrôleur. Ainsi, dès qu'un message ECE du contrôleur est reçu par le véhicule, cette fonction est appelée et, selon la phase concernée, le véhicule va pouvoir **réagir** selon les informations reçues dans ce message.

- Réception de messages venant des autres véhicules : une autre fonction callback permet de prendre en compte les informations données dans les messages **ECE** et les messages **CAM** des **autres véhicules**.

- Initialisation : les véhicules envoient tous au début un message ECE en **phase d'initialisation** au contrôleur dans le but d'être **connus** par ce dernier. Afin de s'assurer de la bonne réception de ce message, les véhicules attendent que le **contrôleur** soit prêt à recevoir leurs messages.

- Insertion : les véhicules reçoivent un message ECE du contrôleur en **phase d'insertion**. Ils disposent alors des **informations du convoi**, comme l'ID du convoi, le nombre de véhicules appartenant à ce convoi ainsi que l'ID de la voiture de tête.

- Informations de position au contrôleur : les véhicules envoient **périodiquement** des messages **CAM** au contrôleur pour lui indiquer sa position. Afin de s'assurer que le contrôleur reçoit bien les messages CAM envoyés, les véhicules attendent que le contrôleur soit prêt à recevoir leurs messages.

- Suivi de trajectoire : les véhicules du **convoi** envoient **périodiquement** des messages **CAM** au véhicule se trouvant **derrière eux**, à condition qu'un véhicule se trouve effectivement derrière eux et que celui-ci appartienne bien au convoi.

- Désinsertion : lorsqu'un véhicule est **proche** de sa **zone de désinsertion**, il envoie un message ECE pour indiquer au contrôleur qu'il souhaiterait **sortir** du convoi. Le convoi lui envoie une **confirmation** par la suite pour lui indiquer qu'il autorise la sortie. Une fois que le véhicule est bien désinséré, il envoie une **confirmation de désinsertion** au contrôleur à travers un message ECE. Le contrôleur lui indique alors en retour qu'il a bien pris en compte son **départ** et qu'il ne fait plus partie du convoi.
  Dans notre scénario, c'est la voiture 4 qui souhaitera se désinsérer.

- Prise en compte du feu tricolore : lorsque le véhicule de tête est **à proximité** du **feu** tricolore, le contrôleur lui envoie un message ECE. Les véhicules **s'arrêtent** alors si le contrôleur le décide ou **poursuivent** leur chemin si le contrôleur estime que le convoi a le temps de passer le feu.

6.5.3 Light

Le **feu tricolore** est implémenté dans ce package. Il possède **2 états** : 0 pour indiquer que le feu est **orange et rouge** et 1 **vert** pour indiquer que le feu est vert. L'**état** du feu **change** au fil du temps afin de simuler au mieux un feu tricolore en milieu urbain. Le feu envoie périodiquement des messages **SPAT** au contrôleur pour lui indiquer son état.

6.5.4 ETSI_msgs

Nous avons implémenté les messages **CAM**, **DENM** et **SPAT** définis par l'ETSI en tant que messages ROS. Pour rappel, les messages que nous utilisons sont définis comme suit :

- CAM : principalement pour la **position** et le **suivi de trajectoire**.

- DENM : pour la **détection d'obstacles**. Ces messages sont implémentés sous ROS mais non utilisés pour le moment.

- SPAT : pour la **communication** du **feu** tricolore avec le contrôleur.

6.5.5 ECE_msgs

Dans ce package, les **messages ECE** sont implémentés afin d'être utilisés par le contrôleur et les véhicules. Ils sont donc utilisables par les autres packages en tant que **messages ROS**, tout comme les messages CAM, DENM et SPAT. C'est donc avec les messages ROS de ce package que sont **échangés** les messages ECE entre les **véhicules** et le **contrôleur**.

6.5.6 Simu_msgs

Ce package est assez particulier puisqu'il permet d'**envoyer** des messages à la **simulation** afin de faciliter le traitement des messages de la **partie robotique**. Les messages **CAM** et **ECE** sont ainsi **dupliqués** pour être envoyés en simulation et ainsi permettre à la partie robotique de mener au mieux l'asservissement des robots. Par exemple, lorsque les véhicules envoient des messages CAM aux véhicules suiveurs, ces messages sont aussi envoyés avec des "**simu_msgs**". Ainsi, la partie robotique peut exploiter ces messages le plus facilement possible. De plus, des messages ECE sont eux aussi dupliqués afin de faciliter l'arrêt des véhicules à proximité d'un feu mais aussi pour la désinsertion.

6.6 Communication des différentes entités

Se référer à la partie **8.1.1 "L'outil ROS et GAZEBO"** pour mieux comprendre l'implémentation des topics ROS présentés dans cette partie.

6.6.1 Communication connectivité

Chaque nœud ROS, c'est-à-dire le contrôleur, les véhicules et le feu communiquent grâce à des **topics ROS**. Pour ce faire, nous implémentons plusieurs "publishers" et "subscribers" dans chaque nœud, chacun abonné à un **topic différent**. A chaque fois qu'un publisher publie sur un topic, le subscriber reçoit le message et le stocke dans une **file d'attente**.

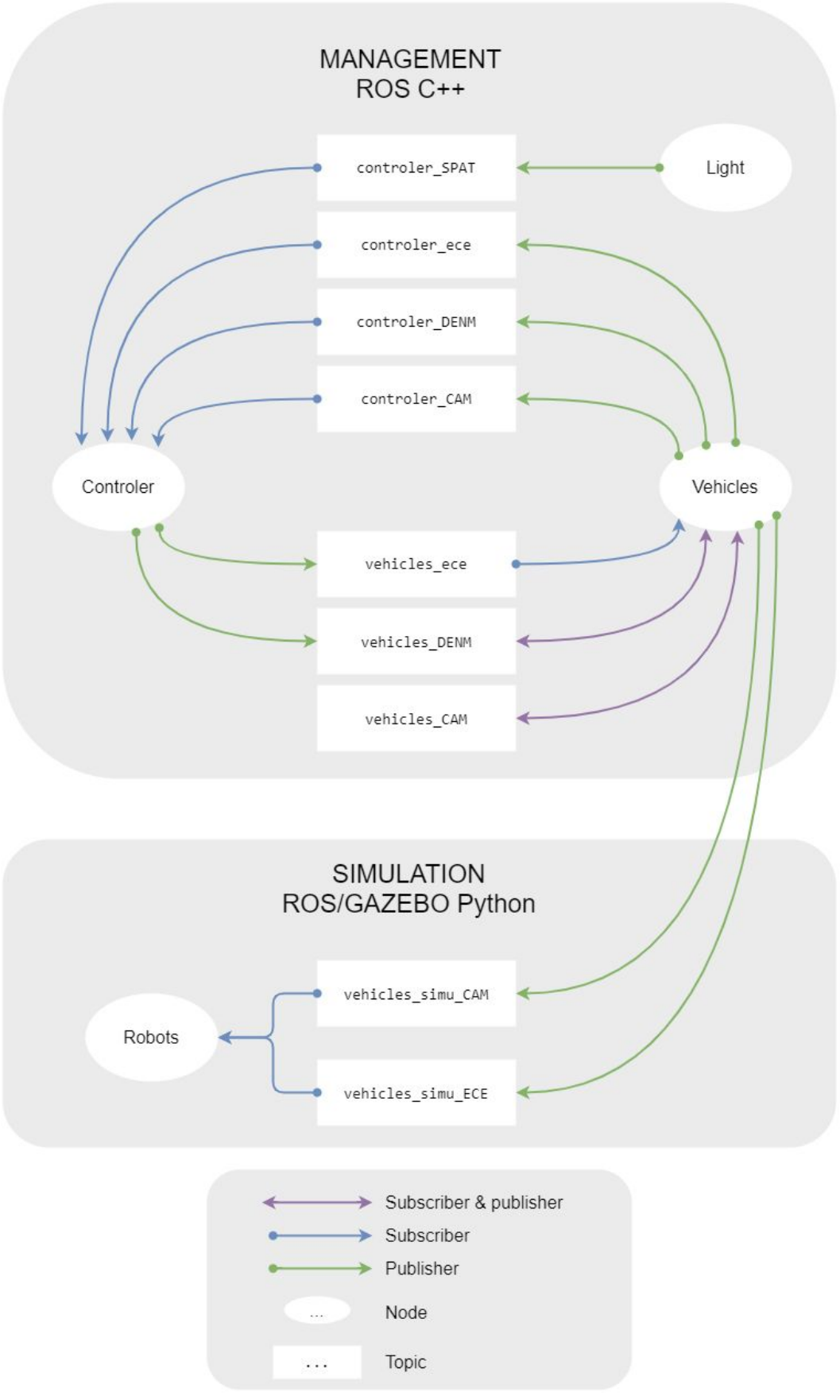
Nous avons réduit le plus possible le nombre de topics afin de simplifier l'implémentation, c'est pourquoi lorsque plusieurs entités sont abonnées au même topic et donc reçoivent toutes le même message, nous envoyons l'**ID du destinataire** pour traiter les messages destinés seulement au nœud concerné.

