État de l'Art

LAMONT Jonathan - ELOUKLI Hossam

Environnement P2P et Routage Ad-Hoc

Table des matières

1. Introduction

1.1. Présentation du sujet & Motivations

Nous nous sommes imposés d'explorer les usages et coutumes dans les milieux du routage ad-hoc et des réseaux pair-à-pair. Nous savions déjà que les systèmes P2P peuvent être plus résilients que les réseaux hiérarchisés classiques. Et quoi de mieux pour s'intéresser à ça que de commencer par la première brique fondamentale qu'est le routage !? Et c'est ainsi que nous avons choisi de faire un état de l'art de ces deux domaines. Remarquons que par la suite, par extension au sujet, nous avons voulu tester et simuler certains protocoles de routage ad-hoc existants grâce au simulateur réseau : NS2.

1.2. Objectifs

Premier objectif, un état de l'art sous deux aspects : les environnements P2P et les protocoles de routages ad-hoc.

Deuxième objectif, tester certains protocoles de routage ad-hoc existant par simulation.

2. État de l'art

Cette section vise à faire un état de l'art non-exhaustif des pratiques contemporaines en la matière du routage au sein de réseau P2P pur. Nous allons en premier lieu faire quelques rappels historiques sur les réseaux P2P. Puis nous poserons, quelques généralités sur le contexte de ce sujet, ainsi que les définitions dont nous aurons besoin par la suite. Et enfin, nous nous intéresserons à quelques protocoles de routages existants pour les milieux ad-hoc.

2.1. Histoire des réseaux P2P

Depuis la naissance d'internet, le modèle **client - serveur** s'est imposé. Ce modèle place sur le réseau un serveur détenteur d'un service (ou d'une ressource de manière générale), et les clients souhaitant disposer de ce service ou de cette ressource, se connectent au serveur en question. Aux prémices d'internet ce modèle s'est imposé naturellement car il y avait peu d'utilisateurs sur le réseau internet, et ces quelques utilisateurs présents ne disposaient que de peu de ressources informatiques (les constructeurs étant à leur balbutiement, les ressources matérielles étaient onéreuses). L'avantage de ce modèle est que seul le serveur est maître. Et par conséquent, seul lui nécessite une infrastructure coûteuse et robuste.

Aujourd'hui c'est l'effet inverse qui se profile à l'horizon. Le nombre d'utilisateur faramineux occupant internet pousse les entrepreneurs à construire des infrastructures serveurs toujours plus complexes et onéreuses.

De là émerge l'idée que nous pourrions distribuer les services sur les clients pour diminuer les coûts. Et également supprimer la contrainte de disponibilité directement imputée au maillon faible qu'est le serveur (i.e. *single point of failure*). En effet, dans le modèle client - serveur classique, le serveur est le seul détenteur du service. Il est donc sensible aux pannes et aux attaques. Le fait de distribuer ce service sur une communauté d'utilisateurs répartis au sein d'un réseau, permettrait d'augmenter la tolérance aux pannes et de manière générale augmenter la disponibilité du service. Et c'est cette problématique qui, un peu avant les années 2000, fera émerger le concept de **réseau pair-à-pair** (Peer-to-Peer en anglais, abrégé P2P).

Les **réseaux P2P** sont des systèmes dont les pairs (nœuds) sont à la fois client et serveur. Les nœuds sont libres et autonomes. Deux caractéristiques essentielles découlent de cela. Premièrement, aucune entité ne contrôle le réseau. Ils interagissent entre eux directement pour s'échanger des informations. Et deuxièmement, ces réseaux sont dynamiques. Un pair peut se déconnecter à tout moment, et ré-accéder au réseau à tout moment.

Plusieurs modèles de réseau P2P sont apparus au fil du temps depuis le milieu des années 90. Citons les plus connus : *KaZaA*, *Gnutella*, *BitTorrent*... Ces modèles de réseau P2P peuvent être classés en **3 types** : les modèles P2P **centralisés**, les modèles P2P **acentralisés**, et les modèles P2P **hybrides** s'inspirant des deux modèles précédents.

