

Rapport de Projet de Fin d'Etudes

Simulation et expérimentation de Trust Management dans VANET

Encadrants : Ye-Qiong SONG et Runbo SU

Résumé

De nombreux accidents de voiture se produisent dans le monde et la plupart sont dus à une erreur humaine, car nous ne pouvons pas tout voir et tout prévoir. L'idée de VANET est qu'une voiture communique sa vitesse, sa position et d'autres paramètres de conduite utiles aux autres véhicules à proximité. Le fait de pouvoir communiquer des informations entre plusieurs voitures apporte des informations supplémentaires au conducteur et ainsi lui permettre d'anticiper sa conduite. De plus, un ordinateur ayant accès aux informations des autres véhicules peut réagir plus rapidement aux différents événements de la route. Partager des informations avec tout le monde présente l'inconvénient de s'exposer à des actions malveillantes. Le principe du Trust Management est de contrer ces actions en mettant à l'écart les véhicules dont la note de confiance est inférieure à un seuil, la note est calculée en fonction de ces paramètres de conduite. La simulation peut donc fournir des preuves de la sécurité de ce principe.

Mots-clés : Véhicules, Gestion de confiance, Communication, Malveillant, Sécurité

Abstract

Many car accidents occur in the world and most of them are due to human error because we cannot see and anticipate everything. The idea of VANET is that a car will communicate its speed, position and other useful driving parameters to other nearby vehicles. Being able to communicate information between several cars could bring additional information to the driver and thus allow him to anticipate his driving. Moreover, a computer with access to information from other vehicles could react faster to different road events. Sharing information with everyone has the disadvantage of exposing oneself to malicious actions. The principle of Trust Management is to counter these actions by putting aside vehicles with a trust rating below a threshold, a rating that is calculated permanently according to these driving parameters. Simulation can therefore provide evidence on the security of this principle.

Keywords : Vehicles, Trust Management, Communication, Malicious, Security

Table des matières

1. Introduction	4
2. Etat de l'art.....	4
2.1. VANET et la sécurité dans VANET	4
2.2. Trust Management	6
3. Déroulement du PFE	11
3.1. Etudes bibliographiques	11
3.2. Ns-3.....	11
3.3. Veins	16
3.4. SUMO.....	18
3.5. Omnet++	20
3.6. Veins et Inet.....	26
3.7. La simulation.....	31
4. Conclusion	41
5. Références bibliographiques	42
6. Sources.....	42

1. Introduction

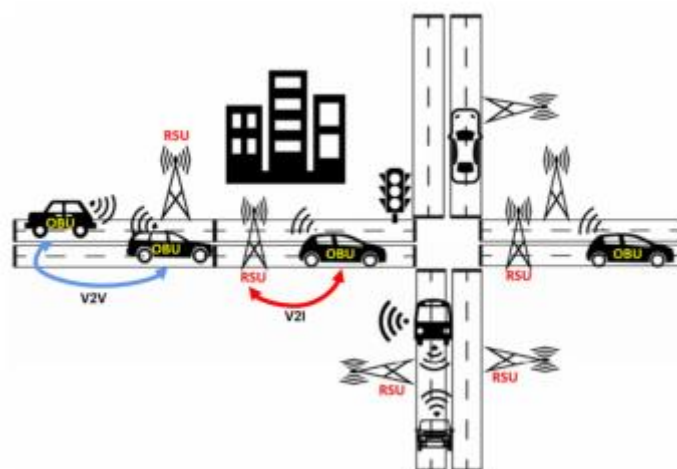
Le but de ce projet est de concevoir une simulation d'un réseau de type VANET dans laquelle on implémente un modèle de Trust Management permettant de maintenir la fiabilité de VANET et en même temps d'identifier les attaques de nœuds malveillants.

Ce PFE a été encadré par Ye-Qiong SONG, professeur d'informatique à l'Université de Lorraine et chercheur au Loria, et Runbo SU, étudiant en deuxième année de doctorat au Loria. C'est au Laboratoire lorrain de recherche en informatique aussi appelé Loria que s'est déroulé ce PFE d'une durée de 4 mois. Le projet s'est déroulé en plusieurs étapes. En premier lieu, il y a eu une étude bibliographique sur la sécurité de Trust Management dans VANET. Puis, le modèle de Trust Management qui a permis d'évaluer dynamiquement la confiance des véhicules a été conçu. Enfin, La simulation numérique du modèle de Trust Management sur Veins avec Omnet++ et SUMO a été réalisé.

2. Etat de l'art

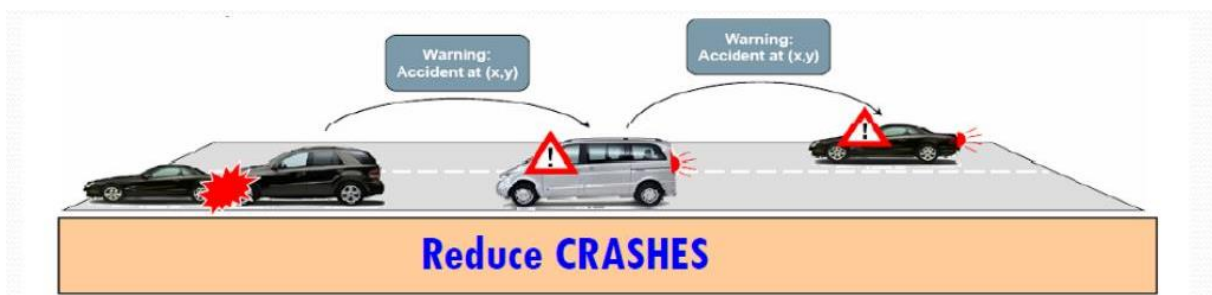
2.1. VANET et la sécurité dans VANET

Vehicular Ad-Hoc Network ou VANET est un type de réseau particulier qui permet à des véhicules de communiquer entre eux ainsi qu'avec des équipements fixes au bords de la route. Ce réseau est constitué de nœuds mobiles appelés OBU (On-Board Unit) qui sont des unités de communications embarquées dans les véhicules et de nœuds fixes appelés RSU (RoadSide Unit) qui sont des unités de communications fixes au bord de la route. Dans ce type de réseau, les informations sont transitées en passant par un ou plusieurs nœuds de manière sans fil afin d'atteindre un ou plusieurs nœuds distants. Un véhicule peut transmettre vers un autre véhicule (V2V), il peut aussi transmettre vers une infrastructure (V2I) ou alors peut transmettre à n'importe quel nœud (V2X). Les unités de communications fixes sont utilisées pour relayer des informations à des véhicules éloignés et qui peuvent parfois être hors de portée du véhicule qui transmet. VANET possède la contrainte d'un réseau qui change de topologie constamment du fait des nombreuses connexions et déconnexions de différents nœuds au sein d'un réseau à cause de la mobilité des nœuds.



Architecture de VANET [Hasrouny17]

La communication entre véhicule permet d'apporter de la sécurité au trafic car les nœuds vont pouvoir analyser le trafic en temps réel afin de réagir au plus vite aux différents événements routiers. Deux types de message ont été standardisés par l'ETSI (European Telecommunications Standards Institute) afin d'échanger des informations routières coopérativement : le premier message est le message CAM (Cooperative Awareness Message), ce message est envoyé périodiquement par chaque véhicule afin de donner sa position, sa vitesse, son type (véhicule normal, véhicule prioritaire...), son accélération... Ces messages périodiques permettent de connaître la situation routière précisément et en temps réel, ce qui permet d'adapter le trafic en conséquence. Cela permet par exemple d'anticiper le freinage d'un véhicule qui roule devant en étant prévenu à l'avance, cela permet d'éviter les collisions ainsi que les effets d'accordéon dans les bouchons par exemple. Le deuxième message est le message DENM (Decentralized Environmental Notification Message), ce message est généré lorsqu'un événement spécifique est détecté (bouchons, collision, accident, obstacle sur la route...). Cela permet d'avertir les véhicules concernés de près ou de loin par l'événement le plus rapidement possible afin de les faire réagir au mieux. Par exemple, le véhicule sera averti des différents événements routiers dans la zone comme un accident ou un bouchon et pourra décider d'un chemin différent pour les éviter. La sécurité et le confort de conduite sont donc améliorés grâce à ces deux types de messages.



Source : <https://www.slideserve.com/trilby/les-r-seaux-v-hiculaires-vanet>

Le problème avec un tel réseau est qu'un véhicule peut recevoir des messages provenant de beaucoup de nœuds différents qui se connectent et se déconnectent du réseau en permanence à cause de la mobilité des nœuds. La sécurité d'un réseau mobile comme celui-ci est difficile à mettre en œuvre. Cependant, si la sécurité n'est pas prise en compte, il est très facile pour un véhicule malveillant d'envoyer de fausses informations à plusieurs autres véhicules. Par exemple, un véhicule malveillant pourrait se faire passer pour un véhicule prioritaire afin de voyager plus vite. Les attaques peuvent être de diverses natures, par exemple, un nœud malveillant pourrait tout simplement détruire tous les paquets qu'il reçoit et donc perturber la transmission des informations dans le réseau (Black hole attack). Un nœud malveillant pourrait aussi faire router tous les paquets vers lui en faisant croire qu'il a le plus court chemin de routage (Sink hole attack) ou encore le réseau peut se faire inonder de messages provenant d'un nœud malveillant perturbant les autres communications (Flooding attack).

Un grand nombre d'attaques sont répertoriés dans [Hasrouny17] :

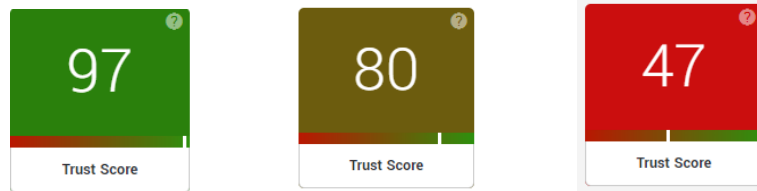
Attacks on	Attack name	Attack on VANET communication mode
Wireless interface	<ul style="list-style-type: none"> - Location Tracking - DoS, DDoS - Sybil - Malware and spam. - Tunnelling, Blackhole, Greyhole. - MIM - Brute force - DoS 	V2V
Hardware and software	<ul style="list-style-type: none"> - Spoofing and forgery. - Cheating with position info (GPS spoofing). - Message suppression/alteration/fabrication. - Replay - Masquerade - Malware and spam - MIM - Brute force - Sybil - Injection of erroneous messages (bogus info). - Tampering hardware - Routing, Blackhole, wormhole and Greyhole. - Timing. 	V2V, V2I
Sensors input in vehicle	<ul style="list-style-type: none"> - Cheating with position info (GPS spoofing) - Illusion attack - Jamming attack 	V2V
Infrastructure	<ul style="list-style-type: none"> - Session hijacking - DoS, DDoS - Unauthorized access - Tampering hardware - Repudiation - Spoofing, impersonation or masquerade 	V2I and V2V

2.2. Trust Management

Pour améliorer la sécurité d'un réseau de type VANET, en plus des solutions de cybersécurité classiques comme la cryptographie, l'authentification, les mécanismes anti Dos... ; la solution de Trust Management a été proposée. Cette solution consiste à attribuer une note de confiance à chaque véhicule qui permet de quantifier la confiance que l'on peut avoir envers lui. Quand un nœud reçoit un message provenant d'un autre nœud, il va consulter la note de confiance de celui-ci et si cette note est trop basse comparée à un seuil, alors le message sera jeté, sinon, le message sera lu et pris en compte. Cette note est comparable à la confiance dans une société, chaque nœud possède son avis sur les autres nœuds s'ils les ont déjà rencontrés, et donc lui fait plus ou moins confiance. Cependant, s'il ne connaît pas ce nœud, il va lui attribuer une note de confiance de départ. Cette note de départ peut être attribuée de deux manières : soit la note de départ est définie de manière fixe, c'est-à-dire que l'on donne toujours la même note de départ aux nouveaux nœuds ou alors on peut la définir en fonction de différents paramètres comme par exemple, sa puissance de calcul ou son type, la note de départ sera donc différente selon le nœud.

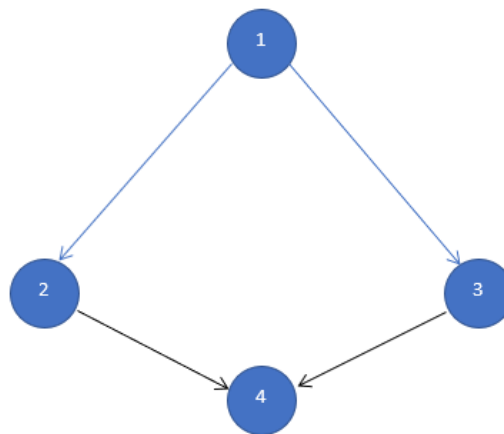
Chaque nœud attribue donc une note aux autres nœuds du réseau selon l'expérience qu'il a eu avec ce nœud et selon le comportement de ce dernier. Cette note est calculée constamment en fonction de différents paramètres comme la vitesse, la position ou le type du véhicule ou encore selon la manière dont il se comporte dans le réseau, par exemple, s'il ne transmet pas correctement les informations qu'il doit relayer ou s'il envoie de fausses informations, sa note va diminuer, de même s'il ne respecte pas la distance entre deux véhicules. Au contraire, s'il transmet rapidement et correctement les informations qu'il doit

relayer ou qu'il avertit d'un obstacle ou encore s'il respecte les vitesses et les distances entre véhicules, sa note augmentera.



