Chapitre III. Conception

Introduction

L'objectif principal de notre travail est de concevoir une application capable de fournir au conducteur d'un véhicule autoritaire la localisation d'un autre véhicule donné, dans un moment quelconque donné, le choix de la plus courte route en fonction de trafic, des obstacles…etc., reste parmi les tâches exécutées par le conducteur.

La mise en place d'une telle application dans un réseau véhiculaire impose la présence préalable de plusieurs mécanismes, la communication entre véhicules est une d'elles. Cette dernière permet l'échange d'informations locales entre véhicules telles que la vitesse des voisins, la situation de la route, la détection des accidents ou d'incidents criminels (vol par exemple), la position locale…etc. La collecte de ces données se fait donc en suivant différentes approches à propos desquelles sont implémentés divers protocoles.

En s'appuyant sur le choix fait dans une thèse de master [1], nous avons fini par choisir le mécanisme de collecte présenté par COL [23], et essayer de l'optimiser et de l'adapter pour répondre aux besoins de notre application. Cette dernière sera nommée Tracker.

Avant de passer à la conception, nous avons vu avantageux le fait de présenter en premier lieu les hypothèses devant être satisfaites pour le bon fonctionnement de notre application.

1. Hypothèses

Afin de déterminer notre travail, nous avons jugé important de délimiter le cadre dans lequel nous allons le développer. Nous avons donc fixé certaines hypothèses qui se présentent comme suit :

- Absence d'infrastructure : n'importe quel véhicule a potentiellement la possibilité de collecter des données au-delà de son voisinage direct, et ce, en utilisant uniquement des communications inter-véhiculaires. Cela est avantageux en terme d'efficacité car une application fonctionnelle sans infrastructure, est fonctionnelle avec, alors que le contraire n'est pas toujours vrai.

- Chaque véhicule doit avoir les caractéristiques suivantes :
  - Il doit être équipé d'un dispositif IEEE 802.11 pour la communication V2V.
  - Il doit être équipe d'un capteur lui permettant de déterminer sa position, ou un système GPS.
  - Il doit avoir une copie de la carte routière (digital map).
  - Il doit être équipe d'une interface permettant au conducteur d'interagir avec l'application.
  - Il doit avoir un dispositif de photographie d'immatriculation, et un système de reconnaissance visuelle.

2. Description de l'application

Pour réduire le coût de communication induit par la diffusion périodique par tous les véhicules, nous proposons pour chaque véhicule de garder localement sa vue locale, et de la communiquer qu'on cas d'une demande. Nous proposons aussi, pour tout message communiqué, d'inclure que les informations fortement utiles. Pour remédier à cela nous proposons de diviser ces messages en plusieurs types.

Dans notre application, toute collecte est déclenché par un véhicule spécifique appelé ***initiateur principal*** (racine) [1]. Celui-ci lance une demande de collecte (diffusée au niveau de son voisinage) contenant deux identifiants, un pour identifier la collecte en cours, et l'autre identifiant le véhicule à suivre. Cette demande est un message appelé (***REQ***).

Suite à une demande de collecte, Les véhicules voisins répondent (***RP*** : réponse positive, ou ***RN*** : réponse négative) avec leur vues locales accompagnées de quelque informations de contrôle utiles pour choisir le prochain initiateur (***initiateur secondaire***), afin de transmettre la requête au saut suivant.

L'initiateur secondaire reçoit un message (***UNI***), lui informant qu'il est le prochain initiateur, ce message contient plusieurs informations lui permettant de continuer la collecte.

Toute repense positive (***RP***) doit être transmise (en multi saut) à l'initiateur principal en forme de message (***Obj***). Pour cela, chaque initiateur secondaire doit être au courant de l'adresse IP de l'initiateur principal.

Une collecte doit être finie, dans notre application elle est limitée par une durée (***MaxDur***) et un nombre maximal de sauts depuis l'initiateur principal (***MaxDst***).

2.1. L'initialisation de l'application

Au départ de chaque collecte, seul l'initiateur principal est impliqué dans le processus de collecte. Mais le nombre de véhicules concernés augmente au fur et à mesure de la propagation des messages issus de l'initiateur principal à travers le réseau. Cependant, l'exploration de réseau est bornée à *MaxDst* (nombre maximal de sauts) de l'initiateur principal, et la durée de cette exploration est aussi bornée à *MaxDur* (Durée maximale). Chaque collecte possède un identifiant unique et propre à elle.