2.1.1. Le modèle P2P centralisé

Ces systèmes reposent sur l'utilisation d'un serveur central qui recense les ressources partagées. C'est une sorte de gros index, auquel tous les clients soumettront une requête pour faire une recherche de ressource. Là où cela diffère avec le modèle client-serveur classique, c'est qu'une fois la ressource localisée suite à la requête au serveur central, la communication permettant l'échange de la ressource est quant à elle, directement effectuée auprès des pairs du réseau entre eux même (permettant ainsi de désengorger le réseau vers le serveur central).

On voit dans ce système, toujours la même contrainte de centralisation : si le serveur tombe en panne, les pairs du réseau se retrouvent aveugles et ne peuvent plus rien rechercher au sein du réseau.

2.1.2. Le modèle P2P acentralisé

Ces systèmes P2P décentralisés peuvent être de **2 types** : **structurés** ou **non structurés**.

Un modèle P2P **décentralisé** est dit **non structuré** si les connexions entre les pairs du réseau se font de manière totalement aléatoire. Le mécanisme de recherche est souvent mis en œuvre par des techniques de diffusions par inondation. Ce qui consiste à inonder ses voisins par la même requête pour que cela touche un maximum de pairs dans l'espoir de recevoir une réponse indiquant la localisation de la ressource. Ces techniques d'inondations sont connues pour être coûteuses en bande passante et assez lentes.

Ces système acentralisés (ou décentralisés) sont dit «**Pair-à-Pair pur**», car ils sont entièrement décentralisés. Aucun nœud central n'est requis. Les pairs sont tous égaux au sein du réseau. Ils disposent tous des mêmes fonctionnalités de base.

Remarque de vocabulaire personnelle : "décentralisé" suppose l'existence d'un centre ("éloigné du centre"). Alors que "acentralisé" nie l'existence dudit centre ("sans centre"). Je comprends ces mots ainsi. J'ai une préférence pour «acentralisé». Seulement dans la littérature contemporaine, j'ai l'impression que ces deux mots sont majoritairement synonymes. Considérons les comme tel dans ce rapport.

Un modèle P2P **décentralisé** est dit **structuré** s'il existe une organisation des pairs au sein du réseau. Typiquement, ils sont organisés par une table de hachage distribuée (DHT*, i.e. Distributed Hash Table*). Les pairs possèdent un identifiant qui est haché et rangé dans la DHT. Cette DHT permet des recherches peut coûteuses en O(log N) (si N est le nombre de nœuds/pairs du réseau). Cela permet une localisation relativement performante des pairs et des données, et de manière distribuée. Remarque, pour garder ces performances, il est recommandé d'avoir une capacité totale dans la DHT d'environ deux fois le nombre de nœuds du réseau. Car l'absence de «blancs» aléatoirement répartis dans la DHT ont tendance à rendre l'algo linaire en O(N).

2.1.3. Le modèle P2P hybride

Ce modèle hybride s'inspire des qualités des deux modèles précédents. Souvent nommé «**modèle super-peer**». Il repose sur l'utilisation de super nœuds qui forment une maille grossière du réseau. Les nœuds simples se connectent à des super nœuds qui gèrent un ensemble de nœud simple. Nous avons donc une hiérarchisation du réseau sur deux niveaux. Cela a pour but d'accélérer les recherches, car un nœud simple demandera à son super nœud qui recherchera dans

l'ensemble des nœuds qu'il gère ; et éventuellement s'il ne trouve rien, il demandera à ses super nœuds voisins.

Le choix des super nœuds par le protocole devient un problème critique. Sa disponibilité, ses capacités de traitement, de bande passante, etc... sont tous des facteurs qui peuvent directement impacter les performances du réseau. C'est pourquoi le protocole doit prendre ces aspects en compte lors du choix d'un super nœud parmi des nœuds candidats.

