Chapitre IV. Implémentation

Introduction

La simulation des réseaux de capteurs véhiculaires consiste principalement en la reproduction du comportement des véhicules et des interactions entre eux. C'est une étape incontournable pour l'évaluation des modèles ou des protocoles de communication. De plus, la simulation offre un gain considérable en temps, une flexibilité en permettant la variation des paramètres et une meilleure visualisation des résultats.

Ce chapitre contient la présentation de simulateur ns-3 et des différentsdétails de l'implémentation.

1. Différentes approches de test [1]

Lors du développement d'un protocole ou d'une application répartie, il est nécessaire d'évaluer le comportement des différents mécanismes proposés, de pouvoir les comparer avec ceux existants et de mesurer les performances dans des conditions réseau particulières. Pour ce faire, il est possible d'utiliser plusieurs méthodes : le test en environnement réel, la simulation et l'émulation.

1.1. Test en environnement réel

Ce test permet d'avoir des conditions parfaitement réalistes, mais n'est pas complètement satisfaisante car il ne permet pas de contrôler l'ensemble des paramètres de l'environnement. De plus, il ne permet pas de reproduire plusieurs fois de suite les mêmes conditions avec précision. En effet, les conditions de propagation du signal à l'intérieur du réseau ne sont pas contrôlables, si bien que d'une expérience à l'autre les conditions observées peuvent évoluer et parfois modifier considérablement le résultat des tests [réf].

1.2. Simulation

La simulation permet d'évaluer un modèle d'application ou de protocole dans un environnement entièrement contrôlable. Pour cela, la simulation s'appuie sur des modèles décrivant l'environnement et les couches de communication utilisées par les terminaux sans fil ainsi que d'autres équipements du réseau et un modèle du trafic circulant sur le réseau. Cependant, l'utilisation d'un modèle au lieu de l'implémentation réelle influe sur le déroulement de l'implémentation et son déploiement [réf].

1.3. L'émulation

L'émulation peut être vue comme un compromis entre les deux tests précédents qui simule en temps réel dans un environnement contrôlable et reproductible les conditions telles que : les débits, les délais et les pertes que l'on observe dans le réseau cible [réf].

Comme l'émulation travaille en temps réel, les modèles utilisés pour simuler les couches basses ne peuvent pas être trop complexes ce qui implique un impact négatif sur le réalisme de l'émulation rendue.

A cet effet, l'approche de simulation détient toujours sa place comme solution pour le test et la validation de protocoles et d'applications.

2. Environnement de simulation

Les besoins croissants de tester les nouvelles technologies et les nouveaux protocoles avant leur déploiement, ont conduit à la prolifération des simulateurs. On peut les classer en deux types : les logiciels libres et gratuits tels que : **OMNet++ [réf]**, **J-Sim [réf]**, **NS2 [réf]** et **ns-3[réf]**…et les logiciels commerciaux tels que : **OPNET [réf]** et **NetRule [réf].**

Dans ce qui suit, nous présentons l'environnement de simulation utilisé.

2.1. Choix de simulateur ns-3

Ns (de l'anglais « network simulator ») est un logiciel libre de simulation à événements discrets très largement utilisé dans la recherche académique et dans l'industrie. Il est considéré par beaucoup de spécialistes des télécommunications comme le meilleur logiciel de simulation à événements discrets, en raison de son modèle libre, permettant l'ajout très rapide de modèles correspondant à des technologies émergentes.

La version 2 était basée sur l'utilisation de langages de scripts pour la commande des simulations (Tcl/Tk) alors que seul le cœur des simulations était implémenté avec le langage C++.

La nouvelle version ns-3[réf] est entièrement écrite en C++ avec des liaisons Python en option, ce qui permet l'écriture de scripts de simulation en C++ ou Python.

Le logiciel NS est fourni avec une interface graphique (NetAnim) permettant de démontrer le fonctionnement des réseaux, comme le montre la figure 7 ci-dessous. ce qui en fait un outil à la valeur pédagogique très intéressante.