Le véhicule initiateur principal initialise les données propres à la collecte en cours à l'aide de cet Algorithme :

Algo 1 :

```
1. initialization (MaxDist, MaxDur, Target)
2. Debut
3.     col_active <- faux ;
4.     col_id <- 0 ; // Le numéro de la collecte
5.     maxdst <- MaxDist ;
6.     maxdur <- MaxDur ;
7.     target_id <- Target ; // Identifiant de véhicule à suivre.
8.     IPr <- GetIp () ; // Initialiser IPr a l'ip locale de véhicule.
9. Fin.
```

2.2. Le lancement d'une collecte

Dès l'instant que l'initiateur principal décide de débuter une opération de suivi (collecte), il envoie un message (requête) composé de trois champs à destination de tout véhicule voisin. Ces champs correspondent respectivement au type de message (***Type_Message***), numéro de la collecte (***Collect_ID***), et l'identifiant de véhicule à traquer (***Target_ID***).

Le format de message :

Type_Message = REQ	Collect_ID	Target_ID

L'algorithme suivant permet de démarrer une collecte de données :

Algo 2 :

```
1.  StartCollect ()
2.  Debut
3.     si col_active == vrai alors :
4.        sortir () ;
5.     col_active <- vrai ;
6.     col_id += 1 ;
7.     message : Packet; // Declaration de message de type packet.
8.     message << REQ ; // Type de message : requête.
9.     message << col_id ;
10.    message << target_id ;
11.    DoBroadcast (message) ; // envoyer le message aux voisins en diffusion.
12. Fin.
```

Avant de commencer une collecte, l'initiateur principal vérifie s'il y a déjà une collecte en cours, et quitte si c'est le cas, car une seule collecte est autorisée à la fois (s'il est nécessaire de lancer plusieurs collectes à la fois, on peut recourir à la programmation parallèle). Dans le cas contraire il déclare qu'il y a une collecte en cours, incrémente le numéro de collecte, remplis le message à envoyer par les informations requises, et l'envoie enfin.

2.3. Le choix de prochain initiateur

Le véhicule initiateur (principal ou secondaire) doit être capable d'élire un autre véhicule pour continuer la collecte au-delà de son voisinage, et de relancer la collecte avec les paramètres actuels.

Après avoir choisi le prochain initiateur, le véhicule (initiateur courant) doit lui envoyer un message pour l'informer de la situation. Ce message doit contenir le type de message (***Type_Message***), L'adresse ip d'initiateur principal (***IPr***), le numéro de la collecte (***Collect_ID***), Le nombre de sauts restants (***MaxDst***), La durée restante (***MaxDur***), et l'identifiant de véhicule à traquer (***Target_ID***).

Le Format de message :

UNI	IPr	Collect_ID	MaxDst	MaxDur	Target_ID

Pour ce but, nous fournissons l'algorithme suivant :

Algo 3 :

```
1.  NextInitiatorElection ()
2.  Debut
3.      si MaxDur == 0 ou MaxDst == 0 alors
4.              sortir () ;
5.      nextInitiator : Candidat; // Déclaration de « nextInitiator » de type Candidat.
6.      nextInitiator <- Candidats [0]; // La première entrée dans le vecteur des candidats.
7.      Pour i de 0 à Candidats.size () faire
8.      Debut
9.              si Candidats[i].positive < nextInitiator.positive alors
10.                     // Si nextInitiator a répondu avec un message positif
11.                     // et Candidats[i] a répondu avec un message négatif
12.                     // On ignore.
13.             sinon si Candidats[i].positive < nextInitiator.positive alors
14.                     nextInitiator <- Candidats[i] ;
15.                     // Dans le cas contraire on prend Candidats[i].
16.             sinon si Candidats[i] est plus proche au rang/2 que nextInitiator alors
17.                     nextInitiator <- Candidats[i] ;
18.     Fin.
19.     message : Packet;
20.     message << UNI ; // Type de message : prochain initiateur.
21.     message << IPr ; // Adresse ip de l'initiateur principal.
22.     message << col_id ;
23.     message << maxdst - 1 ; // Calculer le nombre de sauts restants.
24.     message << maxdur – (Now () – startTime) ; // Calculer la durée restante.
25.     message << target_id ;
26.     Envoie (nextInitiator) ; // Envoie de message au prochain initiateur.
27. Fin.
```

Lorsqu'un véhicule (initiateur courant) tend à choisir le prochain initiateur, il commence par tester si le nombre maximal des sauts depuis le nœud initiateur principal est atteint ou la durée maximale est écoulée, et termine la collecte par sortir si c'est le cas. Ensuite, il prend initialement le premier véhicule dans le vecteur des candidats ayant répondu au message requête, et le compare avec le reste de vecteur. Il y a trois cas à considérer :

1) On a reçu une réponse positive depuis le véhicule déjà choisi, et négative depuis le véhicule en cours, ce qui fait que les voisins de véhicule choisi ont plus de chance d'avoir rencontré le véhicule suivi, donc on lui donne la priorité.
2) On fait de même dans le cas contraire, on donne la priorité au véhicule ayant une réponse positive.

