
# Communications Véhicule à Véhicule et Véhicule à Infrastructure

Rapport Final

Encadré par Sahar HOTEIT et Salah EL AYOUBI

Leonardo BORGES MAFRA MACHADO
Caio Henrique COSTA SOUZA
Vinicius LOPES SIMÕES
Marcos Paulo PEREIRA MORETTI

CentraleSupélec
2019

# Sommaire

# Introduction

L'objectif de ce rapport est de décrire le travail de simulation et la mise en œuvre des communications entre les véhicules et entre les véhicules et l'infrastructure. Tout d'abord, nous avons analysé les technologies les plus utilisées actuellement dans ce domaine : WiFi et 5G. Alors, nous avons décidé d'utiliser le protocole IEEE 802.11p pour les simulations.

Ensuite, nous avons utilisé Matlab et les toolboxes WLAN et Automated Driving pour faire des simulations de différents situations de conduite. Avec WLAN, c'est possible de simuler, analyser et tester la couche physique des systèmes de communication sans fils. Le toolbox Automated Driving permet de concevoir, simuler et tester des systèmes de conduite autonomes.

Nous avons notamment étudié le taux d'erreur de paquets pour des situations de conduite sans bâtiments et sans autres véhicules, puis dans une zone urbaine avec certains bâtiments et véhicules, puis dans une intersection urbaine, puis dans une autoroute à plusieurs voies et enfin dans une autoroute à plusieurs voies et avec camions.

Nous avons également étudié une simulation réalisée par MathWorks d'une situation où 3 nœuds échangent des paquets afin d'étudier le protocole CSMA/CA qui évite les collisions. En raison de ce protocole, il n'y avait aucune interférence entre les paquets.

Puis, pour l'implémentation, nous avons utilisé des rovers qui nous ont été donnés par MathWorks. Les rovers sont composés d'une carte raspberry pi, un arduino, une caméra, des roues motorisées, une batterie et un support.

L'objectif était de faire en sorte que tous les rovers suivent un tracé avec le format d'un 8 tout en évitant les collisions qui peuvent se produire aux carrefours. Le chemin était marqué par une ligne noire, et les rovers utilisaient un algorithme de traitement d'images pour suivre la ligne. Les rovers transmettaient des images de la caméra et la tension aux roues au serveur qui les utilisait avec un filtre de Kalman pour déterminer leur position.

Dans les rovers il y avait deux codes principaux écrits en Python, le suivi de ligne et le code responsable pour la communication avec le serveur. Afin de contrôler l'accès aux ressources qui ces deux codes partagent, nous avons utilisé la bibliothèque multiprocessing qui a des structures spécifiques pour la communication entre les processus.

Dans le côté du serveur, il y avait le code Matlab responsable pour l'envoi des commandes aux rovers et le code Python responsable pour la communication avec les rovers. Similairement, afin de contrôler l'accès aux ressources partagées, nous avons utilisé des sémaphores et une connexion localhost entre le matlab et le code python.

Pour la communication entre les codes python dans les rovers et le serveur, nous avons défini un protocole simple où les messages ont un en-tête avec 3 champs de longueur fixe, le nombre du rover, le type de données et la longueur des données, suivi du champ de données avec longueur variable. Pour envoyer les messages, nous avons utilisé des sockets avec le protocole TCP.

# 1. Étude et simulation de la communication V2V et V2I

## 1.1. Contexte et analyse de solutions

Le développement des véhicules autonomes apporte une vaste gamme de problèmes à résoudre, parmi lesquels on peut citer l'assurance de communication entre véhicules (V2V) et avec l'infrastructure (V2I).

Parmi les solutions qui existent actuellement, il y en a deux qui sont considérées les plus pertinentes : WiFi et 5G.

Sur le WiFi, le standard IEEE 802.11 est un ensemble de normes pour les réseaux sans fils locaux. Suite à la perspective du besoin d'un canal de communication, l'IEEE a alloué une bande de fréquence autour de 5,9 GHz dédiée à la communication véhiculaire - c'est l'amendement IEEE 802.11p.

Dans la Cinquième Génération (5G), la communication pour les applications V2X est comprise dans une bande de fréquence libre d'environ 5,9 GHz, et avec une portée d'1,4 km. Un des principaux avantages de cette technologie est le volume de données qu'elle permet de transmettre d'un objet connecté à l'autre, ce qui peut atteindre un taux de 1 Gbit/s jusqu'à 10 Gbit/s dans le futur.

Dans la suite des simulations on utilise le standard IEEE 802.11p.

## 1.2. Les toolboxes WLAN et Automated Driving

Les toolboxes WLAN et Automated Driving sont les toolboxes supplémentaires de MATLAB que nous avons utilisées pour faire des simulations spécifiques.

### 1.2.1. Analyse de la Toolbox WLAN

La toolbox WLAN a des fonctions pour la conception, la simulation, l'analyse et le test de systèmes de communication. Cette toolbox possède des standards IEEE 802.11, ce qui permet de configurer les formes d'onde de la couche physique. Ainsi avec sa toolbox est possible de modéliser le canal d'émission et réception, inclusivement avec le codage du canal et sa modulation.

Pour pouvoir faire une simulation, on a besoin de définir quelques éléments :

**a ) Objet de configuration de canal**

C'est un objet qui définit les paramètres d'un canal de communication, notamment la bande passante, la taille de l'unité de données (PSDU), le schéma de modulation et codification OFDM (MCS) et le nombre d'antennes :

```
cfgNHT = wlanNonHTConfig;
```

**b ) Canal**

Avec l'objet de configuration du canal, on peut obtenir la fréquence d'échantillonnage, créer le canal en soi et modifier ses paramètres (SampleRate et DelayProfile, par exemple) :

```
% Create and configure the channel
fs = wlanSampleRate(cfgNHT);
chan = V2VChannel;
chan.SampleRate = fs;
chan.DelayProfile = 'Urban NLOS';
```

**c ) Formes d'onde**

Cet élément est fondamentalement l'ensemble des données qui seront transmis par le canal :

```
% Data bits
bits = randi([0 1], cfgNHT.PSDULength*8, 1);
% Source waveform
tx = wlanWaveformGenerator(bits, cfgNHT);
```

Le passage du signal par le canal est modélisé par la fonction step :

```
% Receiver waveform
rx = step(chan,tx);
```

Finalement, on ajoute un bruit additif blanc gaussien (AWGN) pour pouvoir simuler les processus stochastiques qui se produisent naturellement.

```
% Create an instance of the AWGN channel for each transmitted packet
awgnChannel = comm.AWGNChannel;
awgnChannel.NoiseMethod = 'Signal to noise ratio (SNR)';
awgnChannel.SNR = 25;
awgnChannel.SignalPower = 1; % Unit power
% Add noise
rx = awgnChannel(rx);
```

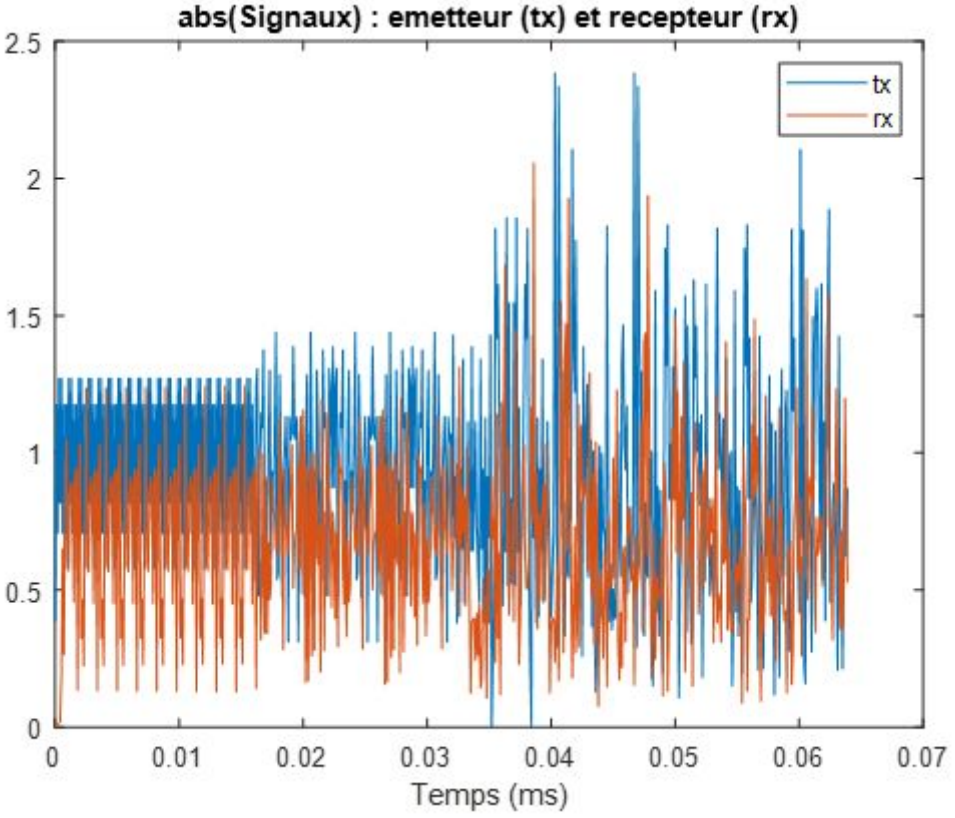


Figure 1 : signaux transmis et reçu (PSDULength = 10 bytes)

L'intérêt de réaliser ces simulations de transmission de paquets est de pouvoir comparer ce qui l'emetteur a envoyé et ce qui le récepteur a reçu. Avec ces données, on peut avoir des statistiques sur le taux d'erreurs et évaluer les performances du canal.