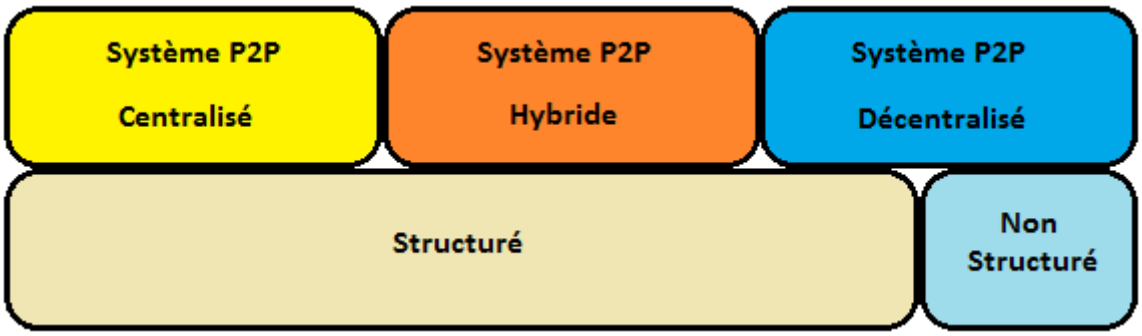

Fig 1 - Différentes architectures des systèmes P2P

2.2. Généralités

2.2.1. Les réseaux P2P

2.2.1.1. Définitions

Nous allons à présent définir quelques éléments de vocabulaire en lien avec les réseaux pair à pair, ce qui nous aidera par la suite.

2.2.1.1.1. Système pair à pair pur

«Les systèmes P2P purs sont des systèmes P2P non structurés et acentralisé.»

En effet, le fait qu'aucune structure réseau ne soit requise permet d'avoir une liberté locale au niveau des nœuds (comportement stochastique : cf. milieu ad-hoc) ; et le fait que le réseau soit entièrement décentralisé a pour effet d'avoir une homogénéité des nœuds (tous les nœuds exécutent le même protocole). *Remarque, les nœuds ne sont pas forcément égaux, cela dépendra des protocoles partagés entre eux. Les protocoles peuvent être uniformes et respecter l'égalité, ou ne pas l'être et auquel cas, les nœuds pourront très bien avoir des comportements positif ou néfaste au réseau, ou bien même exécuter d'autre traitement vu qu'ils sont libres et différents.*

2.2.1.1.2. Réseau ad-hoc

«Les réseaux *ad-hoc* sont des réseaux sans fil capables de s'organiser sans infrastructure définie préalablement.» **[AdHoc]**

Aussi connu sous le nom de réseau mobile, ou MANET dans les pays anglophones (en référence au nom du groupe de travail créé par l'IETF en 1999). Ces réseaux ont la particularité de

communiquer entre eux en se servant d'intermédiaires relais pour atteindre leur destination. Et ce, sans infrastructure préexistante (e.g. comme les antennes relais pour les réseaux GSM).

Cette catégorie de réseau s'adapte très bien avec un contexte pair à pair non structuré, vu qu'aucune topologie réseau précise n'est requise.

2.2.1.1.3. Overlay

«Un réseau "overlay" est un réseau virtuel superposé à un réseau dans le but de masquer la complexité du réseau sous-jacent.»

"Overlay" ("sur couche" de l'anglais) désigne donc le réseau virtuel superposé à un réseau sous-jacent. Les nœuds y sont interconnectés par des liaisons logiques. Et la plus part du temps les nœuds sont affectés d'une identité dans l'overlay permettant leurs localisations (identité correspondant le plus souvent à un hash). Typiquement, l'ensemble des nœuds d'un réseau P2P définissent un réseau virtuel. **[Overlay]**

2.2.1.2. Propriétés

Voici les principales propriétés que nous pouvons dégager des réseaux P2P. Bien entendu, pour les systèmes P2P acentralisés et non structurés, ces propriétés sont conservées.