- Pour communiquer avec le contrôleur, les véhicules et le feu de signalisation utilisent les topics nommés "controller_[...]".
- Pour communiquer avec les véhicules, le contrôleur utilise les topics nommés "vehicles_[...]".
- Pour communiquer entre eux, les véhicules utilisent les topics "vehicles_[...]", c'est pourquoi lorsqu'un véhicule publie un message il le reçoit aussi, mais grâce à l'ID du destinataire, comme expliqué plus haut, cela ne pose pas de problème.

6.6.2 Interface connectivité/robotique

La partie connectivité a été codée en C++ alors que la partie robotique en Python. Afin de **lier ces deux parties**, nous avons utilisé la méthode la plus simple et faisable dans ROS qui consiste à faire communiquer deux nœuds codés dans des langages différents grâce à des topics. En effet, les topics utilisent les messages ROS et de ce fait peuvent être utilisés dans **plusieurs langages**. Les topics utilisés pour faire ce lien contiennent le mot "simu" dans leur nom pour "simulation". Nous avons créé de nouveaux types de messages simplifiés, basés sur les messages ECE et CAM, contenant uniquement les informations **nécessaires** à la partie robotique.

6.6.3 Schéma global des topics



7. Méthodologie de la partie robotique

Nous nous sommes appuyés sur le logiciel de simulation **GAZEBO** [10] exécuté à partir de **ROS** [11] (Robot Operating System) pour reproduire un scénario de **convoi de véhicules autonomes** en milieu urbain.

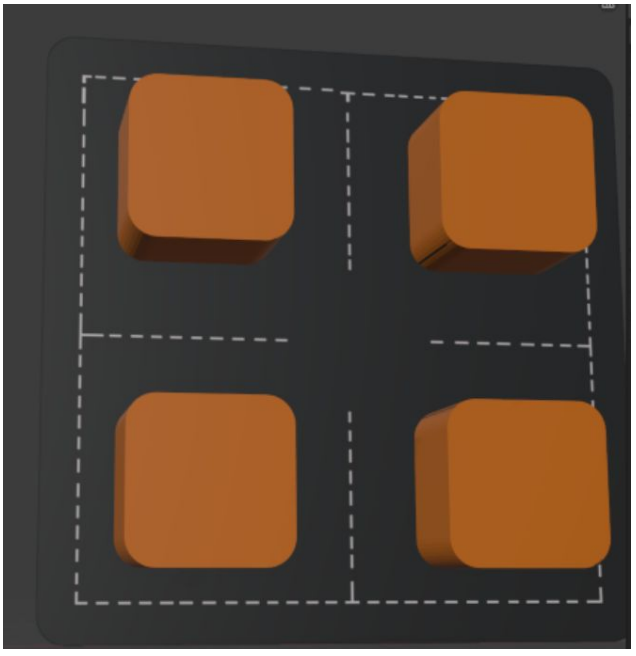
Pour cette simulation, nous avons voulu réaliser un scénario composé de :

- Une carte avec des immeubles.
- Un véhicule de tête (robot) qui suit un itinéraire fixe.
- Deux véhicules (robots) qui suivent le véhicule de tête grâce aux **messages CAM**.
- Un véhicule (robot) immobile qui va s'insérer en bout du convoi et qui se désinsère lors de son arrivée vers la destination.
- Un feu tricolore symbolisé par un poteau rouge

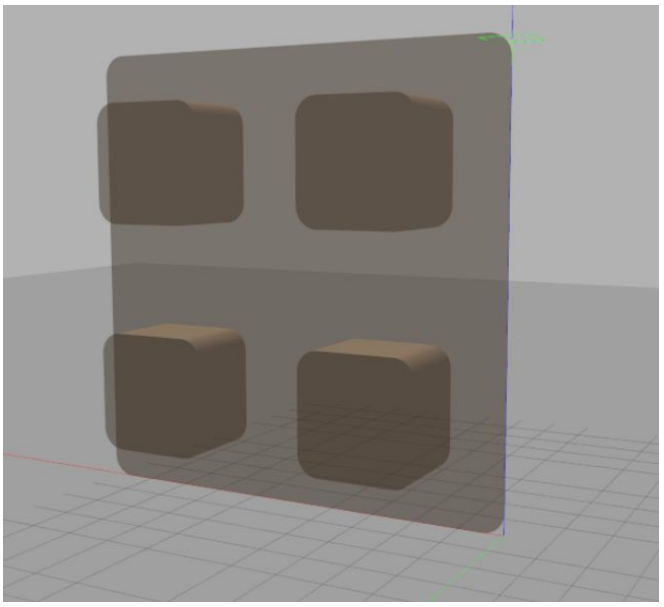
7.1 Création de la map

Avant de commencer à faire de la robotique et à manipuler les robots à proprement parler, nous avons dû concevoir une carte, un environnement dans lequel nos robots seront amenés à évoluer. Pour cela, nous sommes partis sur quatre obstacles statiques représentant des bâtiments, puis une route parcourant le tour des quatres bâtiments en formant un carrefour au centre. Ainsi, nous avions tous les éléments pour mettre en scène les différentes situations exploitant **les messages CAM, SPAT et ECE**.

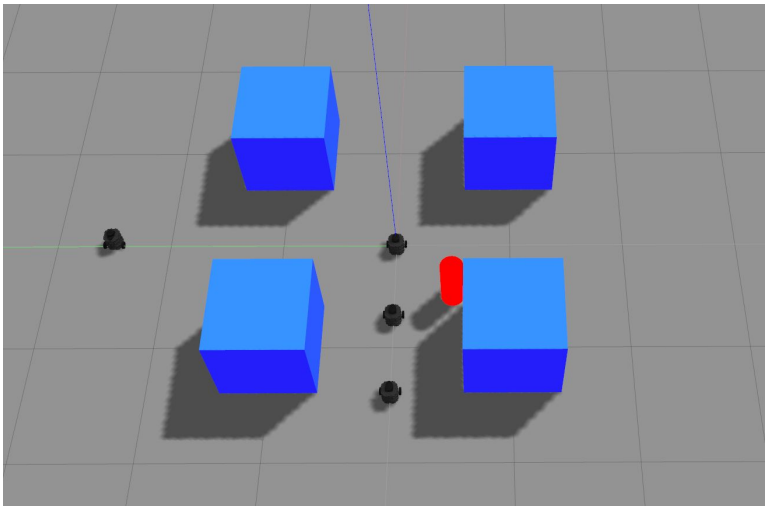
Pour modéliser ce monde, nous nous sommes servis du logiciel **Solidworks** [12] afin de créer les volumes et de produire un fichier ".stl" compatible avec le simulateur **Gazebo**. Cependant, lors de l'importation du modèle ".stl" dans **Gazebo**, nous nous sommes rendus compte que le simulateur n'affichait pas les couleurs et textures de la carte (couleur des blocs et tracés au sol des lignes). Après investigation, nous avons décidé d'utiliser le second format de fichier exploitable par **Gazebo** : ".dae". Ce format de fichier peut être obtenu à partir d'un logiciel de modélisation comme **Blender** [13]. Nous avons donc utilisé **Blender** pour créer à nouveau une carte dont voici le résultat :



Malheureusement nous avons rencontré encore une fois le même problème : les couleurs et textures (marquage au sol) ne s'affichent pas comme le montre cette image prise dans **Gazebo** :



Le modèle est transparent et il manque les détails peints sur les surfaces du modèle. Nous avons donc opté pour une méthode plus simpliste en nous servant de l'éditeur de carte inclus dans **Gazebo** dont voici une image :



Comme vous pouvez le constater, cette version ne présente pas de marquage au sol. Mais notre PFE étant axé sur les **communications entre véhicules et contrôleur**, il n'avait pas particulièrement besoin de ce niveau de détail sur la carte. Cependant, notre projet étant repris par une équipe PPE qui aura besoin des tracés de lignes au sol, nous avons donc initialement passé beaucoup de temps à essayer de les faire apparaître, malheureusement sans succès.