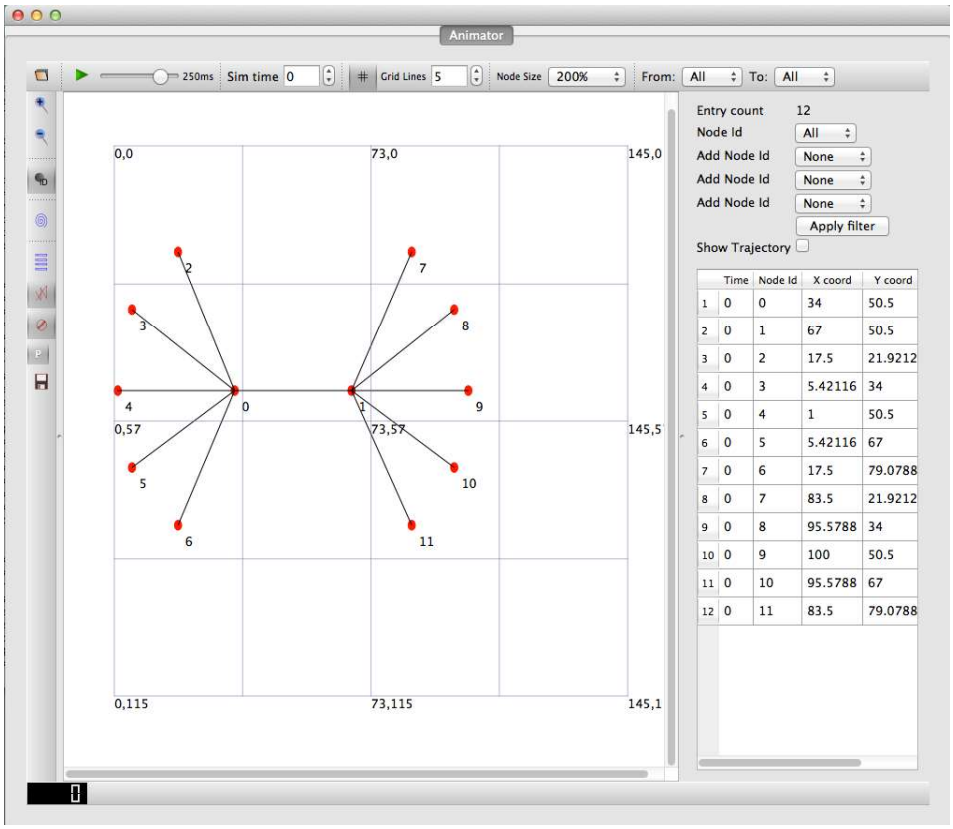

Figure 7 : Interface de NetAnim.

2.2. Comparaison entre les simulateurs NS2 / ns-3

Une étude comparative entre NS2 et NS-3 a été effectuée par [réf] dont le tableau 1 ci-dessous résume les principales différences.

	NS2	ns-3
Langages de programmation	• NS2 est implémenté en utilisant une combinaison d'oTCL (scripts) et de C++ (cœur de simulateur). • Ce système a été choisi au début des années 1990 pour éviter la recompilation du C++, car il était très coûteux en temps en utilisant le matériel disponible à ce moment-là, la recompilation otcl prend moins de temps que celle de C++. • Inconvénient otcl: des frais onéreux sont introduits avec de grandes simulations. • oTCL est le seul langage de script disponible.	• Ns-3 est implémenté en utilisant le C++. • Avec des capacités matérielles modernes, le temps de recompilation n'est plus un problème comme c'était pour NS2, ns-3 peut être développé avec C++ entièrement. • Un script de simulation peut être écrite comme une programme C++, ce qui n'est pas possible dans NS2. • Il existe un support (limité) de python, dans les scripts et les visualisations.
Gestion de la mémoire	• NS2 nécessite l'utilisation des fonctions basiques (manuelles) de gestion de la mémoire.	• Toute les fonctions de gestion de mémoire de C++ sont disponibles (new, delete, malloc, free). • La dés-allocation automatique des objets est supportée. (traquer le nombre de pointeurs vers un objet) ; utile lors de traitement des objets de type Packet.
Paquets	• Un paquet est constitué de deux régions distinctes, l'une pour l'en-tête, et la deuxièmecontient les données. • NS2 ne libère pas la mémoire utilisée pour stocker les paquets jusqu'à ce que la simulation se termine, il réutilise juste le paquet alloué à plusieurs reprises, par conséquent, la région d'en-tête de chaque paquet comprend tous les en-têtes définis dans le cadre du protocole utilisé même si ce paquet particulier n'utilisera pas cet en-tête particulier, mais juste pour	• Un paquet se compose d'un seul tampon d'octets, et éventuellement d'une collection de petites étiquettes contenant des métadonnées. • Le tampon correspond exactement au flux de bits qui seraient envoyés dans véritable réseau. • L'information est ajoutée au paquet à l'aide de sous-classes; Header, ce qui ajoute des informations au début du tampon, Trailer ???, ce qui ajoute à la fin. • Contrairement à NS2, il est généralement facile de

	être disponible au moment de la réattribution de paquet.	déterminer si un en-tête spécifique est attaché.
Performance	• Le temps de calcul exigé pour exécuter une simulation est meilleur en ns-3 qu'en NS2. • Cela est dû à la suppression des frais associés à l'interfaçage d'otcl avec C++, et les frais associés à l'interpréteur otcl.	• Ns-3 est mieux performant que NS2 en termes de gestion de la mémoire. • Le système d'agrégation empêche les paramètres inutiles d'être stockés, et les paquets ne contiennent pas d'espaces d'en-tête réservés non utilisés.
Sortie de simulation	• NAM (Network Animator). Représentation graphique de réseau décrit, basée sur le TCL.	• NetAnim (basé sur XML). • PyViz (basé sur python).

Tableau 1. Comparaison entre les simulateurs.

2.3. Notes importantes à propos de ns-3

- Ns-3 n'est pas rétro compatible avec NS2, il a été bâti à partir de zéro pour remplacer NS2.
- Ns-3 est écrit en C++, Python peut éventuellement être utilisé comme une interface.
- Ns-3 essaie de résoudre les problèmes présents dans NS2.
- Dans NS2 le système bi-langage rend le débogage complexe (C++/TCL), alors que pour ns-3 la connaissance de C++ est suffisante. (l'architecture uni-langage est plus robuste à long terme).
- Ns-3 a un mode émulation, qui permet l'intégration avec les réseaux réels.

3.3. Architecture de ns-3

Ns-3 est une bibliothèque C++ fournissant un ensemble de modèles de simulation de réseaux implémentés comme objets C++ et enveloppés par python. Les utilisateurs peuvent interagir normalement avec cette bibliothèque en écrivant une application C++ ou python, qui instancie un ensemble de modèles de simulation pour mettre en place le scénario de simulation en question, entre dans la boucle principale de la simulation et quitte lorsque la simulation est terminée.