2.2.1.2.1. Tolérance aux pannes

Généralement la disponibilité d'un service dépend du serveur le fournissant. Dans le cas d'un réseau P2P, la disponibilité dépend de l'ensemble des nœuds du réseau capable de fournir ce service (i.e. la communauté) plutôt qu'un unique nœud.

2.2.1.2.2. Vivacité

Caractère dynamique des nœuds qui peuvent quitter ou se connecter au réseau à tout moment de manière libre et aléatoire. La topologie du réseau se doit donc d'être nécessairement dynamique.

2.2.1.2.3. Auto-Organisation

Du fait du contexte ad-hoc du réseau et l'organisation entièrement libre des systèmes P2P purs, il n'y a donc aucune hypothèse de structure / d'organisation des nœuds du réseau au préalable. C'est pourquoi, afin de s'organiser, les nœuds du réseau ne peuvent s'appuyer que sur leurs coopérations. Le protocole en question se doit d'avoir cette capacité pour qu'une auto-organisation **globale** émerge au sein du réseau, de part des interactions **locales** des nœuds entre eux.

2.2.2. Le routage en réseau ad-hoc

2.2.2.1. Définitions

Nous allons maintenant nous attacher à définir les notions de routage ad-hoc, notamment les différents types et principes d'algorithmes de routage.

2.2.3.1.1. Routage Proactif

«Le routage proactif consiste à établir les routes avant même d'en avoir besoin.»

Un pair souhaitant communiquer aura donc déjà dans sa table de routage les routes pour établir la communication. Sinon, c'est que la destination est inaccessible. Ce protocole agit donc en amont, et souvent de manière continue et périodique.

2.2.3.1.2. Routage Réactif

«Le routage réactif consiste à établir les routes à la demande.»

Un pair souhaitant communiquer avec un autre pair non disponible dans sa table de routage, sollicitera ce protocole pour effectuer une recherche de route immédiate. Suite à la réponse, il pourra ou non établir la communication avec le pair destinataire.

2.2.3.1.3. Routage Hybride

«Le routage hybride consiste à conjuguer un routage proactif et un routage réactif.»

Cette combinaison fourni donc une actualisation des tables de routage en continue grâce à l'aspect proactif du protocole. Et si jamais une destination n'existe pas dans la table de routage locale lors de l'ouverture d'une communication, alors dans ce cas le protocole déclenche une procédure de recherche de route. C'est l'aspect réactif du protocole. L'utilité de ce type hybride de routage réside dans l'amélioration des performances. En effet, là où les protocoles réactifs purs sont obligés d'entretenir des tables de routages complètes du réseau au sein même de chacun des nœuds du réseau ; les protocoles réactifs hybrides peuvent se contenter d'une vue partielle du réseau sachant que l'aspect réactif pourra combler le manque d'exhaustivité de l'aspect proactif.

2.2.3.1.5. Routage par Vecteur Distance

«Un routage par vecteur distance utilise typiquement une métrique de distance au sein de table de routage recensant les routes vers toutes les destinations possibles dans le réseau.»

Le vecteur distance est le plus souvent représenté par un nombre de saut (hop) vers la destination. Les algorithmes mis en œuvre sont souvent simples, mais posent des problèmes d'utilisation importante de la bande passante afin d'entretenir des tables de routage complètes et à jour entre tous les nœuds du réseau.

2.2.3.1.6. Routage par État des Liens

«Un routage par état de lien est assuré par une vision locale au nœud et partielle du réseau, où chaque nœud s'attache à maintenir une liaison avec son voisinage.»