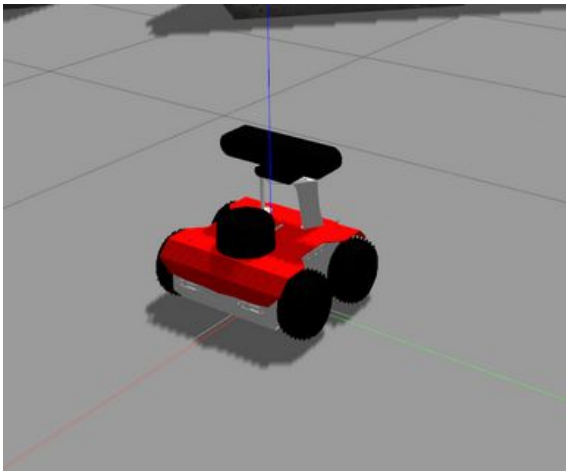
L'image ci-dessus, montre également la disposition des robots au lancement de la simulation : trois robots en ligne se suivant les uns les autres, et un robot seul attendant le signal d'insertion dans le convoi existant. Nous y retrouvons les quatre cubes symbolisant les bâtiments et un cylindre rouge représentant un feu de circulation.

7.2 Définition du path et asservissement du robot de tête

Notre espace de simulation étant maintenant défini, nous implémentons notre robot sur la carte afin de commencer à comprendre comment configurer le robot.

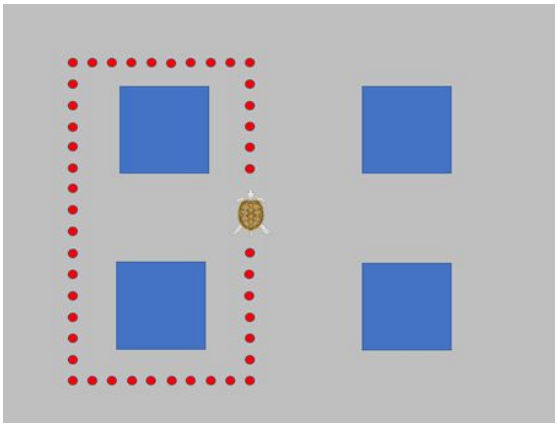
Nous avions au départ prévu de travailler avec un robot **ROSbot** de chez **Husarion**. Ce choix avait été réalisé dû au fait que celui-ci possédait de nombreux capteurs que le PPE allait pouvoir utiliser notamment pour la détection d'obstacles. Cependant, suite à des problèmes pour contrôler la vitesse de manière indépendante sur plusieurs robots de ce type dans le simulateur, nous avons décidé de nous tourner vers des robots **turtlebot3** de chez **Robotis** format “burger”. Ces robots sont plus répandus dans les projets indépendants et open source et nous avions plus de ressources et d’informations à portée.

Robot Robotis Turtlebot3 “burger”



Robot Husarion ROSbot

Nous avons donc commencé par élaborer le chemin de base que doit parcourir le robot de tête. Pour ce faire, nous avons établi une liste qui contient des tableaux de float représentant les coordonnées en x ou en y en fonction de la direction dans laquelle le robot doit se déplacer.



Nous avons défini la trajectoire comme dans le schéma ci-dessus. Le robot étant contrôlable par le biais de sa **vitesse linéaire et angulaire** nous avons mis en place deux **correcteurs proportionnels dérivés (PID : Proportionnel Intégral Dérivé dans sa forme générale)** dont voici la formule :

$$commande = Gain\ proportionnel\ .\ erreur + Gain\ dérivé\ .\ (erreur - erreur\_au\_pas\_précédent)$$

$$erreur = consigne - mesure$$

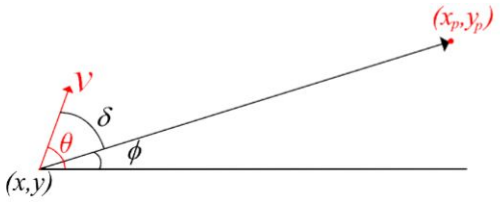
Nous avons choisi ce **correcteur** car nous l'avions déjà étudié en cours d'automatique cette année et parce qu'il convient très bien à ce type d'applications (correcteur très largement utilisé dans l'industrie). Nous avons commencé avec un **correcteur proportionnel**, mais voulant améliorer la précision et réduire les oscillations sur les trajectoires, nous avons introduit la **composante dérivée à notre correcteur**.

Passons désormais à la modélisation de notre système : le robot est défini selon le système d'équations suivant (système discret échantillonné):

$$\begin{cases} x_{k+1} = x_k + \Delta t \cdot v_k \cdot \cos(\theta_k) \\ y_{k+1} = y_k + \Delta t \cdot v_k \cdot \sin(\theta_k) \\ \theta_{k+1} = \theta_k + \Delta t \cdot \omega_k \end{cases}$$

- x_k et y_k les composantes en x (axe des abscisses) et y (axe des ordonnées) de la position du robot au pas courant (k).
- x_{k+1} et y_{k+1} : idem mais au pas de temps suivant ($k+1$).
- Δt : la durée séparant deux pas (la période d'échantillonnage). Dans notre algorithme équivaut à 0.2s.
- v_k : la vitesse linéaire du robot.
- $\theta_k et\ \theta_{k+1}$: respectivement les orientations (angles) du robot par rapport à l'horizontale (axe x).
- ω_k : la vitesse angulaire du robot.

Voici un schéma expliquant les différents points et angles et leurs notations pour la suite (le robot se trouvant au point (x,y)):



- (x,y) : point représentant la position actuelle du robot dans une base cartésienne.
- (x_p,y_p) : point représentant la position souhaitée du robot (la consigne en position) un des points du tableau de points représentant la trajectoire.
- $\vec{v}$: le vecteur vitesse linéaire du robot.
- θ : l'orientation du robot par rapport à l'horizontale (cap actuel du robot).
- $(x,y),(\vec{x_p,y_p})$: vecteur reliant le point actuel et le point de destination dont la norme est l'erreur de position (noté d).
- ϕ : l'angle que forme le vecteur $(x,y),(\vec{x_p,y_p})$ avec l'horizontale (cap souhaité du robot, consigne en angle).
- δ : l'erreur de cap ($\theta - \phi$), différence entre le cap souhaité et le cap actuel.

Comme expliqué plus tôt nous avons mis en place deux **correcteurs PD** :

- un pour **l'asservissement en vitesse linéaire du robot** :

$$v_{robot} = -Kp_v \,.\, \delta + Kd_v \,.\, (d_k - d_{k-1})$$

Autrement dit :

$$vitesse\ linéaire = Gain\ proportionnel\ .\ erreur + Gain\ dérivé\ .\ (erreur - erreur\_au\_pas\_précédent)$$

$$erreur(ici\ c'est\ une\ distance) = position\_a\_rejoindre - position\_actuelle$$

- un autre pour **l'asservissement en vitesse angulaire du robot** (son cap, là où il "regarde") :

$$\omega_{robot} = -Kp_\omega \,.\, \delta + Kd_\omega \,.\, (\delta_k - \delta_{k-1})$$

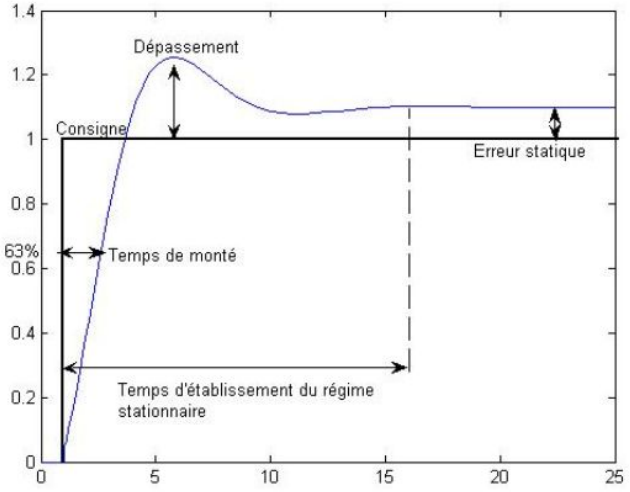
Autrement dit :

$$vitesse\ angulaire = Gain\ proportionnel * erreur + Gain\ dérivé * (erreur - erreur\_au\_pas\_précédent)$$

$$erreur(ici\ c'est\ un\ angle) = cap\ \_a\_\ atteindre - cap\_actuel$$

Lorsque le robot atteint son point de destination, nous passons au suivant dans le tableau de points, ainsi notre robot se déplace de "points en points".