La bibliothèque ns-3 est enveloppée de python en utilisant la bibliothèque *pybindgen* qui délègue l'analyse des en-têtes ns-3 C++ à *gccxml* et *pygccxml*pour générer automatiquement les colles de liaison C++ correspondantes. Ces fichiers C++ automatiquement générés sont finalement compilées dans le module ns-3 python pour

permettre aux utilisateurs d'interagir avec les modèles ns-3 C++ et le cœur à travers des scripts python.

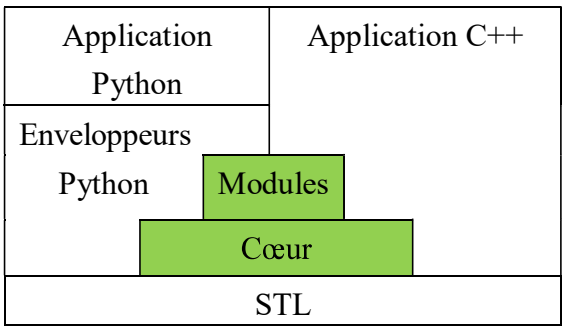

Figure 8. Architecture ns-3.

En comparaison avec d'autres simulateurs à événements discrets, ns-3 se distingue par les objectifs de conception de haut niveau. De nombreux simulateurs utilisent un langage de modélisation spécifique à un domaine pour décrire des modèles et des flux de programme. Ns-3 utilise C++ ou Python, permettant aux utilisateurs de profiter du plein appui de chaque langage.

Les événements de simulation Ns-3 sont simplement les appels de fonctions programmés pour s'exécuter à un instant de simulation prescrit. Toute fonction peut être transformée en un événement prévu par l'utilisation d'une fonction de rappel (*callback function*). Les rappels (*callbacks*) sont également largement utilisés dans le simulateur pour réduire le temps de compilation des dépendances entre les objets de simulation.

La conception de la simulation est orientée vers les cas d'utilisation qui permettent au simulateur d'interagir avec le monde réel. Les objets de paquets dans ns-3 sont stockés intérieurement en tant que tampons d'octets (semblables à des paquets des systèmes d'exploitation réelles) prêts à être sérialisés et envoyés sur une interface de réseau réel.

Les nœuds de ns-3 sont calqués sur l'architecture du réseau Linux. Les interfaces et les objets (sockets, net devices) sont alignées avec celles d'un ordinateur Linux. Cela facilite la réutilisation du code et améliore le réalisme des modèles, et rend les flux de contrôle de simulateur facile de comparer avec les systèmes réels.

Ns-3 dispose d'un système basé sur les attributs intégrés pour gérer les valeurs par défaut et les valeurs par-instance pour les paramètres de simulation. Toutes les valeurs par défaut pour les paramètres configurables sont gérées par ce système, intégré avec le traitement d'arguments en ligne de commande, la documentation Doxygen, et un sous-système de configuration basé sur XML, et un autre facultatif basésur GTK.

Le projet n'a pas maintenu un environnement de développement intégré (IDE) permettant de configurer, déboguer, d'exécuter et de visualiser des simulations dans une seule fenêtre, comme on en le trouve dans d'autres simulateurs. Au lieu de cela, le principe est de

travailler à la ligne de commande et d'intégrer des outils de visualisation et configuration si nécessaire.

3. Implémentation de l'application Tracker

Les différents détails de l'implémentation sont présentés dans cette section.

3.1. Les différents messages utilisés dans Tracker

Nous citons ci-dessous les principaux messages utilisés par l'application :

Message	Type de message	Fonction
REQ	Broadcast	Message (requête) envoyé par initiateur principal ou secondaire pour tous les nœuds voisins.
RP/RN	Unicast	Message de données envoyé par les nœuds voisins vers leur initiateur (principal ou secondaire)
UNI	Unicast	Message envoyé par l'initiateur courant (principal ou secondaire) pour choisir le prochain initiateur.
Obj	Unicast (multi saut)	Message de données, transmit depuis un initiateur secondaire vers l'initiateur principal, contient les informations visées par l'application.

Tableau 2 : Types de messages échangés.

3.2. Structure de données

Les différentes structures de donnes utilisées sont ce qui suit :

- **La rencontre entre deux véhicules**

Chaque véhicule de réseau possède un vecteur local dans lequel il inscrit tous les véhicules qu'il a rencontré. Une rencontre est définit comme suit :

```
1  struct Meet
2  {
3      int id; // Identification de véhicule.
4      Time t; // L'instant de la rencontre.
5      Vector p; // Un vecteur pour enregistrer la position.
6  };
7
8
```

- **Le véhicule candidat**

Les véhicules ayant répondus à une requête serons enregistrés localement comme candidats dans un vecteur. Un candidat se présente comme suit :

```
struct Candidat
{
    bool positive; // La positivité de la réponse reçue.
    Ipv4Address ip; // L'adresse de véhicule candidat.
    double dist; // La distance entre les deux véhicules.
};
```

- **Le paramètre d'une collecte**

Chaque initiateur doit être au courant des paramètres actuels de la collecte. Un paramètre est définit comme suit :

```
struct Parameter
{
    int maxdst; // Le nombre maximal de sauts.
    double maxdur; // La durée maximale de la collecte.
};
```

3.3. Le routage des messages multi-saut

En raison de la grande mobilité des nœuds dans les VSNs, concevoir un protocole de routage capable de calculer et de maintenir efficacement les chemins de routage ente les véhicules, représente aujourd'hui un problème de recherche majeur. Jusqu'à présent, plusieurs protocoles de routage ont été développés, certains d'entre eux ont été obtenus, adaptés et améliorés à partir d'algorithmes proposés dans le passé pour les MANET. Ces protocoles, malgré le fait qu'ils ont atteints de bonnes performances pour les MANET (Mobile Ad-hoc Networks), ils ne sont pas en mesure de garantir le même niveau d'efficacité dans l'élaboration de scénarios VANET. Par conséquent, de nouvelles idées et stratégies plus sophistiquées ont été développées. Un grand nombre de ces nouvelles approches calcule des itinéraires à partir de l'information sur la position de nœud où d'autres protocoles divisent les nœuds en cluster (petits groupes)…

A l'aide d'une étude comparative présentée dans [26], nous avons fini par choisir le protocole de routage AODV pour router les messages de type multi-saut.