L'analyse d'erreurs dans la transmission peut être faite à partir de deux approches : soit on fait une reconstitution de l'information en interprétant le signal reçu par le récepteur et on obtient le nombre de bits qui présentent une divergence avec l'information transmise (un seul paquet), soit on utilise la fonction "v2vPERSimulator" pour avoir une estimation du taux d'erreur de façon plus globale (avec plusieurs paquets).

La reconstitution de l'information avec la première approche, on utilise la méthode décrite dans [2], où il faut utiliser l'information dans le préambule du paquet (L-LTF) :

```
% Demodulate the L-LTF and use it to estimate the fading channel.
rxLLTF = rx(fieldInd.LLTF(1):fieldInd.LLTF(2),:);
dLLTF = wlanLLTFDemodulate(rxLLTF,cfgNHT);
chEst = wlanLLTFChannelEstimate(dLLTF,cfgNHT);

noiseVar = helperNoiseEstimate(dLLTF);
rxPSDU = rx(fieldInd.NonHTData(1):fieldInd.NonHTData(2),:);
[recPSDU,~,eqSym] = wlanNonHTDataRecover(rxPSDU,chEst,noiseVar,cfgNHT);

numErr(i) = biterr(bits,recPSDU);
```

La figure suivante nous montre, pour 1000 paquets, le nombre de bits erronés parmi les 4000 bits qui forment le paquet. On voit qu'il y a de l'erreur sur deux paquets et le taux d'erreur binaire est inférieure à 0.00625.

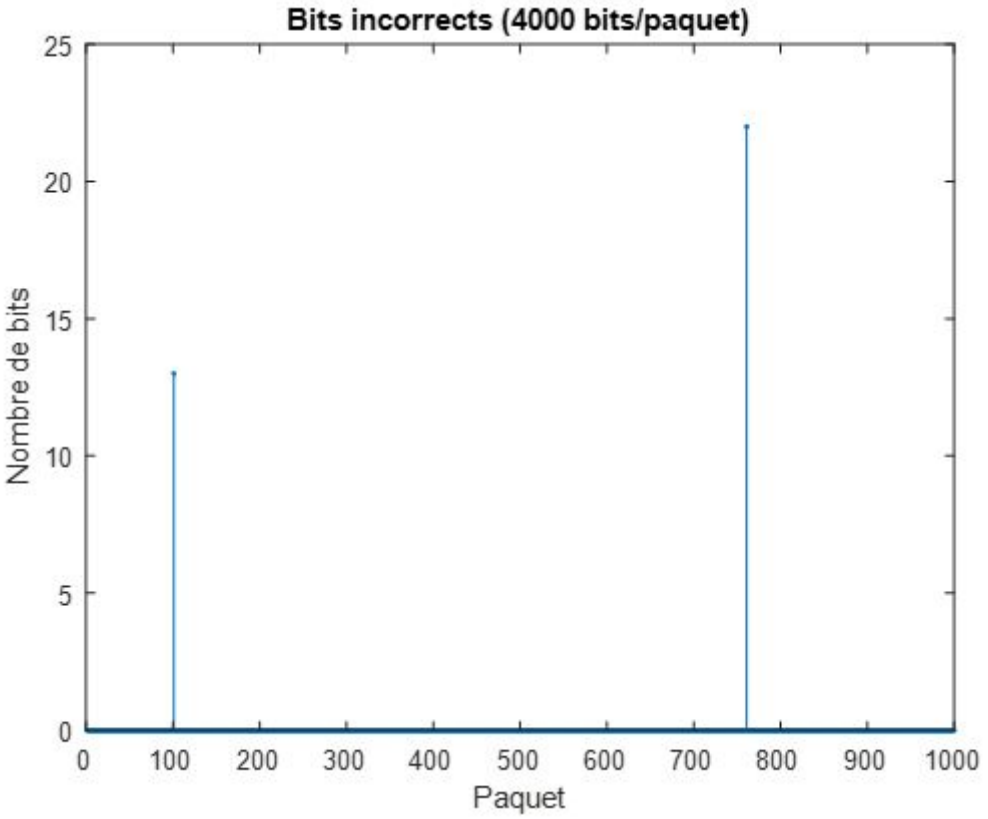


Figure 2 : nombre de bits incorrects par paquet (N = 1000 paquets)

Pour profiter encore plus de cette toolbox, on peut changer le paramètre 'DelayProfile' du canal pour pouvoir jouer sur le scénario de simulation. Les différentes options pour ce paramètre sont décrites ci-dessous :

**a) Rural line-of-sight (LOS)**

Ce scénario est appliqué dans milieux ouverts où les autres véhicules, bâtiments et grands clôtures sont absents.

Figure 3 : Rural line-of-sight

**b) Urban approaching LOS**

Ce scénario comporte deux véhicules qui s'approchent dans une configuration avec des bâtiments à proximité.

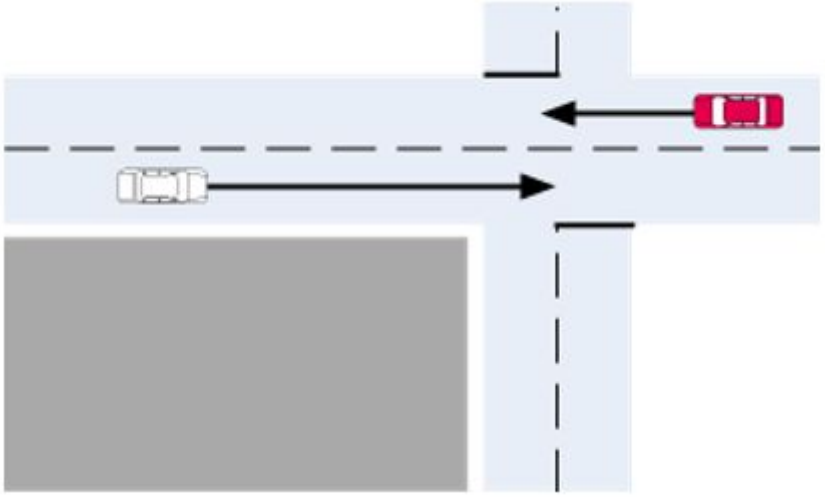

Figure 4 : Urban approaching

**c) Street crossing non-line-of-sight (NLOS)**

Ce scénario définit deux véhicules approchant une intersection urbaine aveugle avec les autres véhicules présents. Les bâtiments / clôtures sont également présents à tous les coins.

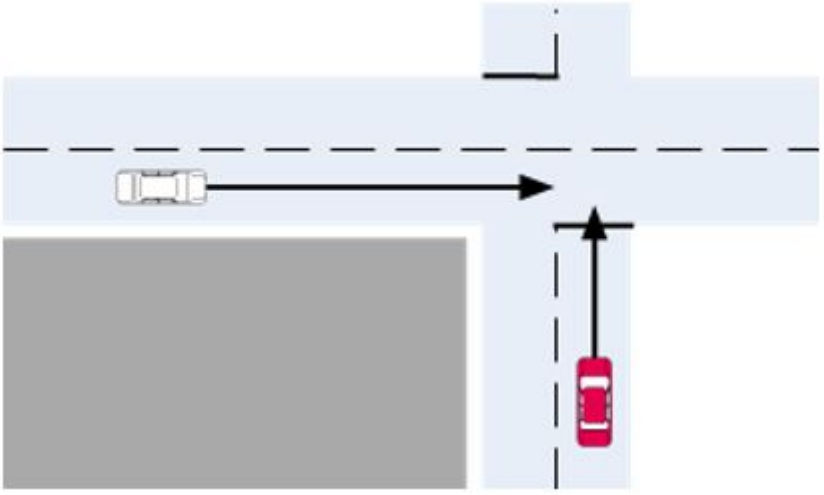

Figure 5 : Street crossing non-line-of-sight

**d) Highway LOS**

Dans ce scénario, deux voitures se suivent sur une chaussée interrégionale à plusieurs voies (par exemple, Autobahn). Des panneaux, des passages supérieurs, des flancs de colline et d'autres types de circulation sont également présents.

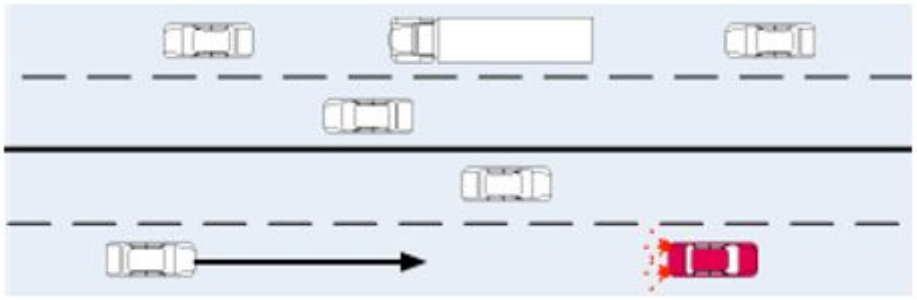

Figure 6 : Highway

**e) Highway NLOS**

Ce scénario ressemble à celui de *Highway LOS*, mais avec des camions bouchés entre les véhicules.

