X2012
30/06/2014

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

Implémentation d'un protocole de routage pour réseaux ad-hoc sans fil

Projet MODAL-NET

Malo Huard, Gurvan L'Hostis

{malo.huard, gurvan.lhostis}@polytechnique.edu

Abstract: Dans ce projet, nous cherchons à écrire un protocole de routage pour des réseaux *ad-hoc* sans fils. Ceci implique que le réseau soit décentralisé et donc le protocole distribué. Il doit pouvoir réagir rapidement à l'arrivée ou le départ de nœuds. Nous choisissons pour cela de se baser sur le protocole de Link State Routing.
Le rapport contient une explication des tenants et des aboutissants du sujet, le fonctionnement abstrait du protocole, l'écriture d'une spécification, le détail de notre implémentation, quelques résultats sur machine virtuelle ainsi que des pistes de développement.

Keywords: Ad-hoc network • LSR • Wireless

1. Sujet

Le point de départ de notre recherche de sujet est le constat que les réseaux WiFi *ad-hoc* proposés par Windows ne sont pas réellement *ad-hoc*.

En effet, ces prétendus réseaux *ad-hoc* utilisent le nœud créateur du réseau comme point d'accès et il existe donc bel et bien une hiérarchie. Puisque le créateur sert de borne WiFi, le réseau ne va donc pas plus loin que la portée de sa carte WiFi et il meurt quand le créateur le quitte.

Ce n'est pas ce que l'on peut attendre d'un réseau *ad-hoc*. Il nous a semblé qu'un tel réseau devrait réunir les propriétés suivantes :

- **Décentralisé** : tant que des individus participent au réseau, le réseau subsiste.
- **Portée illimitée** : un ordinateur à portée de n'importe lequel des membres du réseau peut s'y connecter.

Nous avons donc choisi pour sujet l'implémentation d'un protocole de routage pour réseaux *ad-hoc* sans fil. Ces types de réseaux, regroupés sous le terme de *MANET*[1], sont d'ores et déjà dotés de quatre protocoles de routage

[1] *MANET : Mobile ad hoc networks*

reconnus par *l'IETF*[2]. Celui sur lequel nous nous sommes basés pour écrire notre implémentation porte le nom de *Link State Routing Protocol*.

1.1. Fonctionnement

Le protocole que nous avons choisi fait partie de la catégorie des protocoles **proactifs**. Cela signifie que la découverte du réseau se fait par diffusion périodique sur le réseau des informations que chacun connait, de telle sorte que chacun des nœuds du réseau puisse en mettre à jour la topologie. Au contraire, l'autre méthode, dite **réactive**, consiste à demander la route systématiquement.
La découverte du réseau se fait *via* deux fonctionnalités :

- détection des voisins
- diffusion de la topologie

À chacune de ces fonctionnalités correspond un type de message que l'on envoie périodiquement, à savoir les `HELLO` et les `LSA`[3].
Il est à noter que les réseaux WiFi sont sujets aux collisions, c'est une des raisons pour lesquelles les messages sont renvoyés périodiquement même si aucun changement n'a lieu dans la topologie du réseau.
De plus, les délais de réémission des messages sont définis avec une marge d'écart aléatoire, ce qui évite des possibles collisions systématiques, c'est-à-dire deux nœuds qui envoient tous leurs messages simultanément.

1.1.1. Détection des voisins

Quand un nouveau nœud se connecte au réseau, il doit commencer par détecter les nœuds qui sont sur le même lien que lui, c'est à dire les voisins à un saut.
On utilise pour cela les messages `HELLO`, et c'est pourquoi les messages `HELLO` ne sont pas relayés.
Nous considérons que deux nœuds sont sur un même lien s'ils sont à portée WiFi l'un de l'autre. Il est à noter que ces liens ne sont pas nécessairement symétriques, en effet un nœud peut être à portée WiFi de la carte d'un autre sans que cela soit réciproque (voir figure 1). Ces liens ne sont pas utilisables pour le routage.
Les messages `HELLO` contiennent donc naturellement deux types d'information.
D'une part, on a la liste des voisins qui ont été entendus (*heard*), c'est-à-dire ceux dont on a reçu des `HELLO`.
D'autre part, les voisins qui ont été entendus et dont on sait qu'il nous ont entendu grâce à l'information contenue dans leurs `HELLO` sont classés *symmetric*. Le lien bidirectionnel nécessaire au routage est alors valide.