Source : <https://www.forexbrokers.com/trust-score>

Pour évaluer la confiance que l'on attribue à une personne, on peut se fier uniquement à notre expérience avec cette personne ou alors demander l'avis de nos amis sur cette personne afin de savoir ce que nos amis pensent de celle-ci, grâce à leurs avis, nous pouvons donc décider de la confiance que l'on attribue à cette personne. La première méthode est le Direct Trust et la deuxième est l'Indirect Trust. En Trust Management, ces deux méthodes sont appliquées et peuvent aussi être combinées entre elles en appliquant une pondération, c'est-à-dire que par exemple, si on connaît très bien le nœud en question, on prendra plus en compte notre propre avis plutôt que celui des autres, en revanche, si on connaît très peu un nœud, on prendra plus en considération l'avis des autres nœuds. Pour utiliser l'Indirect Trust, il faut prendre en considération la confiance que l'on attribue aux nœuds à qui on demande l'avis. En effet, on prendra plus en compte l'avis d'un nœud auquel on a très confiance plutôt que l'avis d'un nœud auquel on a moyennement confiance.

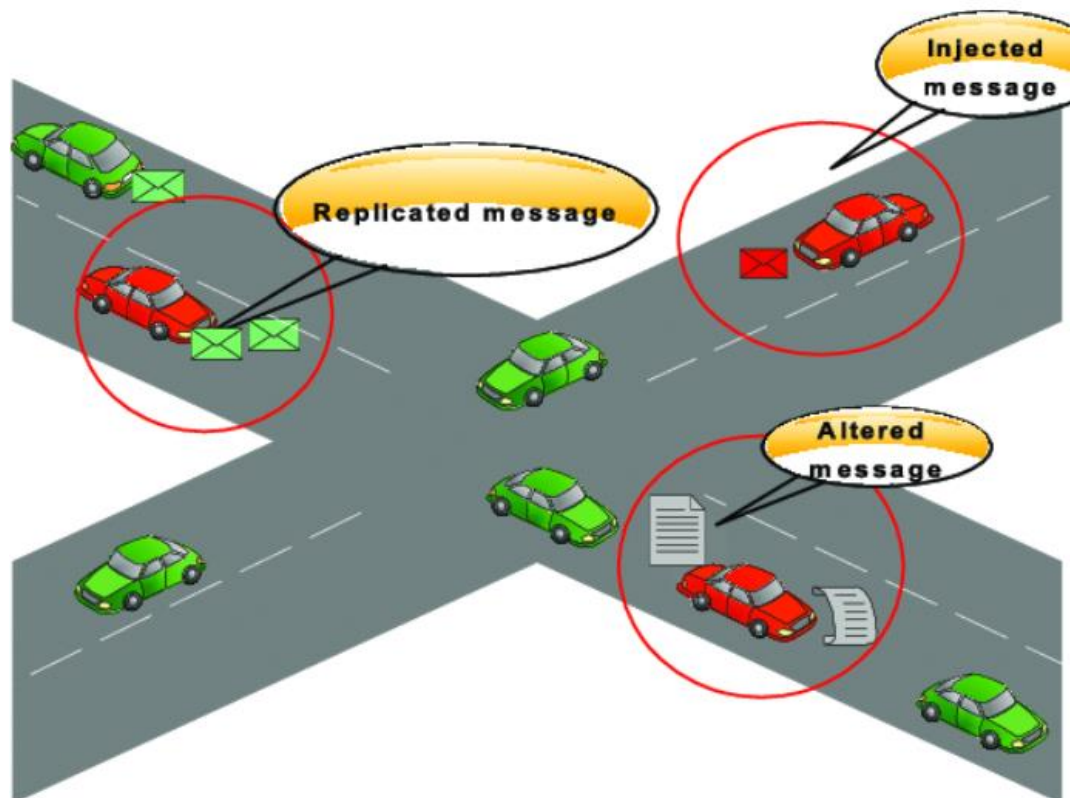


Sur ce schéma, le nœud 1 demande l'avis du nœud 2 et 3 sur le nœud 4.

La solution de Trust Management a pour but d'apporter de la sécurité dans un réseau mobile difficile à sécuriser mais cette solution doit respecter certains critères et contrer plusieurs attaques possibles sur les notes de confiance : par exemple, un nœud malveillant pourrait alterner entre avoir un bon comportement et un mauvais comportement pour garder une note assez haute pour pouvoir communiquer avec les autres nœuds mais continuer à effectuer de mauvaises actions, il faut donc pénaliser les mauvaises actions plus sévèrement. Il y a donc plusieurs contraintes auquel doit répondre la solution de Trust

Management puisque plusieurs détournements de cette solution sont envisageables (par exemple un nœud malveillant peut mal noter un nœud qui se comporte correctement ou inversement afin de perturber le système de confiance du réseau), il faut donc trouver des compromis dans la façon de calculer les notes et dans la façon de pénaliser et récompenser les actions afin de gérer ces contraintes.

Les types d'attaques sont variés et l'attaquant peut avoir des motivations variées, comme par exemple, des conducteurs égoïstes qui cherchent à améliorer leur propre confort de conduite au détriment des autres usagers, ou alors un attaquant malveillant qui cherche à perturber le trafic ou à créer un accident, il peut aussi y avoir des personnes qui cherchent à récupérer des informations sur les différents véhicules.



Source : https://www.researchgate.net/figure/Replayed-altered-and-injected-messages-attacks_fig4_311910271

Pour faire fonctionner ce modèle de Trust Management, il faut dans un premier temps, collecter les informations nécessaires à l'établissement de la note de confiance. Il faut donc sélectionner les paramètres qui serviront à établir cette note comme par exemple la position, la vitesse, le type de véhicule, le carburant utilisé ou encore, son temps de réponse, le nombre de paquets qu'il émet et reçoit... Ces paramètres peuvent donc être récoltés par le nœud lui-même (Direct Trust) ou alors peut demander les recommandations des autres nœuds pour établir la note (Indirect Trust).

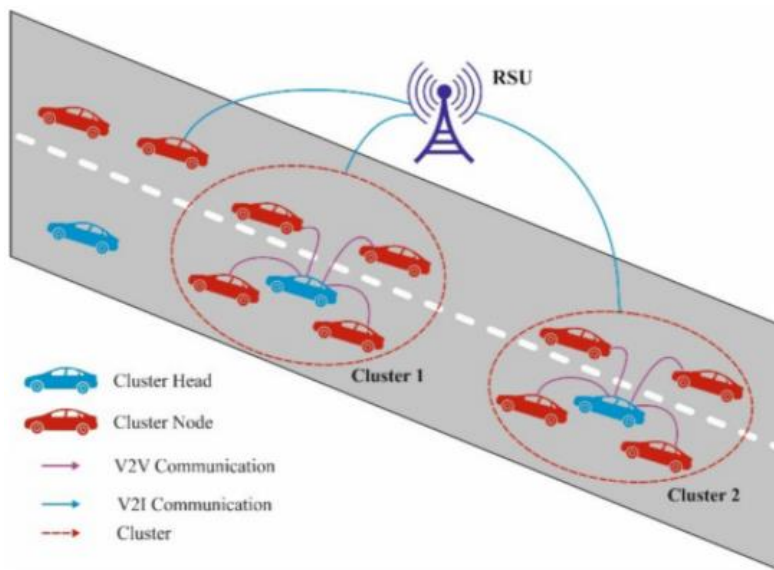
Ensuite, il est nécessaire de choisir une manière de calculer la note de confiance en fonction des différents paramètres choisis précédemment. La manière de calculer cette note doit être choisie en fonction des contraintes que l'on a et de l'utilisation que l'on veut en faire. Les

modèles les plus utilisés sont statistiques, probabilistes, logique floue (Fuzzy Logic), Machine Learning, Théorie des jeux et des modèles hybrides. Chaque modèle possède des avantages et des inconvénients correspondant à différentes utilisations que l'on veut faire. Par exemple, le Machine Learning sera efficace dans le cas où de nombreux paramètres sont utilisés pour calculer la note, cependant, il sera moins efficace s'il faut que la note soit calculée vite car le modèle est lent. Il est nécessaire de trouver un compromis entre la sécurité du modèle qui nécessite un calcul fiable de la note de confiance et la rapidité du modèle qui assure une fluidité du réseau.

Enfin, il faut décider de la façon dont la note va être mise à jour. En effet, il est nécessaire de pouvoir changer la note des différents nœuds selon les actions que celui-ci entreprend. Il y a plusieurs façons de mettre à jour la note de confiance, la première consiste à mettre à jour la note à chaque fois qu'il se produit un événement (demande d'accès, envoi d'information...). La deuxième méthode consiste à mettre à jour la note périodiquement au lieu d'attendre un événement précis.

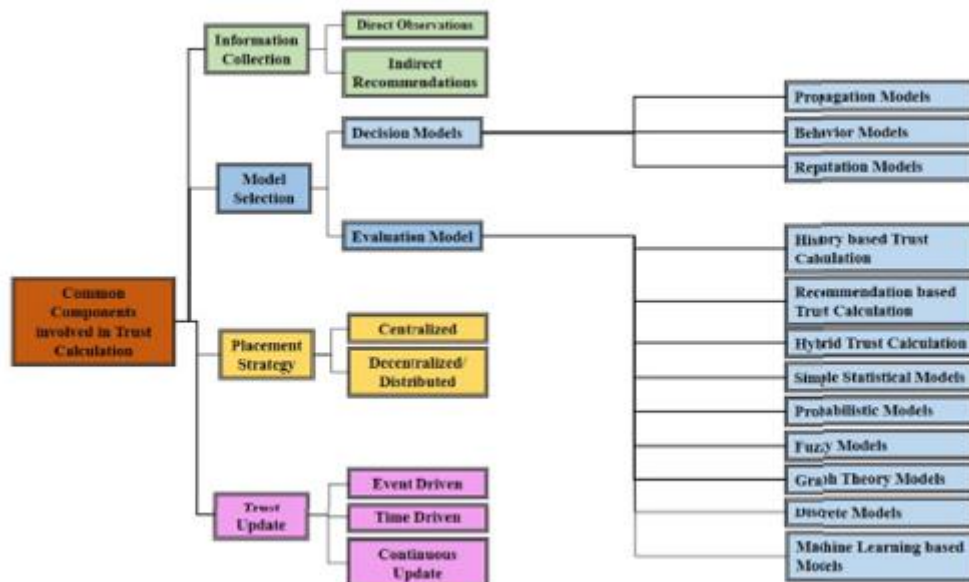
La façon de gérer ce modèle de Trust Management peut se faire de manière distribué, semi-distribué ou alors centralisé. Dans l'approche distribué, tous les nœuds doivent effectuer chacun toutes les étapes nécessaires au fonctionnement du modèle de Trust Management, c'est-à-dire, la collecte d'information, le calcul de la note de confiance, la mise à jour, le stockage et la diffusion. Pour l'approche semi-distribué, certains nœuds sont choisis pour effectuer les tâches de calcul, de mise à jour, de stockage et de diffusion dans un groupe en utilisant les données que les autres nœuds auront récupéré. Dans cette approche, les nœuds sélectionnés sont souvent ceux ayant la plus grande puissance de calcul dans un groupe. Il y a donc plusieurs groupes de nœuds qui comportent chacun un nœud central. Dans l'approche centralisé, il y a une seule entité centrale qui effectue les tâches de calcul, de stockage, de mise à jour et de diffusion.

Exemple de l'approche semi-distribu   :



Source : <https://www.mdpi.com/2079-9292/9/9/1358/htm>

R  sum   de la gestion du mod  le de Trust Management [Altaf19] :



3. Déroulement du PFE

3.1. Etudes bibliographiques

Dans un premier temps, pour m'imprégner du sujet, j'ai lu des enquêtes ([Hasrouny17] ; [Sharma20] ; [Hussain20] ; [Liu16] ; [Su2021]) traitant des problématiques liées à VANET surtout en matière de sécurité et traitant de Trust Management et des solutions apportés par le Trust Management dans VANET, j'ai également lu les normes des deux messages CAM et DENM afin de connaître en profondeur le type de message qu'il y a besoin de sécuriser. Grâce à ces différents documents, j'ai pris connaissance des différentes caractéristiques d'un réseau VANET, les points forts et les points faibles d'un tel réseau ainsi que les solutions pour combler les failles de sécurité. Cela m'a aussi permis de rentrer dans le sujet en m'imprégnant des termes techniques et des acronymes. J'ai pu me rendre compte des travaux qui ont déjà été effectués sur les différents aspects de VANET et de Trust Management, les problématiques et les solutions possibles qui ont été proposées pour améliorer la sécurité ou la rapidité du réseau.

3.2. Ns-3

Ensuite, afin de simuler un réseau de type VANET, le premier outil que j'ai utilisé est Ns-3. Ns-3 est un simulateur de réseau à événement discret qui permet de simuler des réseaux tels que les réseaux Wi-Fi, réseaux Ethernet, réseaux mobiles ad-hoc ... Ce simulateur permet de simuler les protocoles IP, TCP, UDP... Ns-3 fournit des modèles permettant de comprendre le fonctionnement et les performances des réseaux de données par paquets, il permet aussi de réaliser des expériences de simulation grâce au moteur de simulation.

Afin de comprendre au mieux cet outil, j'ai tout d'abord lu de la documentation sur Ns-3 (NS-3) ; [Campanile20]) pour savoir si ce simulateur pouvait correspondre aux besoins de ce PFE. Ns-3 est écrit en c++ et en python, il est disponible sur Linux, Mac OS et Windows, il est open-source et gratuit. Il y a de la documentation et des tutoriels pour son installation et son utilisation. L'outil semble donc correspondre aux besoins de ce PFE.

Ns-3 utilise des classes qui permettent de définir les différents paramètres de la simulation comme par exemple la classe nodes qui regroupe les nœuds qui seront utilisés dans la simulation comme par exemple un ordinateur ou un capteur. La classe Channel permet de définir le type de réseau que l'on veut faire, par exemple, Point to point, Csma ou encore Wi-fi. La classe adresse permet de définir le type d'adresse ainsi que son nom.