❖ AODV (Ad hoc On Demand Distance Vector)

AODV (pour Ad hoc On Demand Distance Vector) [réf] est un protocole de routage destiné aux réseaux mobiles (réseau ad hoc). Il est à la fois capable de routage unicast et multicast. Il est libre de boucle et s'assure que la route soit la plus courte possible, auto-démarrant et s'accommode d'un grand nombre de nœuds mobiles (ou intermittents). Lorsqu'un nœud source demande une route, il crée les routes à la volée et les maintient tant que la source en a besoin. Pour les groupes multicast, AODV construit une arborescence.

Ce protocole de routage est peu gourmand en énergie et ne nécessite pas de grande puissance de calcul, il est donc facile à installer sur de petits équipements mobiles.

3.4. La mobilité véhiculaire

Pour avoir un environnement de simulation des VSNs proche de la réalité, chaque nœud doit avoir un modèle de mobilité similaire à celui d'un vrai véhicule.

❖ SUMO

SUMO [réf] (pour Simulation of Urban Mobility) est un paquage microscopique open source, largement portable, conçu pour la simulation de trafic routier. Capable de gérer de grands réseaux routiers. Il est principalement développé par des employés de l'Institut des systèmes de transport au Centre aérospatial allemand. SUMO est sous licence GPL.

SUMO est accompagné par une interface graphique appelée SUMO-GUI (voir figure 9 ci-après), permet de visualiser les réseaux routiers crées à l'aide de SUMO, et les flux des véhicules circulant dans ces réseaux.

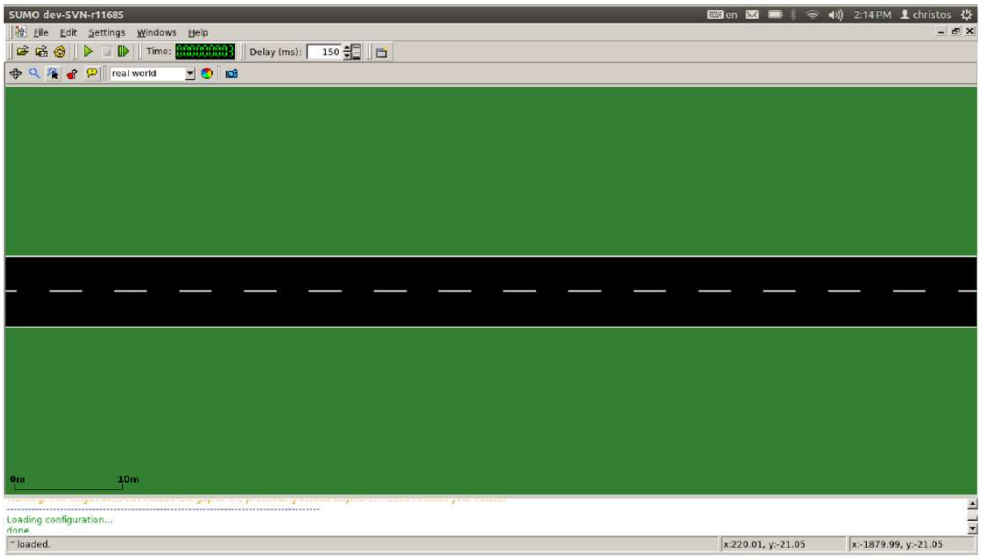

Figure 9 : SUMO-GUI.

❖ SUMO et ns-3

La figure 10 ci-dessous schématise la collaboration entre SUMO et Ns-3 dans la simulation d'un réseau sans fil à mobilité véhiculaire.