[2] *IETF : Internet Engineering Task Force*
[3] *LSA : Link State Advertisement*

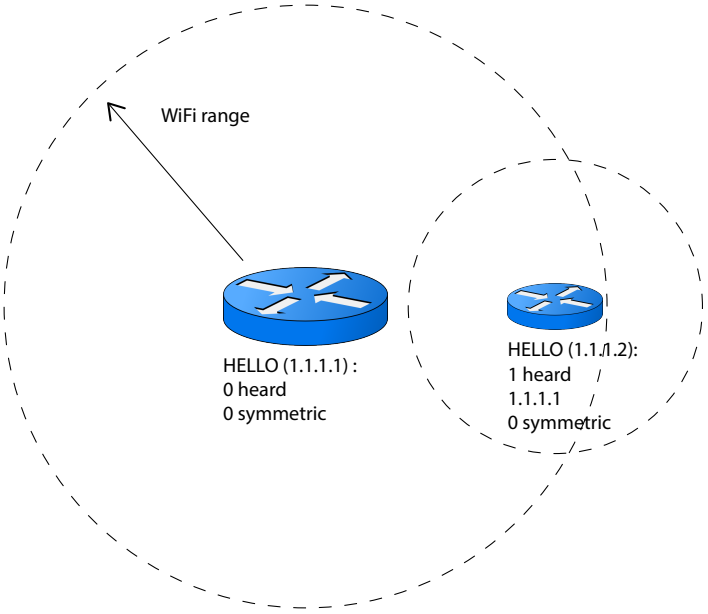

Figure 1. **Situation de lien asymétrique.**

Ces messages `HELLO` sont diffusés *via* l'adresse de broadcast de telle sorte que tous les nœuds du lien puissent le recevoir.

1.1.2. Diffusion de la topologie

Dans un second temps, un nœud qui vient de se connecter au réseau va avoir besoin de récupérer les informations sur l'ensemble des nœuds y appartenant, ainsi que les liens entre eux.

C'est là qu'entrent en jeu les messages `LSA`.

Ces messages contiennent la liste des voisins symétriques de leur expéditeur. Ainsi, la connaissance des `LSA` de tous les nœuds du réseau permet d'en connaître intégralement la topologie. Pour que les messages `LSA` soient connus de tout le monde, il faut les relayer. On dit qu'on inonde le réseau.

Dans la définition actuelle des choses, on voit qu'il y a un problème : les `LSA` peuvent être relayés indéfiniment le long d'un cycle de nœuds, ce qui noie le réseau. Pour que cela ne se produise pas, on fait appel à un **numéro de séquence**.

On assortit les `LSA` d'un numéro de séquence que l'on incrémente à chaque création, ce qui permet de comparer l'âge de deux `LSA`. Ainsi, un nœud du réseau n'a plus besoin de relayer tous les `LSA` qui lui parviennent, il peut se contenter de relayer une fois chaque version, ce qui suffit à ce que chaque nœud dispose toujours du `LSA` le plus récent possible tout en limitant l'inondation du réseau.

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Message type								Reserved												Message size											
Source address																															
Number of heard								Reserved																							
Heard address																															
...																															
Number of sym.								Reserved																							
Symmetric address																															
...																															

Table 1. **Datagramme `HELLO`**

1.1.3. Construction du graphe représentant le réseau

La construction du graphe représentant le réseau se fait à l'aide de la table regroupant les `LSA`. Puisque chaque `LSA` correspond à un nœud du réseau et ses voisins, traduire ceci en une structure de graphe se fait directement en utilisant une table de hachage, on obtient un graphe représenté par listes d'adjacence.

Il est à noter cependant qu'on ne reporte un lien dans le graphe du réseau que s'il est reconnu par les deux extrémités (un décalage lié au temps de propagation de leurs `LSA` respectifs peut être la cause de ceci).