J'ai donc installé Ns-3 et suivi plusieurs tutoriels afin de comprendre les exemples fournis ainsi que les différentes classes et paramètres du simulateur pour ensuite faire ma propre simulation de VANET. Des codes exemples sont donnés dans le dossier d'installation, ces codes permettent de comprendre la base de Ns-3. C'est pourquoi j'ai suivi des tutoriels pour comprendre les trois codes de bases donnés, first.cc, second.cc et third.cc (Voir annexe 1 pour les codes). Le premier code permet de simuler un réseau simple qui fait communiquer deux nœuds en mode point à point, le deuxième permet de simuler un réseau qui fait communiquer deux nœuds en mode point à point et 4 nœuds en mode LAN (dont un nœud qui fait les deux), le troisième réseau permet de simuler un réseau qui utilise une communication Wifi, point à point et en LAN.

Nous allons voir maintenant, le code first.cc :

La topologie du réseau simulé par first.cc est :

```
// Default Network Topology
//
//      10.1.1.0
// n0 ----- n1
//      point-to-point
//
```

Au début du code, il faut définir les nœuds dont on a besoin pour le réseau :

```
NodeContainer p2pNodes;
p2pNodes.Create (2);
```

Puis il faut définir le type de canal (point à point, csma, wifi...) et définir les paramètres de ce canal comme le débit ou le délai de transmission :

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

Ensuite, il faut définir le matériel qui sera attaché aux nœuds, ici on simule une carte d'interface réseau qui est attaché aux nœuds. On doit donc spécifier les nœuds et le canal, ce qui permet aux nœuds d'être associés au canal voulu :

```
NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

Ensuite, on installe sur les nœuds, la pile protocolaire TCP/IP, ce qui permet aux nœuds d'utiliser les protocoles Internet :

```
InternetStackHelper stack;
stack.Install (nodes);
```

Puis, il faut définir les adresses IP que l'on va utiliser ainsi que le masque réseau et appliquer ces adresses aux différents appareils du réseau :

```
Ipv4AddressHelper address;  
address.SetBase ("10.1.1.0", "255.255.255.0");  
  
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

On crée ensuite un serveur UDP sur le port 9 :

```
UdpEchoServerHelper echoServer (9);
```

On installe le serveur sur le nœud numéro 1, puis on définit quand le serveur va démarrer et s'arrêter :

```
ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));  
serverApps.Start (Seconds (1.0));  
serverApps.Stop (Seconds (10.0));
```

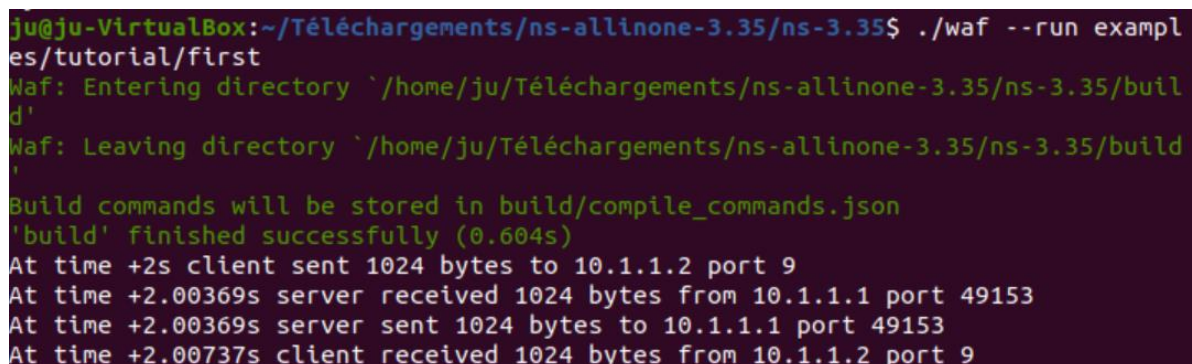
Ensuite, on crée le client, pour cela, il faut l'adresse IP du serveur et le port, puis définir certains autres paramètres :

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);  
echoClient.SetAttribute ("MaxPackets", UIntegerValue (1));  
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));  
echoClient.SetAttribute ("PacketSize", UIntegerValue (1024));
```

Il faut ensuite installer le client sur un nœud (ici, le nœud numéro 0) :

```
ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));  
clientApps.Start (Seconds (2.0));  
clientApps.Stop (Seconds (10.0));
```

L'exécution de first.cc donne :



```
ju@ju-VirtualBox:~/Téléchargements/ns-allinone-3.35/ns-3.35$ ./waf --run examples/tutorial/first  
Waf: Entering directory `/home/ju/Téléchargements/ns-allinone-3.35/ns-3.35/build'  
Waf: Leaving directory `/home/ju/Téléchargements/ns-allinone-3.35/ns-3.35/build'  
Build commands will be stored in build/compile_commands.json  
'build' finished successfully (0.604s)  
At time +2s client sent 1024 bytes to 10.1.1.2 port 9  
At time +2.00369s server received 1024 bytes from 10.1.1.1 port 49153  
At time +2.00369s server sent 1024 bytes to 10.1.1.1 port 49153  
At time +2.00737s client received 1024 bytes from 10.1.1.2 port 9
```

Nous pouvons voir que le client envoie un paquet pour se connecter au serveur, le serveur reçoit ce paquet et envoie un acquittement que le client reçoit.

La topologie du réseau simulé par second.cc est :

```
// Default Network Topology
//
//      10.1.1.0
// n0  ----- n1    n2    n3    n4
//      point-to-point |    |    |    |
//                      =====
//                      LAN 10.1.2.0
```

Pour le code de second.cc, il est assez proche de first.cc mais les nœuds sont définis en deux fois (les nœuds point à point et les nœuds csma (dont un qui est en commun aux deux)). Puis les différentes étapes décrites pour le code first.cc sont effectuées pour les deux types de nœuds. Le serveur est installé sur le dernier nœud csma (ici, c'est n4) et le client est installé sur n0.

Une ligne de code supplémentaire permet de générer une table de routage pour chaque nœud :

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Cela permet au message qui vient de n0 d'être routé jusque n4 en passant par les nœuds n1, n2 et n3.

L'exécution de second.cc donne :

```
ju@ju-VirtualBox:~/Téléchargements/ns-allinone-3.35/ns-3.35$ ./waf --run examples/tutorial/second
Waf: Entering directory `/home/ju/Téléchargements/ns-allinone-3.35/ns-3.35/build'
Waf: Leaving directory `/home/ju/Téléchargements/ns-allinone-3.35/ns-3.35/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.601s)
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.0078s server received 1024 bytes from 10.1.1.1 port 49153
At time +2.0078s server sent 1024 bytes to 10.1.1.1 port 49153
At time +2.01761s client received 1024 bytes from 10.1.2.4 port 9
```

La topologie du réseau simulé par third.cc est :

```
// Default Network Topology
//
// Wifi 10.1.3.0
//
// *      *      *      AP
// |      |      |      |
// n5     n6     n7     n0  10.1.1.0
// ----- n1     n2     n3     n4
// point-to-point |     |     |     |
//                =====
//                LAN 10.1.2.0
```

Le code de third.cc est assez similaire au code de second.cc, la différence est qu'il faut ajouter de nombreux paramètres pour utiliser la communication en wifi.

L'exécution de third.cc donne :

```
ju@ju-VirtualBox:~/Téléchargements/ns-allinone-3.35/ns-3.35$ ./waf --run examples/tutorial/third
Waf: Entering directory `/home/ju/Téléchargements/ns-allinone-3.35/ns-3.35/build'
Waf: Leaving directory `/home/ju/Téléchargements/ns-allinone-3.35/ns-3.35/build'
Build commands will be stored in build/compile_commands.json
'build' finished successfully (0.601s)
At time +2s client sent 1024 bytes to 10.1.2.4 port 9
At time +2.01799s server received 1024 bytes from 10.1.3.3 port 49153
At time +2.01799s server sent 1024 bytes to 10.1.3.3 port 49153
At time +2.03367s client received 1024 bytes from 10.1.2.4 port 9
```

Un outil graphique est accompagné de Ns-3 pour pouvoir voir les simulations graphiquement, cet outil se nomme NetAnim. Cependant, je n'ai pas eu l'occasion de tester cet outil avec Ns-3.

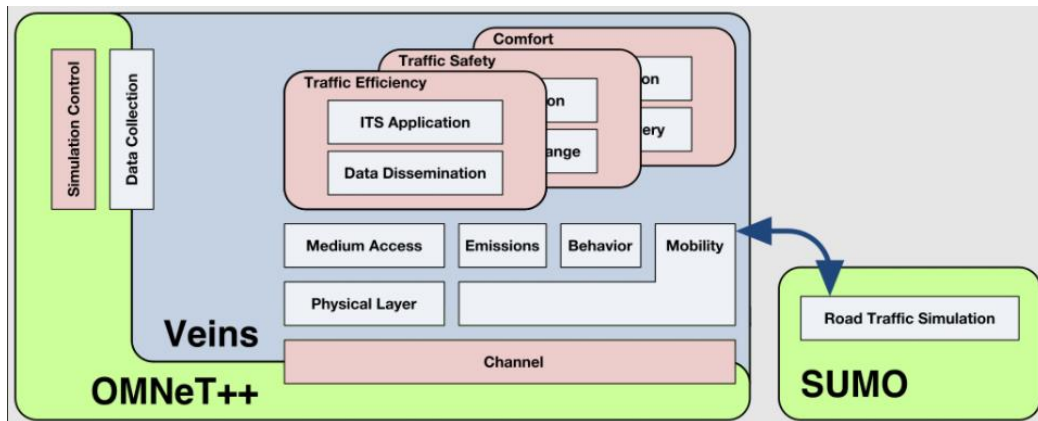
Il y a aussi la possibilité de mettre une ligne de code qui permet de générer un fichier .pcap qui contient les trames échangées durant la simulation, ce fichier peut alors être analysé grâce à un outil comme Wireshark.

A partir des trois codes exemples, j'ai donc essayé de créer une simulation d'un réseau de type VANET. J'ai donc repris une partie du code third.cc afin de faire un réseau Wifi qui ressemble à un réseau de type VANET, ce code correspond plutôt bien à un réseau mobile qui communique sans fil car une classe mobility est appliqué aux nœuds wifi ce qui permet aux nœuds de bouger et grâce à NetAnim, il est possible de voir les nœuds bouger tout en envoyant des informations.

Le problème avec Ns-3, est que c'est un outil qui est difficile à comprendre et à prendre en main. De plus, la documentation est difficile d'accès. Un autre problème se pose, il est nécessaire de combiner un simulateur de réseau avec un simulateur de trafic pour effectuer la simulation du réseau VANET, cependant, l'association des deux n'est pas prise en compte avec Ns-3.

3.3. Veins

C'est pourquoi un autre outil m'a été proposé ensuite, il s'agit de Veins. Veins est un framework open-source qui permet d'effectuer des simulations de réseaux véhiculaires. En effet, Veins combine un simulateur de réseau qui se nomme Omnet++ et un simulateur de trafic qui se nomme Sumo.



Source : <https://veins.car2x.org/documentation/>

J'ai donc cherché à comparer les deux outils, Ns-3 étant recommandé pour les utilisateurs plus expérimentés dans le domaine et Veins étant décrit comme plus facile d'accès. De plus, Veins utilise Omnet++ comme simulateur de réseau, qui est plus récent que Ns-3 et mieux documenté, il est donc plus facile à prendre en main.

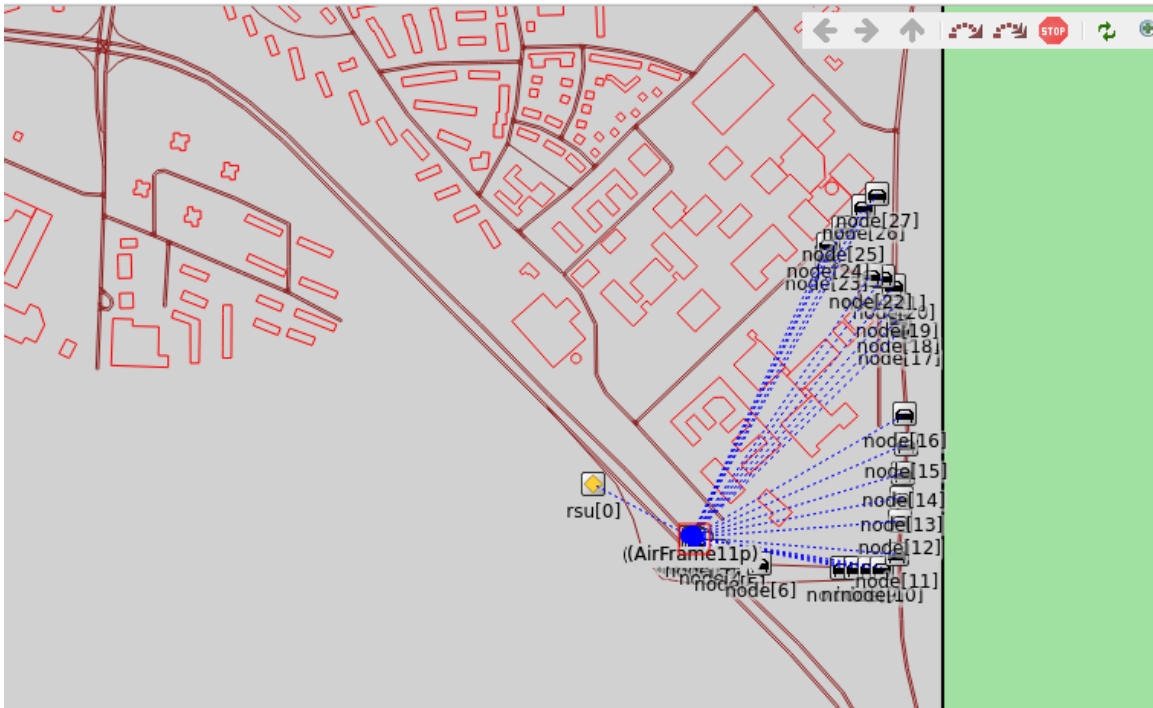
Veins étant spécialisé dans la simulation de réseaux véhiculaires, cet outil est parfaitement adapté pour ce projet car il combine déjà le simulateur de réseau et le simulateur de trafic. De plus, il existe une machine virtuelle sur laquelle les deux simulateurs sont déjà installés et peuvent communiquer entre eux. Il y a aussi des exemples déjà fait qui permettent de comprendre les interactions entre Sumo et Omnet++ ainsi que les différents paramètres qui rentrent en jeu.

Pour la suite, j'ai donc installé la machine virtuelle de Veins qui se nomme instant-veins car cet outil semble plus adapté aux besoins de ce PFE.

J'ai tout d'abord analysé les différents fichiers présents sur la machine virtuelle, il y a plusieurs projets déjà présents :