3) Dans le cas où le véhicule déjà choisi et le véhicule candidat ont tous les deux répondu négativement ou positivement, on prend le plus proche d'eux de moitié du rang de véhicule, car :
   - Si on prend le plus proche, on risque d'augmenter le nombre de sauts inutilement.
   - Si on prend le plus loin, il risque de sortir de portée de véhicules avant d'être choisi comme prochain initiateur.

Cet algorithme se termine toujours par choisir un prochain initiateur.

2.4. La réception d'un message

Lorsqu'un véhicule rencontre un autre véhicule, chacun des deux enregistre l'identifiant de l'autre (numéro d'immatriculation en temps réel) et l'instant courant dans un vecteur local (***Meets***), et permet de rendre ces données disponibles en cas de réception de requête.

Si le véhicule recevant la requête trouve le véhicule traqué dans son vecteur Meets, il répond avec un message ***RP*** (Réponse Positive), contenant le type de message (***Type_Message***), la position locale (***localPosition***) (pour le calcul de la distance), le temps désignant l'instant quand la rencontre a été arrivée (***meet.Time***), et la position de cette dernière (***meet.Position***).

Le format de message :

RP	localPosition	Meet.Time	Meet.Position

Sinon, il répond avec un message ***RN*** (Réponse Négative), contenant que (***Type_Message***), et la position locale (***localPosition***).

Le format de message :

RN	localPosition

Les véhicules ayant répondu (positivement ou négativement) à la requête sont appelés ***candidats***, l'initiateur doit être capable de stocker tous les candidats localement et en élire un initiateur secondaire.

Lorsqu'un initiateur secondaire reçoit une réponse positive, il doit la transmettre à l'initiateur principal (en multi saut), ce message **Obj** (objectif de la collecte), doit contenir la localisation de véhicule suivi, et l'instant de la localisation.

Le format de message :

Obj	Meet.Time	Meet.Position

L'algorithme suivant décrit la façon dont les messages reçus sont traités :

Algo 4 :

```
1.  HandelRead (message)
2.  Debut
3.      Type_message << message ; // Extraire le type de message.
4.      si Type_Message == REQ alors // Le cas d'une requête.
5.      Debut
6.          si local_col_id == message_col_id alors
7.              Sortir () ; // Ignorer si V a déjà été impliqué dans cette collecte.
8.          si target_id est présent dans Meets alors
9.          Debut
10.             message << RP ; // Type de message : repense positive à la requête.
11.             message << localPosition ; // Pour le calcul de la distance.
12.             message << meet.Time ; // Le temps de la rencontre avec le véhicule visé.
13.             message << meet.Position ; // La position de véhicule visé.
14.         Fin.
15.         sinon
16.         Debut
17.             message << R ; // Type de message : repense négative à la requête.
18.             message << localPosition ;
19.         Fin.
20.         SendTo (message.RemoteIp, message) ; // Renvoyer le message d'où il vient.
21.     Fin.
22.
23.     sinon si Type_Message == R alors // Une réponse
24.     Debut
25.         Type_Message << message ;
26.         si Type_Message == N alors // Negative.
27.         Debut
28.             remotePosition << message ;
29.             AddCondidatEntry (0, message.RemoteIp, remotePosition) ;
30.         Fin.
31.         sinon si Type_Message == P alors
32.         Debut
33.             remotePosition << message ;
34.             AddCondidatEntry (1, message.RemoteIp, remotePosition) ;
35.             time << message ;       // Extraire les informations reçu à propos de
36.             position << message ; // véhicule traqué.
37.             // Remplir un message :
38.                 message << Obj ; // Type de message : Objectif.
39.                 message << meet.Time ;
40.                 message << meet.Position ;
41.             si IPr == GetIp () alors
42.                 SendTo (LocalHost, message) ;
43.             sinon
44.                 SendTo (IPr, message) ;
```

```
45.             Fin.
46.             si (!m_election_active)
47.                     ScheduleNextInitiatorElection (); // Le 1er message activera l'election.
48.     Fin.
49.
50.     sinon si Type_Message == UNI alors // C'est le véhicule élu.
51.     Debut
52.             IPr << message ; // Extraire les informations contenues dans le message.
53.             col_id << message ;
54.             maxdst << message ;
55.             maxdur << message ;
56.             target_id << messsage ;
57.             StartCollect () ; // Continuer la collecte avec les paramètres actuelles.
58.     Fin.
59.
60.     sinon si Type_Message == Obj alors
61.     Debut
62.             time << message ;
63.             position << message ;
64.             PrintCollectedData (time, position) ; // Afficher les données au conducteur.
65.   Fin.
```

Lorsqu'un véhicule reçoit un message, il extrait directement le premier champ qui correspond au type de message, et teste son contenu. Il y a quatre cas possibles :