1.1.4. Construction de la table de routage

Pour ce qui est de construire la table de routage, il nous faut d'abord calculer les plus courts chemins pour accéder à chacun des nœuds du réseau.

Ce calcul est relatif à un nœud, c'est pourquoi chacun calcule ses plus courts chemins. La distance considérée est le nombre de sauts.

On se contente de faire un parcours en largeur du graphe à partir du nœud courant en associant les nœuds au nœud par lequel on les a atteints (le *nexthop*) dans une table de hachage. Cette méthode donne directement les plus courts chemins ainsi que la table à utiliser pour effectuer le routage.

1.2. Spécification

La spécification que nous utilisons pour les messages `HELLO` et `LSA` peut être trouvée dans les tables 1 et 2. Le champ `Reserved` est un champ vide, le champ `Internet prov.` vaut 1 si le nœud émetteur du `LSA` possède un accès à Internet. Les autres champs se passent d'explication.

1.3. Implémentation

L'envoi de paquets se fait par UDP[4] car nous avons besoin d'effectuer du *broadcasting* pour inonder le réseau.

[4] *UDP : User Datagram Protocol*

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Message type										Internet prov.										Message size											
Source address																															
Sequence number																				Number of neighbors										
Neighbor address																															
...																															

Table 2. **Datagramme `LSA`**

Nous avons commencé par utiliser la structure Java `DatagramChannel`, mais nous nous sommes rendus compte qu'elle ne permettait pas une fonction pourtant essentielle au fonctionnement du programme, l'utilisation d'un *timeout*. En effet, nous avons besoin de pouvoir écouter pendant une période limitée, ce qui n'est pas possible avec les canaux.

Nous nous sommes donc tournés vers la structure `DatagramSocket` qui remplit cette fonction, nous l'avons implémentée en s'aidant de [2].

1.3.1. Architecture de threads

Chaque nœud du réseau jouant le rôle de *server*, nous avons dû utiliser une architecture à deux threads.

Le premier thread, nommé `PacketManager`, est en charge de

- recevoir les paquets,
- créer et envoyer les `HELLO` et `LSA`,
- relayer les `LSA`.

Le second thread, nommé `MessageThread`, est en charge de

- reconnaitre le type de paquet,
- traduire le datagramme en objet,
- mettre à jour les tables de `HELLO` et `LSA`,
- construire le graphe du réseau,
- mettre à jour la table de routage.

Plusieurs objets sont partagés entre les deux threads. Tout d'abord, on partage les tables contenant les messages `HELLO` et `LSA`. Il existe également un verrou réseau partagé permettant d'éviter que des messages transitent pendant que la table de routage est modifiée. Enfin, une file bloquante est utilisée pour faire passer les paquets du `PacketManager` vers le `MessageThread`.

Un design pattern *Singleton* est mis en place pour s'assurer que le `PacketManager` (et donc le `MessageThread`) n'est lancé qu'une seule fois, le cas contraire produirait une race condition.

Il est à noter également la présence d'un thread graphique dont nous ne détaillerons pas le fonctionnement.

1.3.2. Architecture de classes

On se base sur le diagramme de classes 2 pour décrire l'architecture de classes. Les losanges représentent des attributs multiples de l'objet (une table de l'objet par exemple), la flèche pleine représente un attribut simple et les flèches pointillées représentent des appels de fonction.

Dans tout le programme, les nœuds et les adresses IP sont assimilées, on les représente avec la classe `IP`.

Les `HelloMessage` et `LSAMessage` stockent leurs voisins dans des listes. On dispose pour chacun d'entre eux de fonctions et constructeurs servant à passer de l'objet au datagramme et au paquet, et *vice-versa*.

La `LSATable` sert à stocker les `LSAMessage` de numéro de séquence le plus élévé reçus, elle met à jour la table de routage quand un `LSA` avec des informations différentes lui parvient. Elle dispose également d'une fonction `checkDeadNodes` pour retirer les `LSA` des nœuds qui n'ont pas envoyé de mise-à-jour depuis un certain temps et qui sont donc considérés hors-ligne.