Les **gains** (constantes) $Kp_v, Kd_v, Kp_\omega\ et\ Kd_\omega$ sont réglés à la main en expérimentant avec la simulation. Nous savons que Kp rend le système plus réactif (plus Kp est grand plus le système aura un temps de montée court). Cependant si l'on augmente trop Kp, la correction dépassera la consigne rendant le système peu fiable. Pour ce qui est de Kd, nous savons que l'augmenter réduit les oscillations autour de la consigne, néanmoins cela réduit également le temps de montée. Il s'agit donc de trouver un compromis satisfaisant pour obtenir un **asservissement** précis et fluide.



Pour mettre en place ce **correcteur** nous avons besoin d'un certain nombre de données fournies par le topic odométrie du robot. Il nous envoie à chaque période de temps Δt des informations sur la position et l'orientation actuelle du robot. Grâce à ces valeurs, nous pouvons donc calculer notre correction (v_{robot} et ω_{robot}) et les appliquer via le **topic *cmd_vel*** (command_velocity) pour contrôler le robot dans la simulation.

7.3 Ajout des robots composant le convoi

Lorsque nous avons réussi à implémenter l'itinéraire pour la voiture de tête, nous avons commencé à travailler sur des robots supplémentaires qui devaient la suivre à l'aide des **messages simu_cam**. Nous avons donc configuré nos fichiers de lancement de simulation afin de faire apparaître deux robots derrière le véhicule de tête. Ces véhicules reçoivent un **ID unique**, propre à chaque véhicule lors de la phase d'initialisation du platoon par le contrôleur. Ils nous permettent ainsi d'**envoyer différentes informations sur les topics de chaque véhicule indépendamment des autres**. Chaque turtlebot s'abonne donc au topic des CAM et communiquent par ce même topic leur position.

Avec ces données nous réalisons donc le même procédé que pour la trajectoire du véhicule de tête. A l'exception que chaque véhicule suiveur, possède **un tableau (liste) de points à rejoindre** s'actualisant avec la nouvelle position du véhicule précédent. Autrement dit, à la place d'avoir un tracé de points prédéfinis (véhicule de tête), les robots suiveurs construisent **leur trajectoire à suivre en fonction des positions du robot précédent**. Lorsque le robot arrive près de son point d'arrivée, il supprime celui-ci de la liste pour prendre la position suivante. Cette "construction" de trajectoire en temps réel n'est cependant pas un simple recopiage de trajectoire, il faut également veiller à ce que la trajectoire créée pour le robot suiveur, respecte **l'interdistance** voulue. Pour ce faire, nous avons élaboré une stratégie consistant à **déterminer un point respectant la trajectoire à l'aide de l'angle d'incidence**. En effet, en utilisant les coordonnés du point ainsi que celle du précédent, nous pouvons déterminer **l'angle d'incidence** au point, et ainsi nous pouvons déterminer une **position antérieure** au point qui respecte la trajectoire. Nous avons donc les formules suivantes pour calculer les objectifs d'arrivées des véhicules suiveurs.

Formule pour calculer l'angle d'incidence (le cap) du robot précédent :

$$\theta^{(N-1)} = arctan\left(\frac{y^{(N-1)}{}_k - y^{(N-1)}{}_{k-1}}{x^{(N-1)}{}_k - x^{(N-1)}{}_{k-1}}\right)$$

Avec :

- $\theta^{(N-1)}$: le cap du robot précédent
- $x\ et\ y$: les coordonnées des robots
- k : pas de temps
- N : rang du robot dans le convoi

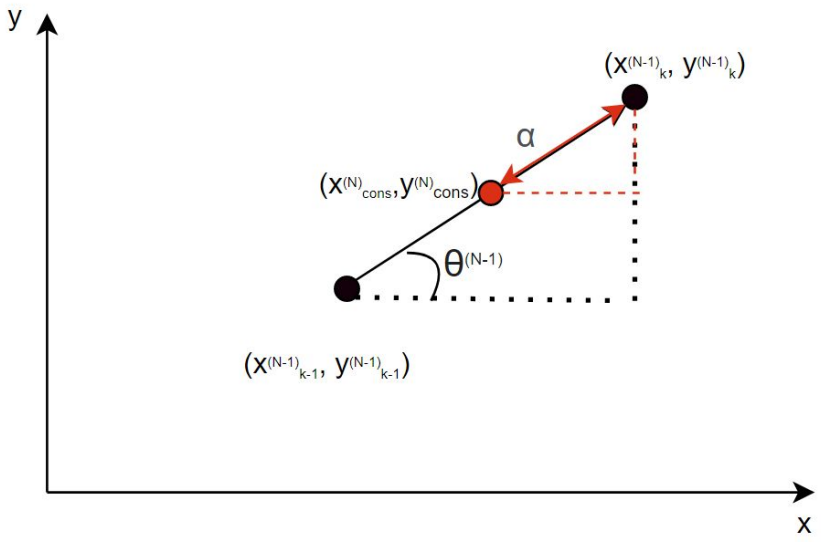
Système d'équation pour calculer la consigne du robot courant en fonction du cap du robot précédent :

$$\left\{\begin{array}{l} x_{cons}^{(N)} = x_k^{(N-1)} - \alpha \cdot cos(\theta^{(N-1)}) \\ y_{cons}^{(N)} = y_k^{(N-1)} - \alpha \cdot sin(\theta^{(N-1)}) \end{array}\right\}$$

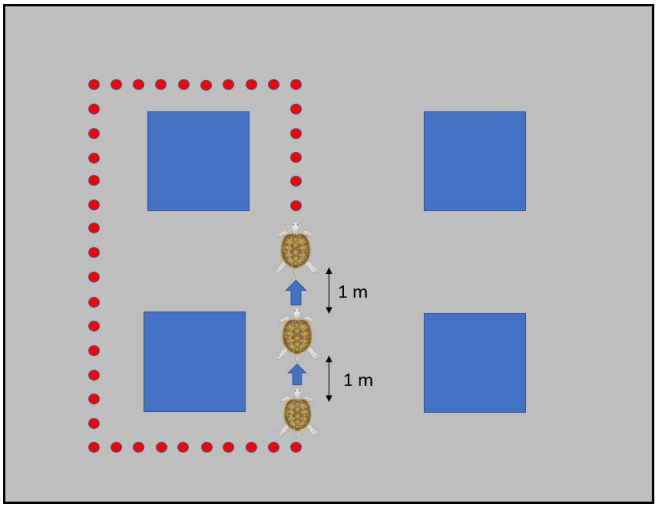
Avec :

- $x^{(N)}{}_{cons}$ et $y^{(N)}{}_{cons}$: la consigne (la destination) du robot actuel.
- $x\ et\ y$: les coordonnées des robots.
- $\theta^{(N-1)}$: l'angle d'incidence (cap) du robot précédent
- N : le rang du robot
- k : le pas de temps
- α : l'interdistance (constante = 1m dans notre simulation)

Voici un schéma expliquant nos notations ainsi que ces formules :



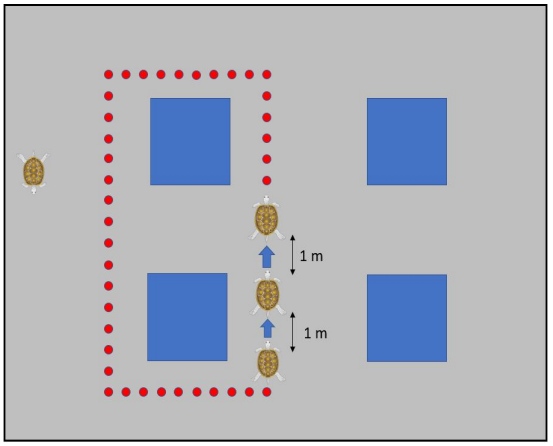
L'algorithme place donc un point respectant l'interdistance dans la liste des consignes (destinations) du robot suiveur. Nous avons ainsi un suivi de trajectoire.
Voici un schéma synthétique du résultat obtenu :



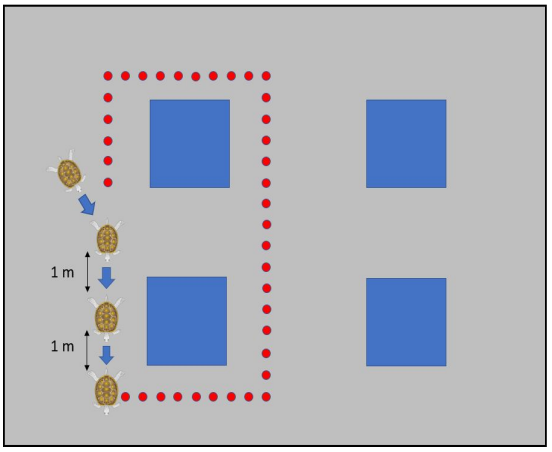
7.4 Ajout du robot destiné à l'insertion

Maintenant que **l'asservissement** est bien implémenté avec notre **correcteur**, nous nous intéressons à la partie de l'insertion. Nous rajoutons donc une **turtlebot** dans une autre partie de la carte mais qui se situe à proximité du passage du **convoi**.