Les liens sont découverts et maintenus par un envoi périodique de message "bonjour". Ces messages permettent de déduire si le lien en question est nul, unidirectionnel, ou bidirectionnel. Le protocole pourra ainsi reconstruire la topologie du réseau de proche en proche en utilisant son voisinage. Ce type de protocole à l'avantage de s'adapter facilement à un changement de topologie du réseau, mais nécessite des ressources et un peu de bande passante au niveau de chaque nœud pour fonctionner. Le principal inconvénient est donc un ralentissement des performances pour de

gros réseau.

2.2.3.1.7. Routage par Sélection de voisin

«Un routage par sélection de voisin est un routage non uniforme où chaque nœud décharge les fonctions de routage à un sous ensemble de voisin direct.»

2.2.3.1.8. Routage par Partitionnement

«Un routage par partitionnement est un routage non uniforme où le réseau est divisé en différentes zones dans lesquelles un unique nœud maître assure le routage au sein de sa zone.»

2.2.3.2. Les différents principes de routage ad-hoc

Comme défini précédemment, les principaux principes de routage en milieu ad-hoc sont classables en deux catégories : les protocoles non-uniformes (typiquement avec hiérarchisation des rôles des nœuds au sein du réseau virtuel), ou les protocoles uniformes (typiquement par distribution à l'identique du même protocole sur chaque nœud).

Protocole non uniforme :

- Routage par choix de voisinage
- Routage par partitionnement

Protocole uniforme :

- Routage par état des liens
- Routage par vecteur distance

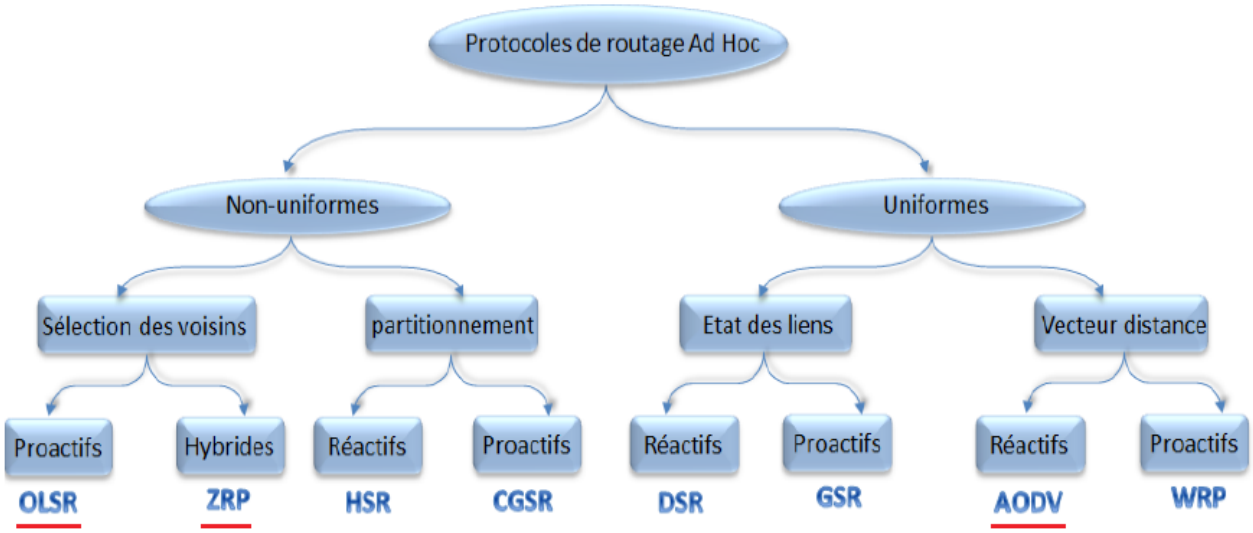

Fig 2 - Les familles de protocoles de routage ad-hoc

2.3. Protocoles de routage ad-hoc existants

Dans cette section nous étudierons succinctement différents protocoles de routages utilisant divers modes de fonctionnements. Cette étude est limitée, et n'a pas vocation d'être exhaustive.