La `HelloTable` stocke les `HelloMessage` des voisins à un saut, c'est donc depuis elle que les `LSA` et `HELLO` du nœud courant sont créés. C'est également là que le numéro de séquence des `LSA` est conservé. Elle dispose comme `LSATable` d'une fonction `checkDeadNodes`.

Le `MessageThread` reçoit les paquets, identifie le type de message puis l'ajoute à la table appropriée.

`RoutingTable` est l'objet représentant la table de routage du système, il est mis à jour dès qu'un `LSA` contenant de nouvelles informations arrive. Pour ce faire, il commence par reconstruire le graphe du réseau `NetworkGraph` puis recalcule les plus courts chemins. Il dispose d'une fonction `writeTable` qui utilise `SystemCommand` pour écrire la véritable table de routage système.

Enfin, le `PacketManager` est le thread principal. Il commence par récupérer les informations d'un fichier de configuration pour connaître ses propres informations réseau. Il ouvre un `DatagramSocket` et initialise la table de routage système via `SystemCommand`. Il initialise les *HelloTable* et `LSATable`, la file bloquante, puis lance `MessageThread`.

Après cela, le `PacketManager` effectue une boucle lors de laquelle il alterne les phases d'écoute et d'envoi des `LSA` et `HELLO` tout en vérifiant les possibles déconnexions via les fonctions `checkDeadNodes`. La fonction d'écoute prend en argument un *timeout*, c'est le temps pendant lequel elle doit écouter, peu importe le nombre de messages reçus. Elle dispose d'un *buffer* de grande taille dans lequel elle reçoit les paquets qui arrivent, puis elle les met dans un nouveau *buffer* faisant la taille du paquet. Quand elle reçoit un `LSA`, elle appelle la fonction `isLatest` de la *LSATable* pour savoir si elle doit le relayer. Aussi, elle ignore les paquets venant de l'adresse du nœud courant.

Enfin, on dispose d'un fichier de configuration qui contient des informations sur la machine du noeud courant.