Le fait de varier le scénario de simulation nous permet d'évaluer les performances de la chaîne de communication (en termes du taux d'erreur sur les paquets) pour chaque valeur de rapport signal sur bruit, comme montré dans la figure suivante.

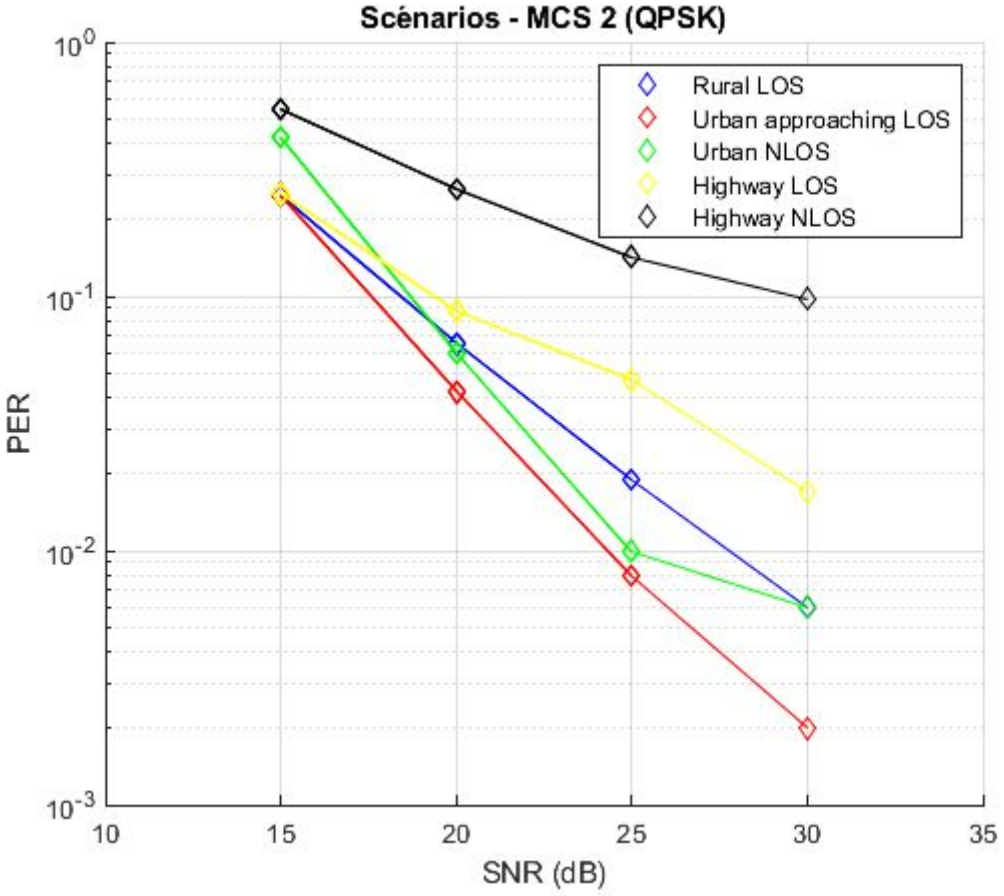


Figure 7 : taux d'erreur sur les paquets pour différents scénarios.

### 1.2.1. Intégration des toolboxes Automated Driving et WLAN

La toolbox Automated Driving fournit des algorithmes et des outils pour la conception et le test des systèmes de conduite autonome et d'aide à la conduite automobile. Ce possible de générer des données de capteurs synthétiques pour des scénarios de conduite, effectuer une fusion multicapteurs, ainsi que concevoir et simuler des systèmes de vision.

L'intégration des deux toolboxes est faite à partir de deux fonctions complémentaires : "getDistanceActors.m" et "PER_distance.m".

La première fonction a pour but de donner deux vecteurs qui représentent les instants et les distances de deux voitures à partir des résultats obtenus de la simulation avec *Automated Driving*.

La deuxième fonction appelle "getDistanceActors.m" et calcule, pour chaque instant donné, la valeur du paramètre SNR à utiliser dans la simulation de la transmission d'un

paquet. Avec ce résultat, on peut faire une estimative de la valeur du taux d'erreur sur les paquets à travers la fonction "v2vPERSimulator" de *WLAN*.

La méthode de calcul a été décrite dans [3], avec l'utilisation de la technologie décrite dans [4], qui nous permet d'avoir une puissance émise de l'ordre de 22,5 dBm. $BW$ est la bande du canal, et la distance est donnée en kilomètres, $A$ et $B$ sont des constantes du modèle simplifié d'affaiblissement de propagation (path loss).

$$SNR_{dB} = P_{emise}(dBm) - P_{bruit} - P_{pertes}$$

$$P_{bruit} = -174 + 10.log_{10}(BW)$$

$$P_{pertes} = A + B.log_{10}(distance)$$

La simulation a été faite à partir de trois scénarios différents conçus avec la toolbox *Automated Driving :*

- "EgoCarTurnsRight_VehicleFromRightMakesUTurn", avec un nombre maximum de paquets avec erreurs égal à 50, le nombre maximum de paquets égal à 500 et avec channel tracking.
- "EgoCarGoesStraight_VehicleFromRightGoesStraight", avec un nombre maximum de paquets avec erreurs égal à 50, le nombre maximum de paquets égal à 500 et avec channel tracking.
- "ELK_OncomingVeh_Vlat", avec un nombre maximum de paquets avec erreurs égal à 50, le nombre maximum de paquets égal à 500 et avec channel tracking.

En analysant les résultats, on peut voir que la distance joue un rôle important dans le taux d'erreur sur les paquets, parce que le rapport signal sur bruit a une forte dépendance avec cette variable.