Nous verrons en premier le protocole **proactif non-uniforme OSLR**, puis le protocole **réactif uniforme AODV**, et enfin le protocole **hybride non-uniforme ZRP**.

2.3.1. OLSR – Protocole Proactif

[OLSR] (*Optimized Link State Routing Protocol*) est une amélioration des protocoles par états des liens classique (notamment LSR). Ce protocole utilise des échanges périodiques afin de maintenir les informations sur la topologie du réseau à jour (il est proactif). L'amélioration provient du fait qu'il utilise des nœuds particuliers nommés MPR (*Relais Multi Point*). Ces relais font l'intermédiaire entre chaque nœud situé à 2 sauts les uns des autres. Ils forment donc un ensemble de voisinage à 1 saut pour chaque nœud du réseau. Et seul les relais MPR peuvent retransmettre les paquets OLSR. Cela a pour but d'améliorer les performances et d'économiser des ressources en bande passante en minimisant l'impact négatif de l'inondation grâce aux MPRs. Ci-dessous un visuel du fonctionnement d'OLSR.

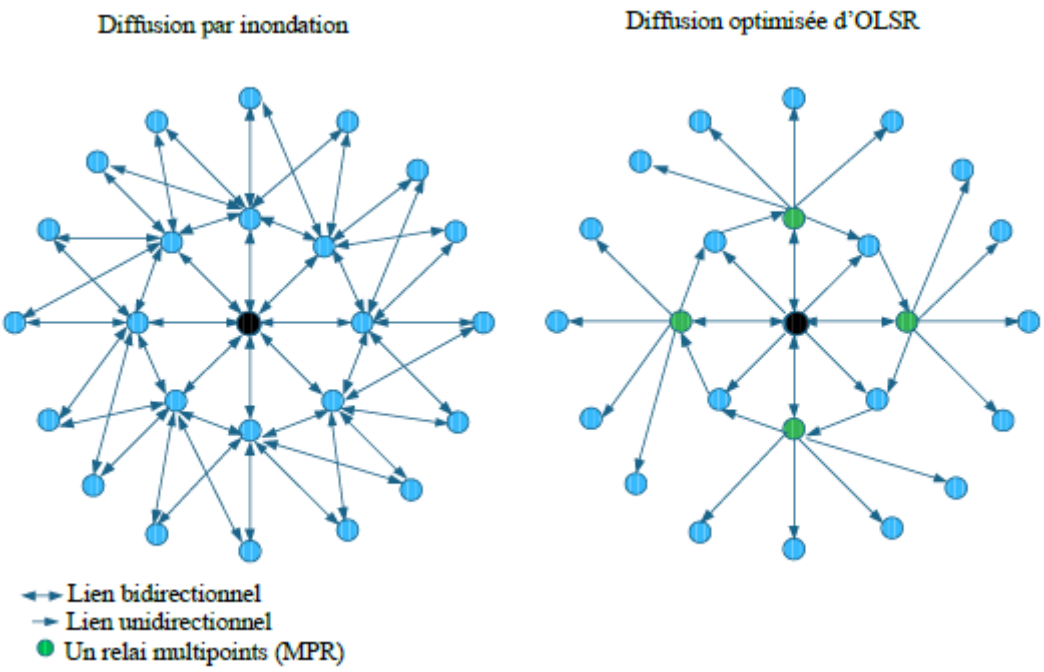

Fig 3 - **[OLSR]** - Diffusion optimisée

2.3.2. AODV – Protocole Réactif

[AODV] (*Ad-hoc On demand Distance Vector*) est un protocole réactif basé sur un routage par vecteur distance. Il est peu gourmand, et sans bouclage. Ce qui le rend intéressant dans le contexte des réseaux ad-hoc, ainsi que pour des petits appareils mobiles.