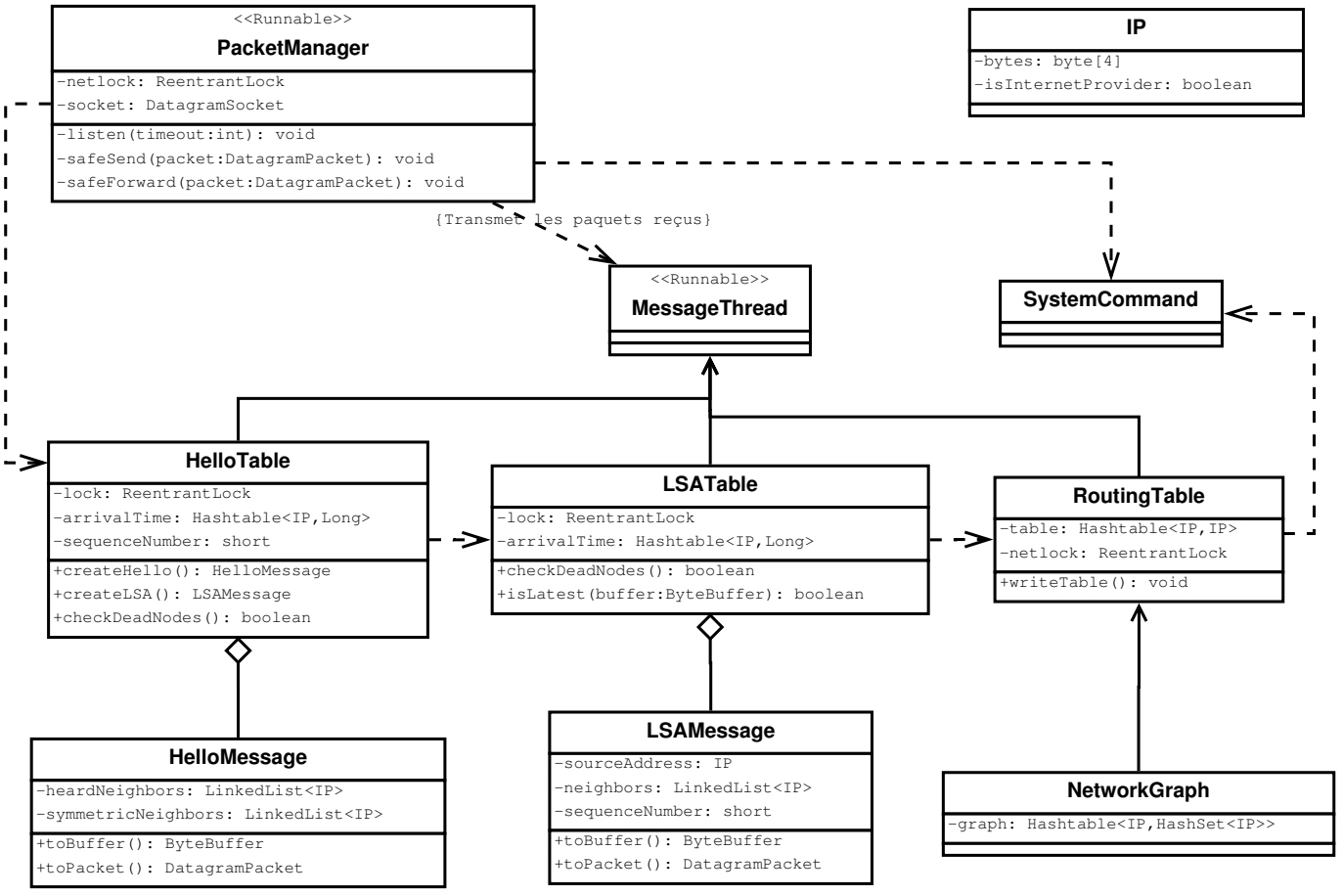

Figure 2. **Diagramme de classes de notre implémentation.**

2. Résultats

2.1. Mise en place d'un réseau de machines virtuelles

Tester un protocole de routage requiert de pouvoir utiliser plusieurs machines, or nous ne sommes dotés que de deux ordinateurs personnels. C'est pourquoi nous avons dû utiliser des machines virtuelles, ce qui nous a permis de lier jusqu'à cinq machines. Il est à noter également qu'utiliser des machines virtuelles permet de maîtriser la configuration du réseau, ce qui est appréciable en comparaison des problèmes de pare-feu parfois posés par Windows.

Contrairement à la machine virtuelle Java qui s'intègre à un système d'exploitation, nous avons utilisé ici des machines virtuelles qui sont complètement opaques pour celui-ci à l'aide de *VMWare*. VMware est un software qui permet de créer des ordinateurs virtuels dont la configuration est modulable. Nous l'avons utilisé pour installer le système d'exploitation Ubuntu ainsi qu'Eclipse et Git afin de pouvoir télécharger et exécuter le code de notre projet pour le tester.

Compte tenu de l'impossibilité de simuler des connections WiFi avec VMware, nous avons cherché à simuler une connexion WiFi pour des machines virtuelles qui sont sur le même lien. Pour cela nous avons intégré dans le code le concept de*Blacklist* afin de couper des connections entre certaines machines, ceci simulant deux machines qui seraient hors de portée l'une de l'autre. En pratique le programme identifie l'adresse depuis laquelle a été relayé le message et le jette si celui-ci est présent dans la Blacklist.
Le nombre de machines virtuelles que nous avons pu faire fonctionner simultanément a été de cinq car celles-ci consomment une grande quantité de mémoire (1,5 Go) et de temps processeur. Ainsi, malgré la puissance de notre ordinateur de test (8 Go de RAM, Core i7) nous n'avons pas pu tester notre algorithme à grande échelle.
Les commandes système de notre implémentation sont donc prévues pour une distribution Linux.

2.2. Mise à l'œuvre du protocole

Les propriétés devant être vérifiées par le protocole sont les suivantes :

- Les paquets sont correctement routés.
- Un nœud peut s'ajouter au réseau et être pris en compte à tout instant.
- Une déconnexion d'un nœud est prise en compte.
- Les chemins utilisés minimisent le nombre de nœuds.
- La table de routage correcte est trouvée rapidement.