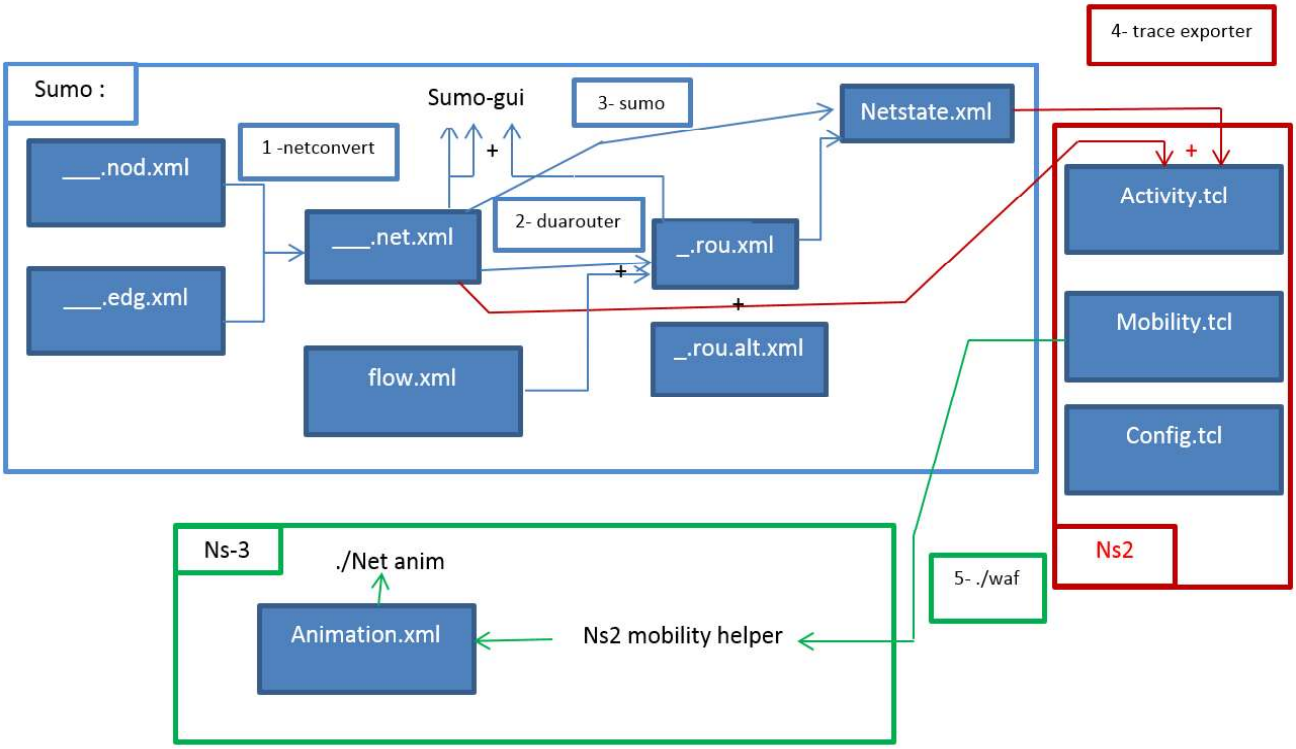

Figure 10 : Diagramme de collaboration de SUMO et ns-3

- Les commandes à utiliser

1- Netconvert : ~# netconvert -n ___.xnod.xml –e ___.edg.xml –o ___.net.xml
2- duaronter : ~# duarouter –n ___.net.xml –f flow.xml -o ___.rou.xml
3- sumo :# sumo –n ___.net.xml –r ___.rou.xml - -netstate-dump netstate.xml
4- trace exporter :~# java-jar/…/sumo-0.16.0/tools/trace exporter/trace exporter.jar ns2
-n___.net.xml –t netstate.xml – a activity.tcl – m mobility.tcl – c config.tcl
- p1 – b0 –e 90
5- ./waf : ~# ./waf –run "scratch/simulationName -- tracefile = mobility.tcl -- node num=..
-- duration = --.- -- log file = ex.log "

- Les fichiers à définir pour SUMO

- **___.nod.xml :** contient les positions des intersections entre les routes (nœuds).

Syntaxe :

```
<nodes>
        <node id="0" x="0" y="0" />
        …
</nodes>
```

- **___.edg.xml :** contient les routes elles-mêmes.

Syntaxe :

```
<edges>
        <edge id="a" from="0" to="1" priority="1" numLanes="3" speed="100" />
        …
</edges>
```

- **___ .flow.xml :** contient la définition des flux de véhicules entre les nœuds.

Syntaxe :

```
<flows>
        <flow id="0" from="a" to="l" begin="0" end="40" number="50" />
        …
</flows>
```

3.5. Implémentation des différentes fonctionnalités

A chaque module ajouté à ns-3 est associé un sous dossier ayant le même nom de module, et contenant les sous dossiers suivants :

- doc
- examples
- helper
- model
- test

Nous avons implémenté deux classes, la première dans le dossier « model », nommée : Tracker, et est la classe principale de l'application. Et l'autre dans le dossier « helper », nommée : TrackerHelper, cette classe aidera les utilisateurs à installer l'application sur leurs nœuds.

Afin d'implémenter les différentes fonctionnalités de notre application nous avons défini / redéfini les méthodes suivantes :

3.5.1. Le script de simulation

Pour tester le fonctionnement de notre application nous avons implémenté le script de simulation en passant par les étapes suivantes

A. L'initialisation

Nous avons commencé par les déclarations nécessaires, et l'affection des valeurs par défaut.

```
1   NS_LOG_INFO ("Initialisation...");
2
3   uint32_t nodeNum (10); // nombre de noeuds à creer
4   double step (100); // Pour le modèle de mobilité fixe
5   double totalTime (60); // Le temps de la simulation.
6   bool pcap (false); // pour pcap tracing.
7   bool printRoutes (false); // Ecrire les tables de routage des noeuds.
8   int t; // For simulation time calculation.
9   std::string traceFile; // For sumo mobility.
10  std::string logFile; // For sumo mobility.
11
12  std::ofstream os;
13
14  NodeContainer nodes; // Conteneur de noeuds.
15  NetDeviceContainer devices; // Conteneur de cartes réseau.
16  Ipv4InterfaceContainer interfaces; // Conteneur d'interfaces réseau.
17
```

B. La configuration

Nous avons ensuite configuré notre application pour recevoir des valeurs par-instance depuis la ligne de commande (valeurs de l'utilisateur).

```
    NS_LOG_INFO ("Configuration...");

    //LogComponentEnable("AodvRoutingProtocol", LOG_LEVEL_ALL); // Comment th
    SeedManager::SetSeed (1234567);
    CommandLine cmd;
    cmd.AddValue ("pcap", "Write PCAP traces.", pcap);
    cmd.AddValue ("printRoutes", "Print routing table dumps.", printRoutes);
    cmd.AddValue ("nodeNum", "Number of nodes.", nodeNum);
    cmd.AddValue ("time", "Simulation time, s.", totalTime);
    cmd.AddValue ("step", "Grid step, m", step);
    cmd.AddValue ("traceFile", "Ns2 movement trace file", traceFile);
    cmd.AddValue ("logFile", "Log file", logFile);
    cmd.Parse (argc, argv);
```

C. L'exécution

Cette étape comprend les sous-étapes suivantes.