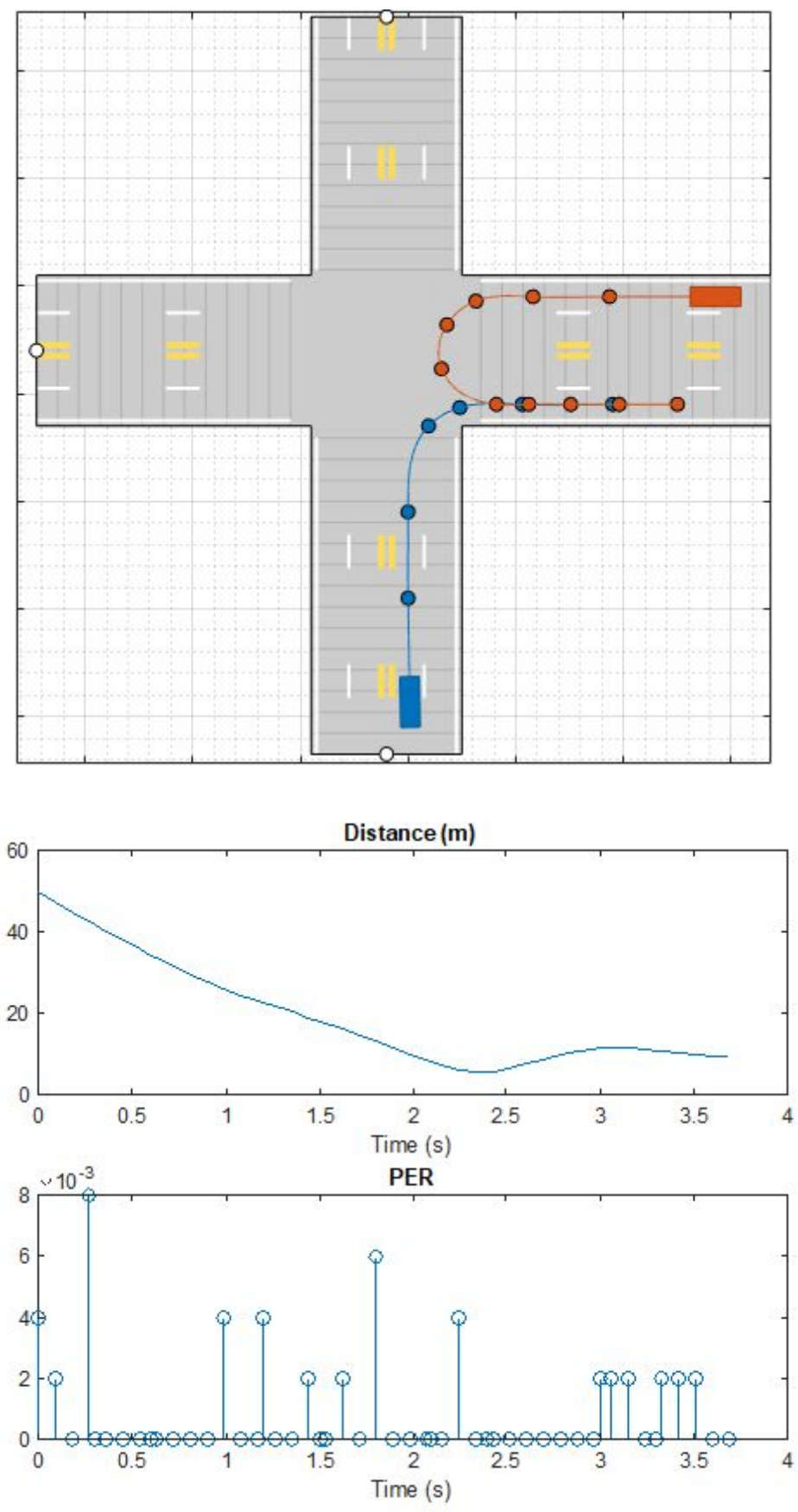


Figure 8 : EgoCarTurnsRight_VehicleFromRightMakesUTurn

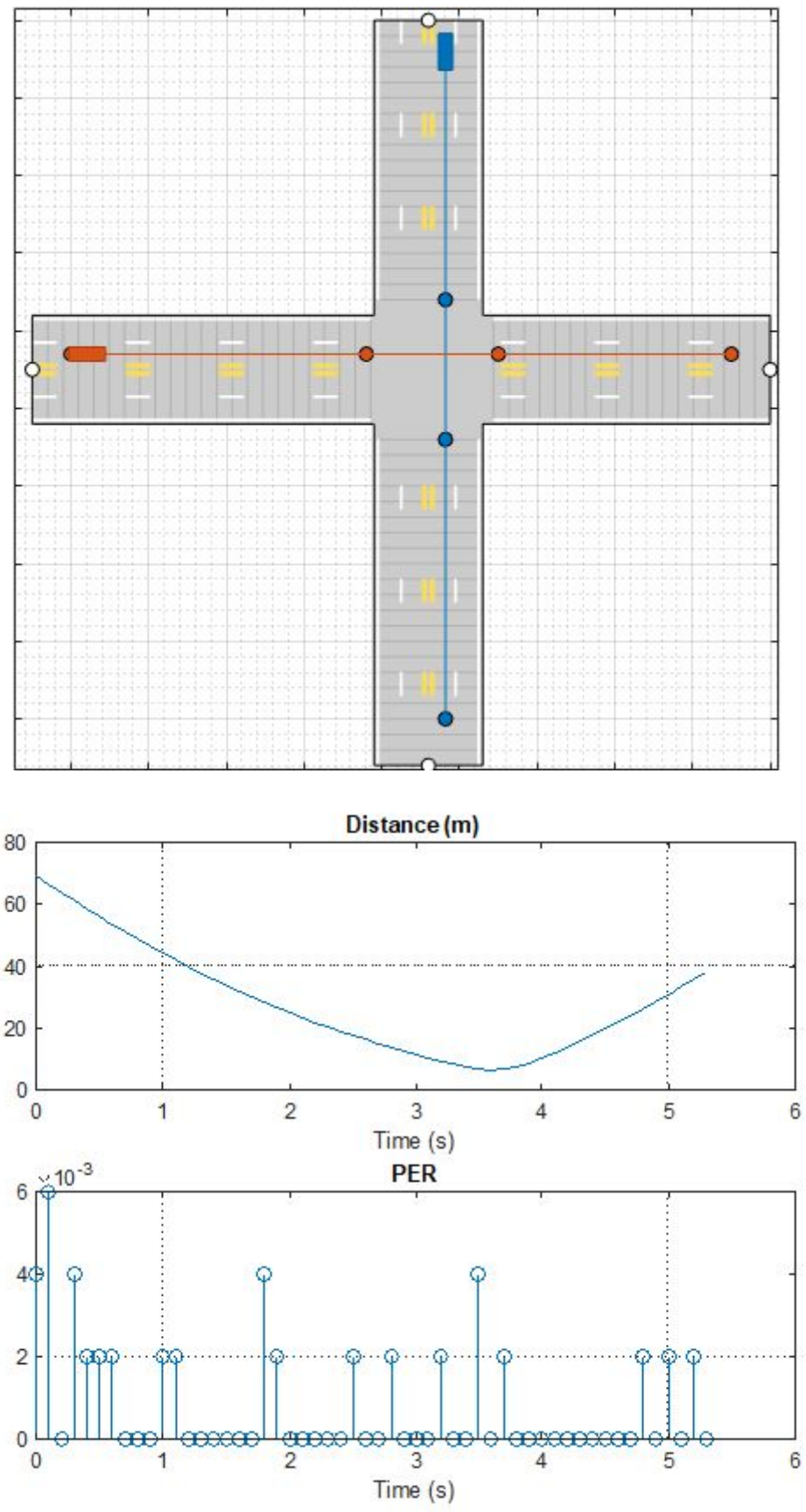


Figure 9 : EgoCarGoesStraight_VehicleFromRightGoesStraight

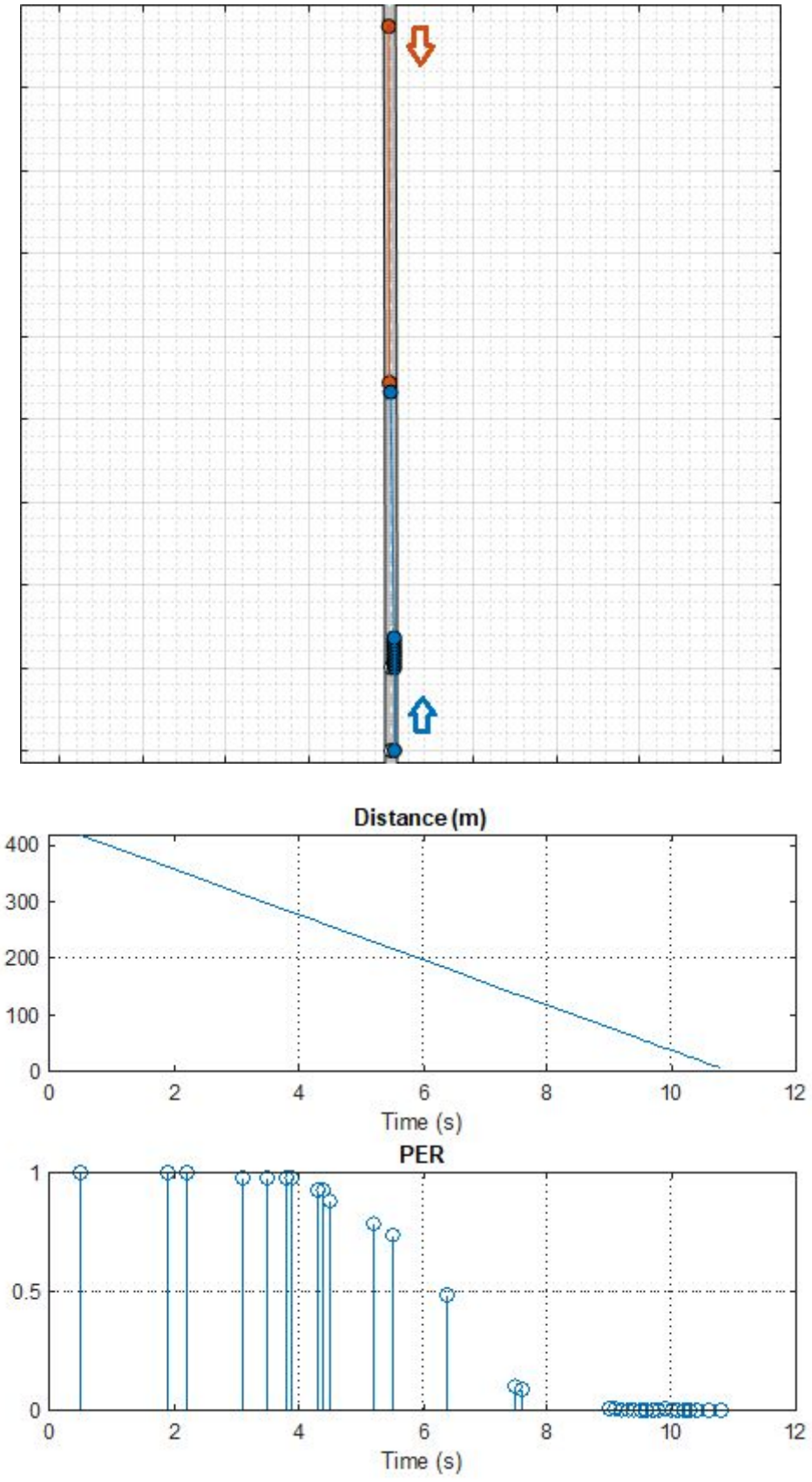


Figure 10 : ELK_OncomingVeh_Vlat

## 1.3. Principes de base pour l'étude de l'impact des interférences

Un des phénomènes qui peut nuire les performances est l'interférence, qui est caractérisé par une éventuelle collision de paquets. Cela modifie le signal pendant son chemin de l'émetteur vers le récepteur. Il y a plusieurs exemples de sources qui peuvent jouer sur la qualité du signal reçu, comme l'interférence électromagnétique et d'autres émetteurs qui envoient leurs paquets simultanément dans le même canal.