Le test que nous reportons en figure 3 est fait avec cinq machines, les arêtes voulues sont exactement celles représentées. On observe une bonne réactivité du protocole au connexions et déconnexions et les chemins utilisés pour le routage sont bien les plus courts.
On vérifie également que les envois de paquets se font en suivant les chemins représentés sur la sortie graphique, et surtout le noeud courant obtient un accès Internet par le noeud 1.1.1.1 à trois noeuds de distance.
À notre échelle de test, notre implémentation du protocole est fonctionnelle.

2.3. Un point dont nous sommes particulièrement fiers

Le système tel que nous l'avions conçu dans un premier temps permettait aux différents périphériques de communiquer entre eux mais il n'existait pas de moyen de partager les connexions à Internet. Afin d'obtenir cette fonctionnalité nous avons eu l'idée d'installer un *NAT*[5] sur les noeud du réseau possédant une connexion et d'ajouter dans les LSA un champs permettant de savoir si ceux-ci pouvaient fournir internet aux autres noeuds

[5] *NAT : Network Address Translation*

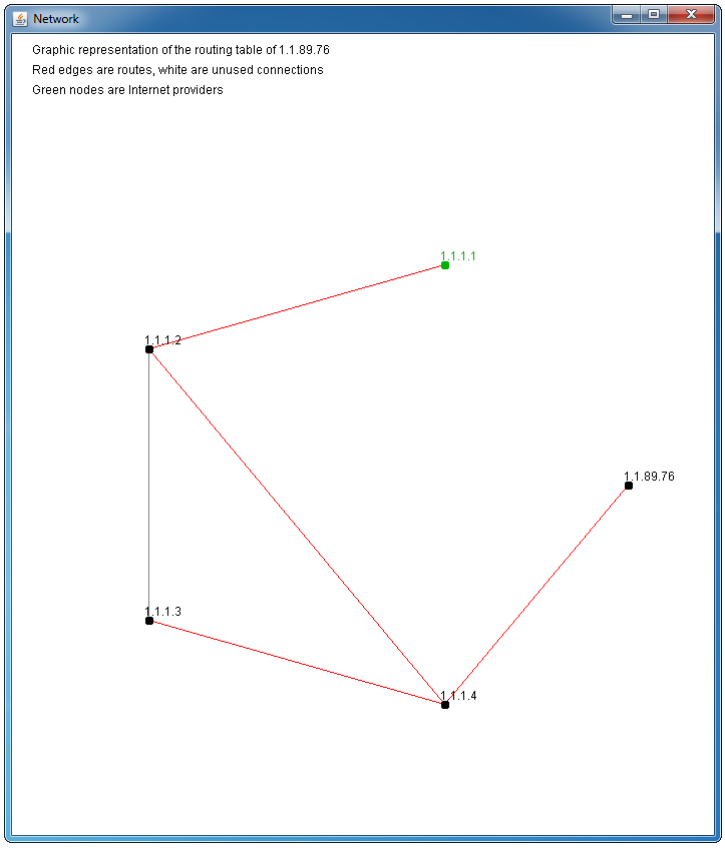

Figure 3. Capture d'écran de la sortie graphique du programme. Le noeud de droite est le noeud courant, les arêtes rouges sont celles utilisées par sa table de routage, les arêtes grises sont non-utilisées, le noeud vert est fournisseur d'accès internet.

du réseau. Lors du calcul de la table de routage, les noeuds du réseau sélectionnent le noeud fournisseur d'accès internet le plus proche s'il en existe au moins un.

Intégrer cette fonctionnalité à été l'occasion d'approfondir notre connaissances des interactions entre la machine virtuelle java et le système d'exploitation, en particulier la gestion de plusieurs interfaces réseaux et des interactions entre celles-ci par le biais de la table de routage et du NAT.

3. Développements possibles

3.1. OLSR

Le protocole de routage sur lequel nous nous sommes basés n'est pas la version retenue par l'IETF. En effet, il existe une version optimisée nommée *OLSR*[6].