1) Type_Message = REQ : Le message correspond à une requête, le véhicule recevant cette requête cherche l'identifiant de véhicule traqué dans son vecteur Meets, et répond par un message RP ou RN.
2) Type_Message = R : Le message est une réponse à une requête. Le véhicule recevant ce message, ajoute le véhicule l'ayant envoyé à son vecteur Candidats, pour pouvoir y choisir un initiateur secondaire, une fois une durée prédéfinit soit écoulé. Cette durée doit être suffisante pour que toutes les réponses soient reçues.
   Si La réponse est positive, on extrait les informations reçus : La position de véhicule suivi, et l'instant de la localisation. On doit envoyer ces informations à l'initiateur principal. Dans le cas où c'est déjà l'initiateur principal qui a reçu ces informations, il les renvoie à lui-même à l'aide de l'adresse de boucle locale.
3) Type_Message = UNI : Le message est un ordre pour que le véhicule recevant devient un initiateur secondaire et continue la collecte avec les paramètres actuelles.
4) Type_Message = Obj : Cela veut dire qu'on a atteint l'objectif de l'application, on extrait les données reçus et on les affiche au conducteur.

3. Exemple d'exécution

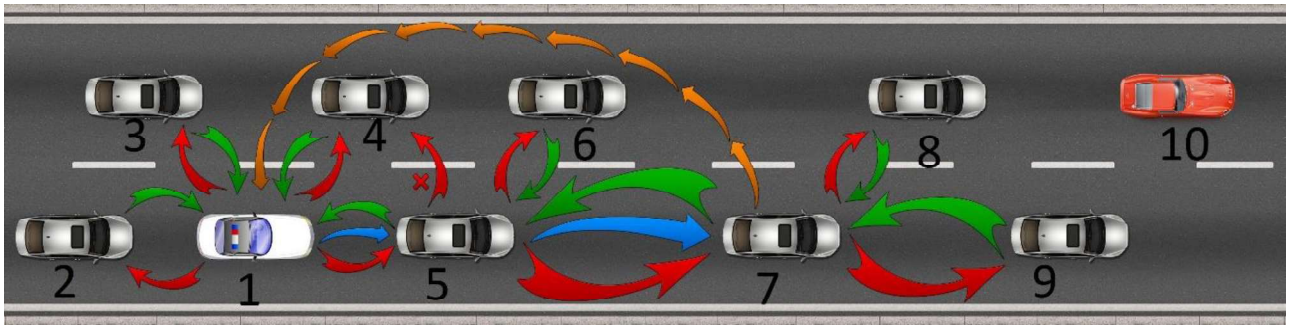

Figure 6 : Illustration d'un exemple d'exécution de l'application.

Légende :

Message REQ. Message UNI.

Message RP/RN. Message Obj.

Etapes d'exécution :

1- L'initiateur principal (véhicule 1) lance la requête ne diffusant un message de type REQ au voisins (véhicules 2, 3, 4, 5). Le véhicule suivi est le véhicule 10.
2- Les véhicules 2, 3, 4, 5 (voisins de l'initiateur principal) répondent avec RN (en ayant pas rencontrés le véhicule à suivre), et deviennent candidats pour l'élection de prochain initiateur secondaire.
3- L'initiateur principal choisit le véhicule 5, en lui envoyant un message UNI contenant les informations actuelles de la collecte. (durée de collecte restante, et nombre de sauts restant).
4- Le véhicule 5 devient un initiateur secondaire, relance la collecte et diffuse un message REQ aux véhicules voisins (véhicule 1, 4, 6, 7).
5- Les véhicules 6 et 7 répondent avec des messages RN, alors que les véhicules 1 et 4 n'envoyant rien, à cause d'avoir déjà participés à la collecte en cours.
6- L'initiateur secondaire courant (véhicule 5) choisit le prochain initiateur (véhicule 7) en lui envoyant un message UNI contenant les informations actuelles de la collecte.
7- Le véhicule 7 diffuse un message REQ aux voisins (véhicule 8 et 9).
8- Les véhicules 8 et 9 répondent avec des messages RP (réponse positive) a l'initiateur courant (véhicule 7).

9- Lorsque le véhicule 7 reçoit les messages RP, il transmit les informations concernant le véhicule suivi à l'initiateur principal en multi-saut.
10- L'initiateur principal (véhicule 1) reçoit les informations et les affiche au conducteur.

Conclusion

Dans ce chapitre, nous avons conçu une application, utilisant un protocole de collecte de données à la demande, adapté et inspiré par le protocole de collecte COL, valable pour les réseaux fortement dynamiques, telles que les VSNs. L'application résultat, nommée Tracker hérite des avantages de COL, et évite quelque diffusions inutiles, et l'inclusion des données inutiles, dans l'espoir de diminuer la surcharge de réseau

Le chapitre suivant sera consacré à l'implémentation et l'évaluation de performances de Tracker sous le simulateur NS-3.