C.1. La création des nœuds

```
NS_LOG_INFO ("-> Creating nodes...");

    nodes.Create (nodeNum);
    // Name nodes
    for (uint32_t i = 0; i < nodeNum; ++i)
    {
        std::ostringstream os;
        os << "node-" << i;
        Names::Add (os.str (), nodes.Get (i));
    }
```

C.2. La mise en place d'une mobilité véhiculaire

Si on ne spécifie pas les fichiers de modèle de mobilité (sortie de SUMO), on utilise un modèle de mobilité statique (nœuds fixes).

```
NS_LOG_INFO ("-> Setting up mobility model...");

 if (traceFile.empty () || logFile.empty ())
 {
     std::cout << "\"logFile\" and \"traceFile\" are not specified... using Constant

     MobilityHelper mobility;
     mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
                                    "MinX", DoubleValue (0.0),
                                    "MinY", DoubleValue (0.0),
                                    "DeltaX", DoubleValue (step),
                                    "DeltaY", DoubleValue (0),
                                    "GridWidth", UintegerValue (nodeNum),
                                    "LayoutType", StringValue ("RowFirst"));
     mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
     mobility.Install (nodes);

     for (NodeContainer::Iterator i = nodes.Begin (); i != nodes.End (); ++i)
     {

     }


 }
 else // Le fichier traceFile (de SUMO) est spécifié.
 {
     // Enable logging from the ns2 helper
     LogComponentEnable ("Ns2MobilityHelper",LOG_LEVEL_INFO);

     // Create Ns2MobilityHelper with the specified trace log file as parameter
     Ns2MobilityHelper ns2 = Ns2MobilityHelper (traceFile);

     // open log file for mobility output
     std::ofstream os;
     os.open (logFile.c_str ());

     ns2.Install (); // configure movements for each node, while reading trace file

     // Configure callback for logging
     Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
                      MakeBoundCallback (&CourseChange, &os));
 }

```

C.3. La création des net devices

La création de la couche mac en mode Ad hoc.

```
NS_LOG_INFO ("-> Creating devices...");

NqosWifiMacHelper wifiMac = NqosWifiMacHelper::Default ();
wifiMac.SetType ("ns3::AdhocWifiMac");
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default ();
YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default ();
wifiPhy.SetChannel (wifiChannel.Create ());
WifiHelper wifi = WifiHelper::Default ();
wifi.SetRemoteStationManager (  "ns3::ConstantRateWifiManager",
                                "DataMode",
                                StringValue ("OfdmRate6Mbps"),
                                "RtsCtsThreshold",
                                UintegerValue (0));
devices = wifi.Install (wifiPhy, wifiMac, nodes);

if (pcap) // Si pcap = true, pcap tracing.
{
    wifiPhy.EnablePcapAll (std::string ("aodv"));
}
```

C.4. L'installation des piles internet

Installation des piles internet sur les nœuds, puis instanciation de protocole de routage AODV, et configuration de l'adressage IP, et enfin associer les interfaces réseau (déclarées dans l'initialisation) au cartes réseau.

```
NS_LOG_INFO ("-> Installing internet stacks...");

AodvHelper aodv;
// you can configure AODV attributes here using aodv.Set(name, value)
InternetStackHelper stack;
stack.SetRoutingHelper (aodv); // has effect on the next Install ()
stack.Install (nodes);
Ipv4AddressHelper address;
address.SetBase ("10.0.0.0", "255.0.0.0");
interfaces = address.Assign (devices);

if (printRoutes) // Si printRoutes = true, écrire les tables de routage des noeu
{
    Ptr<OutputStreamWrapper> routingStream;
    routingStream = Create<OutputStreamWrapper> ("aodv.routes", std::ios::out);
    //aodv.PrintRoutingTableAllAt (Seconds (8), routingStream);
    aodv.PrintRoutingTableAllEvery(Seconds (5), routingStream);
}
```

C.5. L'installation de l'application

Installation de notre application sur les nœuds.

```
NS_LOG_INFO ("-> Installing applications...");

LogComponentEnable ("Tracker", LOG_LEVEL_INFO);

TrackerHelper tracker;
tracker.SetAttribute ("Interval", TimeValue (Seconds (6.0)));

ApplicationContainer applications = tracker.Install (nodes);

applications.Start (Seconds (0));
applications.Stop (Seconds (totalTime));

```

C.6. Simulation

Démarrage de la simulation, et calcul du temps pris par la simulation.

```
std::cout << "Starting simulation for " << totalTime << " s ...\n"; // S
t = time (0);

AnimationInterface anim ("lm.xml"); // Write Netanime .xml trace.
Simulator::Stop (Seconds (totalTime));
Simulator::Run ();
Simulator::Destroy ();

if (! (traceFile.empty () || logFile.empty ()))
{
    os.close (); // close mobility log file
}

t = time (0) - t;
std::cout << "Simulation took " << t << " s." << std::endl;

```

3.5.2. Le helper

Le helper installera l'application sur les nœuds à la place de l'utilisateur, on distingue trois méthodes essentielles dans le helper.