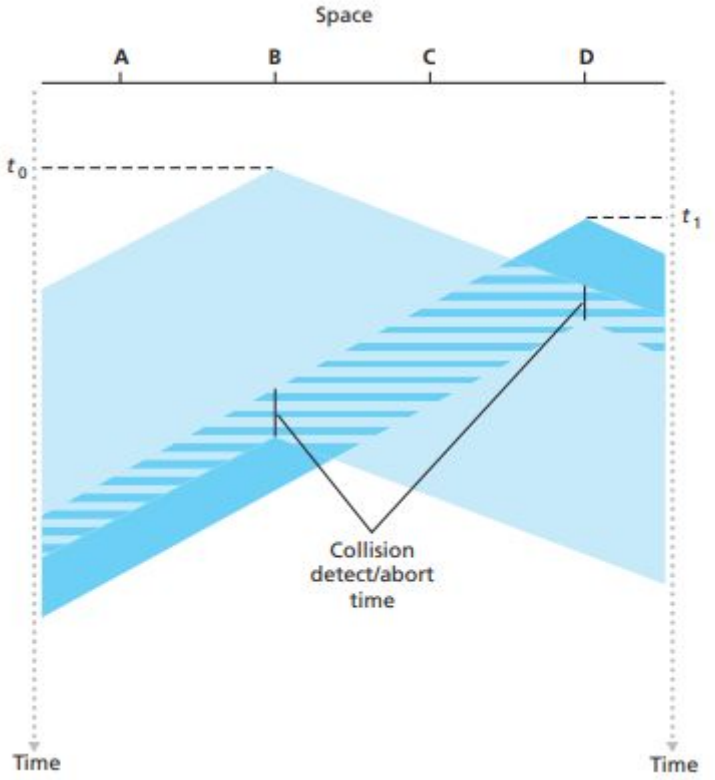


Figure 11 : Illustration du phénomène d'interférence entre deux noeuds

La figure 11 montre le problème de l'interférence entre deux noeuds qui peut arriver dans les systèmes de communication. À $t = t_0$, le noeud B émet un paquet dont les bits sont propagés dans les deux directions. À $t = t_1$, le noeud D n'a pas encore reçu le paquet de B et transmet lui-même un autre paquet. Après un intervalle de temps, on voit que les deux paquet commencent à interférer.

Pour essayer de contourner ce problème, la norme 802.11 utilise une méthode d'accès dans la couche physique dite CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance). D'après Kurose, il y a deux composantes qui servent à définir le fonctionnement de ce protocole :

- Écouter avant de parler : si quelqu'un est en train d'utiliser le réseau, il faut attendre qu'il ne soit plus occupé et, dans ce cas, la transmission est différée. Dans le cas contraire, la transmission peut continuer. Cette composante est liée à la partie "carrier sense multiple sense".
- Esquive de collision : ce mécanisme est basé sur l'algorithme de "exponential backoff". C'est-à-dire que quand l'émetteur a été empêché d'envoyer son paquet $n$ fois, il doit attendre $K$ . $SlotTime$, où $K$ est choisi au hasard parmi $\{0, 1, 2, \ldots, 2^{n-1}\}$ et $SlotTime$ est un intervalle de temps fixe. Donc, l'intervalle d'attente d'envoi d'un paquet varie exponentiellement avec le nombre de collisions que ce paquet a connu.

Soit un émetteur avec un paquet à transmettre, le fonctionnement du protocole 802.11 CSMA/CA est le suivant :

1) Si l'émetteur perçoit le canal inactif, il transmet le paquet après une courte période de temps DFIS (Distributed Inter-Frame Space);

2) Autrement, il doit choisir un temps de backoff au hasard en utilisant l'algorithme de "exponential backoff" décrit ci-dessus et met un compte à rebours à partir du moment où le canal a été perçu inactif;

3) Quand le compteur arrive à zéro, ce noeud transmet le paquet et attend un ACK (accusé de réception);

4) Si l'ACK est reçu l'émetteur sait que son paquet a été émis avec succès. Sinon, il doit recommencer le processus à partir de l'étape 2 avec une valeur de backoff choisie parmi un intervalle plus grand.

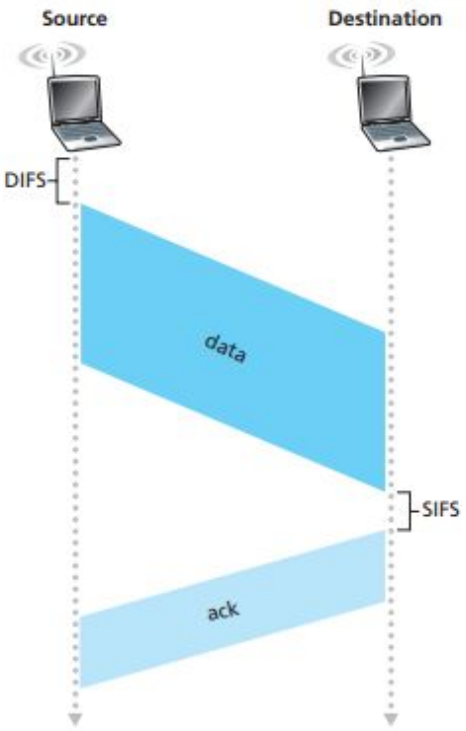


Figure 12 : Intervalles (DFIS et SIFS) avant chaque transmission

L'objectif de ce protocole est d'éviter les collisions dès que possible. Avec 802.11, si deux noeuds sentent le canal actif, ils doivent choisir un temps d'attente aléatoire avant d'envoyer ses paquets, et avec un peu de chance, ils auront choisi des valeurs différentes.

MathWorks a mis en oeuvre une simulation d'une situation avec 3 noeuds qui échangent des paquets sur le même canal (Figure 13). On s'intéresse à analyser les caractéristiques d'une transmission à CSMA/CA et l'étude spécifique de la constitution de chaque bloc dans la simulation est en dehors de la portée du projet.