Le véhicule immobile compare en permanence la distance qui existe entre lui et la dernière voiture du convoi. Dans le cas où celle-ci se trouve à proximité de la dernière voiture du platoon, celle-ci commence sa phase d'insertion et suit le convoi notamment grâce à l'intermédiaire des **messages simu_cam** et de **l'asservissement** du robot. Les schémas ci-dessous montrent le déroulement du scénario que nous avons voulu mettre en place pour tester notre insertion.



Situation de départ

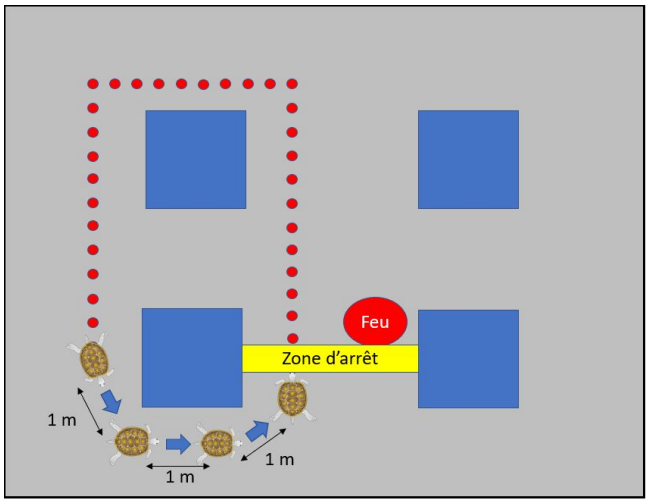


Situation d'insertion

7.5 Ajout du feu de circulation

Afin de donner une dimension plus réaliste à la simulation et respecter au mieux notre cahier des charges, nous avons décidé d'intégrer un feu de circulation dans notre simulation. Notre **turtlebot** étant dépourvu de caméra et la reconnaissance de l'environnement étant réalisé par le PPE, nous nous sommes détournés de la réalisation d'un véritable feu sous gazebo. Nous avons préféré à la place investir notre temps sur **comment nous allions recevoir les messages SPAT** et **interpréter ces messages**.

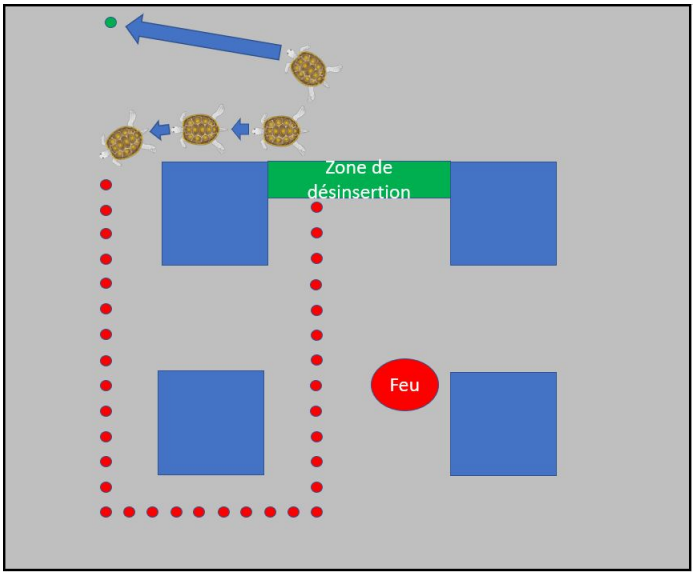
Nous avons donc placé un poteau rouge représentant la zone dans laquelle sera situé le feu de circulation. Les véhicules reçoivent en permanence l'information de l'état du feu. Dans le cas où l'état du feu est au rouge et que le véhicule de tête se trouve proche du feu, celui-ci publish sur son **topic cmd_vel** une vitesse linéaire et angulaire nulle. Les autres véhicules du convoi s'arrêtent également par l'intermédiaire des **messages simu_cam**. A l'inverse, lorsque l'état du feu est au vert, les véhicules poursuivent leur itinéraire prédéfini sans interruption.



Ce schéma représente donc la situation dans laquelle la **turtlebot de tête** approche de la zone du feu lorsque celui-ci est à l'état rouge.

7.6 Désinsertion d'un véhicule

En ce qui concerne la désinsertion, le contrôleur reçoit un message de **demande de désinsertion** de la part du véhicule qui souhaite se désinsérer lorsqu'il atteint une certaine zone de la map. Cette zone se trouve près de son **point d'arrivée** et lorsque le véhicule approche de cette zone, il reçoit **la permission de se désinsérer**. Le véhicule actualise alors son point d'arrivée vers sa nouvelle destination avant de s'arrêter sur celui-ci.



Le schéma ci-dessus permet ainsi de visualiser la situation de la désinsertion dans notre simulation.

8. Réalisation du prototype/expérimentations

8.1 Environnement de développement (Outils, langages, ...)

8.1.1 L'outil ROS et GAZEBO

Nous avons travaillé avec un **logiciel de simulation** appelé **GAZEBO [10]** et exécuté à partir de **ROS (Robot Operating System) [11]** afin de reproduire un scénario de convoi de véhicules autonomes en milieu urbain.

ROS constitue un ensemble d'outils **open source** facilitant le développement robotique. Il est actuellement développé par l'Open Robotics.

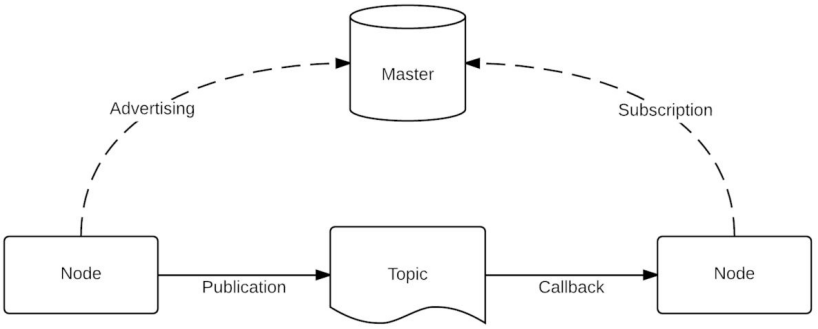
Il s'agit d'un middleware ie. une plateforme de développement logiciel, ou encore un méta-système d'exploitation qui peut fonctionner sur un ou plusieurs dispositifs à la foi et qui fournit plusieurs fonctionnalités: abstraction du matériel, mise à disposition de librairies de développement pour les fonctionnalités les plus couramment utilisées, transmission de messages entre les processus et gestion de briques (packages) technologiques.

Le *middleware* ROS met à disposition une **suite d'outils** comme des simulateurs (RViz ou Gazebo), un serveur de paramètres, un système d'enregistrement et de rejeu (*rosbag*) et certainement le point le plus important étant une architecture de communication inter-processus et inter-machine.

Les processus ROS sont appelés ***nodes*** (noeuds) et chaque node peut communiquer des messages à d'autres nodes via des ***topics*** (des tuyaux, canaux de communication). La connexion est gérée par un ***master*** (node maître) selon le processus suivant :

1. Un premier *node* avertit le *master* qu'il a une donnée à partager
2. Un deuxième *node* avertit le *master* qu'il souhaite avoir accès à une donnée
3. Une connexion entre les deux *nodes* est créée
4. Le premier *node* peut envoyer des données au second

Un *node* qui publie des données est appelé un ***publisher*** et un *node* qui souscrit à des données est appelé un ***subscriber***. Un *node* peut être à la fois *publisher* et *subscriber*. Les messages envoyés sur les topics sont pour la plupart standardisés ce qui rend le système extrêmement flexible et modulaire.



ROS permet une **communication inter-machine**, des *nodes* s'exécutant sur des machines distinctes, mais ayant connaissance du même *master* peuvent communiquer de manière transparente pour l'utilisateur. Idéal donc pour des **systèmes robotiques mobiles** dont les organes ne sont pas tous concentrés au même endroit.