Cette amélioration du protocole sert à limiter le nombre de retransmissions dans le cadre d'une inondation du

[6] *OLSR : Optimized Link State Routing*

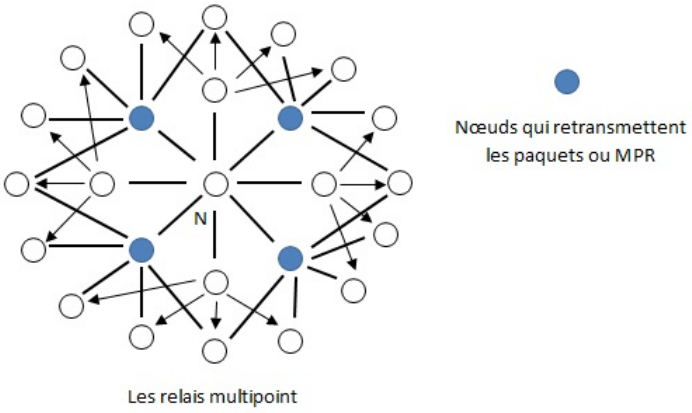

Figure 4. Illustration d'un choix de MPR. Les points bleus suffisent à être lié à tous les voisins à deux sauts, ce qui limite l'inondation du réseau. Schéma tiré de [1].

réseau.

Cette amélioration est basée sur la définition d'une troisième catégorie de voisins en plus des *heard* et des *symmetric*, les *MPR*[7]. Dans la première version d'OLSR, les MPR sont définis comme le sous-ensemble des des voisins à un saut permettant d'accéder à tous les voisins à deux sauts de cardinal minimal (voir figure 4).

On utilise exclusivement lesdits MPR pour faire transiter les `LSA` lors de l'inondation, ce qui limite grandement le nombre de retransmissions.

Pour ce faire, il faut que chaque nœud garde en mémoire de qui il est le MPR. Ainsi, à la réception d'un `LSA`, il saura si il doit le relayer ou non.

Une version modifiée du protocole OLSR consiste non pas à choisir les MPR de cardinalité minimale mais de bande passante maximale. L'intérêt de ceci est que l'on peut réduire les `LSA` à l'ensemble des MPR et des voisins nous ayant choisi comme MPR pour forcer le routage à passer par les MPR, et donc par le chemin à plus grande bande passante.

Dans notre cadre de tests basé sur des machines virtuelles, la notion de bande-passante n'est pas pertinente, c'est pourquoi nous avons voulu implémenter la version originale d'OLSR. Il suffit d'ajouter une classe d'objet contenant l'ensemble de nos MPR et des nœuds nous ayant choisi comme MPR, d'adapter les messages `HELLO` (voir table 3), et de tester le caractère MPR avant de relayer les `LSA`.

Cependant, il faut un nombre trop conséquent de machines pour qu'OLSR soit véritablement testable, nous n'avons donc pas pu aller au bout de l'implémentation.

[7] *MPR : Multi Point Relay*

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Message type								Reserved								Message size															
Source address																															
Number of heard								Reserved																							
Heard address																															
...																															
Number of sym.								Reserved																							
Symmetric address																															
...																															
Number of MPR								Reserved																							
MPR address																															
...																															

Table 3. **Datagramme `HELLO` OLSR**

3.2. Sécurité et attaques

La raison probable pour laquelle les réseaux WiFi prétendument *ad-hoc* de Windows ne le sont pas est le manque de sécurisation des protocoles MANET.

En effet, l'absence d'un point d'accès dans le réseau le rend vulnérable à diverses attaques telles que le brouillage ou simplement l'envoi de messages falsifiés afin d'usurper l'identité d'autre nœuds.

Nous aurions aimé tester ces différents types d'attaques et mettre en place des protections, mais le nombre de machines comme le temps nous a manqué.

Conclusion

Nous avons écrit un protocole de routage pour réseaux *ad-hoc* sans fils, ce qui a nécessité l'écriture d'une spécification et l'appropriation de fonctionnalités réseau de Java ainsi que du système d'exploitation Ubuntu.

Les tests effectués sur machine virtuelle montrent le bon fonctionnement du protocole, il serait maintenant intéressant de pouvoir le tester à plus grande échelle.

References

[1] Optimized Link State Routing Protocol, Wikipédia.

[2] Michiel De Mey, Network discovery using UDP Broadcast (Java), http://michieldemey.be/.