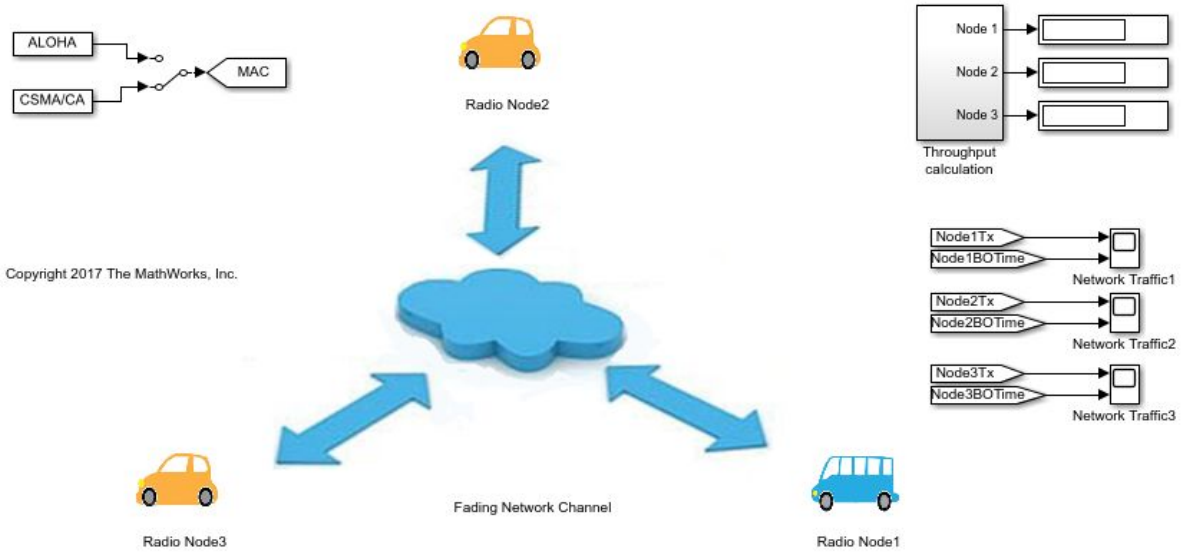


Figure 13 : schéma de simulation conçu par MathWorks

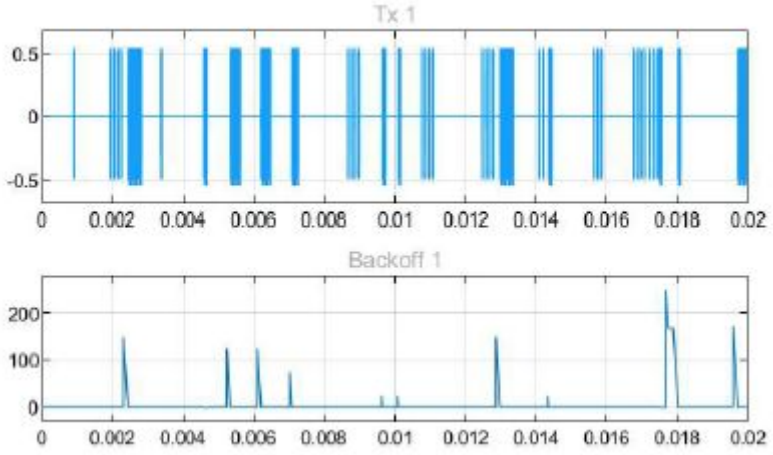


Figure 14.a : signal transmis et temps de backoff pour le premier noeud

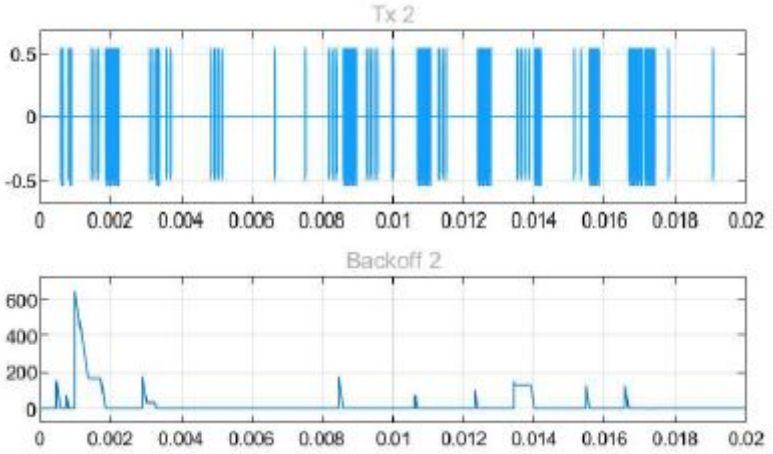


Figure 14.b : signal transmis et temps de backoff pour le deuxième noeud

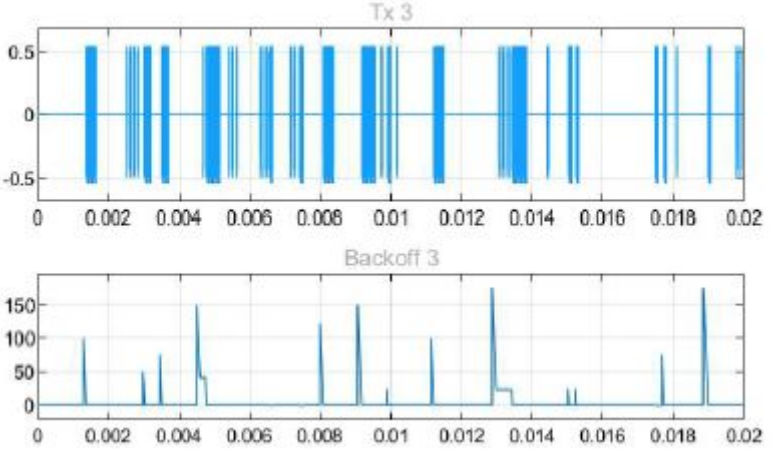


Figure 14.c : signal transmis et temps de backoff pour le troisième noeud

Les résultats de cette simulation nous montre le signal transmis par chaque noeud et le compteur “backoff”. Il est possible de voir que chaque transmission n’a pas d’interférence avec l’autre grâce au protocole CSMA/CA.

# 2. Partie pratique : communication entre le rover et le serveur

## 2.1. L'introduction

Par la suite, nous donnerons une brève introduction à la deuxième partie du projet en commençant par la démonstration faite par Nokia et qui a servi de source de motivation pour cette deuxième partie du projet. Ensuite, nous allons faire une brève description du rover, puis de la raspberry pi. Le but de cette introduction n'est pas de fournir assez des détails, mais de donner une idée générale. Pour plus de détails, voir le rapport final du groupe WP2.

### 2.1.1 La démonstration

La démonstration consiste à mettre en route les rovers dans une piste en format de "8". On doit garantir que les rovers puissent suivre la ligne tracée sur la piste et se coordonner pour éviter toute collision qu'il peut arriver au carrefour.

Voici un exemple de piste avec un format similaire. Celle ci-dessous a été produite pour une démonstration de Nokia et est présentée dans la vidéo "Nokia 5G Car Demo Autonomous Driving (Self Driving Cars)" [1].

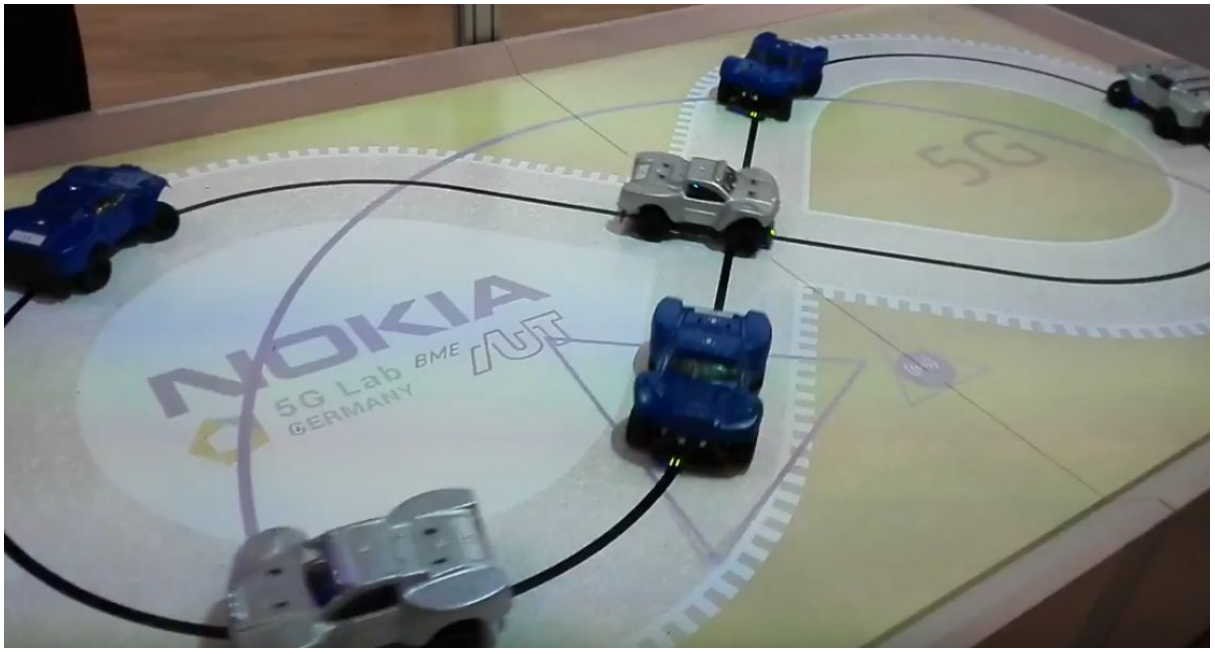


Figure 15 : Piste en "8".

Chaque working package était responsable d'une partie du projet, notre partie (WP3) étant responsable de la communication entre les rovers et le serveur.

### 2.1.2. Le rover

Il est possible de voir dans l'image 16 le rover utilisé dans notre démonstration. Il nous a été fourni par MathWorks et a comme composants principaux la carte raspberry pi, l'arduino, la caméra, les roues motorisées, la batterie et le support.

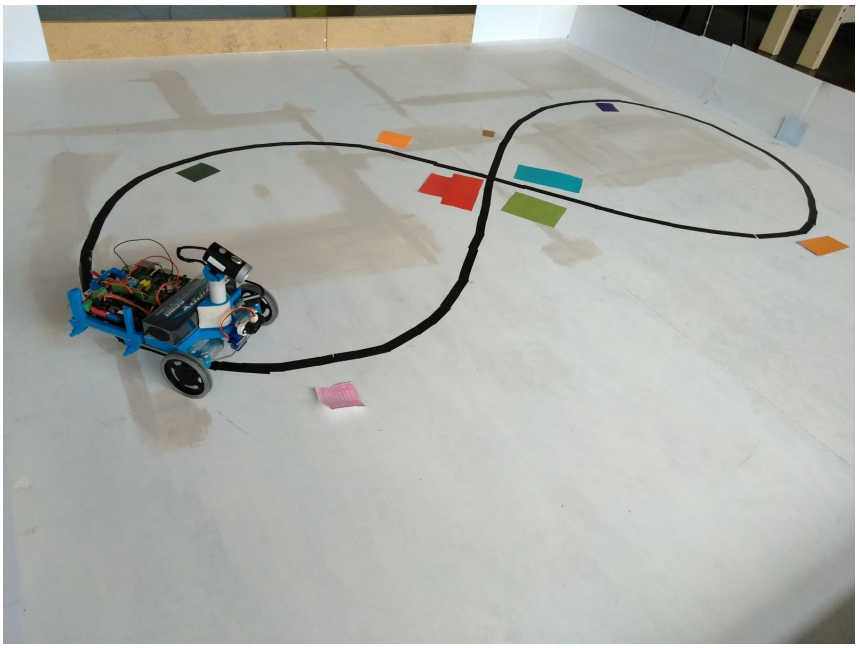

Figure 16 : rover dans le circuit

### 2.1.3. La raspberry pi

Chaque rover dispose d'une carte Raspberry Pi 3 B. Voici une image de la carte:


Figure 17 : Carte Raspberry 3 B.

La préparation de la carte raspberry pi est détaillée dans le rapport du groupe WP2, il nous suffit savoir que la carte possède un système d'exploitation Linux et qu'il est possible de la connecter à un réseau Wi-Fi.  La carte est également connectée à l'arduino.

Dans le cadre de notre groupe, la communication, la carte raspberry pi est le composant  principal du rover sur lequel nous allons travailler.

## 2.2. La communication

La communication entre les rovers et le serveur est importante pour la bonne réalisation du circuit sans collisions. Le serveur fonctionne comme un centre de commande qui reçoit les informations de chacun des rovers présents dans le circuit et traite ces données pour tenter de reconstruire le scénario par calcul et décider quelle instruction envoyer à chaque rover.

Pour ce faire, le serveur a besoin de certaines données qui ont été collectées par le groupe WP4 et confiées à WP2 pour les fournir (les images et la tension sur chaque roue). C'est à ce moment précis que WP3 apparaît lorsque nous effectuons cette intégration, ainsi que la communication, permettant cet échange d'informations.