- **SetAttribute ()** : permet d'envoyer des attributs utiles pour la configuration de l'application.

```
void
TrackerHelper::SetAttribute (std::string name, const AttributeValue &value)
{
    m_factory.Set (name, value);
}
```

- **Install ()** :

```
ApplicationContainer
TrackerHelper::Install (Ptr<Node> node) const
{
    return ApplicationContainer (InstallPriv (node));
}

ApplicationContainer
TrackerHelper::Install (std::string nodeName) const
{
    Ptr<Node> node = Names::Find<Node> (nodeName);
    return ApplicationContainer (InstallPriv (node));
}

ApplicationContainer
TrackerHelper::Install (NodeContainer c) const
{
    ApplicationContainer apps;
    for (NodeContainer::Iterator i = c.Begin (); i != c.End (); ++i)
    {
        apps.Add (InstallPriv (*i));
    }

    return apps;
}
```

- **InstallPriv ()** : (méthode privée).

```
Ptr<Application>
TrackerHelper::InstallPriv (Ptr<Node> node) const
{
    Ptr<Application> app = m_factory.Create<Tracker> ();
    node->AddApplication (app);

    return app;
}
```

3.5.3. L'application Tracker

Nous présentons ci-dessous les principales fonctions implémentées dans notre application.

- **DoDispose ()** : Cette méthode est appelée juste avant la destruction d'un objet de type Tracker, elle déclenche une série d'appels de fonctions aux classes prédécesseurs.

```
void
Tracker::DoDispose (void)
{
    NS_LOG_FUNCTION (this);
    Application::DoDispose ();
}
```

- **GetTypeId ()** : Cette méthode permet de gérer les valeurs par défaut et valeurs par-instance fournis par l'utilisateur.

  Cette méthode permet aussi de donner une signature distinctive aux objets de type Tracker.

```
TypeId
Tracker::GetTypeId (void)
{
    static TypeId tid = TypeId ("ns3::Tracker")
        .SetParent<Application> ()
        .AddConstructor<Tracker> ()
        .AddAttribute ("Port", "Port on which we listen for incoming packets.",
                       UintegerValue (9),
                       MakeUintegerAccessor (&Tracker::ms_port),
                       MakeUintegerChecker<uint16_t> ())
        .AddAttribute ( "Interval",
                       "The time to wait between hello packets",
                       TimeValue (Seconds (3.0)),
                       MakeTimeAccessor (&Tracker::m_interval),
                       MakeTimeChecker ())
        .AddAttribute ( "RemoteAddress",
                       "The destination Address of the outbound packets",
                       AddressValue(),
                       MakeAddressAccessor (&Tracker::mc_peerAddress),
                       MakeAddressChecker ())
        .AddAttribute ( "RemotePort",
                       "The destination port of the outbound packets",
                       UintegerValue (0),
                       MakeUintegerAccessor (&Tracker::mc_peerPort),
                       MakeUintegerChecker<uint16_t> ())
        .AddAttribute ( "Initiator",
                       "The collect initiator",
                       UintegerValue(4),
                       MakeUintegerAccessor (&Tracker::m_col_initiator),
                       MakeUintegerChecker<uint16_t> ())
        .AddAttribute ( "Target",
                       "Target Vehicle",
                       UintegerValue(6),
                       MakeUintegerAccessor (&Tracker::m_col_target),
                       MakeUintegerChecker<uint16_t> ())
        .AddAttribute ( "startTime",
                       "The collect start time",
                       TimeValue (Seconds (50.0)),
                       MakeTimeAccessor (&Tracker::m_col_startTime),
                       MakeTimeChecker ())
        .AddTraceSource ("Tx", "A new packet is craeted and is sent",
                       MakeTraceSourceAccessor (&Tracker::mc_txTrace))
        /* Add attributes */
        ;
    return tid;
}

```

- **StartApplication** () : Cette méthode permet de démarrer l'application.

  Puisque notre application sera installée dans un environnement ad hoc, elle doit avoir un aspect peer-to-peer, donc avoir jouer le roule d'un serveur, et d'un client au même temps.

```
void
Tracker::StartApplication (void)
{
    StartServer ();
    StartClient ();
//------------------------------------------
    unsigned int x = GetNode ()->GetId ();
    if (x == m_col_initiator)
    {
        Simulator::Schedule (m_col_startTime, &Tracker::Track, this);

        Ptr<Ipv4> ipv4 = GetNode ()->GetObject<Ipv4>();
        m_ip_initiator = ipv4->GetAddress (1, 0).GetLocal ();
    }
}

```

- **StartServer ()** : Permet de démarrer le coté serveur de l'application.

```
void
Tracker::StartServer (void)
{
    NS_LOG_FUNCTION (this);

    if (ms_socket == 0)
    {
        TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
        ms_socket = Socket::CreateSocket (GetNode (), tid);
        InetSocketAddress local = InetSocketAddress (Ipv4Address::GetAny (), ms_port);
        ms_socket->Bind (local);

        ms_socket->SetAllowBroadcast (true);

        if (addressUtils::IsMulticast (ms_local))
        {
            Ptr<UdpSocket> udpSocket = DynamicCast<UdpSocket> (ms_socket);
            if (udpSocket)
            {
                udpSocket->MulticastJoinGroup (0, ms_local);
            }
            else
            {
                NS_FATAL_ERROR ("Error: Failed to join multicast group");
            }
        }
    }

    ms_socket->SetRecvCallback (MakeCallback (&Tracker::HandleRead, this));
}