Tous les membres n'ayant pas d'environnement Linux adapté sur leur PC, nous avons utilisé le site ***TheConstructSim*** [14] qui est un outil (en partie gratuit) permettant de choisir d'utiliser ROS de la version souhaitée et **en ligne**. L'environnement fourni est identique à celui de ROS et les projets peuvent être partagés entre les utilisateurs. Il permet aussi d'utiliser, toujours en ligne, les outils de simulation de ROS comme GAZEBO par exemple.

8.1.2 Langages

Pour la partie robotique nous avons décidé d'opter pour le langage **Python 2.7**. Ce choix a été fait pour plusieurs raisons. Tout d'abord, nous avons eu un enseignement en automatique lors du premier semestre d'ING5 où nous avons pu nous familiariser avec ROS, tous les travaux pratiques et les devoirs à réaliser étaient à rendre en Python. En effet, ROS s'interface particulièrement bien avec le langage via son module rospy. De plus, il existe une grande quantité de ressources et une communauté très active et présente sur internet. Ensuite, la partie robotique nécessitant l'usage d'outils mathématiques assez avancés, nous avons opté pour Python grâce à ses multiples modules facilitant entre autres : le calcul matriciel (module **numpy**) et les fonctions ou opérateurs mathématiques de base (module **math**).

Nous sommes conscients que le langage Python n'est pas un langage particulièrement efficace et rapide d'exécution. Pour nous, il était cependant préférable de développer un prototype fonctionnel rapidement avec un langage certes moins efficace mais plus facile à prendre en main.

Pour la **partie connectivité**, contrairement à la partie robotique, nous avons décidé d'implémenter notre code en **C++** puisqu'il est possible d'utiliser ROS avec du Python et du C++. De plus, nous avions la possibilité d'utiliser un compilateur open source pour implémenter nos messages ASN.1. Ce compilateur nous permettait d'obtenir des fichiers sources en C++.

Même si nous n'avons pas réussi à aboutir à des messages ECE en **ASN.1** fonctionnels avec ROS, nous avons tout de même pu les implémenter en C++, comme cela a été mentionné précédemment.
Le C++ étant un langage **orienté objet**, nous avons pu profiter de cet aspect technique pour programmer les différentes entités existantes dans notre simulation ROS.

8.2 Scénario de fonctionnement/montage,...

Pour faire fonctionner la simulation qui tournera sur Ubuntu 16.04 avec ROS Kinetic et le package turtlebot3 pour ROS Kinetic installable via apt : `sudo apt install ros-kinetic-turtlebot3-*`, nous avons besoin de créer un workspace dans lequel stocker les packages (briques logicielles ROS) et leurs fichiers. Ce workspace s'appelle `catkin_ws` dans notre cas.

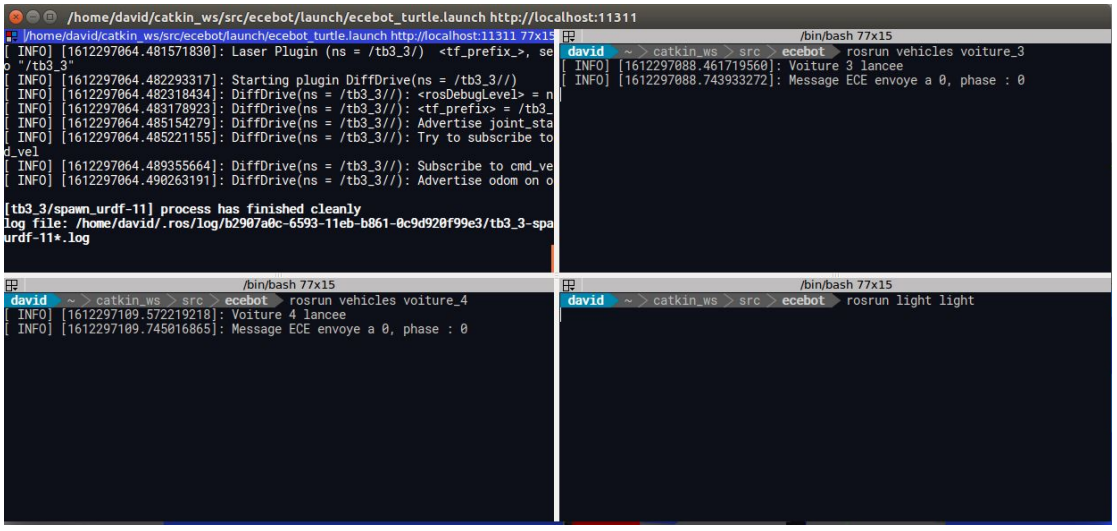
Après avoir initialisé ce workspace avec la commande `catkin_make` (catkin étant l'utilitaire de gestion de paquets ROS), nous pouvons récupérer le code source du projet depuis notre dépôt git et refaire un `catkin_make` pour compiler une nouvelle fois le workspace avec les nouveaux fichiers téléchargés depuis Github.

Nous avons ensuite besoin d'actualiser les variables d'environnement dans chaque terminal ouvert en tapant `source ~/catkin_ws/devel/setup.bash`. Ensuite, pour exécuter les nœuds python, nous avons besoin de modifier les autorisations d'exécution des scripts .py.

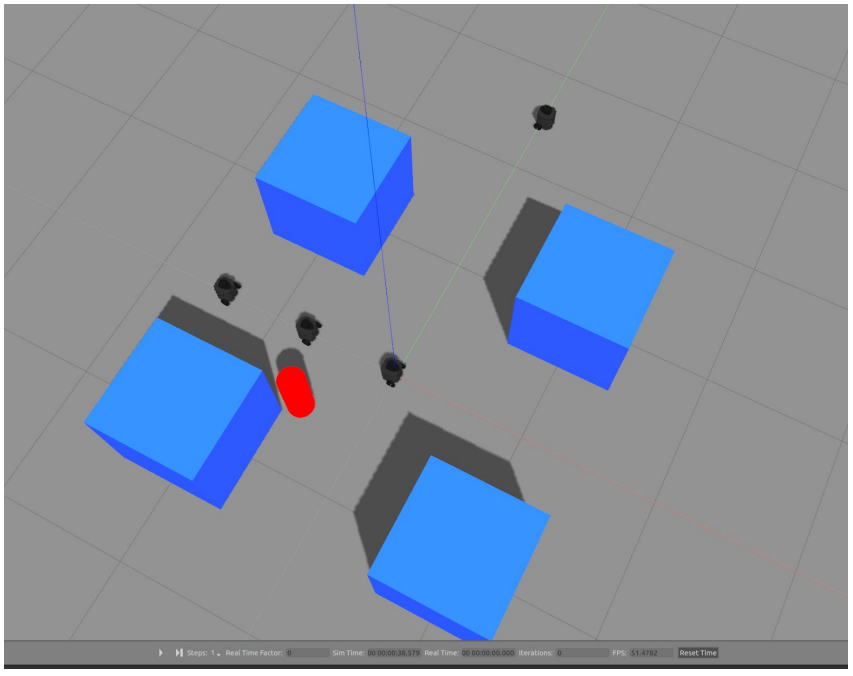
Pour cela il faut taper la commande :
`cd ~/catkin_ws/src/ecebot/src && chmod +x *.py`. Puis, il faut spécifier le modèle de turtlebot3 que nous voulons utiliser dans la simulation. Etant donné que nous utilisons le modèle *"burger"* il faut exporter une variable d'environnement qui va être lue par les fichiers de lancement de la simulation, pour cela il faut taper : `echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc` puis `source ~/.bashrc`

La simulation se lance à l'aide de 4 terminaux différents : dans le premier, taper la commande `roslaunch ecebot ecebot_turtle.launch`, cette commande va lancer la simulation (carte + véhicules) ainsi que les noeuds de communication des véhicules 1 et 2 ainsi que le noeud de communications du contrôleur. Dans le terminal 2, lancer le nœud du véhicule 3 : `rosrun vehicles voiture_3`. Idem pour le terminal 3, cette fois-ci pour le véhicule 4 : `rosrun vehicles voiture_4`. Dans le terminal 4, lancer le nœud de communications avec la commande : `rosrun light light`.