```
▶ > inet [inet v4.2.2-work]
▶ > src [veins veins-veins-5.1-work]
▶ > veins [veins veins-veins-5.1-work]
▶ > veins_inet [veins veins-veins-5.1-work]
```


Le projet auquel je me suis intéressé en premier lieu est le projet veins, j'ai ensuite fait tourner l'exemple présent dans le dossier ce projet :



Nous pouvons voir que dans cet exemple, il y a les deux types de nœuds spécifiques de VANET, les véhicules (OBU) sont représentés à droite et sont nommés node[x] et il y a un RSU au bord de la route. Nous pouvons voir que le véhicule de tête envoie un message aux autres voitures ainsi qu'au RSU. Une grande carte a été générée dans cet exemple et les véhicules apparaissent à l'endroit où on voit la voiture nommée node[27], puis elles se dirigent vers le RSU. Au bout d'un moment, la voiture de tête s'arrête et envoie un message en broadcast.

Dans la simulation, on peut voir en bas les logs, dans lesquels on retrouve par exemple des événements de la simulation ou des initialisations de composants, on peut voir ici une initialisation de node[27] ainsi que tous ses paramètres. On trouve aussi dans les logs les variables d'environnements que l'on décide d'afficher :

```

Initializing module RSUExampleScenario.node[27].nic.phy80211p, stage 1
Initializing module RSUExampleScenario.node[27].nic.mac1609_4, stage 1
Initializing module RSUExampleScenario.node[27].veinsmobility, stage 1
** Event #85 t=83 RSUExampleScenario.manager (TraCIScenarioManagerLaunchd, id=6) on selfmsg step (omnetpp::cMessage, id=15)
** Event #86 t=84 RSUExampleScenario.manager (TraCIScenarioManagerLaunchd, id=6) on selfmsg step (omnetpp::cMessage, id=15)
** Event #87 t=84 RSUExampleScenario.node[0].nic.mac1609_4 (Mac1609_4, id=19) on (veins::TraCIDemo11pMessage, id=414)
** Event #88 t=84.000001 RSUExampleScenario.node[0].nic.mac1609_4 (Mac1609_4, id=19) on selfmsg next Mac Event (omnetpp::cMessage, id=29)
** Event #89 t=84.000001 RSUExampleScenario.node[0].nic.mac1609_4 (Mac1609_4, id=19) on Radio switching over (omnetpp::cMessage, id=417)
** Event #90 t=84.000002 RSUExampleScenario.node[0].nic.phy80211p (PhyLayer80211p, id=18) on (veins::Mac80211Pkt, id=415)
** Event #91 t=84.000002016695 RSUExampleScenario.node[1].nic.phy80211p (PhyLayer80211p, id=24) on (veins::AirFrame11p, id=421)
** Event #92 t=84.000002016695 RSUExampleScenario.node[1].nic.mac1609_4 (Mac1609_4, id=25) on ChannelStatus (omnetpp::cMessage, id=448)
** Event #93 t=84.000002033674 RSUExampleScenario.node[2].nic.phy80211p (PhyLayer80211p, id=30) on (veins::AirFrame11p, id=422)

```

On peut aussi voir les messages échangés par les différents nœuds dans la simulation :

84.000002	node[0] --> rsu[0]	420	22002
84.000002	node[0] --> node[1]	421	22002
84.000002	node[0] --> node[2]	422	22002
84.000002	node[0] --> node[3]	423	22002
84.000002	node[0] --> node[4]	424	22002
84.000002	node[0] --> node[5]	425	22002
84.000002	node[0] --> node[6]	426	22002
84.000002	node[0] --> node[7]	427	22002
84.000002	node[0] --> node[8]	428	22002
84.000002	node[0] --> node[9]	429	22002

On trouve sur la gauche de la fenêtre de simulation, les différents éléments qui composent la simulation :

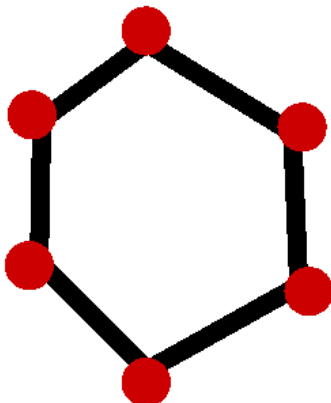
- ▼ RSUExampleScenario (RSUExampleScenario) id=1
 - ◆ playgroundSizeX (cPar) 2500m
 - ◆ playgroundSizeY (cPar) 2500m
 - ◆ playgroundSizeZ (cPar) 50m
 - ▶ obstacles (ObstacleControl) id=2
 - ▶ annotations (AnnotationManager) id=3
 - ▶ connectionManager (ConnectionManager) id=4
 - ▶ world (BaseWorldUtility) id=5
 - ▶ manager (TraCIScenarioManagerLaunchd) id=6
 - ▶ roadsCanvasVisualizer (RoadsCanvasVisualizer) id=7
 - ▶ rsu[0] (RSU) id=8
 - ▶ canvas (cCanvas) 3 toplevel figure(s)
 - ▶ node[0] (Car) id=14
 - ▶ node[1] (Car) id=20
 - ▶ node[2] (Car) id=26

3.4. SUMO

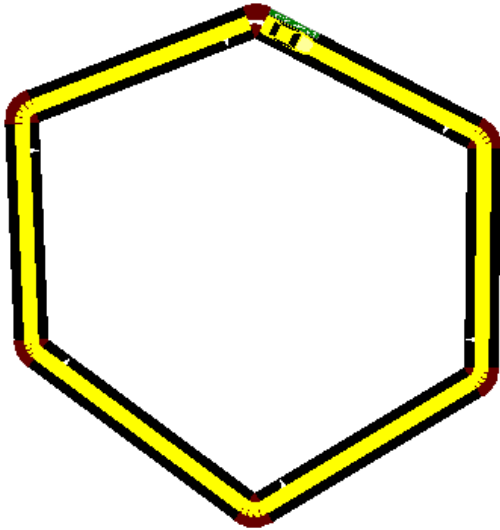
Ensuite, afin de comprendre comment fonctionne le simulateur de trafic, j'ai décidé de suivre des tutoriels sur Sumo. Le tutoriel que j'ai suivi consiste à faire une route en cercle dans laquelle deux véhicules circulent dans le sens des aiguilles d'une montre.

Pour créer une simulation de trafic avec Sumo, j'ai utilisé le logiciel NetEdit qui permet de créer la simulation en utilisant des outils graphiques et avec la possibilité d'éditer le code source.

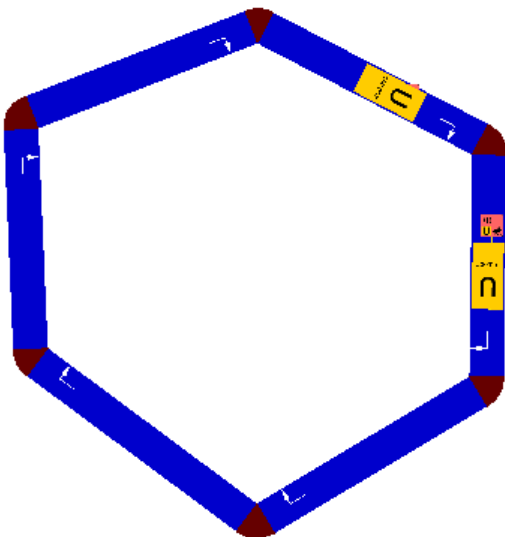
Dans un premier temps, il faut déjà créer la route grâce à des arêtes :



Il faut ensuite nommer les arêtes et définir la route entière en comme étant le regroupement de toutes les arêtes. Puis on ajoute un véhicule sur une arête afin de définir le lieu où le véhicule va apparaître :



Il faut ensuite ajouter des routeurs sur les arêtes 1 et 2 et ajouter un bout de code dans un des fichiers produits par le logiciel afin de définir le sens de rotation des véhicules :



En effet, plusieurs fichiers sont produits quand on enregistre les modifications que l'on a apporté à la simulation :



Le fichier .rou.xml gère les informations relatives aux véhicules, par exemple, le point de départ, son identifiant ou la route qu'il empreinte. Le fichier .net.xml gère la topologie de la route. Le fichier .add.xml gère tous les composants additionnels de la simulation comme par exemple des arrêts de bus ou des routeurs.













Après avoir appris le fonctionnement du simulateur de trafic, j'ai voulu apprendre à utiliser Omnet++, le simulateur de réseau compris dans Veins.

Cependant, Omnet++ est basé sur le langage c++, alors afin de comprendre au mieux les codes qui composent cet outil, je me suis renseigné et fait plusieurs tutoriels sur ce langage dans le but de me familiariser avec celui-ci.







3.5. [Omnet++](#)

Après avoir appris les bases du langage, j'ai essayé de comprendre les fichiers qui composent les différents projets compris dans la machine virtuelle de Veins. J'ai commencé par analyser les fichiers headers qui correspondent aux classes utilisées pour envoyer des messages, comme par exemple la classe Cmessage ou encore la classe Cpacket.

Je suis également allé voir les fichiers utilisés pour la communication, comme par exemple le dossier message dans lequel sont présents tous les types de messages utilisés par le projet Veins :

- ▶  AckTimeOutMessage_m.cc
- ▶  AckTimeOutMessage_m.h
- ▶  AirFrame11p_m.cc
- ▶  AirFrame11p_m.h
- ▶  BaseFrame1609_4_m.cc
- ▶  BaseFrame1609_4_m.h
- ▶  DemoSafetyMessage_m.cc
- ▶  DemoSafetyMessage_m.h
- ▶  DemoServiceAdvertisement_m.cc
- ▶  DemoServiceAdvertisement_m.h
- ▶  Mac80211Ack_m.cc
- ▶  Mac80211Ack_m.h

Nous pouvons remarquer l'utilisation du protocole 802.11p qui est un protocole dérivé du protocole 802.11 et est utilisé pour la communication véhiculaire :

- ▼  ieee80211p
 - ▶  DemoBaseApplLayerToMac1609_4
 - ▶  Mac1609_4.cc
 - ▶  Mac1609_4.h
 - ▶  Mac80211pToPhy11pInterface.h
 -  Mac1609_4.ned

Nous pouvons aussi remarquer que Veins définit des couches protocolaires comme la couche physique, la couche mac et la couche application.

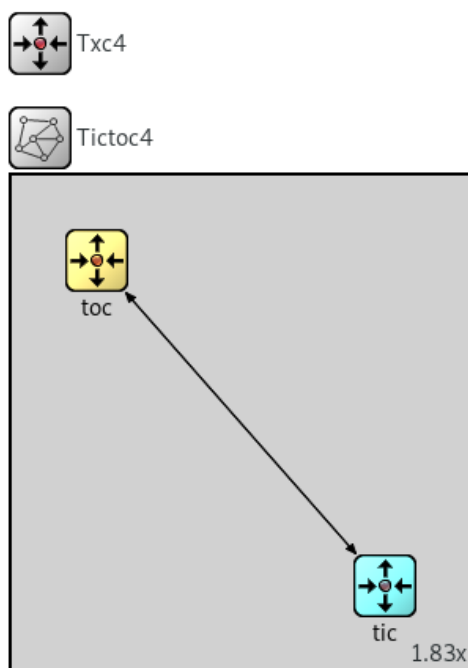
Après avoir consulté plusieurs fichiers du projet veins afin de comprendre comment fonctionne l'exemple qui est donné, j'ai décidé de suivre des tutoriels sur Omnet++ dans le but de comprendre en profondeur comment fonctionne ce simulateur et pour apprendre à créer mon propre réseau ou à modifier un réseau existant.

Dans la documentation d'Omnet++, il y a une partie tutoriel dans laquelle on apprend les bases d'Omnet++ progressivement grâce à plusieurs exemples guidés. On y apprend la fonction des différents fichiers de bases nécessaires pour construire la simulation.

Le premier fichier de base est le Network Description File (NED), ce fichier permet de définir les composants nécessaires et à les assembler pour constituer le réseau.

Nous allons voir un exemple de fichier NED, les fichiers NED peuvent se lire et s'éditer en mode source ou en mode design.

Par exemple, le mode design du fichier NED du réseau tictoc4 est le suivant :



Nous pouvons voir un objet Txc4 qui représente le type de composant utilisé par le réseau ainsi qu'un bloc représentant le réseau lui-même, on peut voir au-dessus du bloc, le nom du réseau : Tictoc4. Et dans ce bloc, on peut voir les deux composants du réseau (des composants de type Txc4), l'un nommé tic et l'autre nommé toc.

Nous pouvons passer en mode source pour avoir plus de détails, le début du fichier se présente comme ceci :