Nous pouvons alors définir un serveur (étant le serveur lui-même comme son nom l'indique) et un client qui dans ce cas serait chacun des rovers. Après avoir défini les agents impliqués dans l'échange d'informations, nous devons parler de la chaîne. Après notre recherche théorique approfondie, nous sommes arrivés à la conclusion que le meilleur protocole à utiliser serait le 8012.11p, mais comme nous effectuons la communication sur des distances considérablement courtes, le protocole Wi-Fi standard est suffisant et nous l'utiliserons.

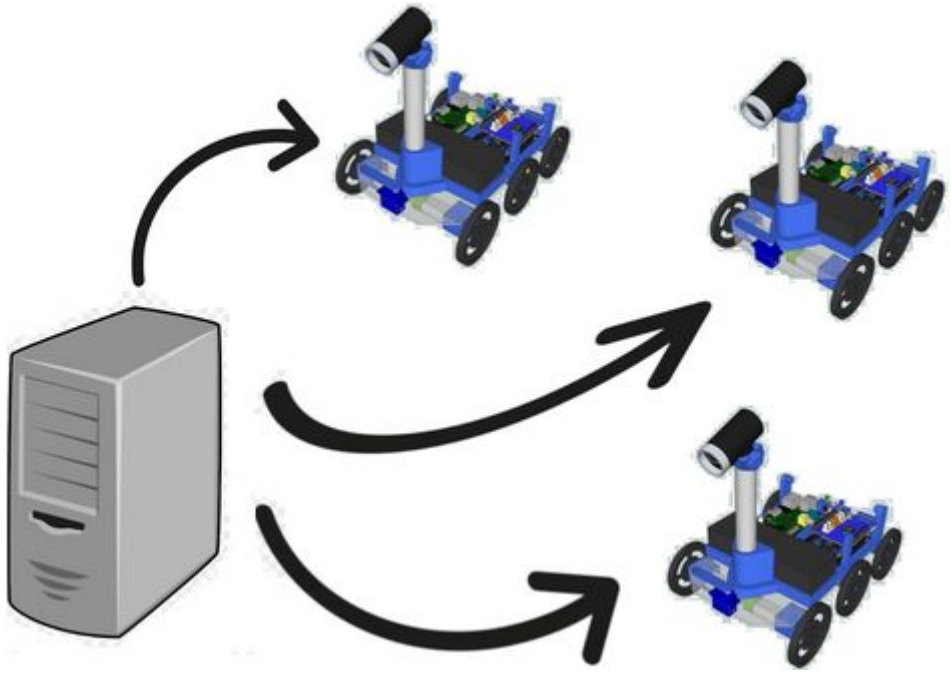

Figure 18 : Représentation du serveur et des rovers.

Comme mentionné dans l'introduction, la raspberry pi a Linux comme système d'exploitation, donc le langage choisi pour travailler dessus était python. Déjà sur le serveur nous avons deux langages utilisées, python et MATLAB. Après avoir étudié la possibilité de n'utiliser MATLAB que pour réaliser le serveur, nous sommes arrivés à la conclusion qu'il ne serait pas possible en raison des difficultés qui existent dans MATLAB de réaliser différents threads avec plusieurs connexions TCP simultanées (ce qui est absolument nécessaire car plusieurs rovers peuvent envoyer des données en même temps). Nous avons donc choisi d'écrire la majeure partie du code en python et d'écrire deux interfaces en MATLAB qui seraient la connexion du serveur python avec le code fait par WP4 en MATLAB.

Enfin, un dernier point qui n'a pas encore été mentionné est que nous avons besoin d'un dispositif qui peut générer un réseau Wi-Fi où les rovers et le serveur peuvent se connecter. Tout d'abord, l'idée d'utiliser un routeur a été évoquée, mais en raison de la difficulté que cela pourrait générer pour les groupes de l'année prochaine, nous avons choisi la solution d'utiliser un smartphone qui a la possibilité de créer un réseau (fonction hotspot).

Cette solution est très pratique et de facile accès (presque tous les smartphones ont cette option), mais il y a un coût de ne pas pouvoir contrôler les adresses IP de chaque rover et du serveur. De cette façon, si un rover (ou le serveur) perd la connexion au réseau, lors de la reconnexion, il y a la possibilité qu'il assume une autre adresse IP, ce qui nécessiterait de modifier les codes de ce rover particulier ainsi que ceux du serveur.[1]

[1] Ce problème éventuel sera traité plus en détail dans les pages suivantes

Le schéma général que nous allons suivre jusqu'à la fin est le suivant (image 19):

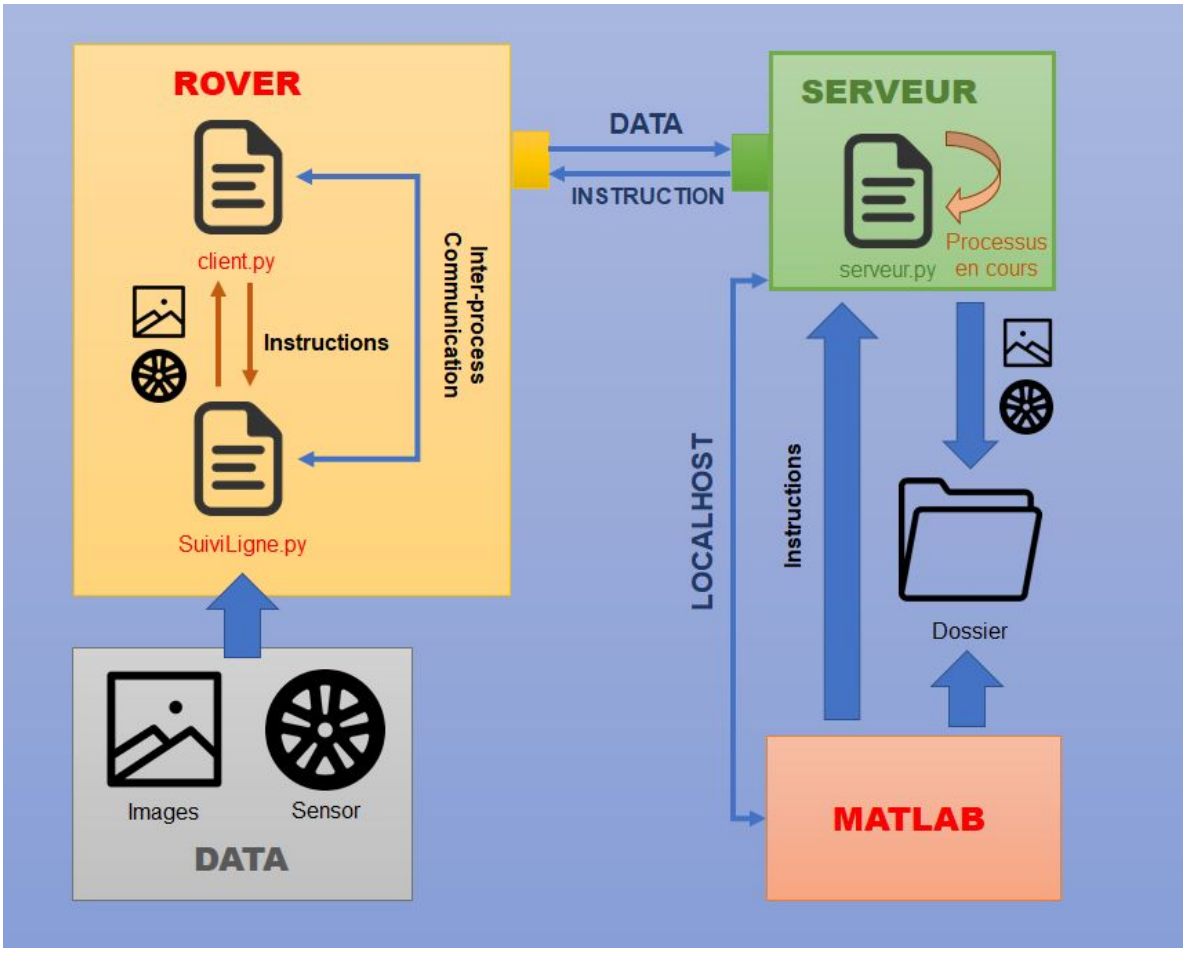


Figure 19 : Schéma proposé

Comme c'est possible de voir, le schéma est déjà complet dans l'image ci-dessus et notre objectif sera désormais de le décomposer en plusieurs morceaux et de travailler sur chacun d'eux jusqu'à ce qu'il soit possible de reconstituer ce même schéma général.

## 2.3. Le client (rover)

Dans le rover nous avons un code unique que nous appelons génériquement client.py. Ce code tel qu'illustré sur l'image 19 se connecte d'un côté au serveur et de l'autre côté au code SuiviLigne.py réalisé par WP2 et responsable du contrôle de l'arduino. De cette façon, nous pouvons voir le code client.py comme l'intégration du code du groupe WP2 avec la communication du rover avec le serveur. La façon dont ces deux liens ont été établis sera détaillée dans les deux prochaines sections.

### 2.3.1. Connexion client et serveur

La connexion entre le client et le serveur s'est faite par l'utilisation de sockets et du protocole TCP. Nous utilisons également la bibliothèque de selectors[2] pour faciliter le codage mais ce n'est pas nécessaire. Chaque rover se connectant au serveur pour la première fois effectuera le handshake TCP standard et commencera ensuite à envoyer les images et/ou les tensions des roues du rover.