Les terminaux doivent se présenter de cette manière une fois les commandes tapées :

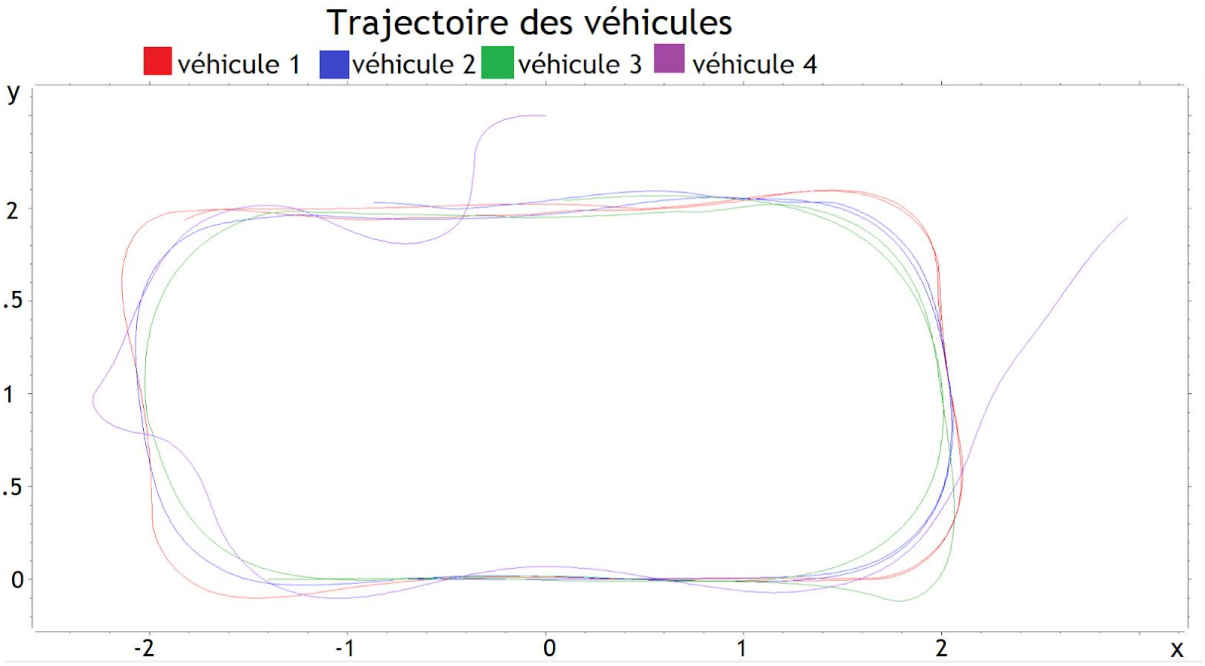


Et voici le résultat en simulation, pour lancer celle-ci, cliquez sur le bouton "play" dans la barre en bas de la fenêtre du simulateur :



9. Evaluation du prototype (tests) / Résultats

Afin de **mesurer l'efficacité de la solution** que nous avons développée, nous avons extrait des **données de notre simulation** afin de pouvoir les représenter sous forme de graphiques. De par les outils de simulation à notre disposition, il était **impossible** de créer une étude statistique sur le **temps de traitement des messages** dans notre système. De ce fait, la **partie résultat** traitera de la **partie robotique** de notre projet et de la capacité de nos véhicules à **respecter les consignes** obtenues de ces messages.

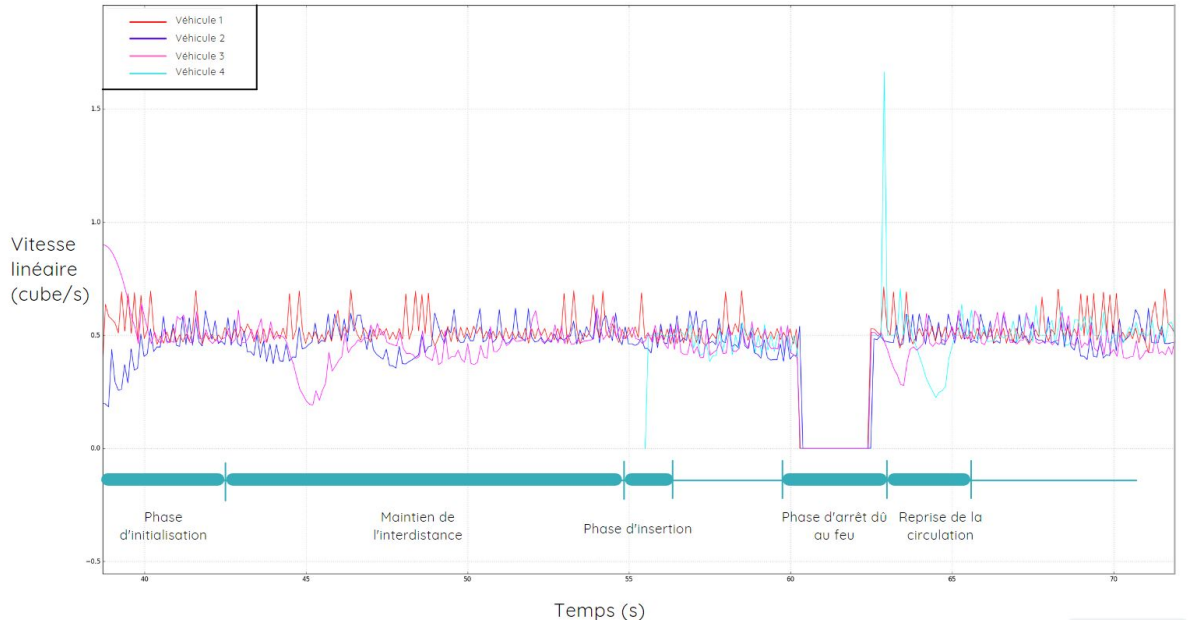


Le graphique ci-dessus représente la **trajectoire des véhicules** lors du premier tour. Il est à noter que le **véhicule 4** commence à **s'insérer** durant le tour et se **désinsère** plus tard. Plusieurs éléments peuvent être remarqués par rapport à ce graphique.

Tout d'abord, le **robot de tête** suit un **chemin de point prétracé**, il prend les virages généralement moins bien que les véhicules suiveurs. D'autre part, le véhicule 4 prend quelques instants pour stabiliser sa trajectoire pour suivre celle du véhicule 1. Néanmoins, ces **oscillations** restent dans un domaine **acceptable** (moins de 0.5 mètre d'écart avec la trajectoire suivie par le véhicule 1) et **disparaissent totalement** une fois le délai écoulé. Le **suivi de trajectoire** est donc une **réussite** néanmoins, il reste à **améliorer** la capacité d'un véhicule à **s'insérer dans le platoon**, ce qui pourrait probablement se faire en **changeant le contrôleur** des véhicules (pour rappel, actuellement un PID).

On peut néanmoins conclure par rapport à ce graphique des trajectoires que tout l'aspect suivi de trajectoire est complété, nous pouvons commencer à supposer que cela est aussi le cas de l'insertion et la désinsertion.

Ce second graphique représente la vitesse des véhicules au cours de la simulation :



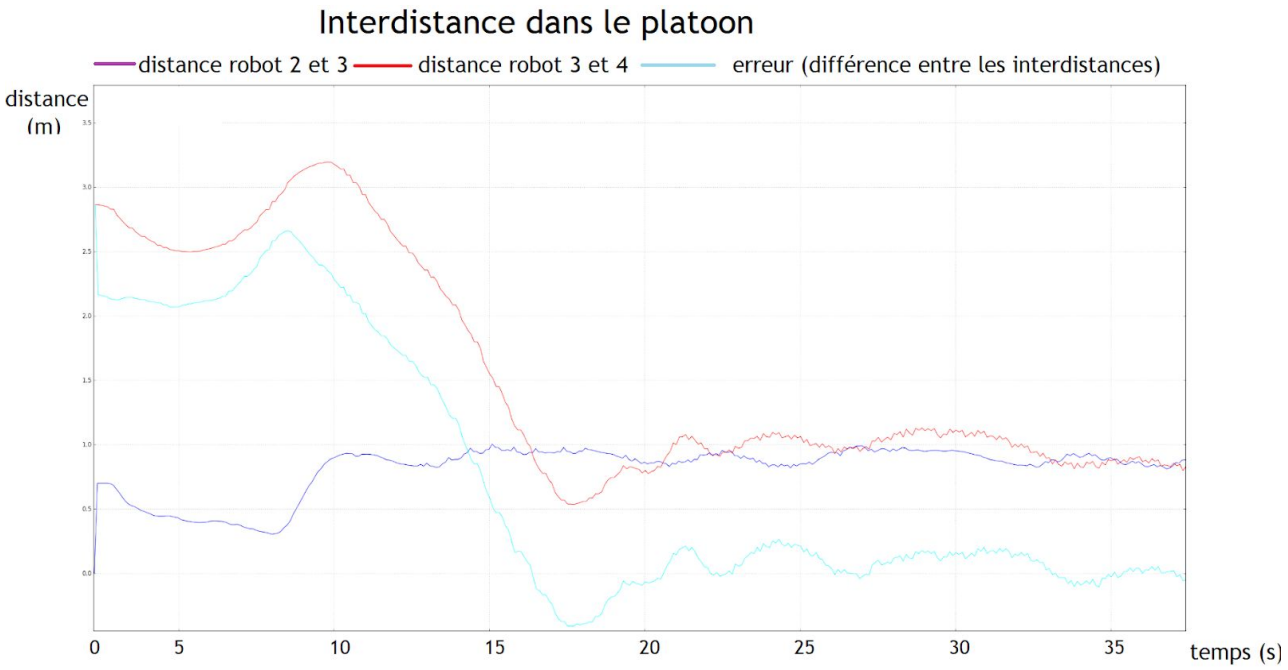
Il est important avant de tirer une quelconque conclusion de se rappeler que les données présentées dans ce graphique ont été tirées d'un **environnement simulé dans gazebo**, ce qui fait que les **commandes du correcteur sont périodiques**, causant des **oscillations** avec une **fréquence beaucoup plus élevée** que les autres. De plus, sur le graphique sont identifiés plusieurs **phases de fonctionnement**, dont la **phase de maintien de l'interdistance** (phase de fonctionnement normal du système).