```
simple Txc4
{
    parameters :
        bool sendMsgOnInit = default(false);
        int limit = default(2);
        @display("i=block/routing");
    gates:
        input in;
        output out;
}
```

Nous pouvons voir au début la définition d'un composant qui se nomme Txc4 et qui a pour paramètre un booléen qui représente l'envoi d'un message à l'initialisation (qui est par défaut à false), ainsi qu'un entier qui correspond à la limite de message reçu (qui est par défaut initialisé à 2). Le dernier paramètre est un paramètre graphique qui permet de mettre une image sur les composants, comme on peut le voir dans le mode design. On définit aussi les ports d'entrées et de sorties du composant.

La deuxième partie du code source se présente comme ça :

```
network Tictoc4
{
    @display("bgb=160,173");
    submodules:
        tic: Txc4 {
            parameters:
                sendMsgOnInit = true;
                @display("i=cyan");
                @display("p=32,101");
        }
        toc: Txc4 {
            parameters:
                sendMsgOnInit = false;
                @display("i=gold");
                @display("p=95,32");
        }
    connections:
        tic.out --> { delay = 100ms; } --> toc.in;
        tic.in <-- { delay = 100ms; } <-- toc.out;
}
```

Dans cette partie, le réseau est défini selon les modules qui le composent, ici on peut voir que deux modules de type Txc4 sont créés, l'un nommé tic et l'autre toc. Le module tic reçoit true dans son paramètre sendOnInit ce qui signifie que c'est lui qui va envoyer le premier message. Les fonctions @display sont des options graphiques qui permettent de changer de couleur les modules ou de les placer suivant un repère (ces options se modifient toutes seules quand on déplace les différents objets dans le mode design). Enfin, les connexions entre les modules sont définies ainsi que le délai de transmission pour chaque connexion. Le port de sortie de tic est relié au port d'entrée de toc et inversement.

Le deuxième fichier de base est un fichier source (écrit en c++) qui permet d'implémenter les fonctionnalités du composant de type Txc4. Nous allons donc voir comment ce fichier est construit :

```
#include <string.h>
#include <omnetpp.h>

using namespace omnetpp;

class Txc4 : public cSimpleModule
{
private:
    int counter; //Note the counter here

protected:
    // The following redefined virtual function holds the algorithm.
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};
```

Dans la première partie du code, on peut voir une définition de la classe Txc4, avec un paramètre privé qui est le compteur ainsi que deux fonctions, initialize() et handleMessage(). La première fonction est invoquée par la simulation à l'initialisation de celle-ci et la deuxième fonction est appelée à chaque fois qu'un message arrive dans le module.

Ensuite, on définit la fonction initialize() :

```
void Txc4::initialize()
{
    counter = par("limit");
    WATCH(counter);

    if (par("sendMsgOnInit").boolValue() == true) {
        EV << "Sending initial message\n";
        cMessage *msg = new cMessage("tictocMsg");
        send(msg, "out");
    }
}
```

Quand la simulation se lance, le compteur est initialisé à la valeur de la variable « limit » du composant en question (la fonction WATCH permet de visualiser la variable compteur dans la simulation graphique). Puis on vérifie si le composant possède son paramètre sendMsgOnInit égal à vrai, et si c'est le cas, un message appelé « tictocMsg » est créé et ce message est envoyé par le port « out » du composant.

Ensuite, on définit la fonction handleMessage() :

```
void Txc4::handleMessage(cMessage *msg)
{
    counter--;
    if (counter == 0){
        EV << getName() << "'s cou_nter reached zero, deleting message\n";
        delete msg;
    }
    else {
        EV << getName() << "'s counter is " << counter << ", sending back message\n";
        send(msg, "out"); // send out the message
    }
}
```

Cette fonction s'active quand un message est reçu par un composant. Dans un premier temps, le compteur est décrémenté, puis si le compteur est à zéro, le message est détruit car le nombre limite de messages reçu est dépassé et on envoie un message visible dans les logs de la simulation. Si le compteur n'est pas à zéro, on renvoie le message par le port de sortie.




Le dernier fichier de base est le fichier d'initialisation, qui permet à la simulation de choisir quel réseau il doit simuler (car les fichiers NED peuvent contenir plusieurs réseaux), ce fichier permet aussi d'initialiser certains paramètres pour la simulation.

Voici le code du fichier d'initialisation nécessaire pour le réseau Tictoc4 :

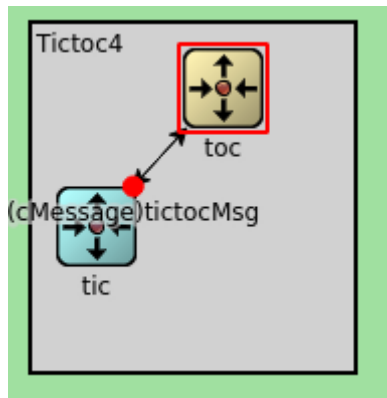
```
[Config Tictoc4]
network = Tictoc4
**.limit = 5|
```

Nous pouvons voir que si nous lançons le fichier Tictoc4.ned, alors la simulation va choisir le réseau nommé Tictoc4 et le paramètre « limit » de tous les composants sont initialisés à 5.

Le projet Tictoc4 se présente donc comme ça :

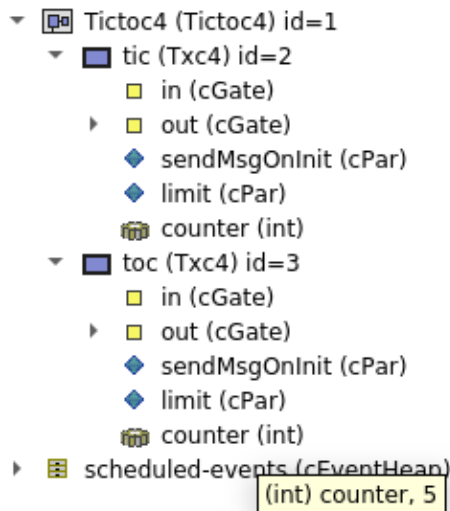
- ▶  txc4.cc
- ▶  omnetpp.ini
- ▶  tictoc4.ned

On peut ensuite lancer la simulation en exécutant le fichier tictoc4.ned avec Omnet++ :



On peut voir à l'initialisation, le premier message est envoyé par tic.

Grâce au WATCH(counter), la variable « counter » apparaît dans les variables visibles dans la simulation et on peut suivre l'évolution de la variable en passant la souris dessus :



On peut aussi voir l'évolution des logs durant de la simulation

```

Initializing module Tictoc4.tic, stage 0
INFO (Txc4)Tictoc4.tic:Sending initial message
Initializing module Tictoc4.toc, stage 0
** Event #1 t=0.1 Tictoc4.toc (Txc4, id=3) on tictocMsg (omnetpp::cMessage, id=1)
INFO:toc's counter is 4, sending back message
** Event #2 t=0.2 Tictoc4.tic (Txc4, id=2) on tictocMsg (omnetpp::cMessage, id=1)
INFO:tic's counter is 4, sending back message
** Event #3 t=0.3 Tictoc4.toc (Txc4, id=3) on tictocMsg (omnetpp::cMessage, id=1)
INFO:toc's counter is 3, sending back message
** Event #4 t=0.4 Tictoc4.tic (Txc4, id=2) on tictocMsg (omnetpp::cMessage, id=1)
INFO:tic's counter is 3, sending back message

```

Une fois que toc a reçu 5 messages, il supprime le message et la simulation s'arrête :

```
INFO:tic's counter is 1, sending back message
** Event #9 t=0.9 Tictoc4.toc (Txc4, id=3) on tictocMsg (omnetpp::cMessage, id=1)
INFO:toc's counter reached zero, deleting message
<!-- No more events, simulation completed -- at t=0.9s, event #9
** Calling finish() methods of modules
```

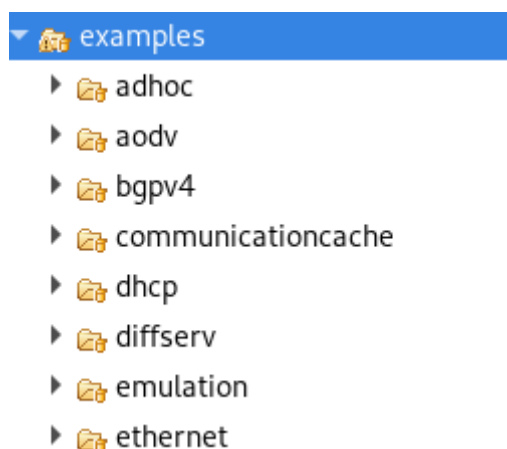
Chaque exemple guidé permet d'en savoir un peu plus sur le fonctionnement d'Omnet++, par exemple, on apprend à faire des héritages de classe, à envoyer des messages avec du délai, à ajouter de l'aléatoire dans le temps de transmission des messages ou encore à faire un réseau avec 6 modules avec un pseudo routage.

Finalement, ces différents exemples m'ont permis de mieux appréhender Omnet++, à mieux comprendre les différents liens entre les fichiers, à comprendre à quoi sert chaque fichier et à savoir l'utilité des différents arguments et des fonctions dans ces fichiers.

Après avoir appris les bases du simulateur de trafic et du simulateur de réseau, j'ai décidé de chercher des exemples de réseaux déjà existant qui pourrait m'aider à faire la simulation que je souhaite. J'ai donc essayé d'étudier plus en détail l'exemple dans le projet veins. Après avoir longtemps cherché dans les différents fichiers qui permettent l'échange de messages, qui génèrent les nœuds, qui gère le scénario ainsi que dans les logs de la simulation, je n'ai pas réussi à trouver la condition de l'envoi de message (je cherchais l'endroit dans le code qui définissait l'envoi d'un message). Je n'ai donc pas utilisé cet exemple pour réaliser ma simulation.

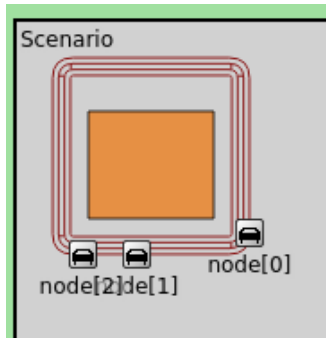
3.6. [Veins et Inet](#)

J'ai donc cherché dans les autres projets présents dans la machine virtuelle et j'ai surtout été intéressé par les projets veins_inet et inet. En effet, INET est une suite de modèles Omnet++ open-source pour les réseaux câblés, les réseaux sans fil et les réseaux mobiles. En effet, INET est composé de plusieurs exemples qui permettent de simuler différents types de réseaux :

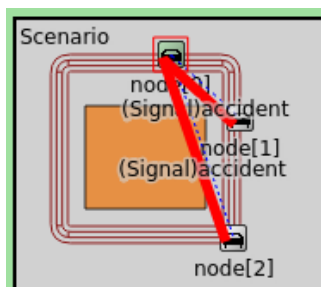


Nous pouvons voir ici des exemples de réseaux mobiles dans le dossier adhoc ainsi que l'utilisation du protocole de routage aodv, protocole destiné aux réseaux mobiles.

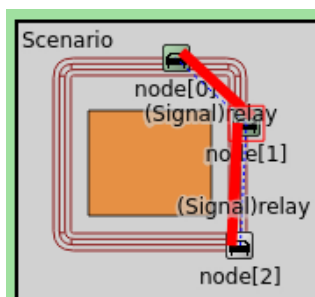
Après avoir regardé les différents exemples présents dans le projet inet, je suis allé voir `veins_inet`, qui est un petit projet dans lequel se trouve un exemple d'un réseau de type VANET avec un scénario qui est assez intéressant pour la simulation que je souhaite réaliser. Dans cet exemple, il y a une route en forme de carré dans laquelle se déplace 3 voitures. Les voitures apparaissent toutes les 5 secondes en bas à gauche de la route :



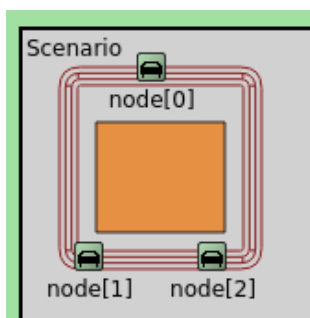
Les 3 voitures suivent la même route et au bout de 20 secondes de simulation, un accident va être simulé, la voiture de tête va s'arrêter et va envoyer un message nommé « accident » en broadcast aux autres voitures :



Les deux voitures reçoivent donc le message « accident » et vont relayer ce message :




Les deux voitures ainsi prévenu de l'accident vont changer de route afin d'atteindre leur point d'arrivée qui est en haut à gauche :





La voiture qui s'est arrêté reprend son chemin après 30 secondes de simulation.

Cet exemple combine un réseau véhiculaire qui permet une communication entre plusieurs voitures ainsi qu'un scénario de trafic simulé par SUMO. D'ailleurs, nous pouvons voir dans le dossier de cet exemple, qu'il y a les fichiers .xml qui ont été créé par SUMO :

 square.launchd.xml

 square.net.xml

 square.poly.xml

 square.rou.xml

Dans ces fichiers, nous retrouvons le fichier square.net.xml qui définit la topologie de la route, le fichier square.poly.xml qui permet de définir le bâtiment au milieu, le fichier square.launchd.xml qui permet de regrouper tous les fichiers dont SUMO a besoin pour la simulation et le fichier square.rou.xml qui permet de définir les paramètres de la simulation de trafic.
















Dans le fichier square.rou.xml, il y a ceci :