L'image sera envoyée au format jpg et la tension de la roue comme une chaîne de caractères et il est de la responsabilité du serveur de stocker correctement ces données dans les formats appropriés.

À ce moment, nous attirons l'attention du lecteur sur la pièce jointe numéro 1 où se trouve le code du client. Le code est commenté pour faciliter la compréhension et il est recommandé de l'étudier à ce stade avant de procéder à la connexion entre client.py et SuiviLigne.py.

### 2.3.2. Connexion client et code WP2

La connexion entre client.py et SuiviLigne.py a été problématique dès le début. Tout d'abord, le code suivilligne du groupe WP2 n'était pas au format classe, ce qu'il était nécessaire de faire en raison de l'utilisation de différents threads dans le fichier client.py. Après l'avoir transformé en une classe, le problème est devenu la puissance maximale que la raspberry pi fournit à un processus approprié.

Pour expliquer ce problème, nous devons d'abord comprendre la différence entre les threads et les processus et comment le processeur fournit la puissance à chacun d'eux.

Un processus peut être décomposé en différents threads, chacun s'exécutant simultanément. L'avantage de travailler avec des threads dans le même processus est qu'ils partagent la même mémoire heap, ce qui nous permet d'utiliser certaines structures de synchronisation des threads comme les sémaphores. Cependant, les différents processus ne partagent pas de mémoire, ce qui rend difficile l'échange d'informations entre deux processus distincts, c'est a dire, il n'y a pas de "sémaphore" dans ce cas.

Ainsi, nous pouvons conclure que l'utilisation de threads est plus recommandée exactement pour cette facilité de communication entre eux et c'est exactement ce que nous avons fait dans notre première tentative, les codes client.py et SuiviLigne.py étaient deux threads dans le même processus.

[2] Voir [6] pour plus de détails

Le problème que cela a généré était que la puissance maximale que le processeur libère pour chaque processus n'était pas suffisante pour faire fonctionner nos deux programmes et nous avons rencontré un scénario où le rover ne se déplaçait pas correctement en quittant le chemin désigné.

Pour résoudre ce problème, il était nécessaire de séparer les exécutions de client.py et SuiviLigne.py en deux processus différents. Comment ferions-nous alors pour transmettre l'information appropriée entre ces deux codes ? Pour cela, nous avons utilisé la bibliothèque multiprocessing qui possède des structures spécifiques pour la communication entre les processus.

Donc, ceci résume d'une manière générale ce qui a été fait dans le code client, nous recommandons de revoir le code en annexe 1 avec les commentaires fournis.

## 2.4. Le serveur

Les codes de serveur peuvent être regroupés en deux catégories, les codes python et les codes MATLAB. Nous avons trois codes en python, MainServer.py, ConnectionRover.py et ConnectionMatlab.py et deux en MATLAB, readDataServer.m et sendInstruction.m.

Comme les noms propres l'indiquent, le ConnectionRover est le code responsable de la connexion au rover (client.py) et le ConnectionMatlab responsable de la connexion localhost entre le code MATLAB et le serveur en python. Le code MainServer est un code central utilisé pour initialiser les deux précédents.

Le code readInstruction.m à son tour est responsable de l'envoi de l'instruction au serveur python, cette information arrivera dans le code de ConnectionMatlab qui la transmettra ensuite au ConnectionRover et enfin ce dernier va l'envoyer au rover (l'instruction arrivera dans Client.py et sera envoyée à SuiviLigne.py).

Le code readInstructionServer.m demande à son tour l'accès aux images et aux tensions du serveur python.

### 2.4.1. Connexion serveur python et rover

Le code ConnectionRover.py est responsable de cette partie, il reçoit les données envoyées par les rovers et renvoie les instructions données par ConnectionMatlab qui seront détaillées dans la section suivante.

Le format du message est composé de quatre champs, le premier est l'ID du rover, puis le type du message, puis la longueur des données et enfin les données. Les trois premiers champs (l'en-tête) ont la même longueur fixe de 9.

Pour décoder les messages, le serveur continue de recevoir des paquets jusqu'à ce que la longueur des données reçues soit supérieure à la longueur de l'en-tête. Ensuite, le serveur enregistre les informations d'en-tête (numéro de rover, type de données et taille des données) et continue à recevoir les paquets jusqu'à ce que la longueur des données reçues soit supérieure à la taille des données, puis il acquiert un sémaphore afin de sauvegarder les données, une fois que les données sont complètement sauvegardées, il libère le sémaphore. Le serveur s'assure également que les données des différents rovers ont été stockées dans des dossiers séparés pour chaque rover.

### 2.4.2. Connexion serveur python et MATLAB

Afin d'empêcher le code MATLAB de lire les images ou les tensions en même temps que le code python les réécrit, il était nécessaire de créer cette connexion entre MATLAB et le serveur python. De cette façon, lorsque MATLAB veut accéder à un fichier, il demande la permission au serveur python avant et seulement lorsque le serveur python lui permet de lire les fichiers, il les accède.

Pendant ce temps, logiquement, le serveur python n'aura pas le droit de réécrire ces fichiers. Pour ce faire, nous utilisons des sémaphores et une connexion localhost entre MATLAB et le serveur python. Cette connexion localhost est également utilisée pour recevoir les instructions qui sendIntruction.m envoie et que ConnectionMatlab transmet à ConnectionRover.

À ce moment nous vous conseillons de consulter l'annexe numéro 2 et regarder le code ConnectionMatlab et de lire les commentaires pour une meilleure compréhension.

# Conclusion

Dans ce projet, on a travaillé en deux étapes majeures : étude préliminaire et simulation; et une partie pratique sur le rover.

Pendant la première partie, nous avons pu comprendre les différentes solutions existantes pour la communication V2V et V2I, ainsi que réaliser des simulations pour évaluer les performances d'une chaîne de communication qui utilise le protocole 802.11p. Ces simulations ont été possibles grâce à l'utilisation des toolboxes du logiciel Matlab : WLAN et Automated Driving et elles nous ont montré une grande variation des performances en fonction de la distance entre deux noeuds qui se communiquent.

La deuxième étape a été consacrée au développement d'un système de communication entre les rovers et un serveur sur la plateforme de Raspberry Pi. Notre groupe a travaillé avec le langage Python pour le développement des clients sur les rovers et du serveur. On a aussi développé des fonctions Matlab qui nous permettent d'accéder aux fichiers du serveur (pour avoir des informations sur chaque rover) et d'envoyer des instructions aux rovers. La liaison entre le serveur en Python et les fonctions Matlab est particulièrement importante pour pouvoir développer des stratégies de commandes plus sophistiquées, qui demandent une interaction entre les rovers.

À la fin de l'intégration avec les autres WPs, un nouveau code pour le suivi de ligne a été proposé en utilisant une logique floue avec le traitement des images sur la plateforme de la Raspberry Pi. Malheureusement, cette plateforme a eu des problèmes pour gérer les trois fonctionnalités en même temps (suivi de ligne, communication et traitement des images) à cause du partage des ressources. Une possible solution à ce problème est d'utiliser un langage qui permet le traitement parallèle, comme le C.

Finalement, en faisant un bilan général de ce projet, on peut dire que le groupe a approfondi les connaissances de base en théorie des communications et réseaux avec un grand apport de nouvelles connaissances en informatique, plus particulièrement en Python. Étant inclus dans un projet plus grand sur les véhicules autonomes, la communication est un aspect qui joue un rôle vital dans le bon fonctionnement du circuit de movimentation des rovers et pourra être utilisée pour en améliorer les performances.

# Références

[1] *Nokia 5G Car Demo Autonomous Driving (Self Driving Cars)*. Disponible sur https://www.youtube.com/watch?v=-PVNV_YUEJY

[2] *Non-HT Packet Recovery*, Mathworks. Disponible sur https://www.mathworks.com/examples/wlan-system/mw/wlan-ex06496606-non-ht-packet-recovery?s_tid=examples_p1_Topic

[3] *Ingénierie des réseaux cellulaires.* Salah Eddine El Ayoubi. CentraleSupélec

[4] *A Linear 5.9 GHz Power Amplifier for IEEE 802.11p Applications.* Gao-Ching Lin et al. Disponible sur https://ieeexplore.ieee.org/document/7365084

[5] *Ready to roll: Why 802.11p beats LTE and 5G for V2x - A white paper by NXP Semiconductors, Cohda Wireless, and Siemens*, Alessio FILIPPI, Kees MOERMAN, Gerardo DAALDEROP, Paul D. ALEXANDER, Franz SCHOBER, Werner PFLIEGL

[6] selectors — High-level I/O multiplexing. Disponible sur https://docs.python.org/3/library/selectors.html

[7] *Véhicules autonomes et connectés - Les défis actuels et les voies de recherche*, Inria (livre blanc)

[8] *Fader Channel Model - 802.11p*, Keysight. Disponible sur http://rfmw.em.keysight.com/wireless/helpfiles/n7605/Content/Main/fcm_80211p.htm

[9] *Non-HT Packet Recovery*, Mathworks. Disponible sur https://www.mathworks.com/examples/wlan-system/mw/wlan-ex06496606-non-ht-packet-recovery?s_tid=examples_p1_Topic