```

- **StartClient () :**Permet de démarrer le coté client de l'application.

```
void
Tracker::StartClient (void)
{
    NS_LOG_FUNCTION (this);

    if (mc_socket == 0)
    {
        TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
        mc_socket = Socket::CreateSocket (GetNode (), tid);

        mc_socket->SetAllowBroadcast (true);

        if (Ipv4Address::IsMatchingType (mc_peerAddress) == true)
        {
            mc_socket->Bind ();
            mc_socket->Connect (InetSocketAddress ( Ipv4Address::ConvertFrom(mc_peerAddress),
                                                    mc_peerPort));
        }
    }

    mc_socket->SetRecvCallback (MakeCallback (&Tracker::ClientHandleRead, this));

}
```

- **StopApplication ()** : Cette méthode permet d'arrêter l'application.

```
void
Tracker::StopApplication (void)
{
    StopClient ();
    StopServer ();
}

```

- **StopServer ()** : Permet d'arrêter le coté serveur de l'application.

```
void
Tracker::StopServer ()
{
    NS_LOG_FUNCTION (this);

    if (ms_socket != 0)
    {
        ms_socket->Close ();
        ms_socket->SetRecvCallback (MakeNullCallback<void, Ptr<Socket> > ());
    }
}

```

- **StopClient ()** : Permet d'arrêter le coté client de l'application.

```
void
Tracker::StopClient ()
{
    NS_LOG_FUNCTION (this);

    if (mc_socket != 0)
    {
        mc_socket->Close ();
        mc_socket->SetRecvCallback (MakeNullCallback<void, Ptr<Socket> > ());
        mc_socket = 0;
    }

    Simulator::Cancel (mc_sendEvent);
}
```

- **SetRemote () :** Permet de décider auquel nœud distant seront envoyés les messages au cours de prochain envoie.

```
void
Tracker::SetRemote (Address ip, uint16_t port)
{
    NS_LOG_FUNCTION (this << ip << port);
    mc_peerAddress = ip;
    mc_peerPort = port;
}

void
Tracker::SetRemote (Ipv4Address ip, uint16_t port)
{
    NS_LOG_FUNCTION (this << ip << port);
    mc_peerAddress = Address (ip);
    mc_peerPort = port;
}
```

- **ChangeRemote () :** Permet de changer le nœud distant auquel seront envoyés les messages au cours de prochain envoie.

```
void
Tracker::ChangeRemote (Address ip, uint16_t port)
{
    StopClient ();
    SetRemote (ip, port);
    StartClient ();
}

void
Tracker::ChangeRemote (Ipv4Address ip, uint16_t port)
{
    StopClient ();
    SetRemote (ip, port);
    StartClient ();
}
```

- **Send () :** Permet d'envoyer un message.

```
void
Tracker::Send (void)
{
    NS_LOG_FUNCTION (this);

    NS_ASSERT (mc_sendEvent.IsExpired ());

    Ptr<Packet> p;
    if (mc_dataSize)
    {
        NS_ASSERT_MSG (mc_dataSize == mc_size, "Tracker::Send(): mc_size and mc_dataSize inconsistent");
        NS_ASSERT_MSG (mc_data, "Tracker::Send(): mc_dataSize but no mc_data");
        p = Create<Packet> (mc_data, mc_dataSize);
    }
    else
    {
        p = Create<Packet> (mc_size);
    }

    mc_txTrace (p);
    mc_socket->Send (p);

    ++mc_sent;
}

```

- **GetData () :** Permet d'extraire les données d'un message reçu.

```
std::string
Tracker::GetData (Ptr<Packet> packet)
{
    std::ostringstream oss;
    uint8_t *buffer = new uint8_t [packet->GetSize ()];
    packet->CopyData (buffer, packet->GetSize ());
    for (uint i = 0; i < packet->GetSize () - 1; i++)
    {
        oss << buffer[i];
    }

    std::string data;
    data = oss.str ();
    return data;
}

```

- **SetFill ()** : Permet de modifier le contenu de message à envoyer.

```
void
Tracker::SetFill (std::string fill)
{
    NS_LOG_FUNCTION (this << fill);

    uint32_t dataSize = fill.size () + 1;

    if (dataSize != mc_dataSize)
    {
        delete [] mc_data;
        mc_data = new uint8_t [dataSize];
        mc_dataSize = dataSize;
    }

    memcpy (mc_data, fill.c_str (), dataSize);

    mc_size = dataSize;
}

```

- **HandleRead () :** Cette méthode sera appelée lorsque des données sont reçues.

```
void
Tracker::HandleRead (Ptr<Socket> socket)
{
    NS_LOG_FUNCTION (this << socket);

    Ptr<Packet> packet;
    Address from;
    while ((packet = socket->RecvFrom (from)))
    {
        // Traitement des données reçues.
    }
}
```

- **GetPosition () :** Permet d'avoir la position actuelle de nœud en question.

```
Vector
Tracker::GetPosition ()
{
    Ptr<MobilityModel> mob = GetNode ()->GetObject<MobilityModel>();
    Vector pos = mob->GetPosition ();
    return pos;
}
```

Conclusion

Dans ce dernier chapitre, nous avons présenté l'environnement de simulation ns-3 et son principe de fonctionnement, ainsi que l'outil de simulation de mobilité véhiculaire SUMO.

Nous avons aussi présenté les différentes étapes de l'implémentation de notre application, et fourni des pseudo-codes permettant de décrire les détails de notre implémentation, et servir de base pour des travaux à venir.

Nous avions l'intention d'évaluer les performances de notre application mais par faute de temps, nous nous somme pas arrivé à bout.