```
<routes>
  <vType id="vtype0" accel="2.6" decel="4.5" sigma="0.5" length="4.5" minGap="2.5" maxSpeed="14" color="1,1,0"/>
  <route id="route0" edges="A0toB0 B0toB1 B1toA1 A1toA0"/>
  <flow id="flow0" type="vtype0" route="route0" begin="0" period="5" number="3" arrivalPos="0" />
</routes>
```

Nous pouvons y voir des paramètres comme le type de véhicule généré (vType), son accélération (accel), sa vitesse maximale (maxSpeed), le temps d'apparition du premier véhicule (begin), la fréquence à laquelle les véhicules sont générés (period) ou encore le nombre de véhicule générés (number). On peut donc changer ces paramètres pour que la simulation colle à nos besoins.

Cet exemple dans le projet veins_inet constitue une bonne base pour créer ma propre simulation, c'est pourquoi j'ai décidé de partir de cet exemple en le modifiant pour ajouter les fonctionnalités voulues.

Dans la simulation que je souhaite faire, des messages doivent être envoyés périodiquement en broadcast par chaque véhicule de manière à transmettre des informations sur le trafic (position du véhicule, vitesse, type...). Je dois donc comprendre comment sont envoyés les messages (accident et relay) dans cet exemple. Pour cela, je suis allé voir dans les fichiers sources du projet qui permettent d'implémenter les fonctionnalités des différents composants :

- ▶  veins_inet.h
- ▶  VeinsInetApplicationBase.cc
- ▶  VeinsInetApplicationBase.h
- ▶  VeinsInetManager.cc
- ▶  VeinsInetManager.h
- ▶  VeinsInetManagerBase.cc
- ▶  VeinsInetManagerBase.h
- ▶  VeinsInetManagerForker.cc
- ▶  VeinsInetManagerForker.h
- ▶  VeinsInetMobility.cc
- ▶  VeinsInetMobility.h
- ▶  VeinsInetSampleApplication.cc
- ▶  VeinsInetSampleApplication.h
- ▶  VeinsInetSampleMessage_m.cc
- ▶  VeinsInetSampleMessage_m.h

Parmi ces fichiers sources, le fichier VeinsInetSampleApplication est particulièrement intéressant, voici ce qu'on peut y trouver :

```
bool VeinsInetSampleApplication::startApplication()
{
    // host[0] should stop at t=20s
    if (getParentModule()->getIndex() == 0) {
        auto callback = [this]() {
            getParentModule()->getDisplayString().setTagArg("i", 1, "red");

            traciVehicle->setSpeed(0);

            auto payload = makeShared<VeinsInetSampleMessage>();
            payload->setChunkLength(B(100));
            payload->setRoadId(traciVehicle->getRoadId().c_str());
            timestampPayload(payload);

            auto packet = createPacket("accident");
            packet->insertAtFront(payload);
            sendPacket(std::move(packet));

            // host should continue after 30s
            auto callback = [this]() {
                traciVehicle->setSpeed(-1);
            };
            timerManager.create(veins::TimerSpecification(callback).oneshotIn(SimTime(30, SIMTIME_S)));
        };
        timerManager.create(veins::TimerSpecification(callback).oneshotAt(SimTime(20, SIMTIME_S)));
    }

    return true;
}
```

Nous pouvons voir dans un premier temps, la fonction `startApplication()` qui est appelée une seule fois au démarrage de la simulation. La condition du début permet de sélectionner uniquement le premier véhicule.

On peut voir à la fin du programme, les `timerManager` qui permettent d'entrer dans le bout de code commençant par « auto callback » au bout de 20 secondes puis au bout de 30 secondes. Après 20 secondes de simulation, le véhicule deviendra rouge pour signifier qu'il se passe quelque chose puis il s'arrête (`traciVehicle->setSpeed(0)`). Ensuite, un payload est créé afin d'y insérer un message, son paramètre de taille est défini, puis le paramètre `RoadId` du payload est initialisé au paramètre `RoadId` du véhicule (`RoadId` permet de définir la route que les véhicules doivent suivre). Puis le payload est horodaté. Ensuite, un paquet nommé « accident » est créé puis le payload est inséré au début du paquet. Le paquet est ensuite envoyé en broadcast (puisque'il n'y a pas d'adresse spécifiée). Ensuite, au bout de 30 secondes de simulation, la vitesse du véhicule est de retour à la normale.

Ensuite, la fonction `processPacket()` est définie :

```
void VeinsInetSampleApplication::processPacket(std::shared_ptr<inet::Packet> pk)
{
    auto payload = pk->peekAtFront<VeinsInetSampleMessage>();

    EV_INFO << "Received packet: " << payload << endl;

    getParentModule()->getDisplayString().setTagArg("i", 1, "green");

    traciVehicle->changeRoute(payload->getRoadId(), 999.9);

    if (haveForwarded) return;

    auto packet = createPacket("relay");
    packet->insertAtBack(payload);
    sendPacket(std::move(packet));

    haveForwarded = true;
}
```

Cette fonction est appelée à chaque fois qu'un paquet est reçu par un nœud.

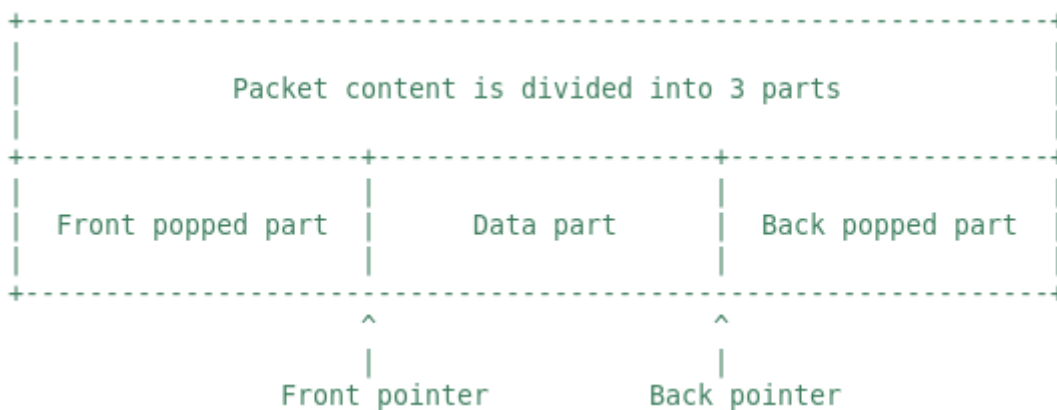
Dans un premier temps, le payload est extrait du paquet en question (comme le payload a été inséré au début du paquet qui a été transmis, on extrait le payload depuis le début du paquet avec la fonction `peekAtFront` de la classe `Packet`). Ensuite, on crée un log dans la simulation qui permet d'informer qu'un paquet a été reçu. Ensuite, le véhicule qui a reçu le paquet devient vert (dans la simulation) pour signifier qu'il a bien reçu le paquet. Puis le véhicule change de route pour éviter l'accident. Enfin, si le véhicule n'a pas déjà relayé le message, alors il envoie un paquet nommé « relay » en broadcast.

Nous pouvons voir que ce code source correspond parfaitement au scénario que nous avons observé lors de la simulation. C'est donc ce code que j'ai ensuite modifié afin de créer mon propre scénario.

3.7. [La simulation](#)

Pour que je puisse envoyer différents paramètres dans un message comme la vitesse, la position, le type de véhicule..., il faut que je sache comment fonctionne la classe « Packet » qui permet d'envoyer les messages entre plusieurs véhicules. Je suis donc allé voir dans le fichier Packet.h (fichier qui se trouve dans le projet inet).

Au début du fichier, une explication de la structure de la classe Packet est présente :



Nous pouvons voir qu'un paquet est divisé en trois parties, le début et la fin sont des parties avec lesquelles on peut travailler, c'est-à-dire qu'on peut ajouter des données dedans, les retirer ou les lire avec les fonctions insertAtBack, insertAtFront, removeAtFront, removeAtBack, peekAtFront ou peekAtBack. Et la partie Data ne peut qu'être lu. Les différentes parties sont délimitées par des pointeurs, donc pour insérer des choses dans la partie Data, il suffit d'insérer au début ou à la fin du paquet puis de faire bouger le pointeur correspondant.

Pour rajouter des données dans le paquet, j'ai essayé d'ajouter un autre payload à la fin du paquet transmis avec comme argument la note du véhicule. Cependant, dans la classe VeinsInetSampleMessage, il n'y a qu'un seul argument, c'est-à-dire qu'il n'existe qu'un seul champ :

```
class VEINS_INET_API VeinsInetSampleMessage : public ::inet::FieldsChunk
{
    protected:
        omnetpp::opp_string roadId;
```

Il est donc nécessaire d'ajouter un champ pour la note afin de transmettre la note dans le message. La classe est apparemment générée par un autre fichier :

Class generated from <tt>veins_inet/VeinsInetSampleMessage.msg:34</tt> by [nedtool](#).

Je suis donc allé voir le fichier en question pour essayer d'y ajouter le champ « note » :

```
class VeinsInetSampleMessage extends inet::FieldsChunk
{
    string roadId;
    string note;
```

Il suffit d'ajouter le champ que l'on souhaite. (Ici, je déclare la note de type string afin de prendre exemple sur RoadId, pour la gestion de ce paramètre).

Après avoir ajouté ce paramètre dans le fichier VeinsInetSampleMessage.msg, la classe VeinsInetSampleMessage a été modifiée en conséquence :

```
class VEINS_INET_API VeinsInetSampleMessage : public ::inet::FieldsChunk
{
    protected:
        omnetpp::opp_string roadId;
        omnetpp::opp_string note;
```

Il y a désormais le paramètre note qui a été ajouté ainsi que les fonctions Get et Set pour ce paramètre :

```
virtual const char * getNote() const;
virtual void setNote(const char * note);
```

Maintenant que je peux mettre le paramètre note dans un payload, je décide de ne plus créer deux payload que j'insère dans un paquet mais je préfère plutôt l'option de mettre les deux paramètres dans le même payload afin de récupérer un seul payload dans le paquet.

J'ajoute donc au code qui permet de simuler l'accident, un morceau qui permet d'insérer une valeur dans le champ réservé à la note dans le payload :

```
auto payload = makeShared<VeinsInetSampleMessage>();
timestampPayload(payload);
payload->setChunkLength(B(100));
payload->setRoadId(traciVehicle->getRoadId().c_str());

char *note = "6";
payload->setNote(note);

auto packet = createPacket("accident");
packet->insertAtFront(payload);
sendPacket(std::move(packet));
```

Je dois ensuite ajouter un bout de code dans la fonction ProcessPacket() (qui permet de gérer les paquets reçus) afin d'afficher ce nouveau paramètre :

```
if (strcmp(pk->getName(), "accident") == 0 ){

    EV_INFO << "note de l'expéditeur: " << payload->getNote() << endl;
```

Je rajoute aussi une condition pour réagir à un paquet nommé « accident », c'est-à-dire que les voitures vont changer de route uniquement si le paquet s'appelle accident. De plus, j'ai renommé le paquet relayé (qui se nommait « relay ») par « accident » pour que les voitures ayant l'information de l'accident par un message relayé puisse changer de route.

Nous pouvons voir dans les logs de la simulation :

```
INFO:Received packet: VeinsInetSampleMessage, length = 100 B
INFO:note de l'emetteur: 6
```

La note est bien transmise et reçue par les véhicules. Je peux donc ajouter autant de paramètres que nécessaire dans un payload afin de transmettre ces informations.

J'ajoute donc des champs pour la vitesse du véhicule, pour sa position (en x et en y), son type et son identifiant :

```
class VeinsInetSampleMessage extends inet::FieldsChunk
{
    string roadId;
    string note;
    double speed;
    double posx;
    double posy;
    string type;
    int id;
}
```

Ensuite, je dois faire en sorte que les véhicules envoient périodiquement des messages contenant ces informations. Pour cela, je vais chercher dans la classe TimerManager qui permet d'activer des événements dans la simulation au bout d'un certain temps de simulation. Je trouve une fonction qui correspond à ce que je veux faire, la fonction interval() :

```
/**
 * Set the period between two timer occurrences.
 */
TimerSpecification& interval(omnetpp::simtime_t interval);
```

Cette fonction permet de déclencher l'entrée dans un morceau de code à intervalle régulier.

La ligne de code qui permet d'envoyer un message toutes les 700ms est :

```
timerManager.create(veins::TimerSpecification(callback_1).interval(SimTime(700,SIMTIME_MS)));
```

(SIMTIME_S permet d'exprimer le temps en secondes et SIMTIME_MS, en millisecondes)

Dans la spécification du standard CAM, un message CAM est envoyé toutes les 100ms à 1s. Avec un message toutes les 700ms, je suis dans la bonne intervalle de valeur.

Ensuite, il faut insérer les valeurs dans les champs correspondant dans le payload et envoyer le message périodiquement. Avant d'insérer les valeurs des paramètres, il faut récupérer ces valeurs. Pour le champ note, pour l'instant, je peux mettre une note arbitraire. Pour la valeur de la vitesse, il faut aller récupérer la vitesse du véhicule grâce à une fonction présente dans le fichier TraCICommandInterface.cc qui est le fichier qui regroupe des fonctions qui permettent de récupérer des informations du simulateur de trafic SUMO.

Par exemple, pour avoir accès à la valeur de la vitesse du véhicule, on utilise la fonction `getSpeed()` :

```
double TraCICommandInterface::Vehicle::getSpeed()
{
    return traci->genericGetDouble(CMD_GET_VEHICLE_VARIABLE, nodeId, VAR_SPEED, RESPONSE_GET_VEHICLE_VARIABLE);
}
```

On récupère le type du véhicule de la même manière avec la fonction `getVType()`.

Pour la position du véhicule, on doit utiliser la classe `VeinsInetMobility` qui permet de connaître la position ainsi que l'orientation des composants dans un réseau mobile simulé par Omnet++ . On utilise la fonction `getCurrentPosition()` :

```
virtual inet::Coord getCurrentPosition() override;
```

Ensuite, pour récupérer l'ID du véhicule, la fonction `getId()` suffit.

Voici ce que donne le code du message périodique :