Tout d'abord, nous pouvons noter des **phases transitoires** durant lesquelles les **vitesses des véhicules** se **comportent comme attendu** lors de celle-ci, hormis un **pic de vitesse de la voiture 4** lors de la reprise de circulation **après le feu**. Nous supposons que cette **anormalité provient de l'environnement simulé**, néanmoins nous n'avons **pas de prototype physique pour vérifier** notre théorie. De plus, cette **anormalité ne semble pas impacter nos résultats** dans les autres domaines.

Par ailleurs, durant les **phases de maintien de l'interdistance,** apparaissent aussi des **oscillations des vitesses** sur des **périodes plus longues**. Nous avons déduit de nos observations que ces **oscillations** sont dues à **trois facteurs essentiels**: les **consignes de maintien de l'interdistance** entre les différents véhicules, la **diminution de la vitesse suite à la prise d'un virage** et enfin le **fonctionnement du correcteur PD** en lui-même. Les **vitesses** des différents véhicules conservent un **écart de 0.1 m/s** (soit 20% de la vitesse de croisière) au **maximum**, hormis un écart du véhicule 3. Néanmoins, cet écart s'explique par l'**accumulation des facteurs** présentés précédemment.

Nous pouvons conclure de ce graphique que les **consignes de vitesse** sont **bien maintenues** dans le cadre des autres consignes, avec toutefois un **épisode d'écart** assez important pour être notifié. Comme pour les trajectoires, il serait intéressant de **tester un nouveau correcteur** pour vérifier si de meilleurs résultats sont obtenus. Un **correcteur** construit à partir de la **théorie de Lyupanov** semble être une bonne première direction pour étendre le sujet [15].

Finalement, nous en sommes arrivés à notre dernier graphique de données, où nous avons mis en parallèle les **interdistances** entre les véhicules 2 et 3 et les véhicules 3 et 4 et avons calculé **l'erreur** (la différence de ces interdistances :



Il est à noter que le graphique commençant au début de la simulation, la voiture 4 n'est **pas encore insérée** dans le platoon. **Aucune nouvelle information** ne transparaît de ce graphique outre que la **consigne d'inter-distance est respectée** avec une **erreur maximale de 0.2 mètre** lors de la **phase de maintien de l'interdistance**, **l'erreur moyenne étant inférieure à 0.1 mètre**. Encore une fois, les résultats semblent être assez bons pour être appliqués sur un prototype réel, avec néanmoins la possibilité **d'essayer un nouveau correcteur** pour **améliorer l'erreur maximale** observée durant la phase normale.

10. Conclusion

Par rapport au cahier des charges que nous avions établi au début du semestre, nous avons réussi à implémenter la plupart des **objectifs** que nous nous étions fixés. Nous avons réussi à implémenter un platoon de robot sous ROS qui peut suivre une **trajectoire précise** tout en conservant une **interdistance** entre les véhicules. Nous avons également réussi à reproduire sous notre **simulation** l'utilisation des **messages** et la **gestion** de ces messages par le **contrôleur**. Les véhicules peuvent également **s'insérer** et se **désinsérer** du platoon grâce aux messages gérés par le contrôleur.

Ainsi, nous avons réalisé le **"must to have"** de notre CDC mais nous n'avons pas réussi à implémenter tout le scénario que nous voulions mettre en œuvre. Nous n'avons malheureusement pas pu mettre en place **d'autres convois** pour gérer leur **interaction** entre eux en fonction de leur "path" respectif. Nous aurions également voulu mettre en place la **reconnaissance d'éléments statique et dynamique** et pouvoir ainsi gérer la partie des messages DENM. Mais cette dernière partie ne pouvait se faire que par le biais du travail effectué par le **PPE**. Néanmoins, un **"nice to have" a été implémenté** notamment par la présence et l'interaction de notre convoi avec les **feux de circulation.**

L'insertion et la **désinsertion** d'un véhicule en milieu urbain sont **délicates.** Il faut, contrairement à l'autoroute, prendre en compte toute la complexité du trafic en ville. C'est pourquoi un **contrôleur omniscient**, pouvant superviser tous les véhicules et infrastructures connectés (feux tricolores, piétons, travaux, etc.) grâce à nos messages ECE, permettrait de réaliser ce projet.

Enfin, au-delà du but premier, l'implémentation de ce type de contrôleur pourrait faciliter la gestion de convois et augmenter la sécurité routière en ville autant pour les piétons que pour les véhicules.

11. Perspectives

Les perspectives d'avenir de ce projet sont multiples. Tout d'abord, il serait intéressant d'**améliorer l'interaction du feu avec le contrôleur**. En effet, en l'état actuel des choses, la taille du peloton n'est pas prise en compte par le contrôleur. Ainsi, le véhicule situé en fin de file pourrait passer au rouge en faisant fi du feu rouge. Dans une autre mesure, il serait intéressant d'étudier et de tester d'autres correcteurs que le PID pour améliorer le suivi de trajectoire.

Par la suite, nous avons pu, au travers de cette année, poser les bases pour les personnes qui reprendront notre travail afin de le compléter et l'améliorer. Ils seront notamment chargés d'implémenter la **détection d'obstacles par le contrôleur**. Pour ce faire, à cette occasion, ils pourront tirer partie des **messages DENM** que nous avons déjà intégrés à la solution. Pour réaliser cette tâche, une nouvelle **carte plus élaborée** contenant des obstacles devra être créée. Ils devront être de nature **statique et dynamique** pour simuler au mieux le monde réel.

Enfin, la crise sanitaire mondiale nous a forcé à travailler dans un environnement simulé. Si le contexte s'améliore et que la crise s'amenuise, nous espérons intégrer notre solution sur **un vrai robot** comme l'Husarion avec de véritables antennes V2X. Nous aurions ainsi des conditions et des paramètres réels.

12. Références Bibliographiques et Sources

[1] Inria, Véhicules autonomes et connectés, Les défis actuels et les voies de recherche, Livre Blanc n°02, 2018.
[2] www.utacceram.com
[3] Niveaux automatisation, SAE INTERNATIONAL, standard J3016 “Levels of Driving Automation”
[4] John Lubbock, « Observations on Ants, Bees, and Wasps.-Part IX. », *Journal of the Linnean Society of London, Zoology*, vol. 16, no 90, janvier 1882, p. 110-121 (DOI 10.1111/j.1096-3642.1882.tb02275.x)
[5] P. Nonacs, *Ant reproductive strategies and sex allocation theory* (1986), Quarterly Review of Biology, no. 61, p. 1-21
[6] Varnava, C. AntBot makes its own way home. *Nat Electron* 2, 93 (2019).
[7] M. Dorigo, V. Maniezzo, et A. Colorni, Ant system: optimization by a colony of cooperating agents, IEEE Transactions on Systems, Man, and Cybernetics--Part B, volume 26, numéro 1, pages 29-41, 1996.
[8] github.com/riebl/ros_etsi_its_msgs
[9] github.com/vlm/asn1c/
[10] www.gazebosim.org
[11] www.ros.org
[12] www.solidworks.com
[13] www.blender.org
[14] app.theconstructsim.com
[15] Denis Arzelier. Théorie de Lyapunov, commande robuste et optimisation. Mathématiques [math]. Université Paul Sabatier - Toulouse III, 2004. fftel-00010257

FIN