```
auto callback_1 = [this]() {
    auto payload_1 = makeShared<VeinsInetSampleMessage>();
    timestampPayload(payload_1);
    payload_1->setChunkLength(B(100));

    char *note_1 = "6";
    payload_1->setNote(note_1);

    double Speed_v = traciVehicle->getSpeed();
    payload_1->setSpeed(Speed_v);

    inet::Coord Pos_v = mobility->getCurrentPosition();
    double x_v = Pos_v.x;
    double y_v = Pos_v.y;
    payload_1->setPosx(x_v);
    payload_1->setPosy(y_v);

    std::string Type_v = traciVehicle->getVType();
    payload_1->setType(Type_v.c_str());

    int id = getId();
    payload_1->setId(id);

    auto packet_1 = createPacket("cam");
    packet_1->insertAtFront(payload_1);
    sendPacket([std::move(packet_1)]{
    });
    timerManager.create(veins::TimerSpecification(callback_1).interval(SimTime(700, SIMTIME_MS)));
};
```

L'ID du véhicule est en fait l'ID de ce qui gère les messages pour chaque véhicule, cet ID est donné par la simulation. Chaque véhicule à un ID unique qui permet de le distinguer des autres. Dans la simulation, on peut voir que le véhicule 0 possède l'id 49 :

 app[0] (VeinsInetSampleApplication) id=49

Donc quand on utilisera la fonction `getId()` pour le véhicule 0, on verra apparaître 49.

Pour afficher toutes les informations qu'un véhicule va recevoir, il faut ajouter une condition à la fonction processPacket() afin d'afficher les informations qu'il y a dans le payload si le nom du message que l'on reçoit est « cam », puis d'afficher toutes les informations contenues dans les champs du payload :

```
else if (strcmp(pk->getName(), "cam") == 0 ){
    EV_INFO << "numero voiture recepteur en cam: " << getParentModule()->getIndex() << endl;
    int id_recepteur = getId();
    EV_INFO << "id voiture recepteur en cam: " << getId() << endl;
    int id_emetteur = payload->getId();
    EV_INFO << "Id de l'emetteur: " << id_emetteur << endl;
    EV_INFO << "vitesse de l'emetteur: " << std::to_string(payload->getSpeed()) << endl;
    EV_INFO << "pos_x de l'emetteur: " << std::to_string(payload->getPosx()) << endl;
    EV_INFO << "pos_y de l'emetteur: " << std::to_string(payload->getPosy()) << endl;
    EV_INFO << "type de l'emetteur: " << payload->getType() << endl;
```

(Il est possible d'avoir le numéro de la voiture en utilisant la fonction getParentModule()->getIndex() mais j'ai préféré travailler avec l'ID pour plus de réalisme. En effet, l'ID permet d'identifier une voiture dans la simulation tout comme une adresse IP permet d'identifier un appareil dans un réseau, j'ai donc décidé de faire une correspondance entre l'ID d'un véhicule et sa note de confiance par la suite.

On peut aller voir dans la simulation si tout se passe comme prévu :

```
numero voiture recepteur en cam: 0
id voiture recepteur en cam: 49
Id de l'emetteur: 111
vitesse de l'emetteur: 8.233304
pos_x de l'emetteur: 53.418943
pos_y de l'emetteur: 129.950000
type de l'emetteur: vtype0
```

Nous pouvons voir ici que la voiture possédant l'ID 49 (voiture 0) reçoit un message de la voiture possédant l'ID 111 (voiture 1). On peut donc voir les informations de la voiture émettrice. Le message périodique permet donc aux véhicules d'envoyer leurs informations aux autres véhicules toutes les 700 ms.

La prochaine étape est de stocker les notes des véhicules quelque part où on pourra les modifier et les consulter. Chaque véhicule attribue une note aux autres véhicules. Le véhicule 0 par exemple attribue une note aux véhicules 1,2 et 3. Le véhicule 1 attribue une note aux véhicules 0,2 et 3 mais la note que le véhicule 0 attribue au véhicule 2 n'est pas la même que la note que le véhicule 1 lui attribue. C'est pourquoi, au début ma première idée était de stocker la note attribuée aux véhicules de la même façon que la vitesse et le type du véhicule, cependant, ces deux paramètres sont récupérés depuis SUMO. J'ai donc décidé de créer un fichier header Note.h dans lequel se trouve un tableau à deux dimensions (voir annexe 2 pour le fichier Note.h). La première dimension correspond à l'identifiant du véhicule qui attribue la note et la deuxième dimension correspond à l'identifiant du véhicule auquel est attribué la note. (Par exemple, pour savoir la note attribuée par le véhicule 0 sur le véhicule 1, il faut aller regarder aux coordonnées [0,1]).

Dans le fichier Note.h, il y a aussi un tableau qui permet de faire correspondre les identifiants des véhicules aux places dans le tableau de note. Par exemple, si le véhicule 0 reçoit un message du véhicule 1 pour la première fois, il va ajouter l'ID du véhicule 1 dans le tableau des Identifiants. Puis il va ajouter la note de départ dans son tableau des notes. Ensuite, quand il voudra consulter la note qu'il a attribué à ce véhicule, il regardera à quel indice (dans le tableau des ID) correspond l'identifiant du véhicule 1, puis il ira regarder au numéro de colonne de cet indice pour trouver la note. Par exemple :

Tableau d'ID du véhicule 0 :

111	172
-----	-----

Tableau de note du véhicule 0 :

5	6
---	---

Ici, on voit que le véhicule 0 a reçu un message premièrement du véhicule avec l'ID 111 puis du véhicule avec l'ID 172, il a stocké leurs ID ainsi que leurs notes dans le même ordre. Le véhicule 0 attribue donc la note de 5 au véhicule possédant l'identifiant 111 et 6 au véhicule possédant l'identifiant 172.

Il faut donc faire les fonctions `getNote()` et `setNote()` afin de pouvoir lire les notes et les modifier :

```
double TraCInterface::Vehicle::getNote(int id_em,int id_re)
```

La fonction `getNote` prend en argument l'ID du véhicule a qui la note est attribué (`id_em`) et l'ID du véhicule qui attribue la note (`id_re`) et sort la note demandée.

Si on ne connaît pas l'ID du véhicule, on donne une note de départ par défaut.

```
void TraCInterface::Vehicle::setNote(int id_em,int id_re,double note)
```

La fonction `setNote` prend en argument l'ID du véhicule a qui la note est attribué (`id_em`), l'ID du véhicule qui attribue la note (`id_re`) et la note que l'on veut attribuer.

(Voir annexe 3 pour les codes en entier).

Nous avons donc maintenant un moyen d'avoir accès à la note attribuée par tous les véhicules sur tous les véhicules, il n'y a donc plus besoin d'envoyer la note dans le paquet transmis.

Pour tester ce système de stockage et de modification de note, je décide de faire un petit scénario avec quatre véhicules roulant à une vitesse constante de 10 (je ne connais pas l'unité de vitesse de la simulation) dans lequel un véhicule va faire baisser sa vitesse subitement. Je vais alors modifier la note de tel manière que si le véhicule descend en dessous de 10 de vitesse, sa note attribuée par les autres véhicules va baisser. Et si le véhicule roule exactement à la vitesse 10, alors sa note augmentera.

Pour cela, j'affiche la note de l'émetteur lorsqu'un véhicule réceptionne un paquet « cam » :

```
EV_INFO << "note de l'emetteur: " << traciVehicle->getNote(id_emetteur,id_recepteur) << endl;
```

Ensuite, j'établis la condition pour baisser la note ou l'augmenter :

```
if (payload->getSpeed() < 10){
    double note_emetteur = traciVehicle->getNote(id_emetteur,id_recepteur);
    traciVehicle->setNote(id_emetteur,id_recepteur,note_emetteur - 0.5);
}
else if (payload->getSpeed() == 10){
    double note_emetteur = traciVehicle->getNote(id_emetteur,id_recepteur);
    traciVehicle->setNote(id_emetteur,id_recepteur,note_emetteur + 0.25);
}
```

Puis j'affiche la nouvelle note :

```
EV_INFO << "note de l'emetteur apres: " << traciVehicle->getNote(id_emetteur,id_recepteur) << endl;
```

Sans oublier de créer l'évènement qui va faire baisser la vitesse d'un véhicule :

```
if (getParentModule()->getIndex() == 2) {
    auto callback_2 = [this]() {
        traciVehicle->setSpeed(5);
    };
    timerManager.create(veins::TimerSpecification(callback_2).oneshotAt(SimTime(20, SIMTIME_S)));
}
```

Le véhicule 2, va descendre à 5 en vitesse au bout de 20 secondes de simulation.

Je lance donc la simulation pour vérifier :

```
Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 1
id voiture recepteur en cam: 111
Id de l'emetteur: 49
vitesse de l'emetteur: 10.000000
pos_x de l'emetteur: 107.416000
pos_y de l'emetteur: 129.950000
type de l'emetteur: vtype0
note de l'emetteur: 6.75
note de l'emetteur apres: 7
ent #582 t=9.200152235329 Scenario.node[2].wlan[0].rac
```

On peut voir ici que la note attribuée par la voiture 111 sur la voiture 49 augmente car la voiture 49 roule à une vitesse de 10. (t = 9.2s)

```

Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 2
id voiture recepteur en cam: 173
Id de l'emetteur: 49
vitesse de l'emetteur: 10.000000
pos_x de l'emetteur: 107.416000
pos_y de l'emetteur: 129.950000
type de l'emetteur: vtype0
note de l'emetteur: 5.25
note de l'emetteur apres: 5.5
Sent #588 t=9.210152235329 Scenario.radioMedium (Ieee8

```

On peut voir ici que la note attribuée par la voiture 173 sur la voiture 49 augmente car la voiture 49 roule à une vitesse de 10. On peut aussi constater que les deux captures sont prises avec peu de temps d'intervalle ($t = 9.20s$ et $t = 9.21s$), il s'agit donc du même message envoyé par la voiture 49, on peut donc voir que les notes attribuées par les véhicules sont différentes. En effet, la voiture 111 est apparue depuis plus longtemps que la voiture 173 et donc elle « connaît » depuis plus longtemps la voiture 49.

Ensuite, on regarde après 20 secondes de simulation ($t = 20.7$) :

```

Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 3
id voiture recepteur en cam: 235
Id de l'emetteur: 173
vitesse de l'emetteur: 6.850000
pos_x de l'emetteur: 129.950000
pos_y de l'emetteur: 111.634000
type de l'emetteur: vtype0
note de l'emetteur: 8
note de l'emetteur apres: 7.5
Sent #2684 t=20.700152138829 Scenario.node[1].wlan[0].r

```

On peut voir que la vitesse du véhicule 2 (ID = 173) commence à diminuer et sa note diminue aussi.

Si on continue encore la simulation ($t=24.9s$) :

```

Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 3
id voiture recepteur en cam: 235
Id de l'emetteur: 173
vitesse de l'emetteur: 5.000000
pos_x de l'emetteur: 129.950000
pos_y de l'emetteur: 90.344000
type de l'emetteur: vtype0
note de l'emetteur: 5
note de l'emetteur apres: 4.5
Sent #3584 t=24.900152210622 Scenario.node[1].wlan[0].r

```

On peut voir que la note attribuée par le véhicule 235 à bien diminué.

Un modèle de calcul de la note de confiance m'a ensuite été proposé (la note de confiance est définie ici entre 0 et 1), ce modèle consiste à prendre des paramètres comme la position relative entre deux véhicules, la vitesse d'un véhicule ou encore son type. Personnellement, j'ai utilisé ce modèle avec la vitesse et le type. Une fois les paramètres choisis, il faut utiliser une formule pour évaluer chaque paramètre. Pour le type, c'est simple, si le véhicule est de type `vtype0`, alors le paramètre sera évalué à 1, sinon il sera évalué à 0.

Pour la vitesse, la formule qui m'a été donnée est la suivante :

- $1 - a$ si $a \leq Ths$
- 0 sinon

Avec $a = \frac{|Ths - Rs|}{Ths}$, Ths est la vitesse idéale et Rs , est la vitesse réelle.

J'ai aussi changé la note de départ à 0.5 pour que les calculs soient faits entre 0 et 1.

J'ai donc implémenté ces formules pour évaluer les deux paramètres :

```
double Speed_thresh = 10;
double Speed_em = payload->getSpeed();
double a = (abs(Speed_thresh - Speed_em))/Speed_thresh;
double pki_speed;
if (a <= Speed_thresh){
    pki_speed = 1-a;
}
else {
    pki_speed = 0;
}
double pki_type;
if (strcmp(payload->getType(), "vtype0") == 0){
    pki_type = 1;
}
else {
    pki_type = 0;
}
```

Ensuite, pour calculer la note, il suffit de faire une moyenne géométrique des valeurs des évaluations des paramètres, puis il faut modifier la note correspondante en conséquence :

```
double Note = sqrt(pki_speed*pki_type);
traciVehicle->setNote(id_emetteur, id_recepteur, Note);
```

J'affiche dans les logs de la simulation la note ainsi calculée :

```
EV_INFO << "note de l'emetteur apres calcul: " << traciVehicle->getNote(id_emetteur, id_recepteur) << endl;
```


On va ensuite voir dans la simulation pour vérifier que le modèle est bien implémenté :

À $t = 4.3s$:

```
Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 1
id voiture recepteur en cam: 111
Id de l'emetteur: 49
vitesse de l'emetteur: 10.000000
pos_x de l'emetteur: 58.416000
pos_y de l'emetteur: 129.950000
type de l'emetteur: vtype0
note de l'emetteur: 0.5
note de l'emetteur apres calcul: 1
ent #155 t=4.310152077347 Scenario.radioMedium (Ieee80211f
```

On peut voir ici que c'est la première fois que la voiture avec l'ID 49 envoie un message « cam » à la voiture avec l'ID 111 car la première note de l'émetteur qui est affiché est la note de départ (0.5), puis avec le calcul, la note est de 1 puisque la voiture est à la vitesse 10 (qui est défini comme la vitesse idéale dans la simulation) et elle est de type vtype0.

À $t = 4.8s$:

```
Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 0
id voiture recepteur en cam: 49
Id de l'emetteur: 111
vitesse de l'emetteur: 1.560000
pos_x de l'emetteur: 36.696000
pos_y de l'emetteur: 129.950000
type de l'emetteur: vtype0
note de l'emetteur: 0.5
note de l'emetteur apres calcul: 0.394968
ent #183 t=4.810152091857 Scenario.radioMedium (Ieee802
```

On peut voir ensuite que la première fois que la voiture avec l'ID 111 envoie à la voiture avec l'ID 49, sa vitesse est de 1.56, sa note calculée est donc plus faible puisque sa vitesse est bien plus basse que la vitesse idéale.

À $t = 5.5s$:

```
Received packet: VeinsInetSampleMessage, length = 100 B
numero voiture recepteur en cam: 0
id voiture recepteur en cam: 49
Id de l'emetteur: 111
vitesse de l'emetteur: 3.380000
pos_x de l'emetteur: 38.516000
pos_y de l'emetteur: 129.950000
type de l'emetteur: vtype0
note de l'emetteur: 0.394968
note de l'emetteur apres calcul: 0.581378
ent #236 t=5.510152108528 Scenario.radioMedium (Ieee802
```

La note de la voiture 111 attribuée par la voiture 49 augmente en fonction de l'augmentation de sa vitesse, jusqu'à atteindre 1 quand sa vitesse sera égale à 10.

Pour le calcul de la note, on peut tout à fait attribuer des poids aux paramètres lors du calcul, par exemple si on considère que la valeur de l'évaluation du paramètre vitesse d'une voiture est plus importante pour le calcul de sa note, on lui attribuera un poids plus important.

4. Conclusion

La simulation fonctionne donc correctement, un message est envoyé périodiquement par tous les véhicules afin d'envoyer des informations sur le trafic en temps réel. Ce message peut être assimilé au message standardisé CAM dans un réseau VANET. Un autre message est envoyé quand un événement routier se produit afin de prévenir les autres véhicules de cet événement. Ce message peut être assimilé au message standardisé DENM dans un réseau VANET. De plus, dans la simulation, les véhicules ont accès aux notes qu'ils ont attribués aux autres véhicules, il est désormais facile à imaginer que l'on peut établir un seuil à partir duquel un véhicule accepte ou non le message provenant d'un autre véhicule.

Il est aussi possible de changer la méthode de calcul de la note de confiance (en prenant plus de paramètres en compte ou en ajustant les poids par exemple) afin de s'adapter aux contraintes du modèle de Trust Management que l'on a choisi.

On pourrait aussi calculer la note de manière indirect dans cette simulation en demandant l'avis d'autres véhicules car le tableau de note est construit pour que tous les véhicules aient accès aux notes attribuées par les autres véhicules.

Il est possible d'utiliser ce modèle sur d'autres simulations de trafic (route différente, apparition des voitures à d'autres endroits, parcours des voitures différent...), on peut donc créer sa propre simulation de trafic avec SUMO et utiliser ce modèle dessus.

5. Références bibliographiques

[Hasrouny17] H. Hasrouny, A. E. Samhat, C. Bassil, and A. Laouiti, “Vanet security challenges and solutions: A survey”, Vehicular Communications, vol. 7, pp. 7–20, 2017.

[Sharma20] A. Sharma, E. S. Pilli, A. P. Mazumdar, and P. Gera, “Towards trustworthy internet of things: A survey on trust management applications and schemes,” Computer Communications, 2020.

[Hussain20] R. Hussain, J. Lee, and S. Zeadally, “Trust in vanet : A survey of current solutions and future research opportunities,” IEEE transactions on intelligent transportation systems, vol. 22, no. 5, pp. 2553– 2571, 2020.

[NS-3] Wikipédia, “Network simulator — wikipédia, l’encyclopédie libre,” 2020. [En ligne ; Page disponible le 16-septembre-2020].

[Campanile20] L. Campanile, M. Gribaudo, M. Iacono, F. Marulli, and M. Mastroianni, “Computer network simulation with ns-3 : A systematic literature review,” Electronics, vol. 9, no. 2, p. 272, 2020.

[Liu16] W. Liu, X. Wang, W. Zhang, L. Yang, and C. Peng, “Coordinative simulation with sumo and ns3 for vehicular ad hoc networks,” in 2016 22nd Asia-Pacific Conference on Communications (APCC), pp. 337– 341, 2016.

[Su2021] Runbo Su, Arbia Riahi, Enrico Natalizio, Pascal Moyal, Ye-Qiong Song: PDTM: Phase-based dynamic trust management for Internet of things. ICCCN 2021: 1-7.

6. Sources

Ns-3 :

<https://www.nsnam.org/>

<https://www.nsnam.org/docs/release/3.35/tutorial/ns-3-tutorial.pdf>

https://www.youtube.com/watch?v=KVcZbrMNxvw&list=PLX6MKaDw0naZILVYjo8_quA4JI8Jz-idL&index=37

<https://www.youtube.com/watch?v=cX2HrTnl80E&list=PLmcMMZCV897qkb1fv177Y69hX8YhcmVpS&index=3>

SUMO:

<https://sumo.dlr.de/docs/Tutorials/index.html>

<https://sumo.dlr.de/userdoc/>

<https://www.youtube.com/watch?v=Y37GXkyqfow>

Veins :

<https://veins.car2x.org/>

<https://veins.car2x.org/tutorial/>

<https://stackoverflow.com/questions/tagged/veins>

<https://veins.car2x.org/documentation/faq/>

Omnet ++ :

<https://omnetpp.org/documentation/>

<https://docs.omnetpp.org/tutorials/tictoc/part1/>

Annexe 1

First.cc :

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");

int
main (int argc, char *argv[])
{
    CommandLine cmd (__FILE__);
    cmd.Parse (argc, argv);

    Time::SetResolution (Time::NS);
    LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);

    NodeContainer nodes;
    nodes.Create (2);

    PointToPointHelper pointToPoint;
    pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
    pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

    NetDeviceContainer devices;
    devices = pointToPoint.Install (nodes);

    InternetStackHelper stack;
    stack.Install (nodes);

    Ipv4AddressHelper address;
    address.SetBase ("10.1.1.0", "255.255.255.0");

    Ipv4InterfaceContainer interfaces = address.Assign (devices);

    UdpEchoServerHelper echoServer (9);

    ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
    serverApps.Start (Seconds (1.0));
    serverApps.Stop (Seconds (10.0));

    UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
    echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
    echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
    echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

    ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
    clientApps.Start (Seconds (2.0));
    clientApps.Stop (Seconds (10.0));

    Simulator::Run ();
    Simulator::Destroy ();
    return 0;
}
```

Second.cc :

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("SecondScriptExample");

int
main (int argc, char *argv[])
{
    bool verbose = true;
    uint32_t nCsma = 3;

    CommandLine cmd (__FILE__);
    cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
    cmd.AddValue ("verbose", "Tell echo applications to log if true",
verbose);

    cmd.Parse (argc,argv);

    if (verbose)
    {
        LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
        LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
    }

    nCsma = nCsma == 0 ? 1 : nCsma;

    NodeContainer p2pNodes;
    p2pNodes.Create (2);

    NodeContainer csmaNodes;
    csmaNodes.Add (p2pNodes.Get (1));
    csmaNodes.Create (nCsma);

    PointToPointHelper pointToPoint;
    pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
    pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

    NetDeviceContainer p2pDevices;
    p2pDevices = pointToPoint.Install (p2pNodes);

    CsmaHelper csma;
    csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
    csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

    NetDeviceContainer csmaDevices;
    csmaDevices = csma.Install (csmaNodes);
```

```

InternetStackHelper stack;
stack.Install (p2pNodes.Get (0));
stack.Install (csmaNodes);

Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);

address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);

UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get
(nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));

UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UIntegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UIntegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (p2pNodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

pointToPoint.EnablePcapAll ("second");
csma.EnablePcap ("second", csmaDevices.Get (1), true);

Simulator::Run ();
Simulator::Destroy ();
return 0;

```


Third.cc :

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("ThirdScriptExample");

int
main (int argc, char *argv[])
{
    bool verbose = true;
    uint32_t nCsmas = 3;
    uint32_t nWifi = 3;
    bool tracing = false;

    CommandLine cmd (__FILE__);
    cmd.AddValue ("nCsmas", "Number of \"extra\" CSMA nodes/devices", nCsmas);
    cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
    cmd.AddValue ("verbose", "Tell echo applications to log if true",
verbose);
    cmd.AddValue ("tracing", "Enable pcap tracing", tracing);

    cmd.Parse (argc,argv);

    // The underlying restriction of 18 is due to the grid position
    // allocator's configuration; the grid layout will exceed the
    // bounding box if more than 18 nodes are provided.
    if (nWifi > 18)
    {
        std::cout << "nWifi should be 18 or less; otherwise grid layout
exceeds the bounding box" << std::endl;
        return 1;
    }

    if (verbose)
    {
        LogComponentEnable ("UdpEchoClientApplication", LOG_LEVEL_INFO);
        LogComponentEnable ("UdpEchoServerApplication", LOG_LEVEL_INFO);
    }

    NodeContainer p2pNodes;
    p2pNodes.Create (2);

    PointToPointHelper pointToPoint;
    pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
    pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

    NetDeviceContainer p2pDevices;
    p2pDevices = pointToPoint.Install (p2pNodes);

    NodeContainer csmaNodes;
    csmaNodes.Add (p2pNodes.Get (1));
    csmaNodes.Create (nCsmas);

    CsmaHelper csma;
    csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
    csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));
}
```

```

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);

NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi);
NodeContainer wifiApNode = p2pNodes.Get (0);

YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper phy;
phy.SetChannel (channel.Create ());

WifiHelper wifi;
wifi.SetRemoteStationManager ("ns3::AarfWifiManager");

WifiMacHelper mac;
Ssid ssid = Ssid ("ns-3-ssid");
mac.SetType ("ns3::StaWifiMac",
             "Ssid", SsidValue (ssid),
             "ActiveProbing", BooleanValue (false));

NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes);

mac.SetType ("ns3::ApWifiMac",
             "Ssid", SsidValue (ssid));

NetDeviceContainer apDevices;
apDevices = wifi.Install (phy, mac, wifiApNode);

MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                              "MinX", DoubleValue (0.0),
                              "MinY", DoubleValue (0.0),
                              "DeltaX", DoubleValue (5.0),
                              "DeltaY", DoubleValue (10.0),
                              "GridWidth", UIntegerValue (3),
                              "LayoutType", StringValue ("RowFirst"));

mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
                           "Bounds", RectangleValue (Rectangle (-50, 50,
-50, 50)));
mobility.Install (wifiStaNodes);

mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNode);

InternetStackHelper stack;
stack.Install (csmaNodes);
stack.Install (wifiApNode);
stack.Install (wifiStaNodes);

Ipv4AddressHelper address;

```



```

address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);

address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);

address.SetBase ("10.1.3.0", "255.255.255.0");
address.Assign (staDevices);
address.Assign (apDevices);

UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get
(nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));

UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps =
    echoClient.Install (wifiStaNodes.Get (nWifi - 1));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));

Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

Simulator::Stop (Seconds (10.0));

if (tracing)
{
    phy.SetPcapDataLinkType (WifiPhyHelper::DLT_IEEE802_11_RADIO);
    pointToPoint.EnablePcapAll ("third");
    phy.EnablePcap ("third", apDevices.Get (0));
    csma.EnablePcap ("third", csmaDevices.Get (0), true);
}

Simulator::Run ();
Simulator::Destroy ();
return 0;

```

Annexe 2

Note.h :

```
#pragma once

namespace veins {
namespace Note {

//double note = 5.0;
std::vector<int> id_re;
std::vector<std::vector<int>> id_em;
std::vector<std::vector<double>> notes;

} // namespace Note
} // namespace veins
```

Annexe 3

Fonction getNote() :

```
double TraCICCommandInterface::Vehicle::getNote(int id_em,int id_re)
{
    int indice_re = 0;
    int n = (Note::notes).size();
    while (indice_re < n && (Note::id_re)[indice_re] != id_re ){
        indice_re++;
    }
    if (indice_re < n){
        std::vector<double> note_re = Note::notes[indice_re];
        std::vector<int> note_id_re = Note::id_em[indice_re];
        int indice_em = 0;
        int m = note_re.size();
        while (indice_em < m && (note_id_re)[indice_em] != id_em ){
            indice_em++;
        }
        if (indice_em < m){
            return note_re[indice_em];
        }
        else {
            double note_depart = 5.0;
            Note::id_em[indice_re].push_back(id_em);
            Note::notes[indice_re].push_back(note_depart);
            return(note_depart);
        }
    }
    else {
        double note_depart = 5.0;
        Note::id_re.push_back(id_re);
        Note::id_em.push_back({id_em});
        Note::notes.push_back({note_depart});
        return(note_depart);
    }
}
```

Fonction setNote() :

```
void TraCICommandInterface::Vehicle::setNote(int id_em, int id_re, double note)
{
    int indice_re = 0;
    int n = (Note::notes).size();
    while (indice_re < n && (Note::id_re)[indice_re] != id_re ){
        indice_re++;
    }
    if (indice_re < n){
        std::vector<double> note_re = Note::notes[indice_re];
        std::vector<int> note_id_re = Note::id_em[indice_re];
        int indice_em = 0;
        int m = note_re.size();
        while (indice_em < m && (note_id_re)[indice_em] != id_em ){
            indice_em++;
        }
        if (indice_em < m){
            Note::notes[indice_re][indice_em] = note;
        }
        else {
            int note_depart = 5.0;
            Note::id_em[indice_re].push_back(id_em);
            Note::notes[indice_re].push_back(note_depart);
        }
    }
    else {
        double note_depart = 5.0;
        Note::id_re.push_back(id_re);
        Note::id_em.push_back({id_em});
        Note::notes.push_back({note_depart});
    }
}
```