AODV définit un mécanisme de découverte de route, ainsi qu'un mécanisme de maintenance des routes. La découverte de route est assurée par l'envoi en broadcast d'un message RREQ (*Route Request*). Si le message RREP (*Route Reply*) est reçu, l'opération est un succès. Dans le cas

contraire, on procède à un délai d'attente NET_TRANVERSAL_TIME avant de réessayer d'envoyer un RREQ dans la limite du nombre d'essais défini par RREQ_RETRIES+1 (par défaut à 2+1). La maintenance des routes est effectuée tant que des données transitent entre le nœud et sa destination. Après un délai ACTIVE_ROUTE_TIMEOUT, si aucune donnée n'a transité, alors le lien expire. Et une nouvelle découverte de route devra être effectuée au besoin.

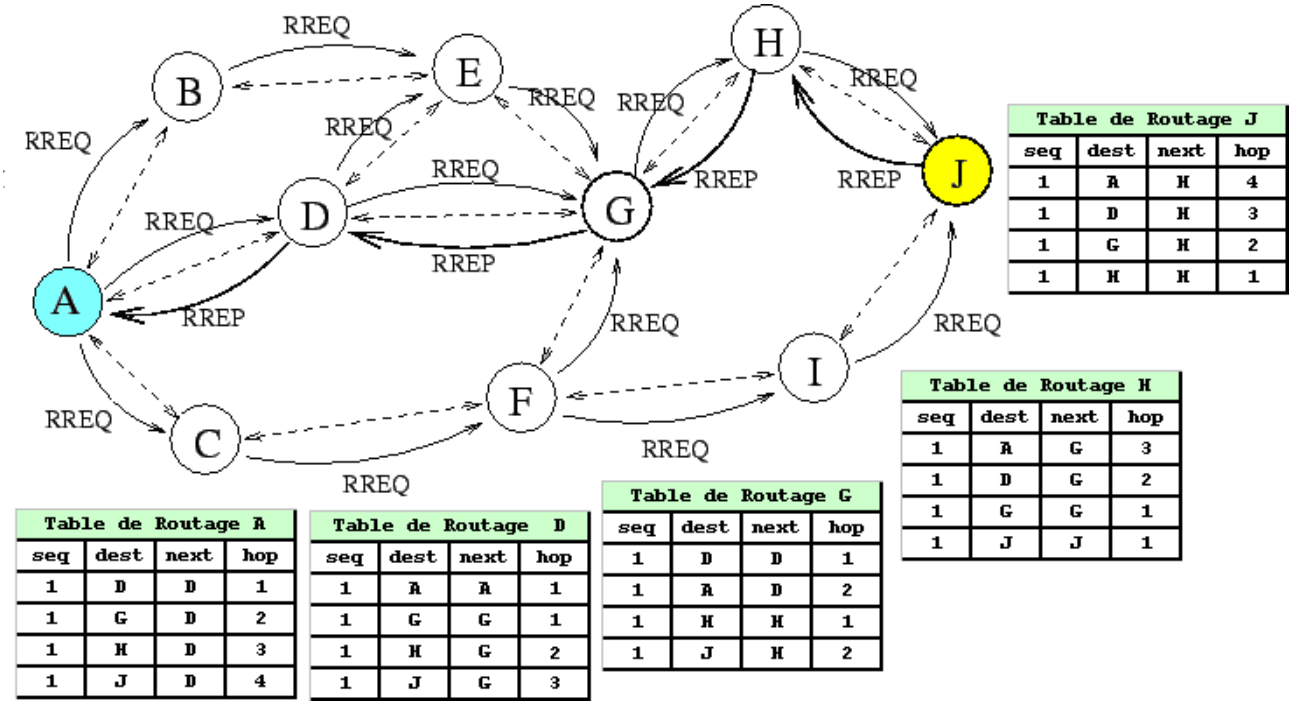

Table de Routage J

seq	dest	next	hop
1	A	H	4
1	D	H	3
1	G	H	2
1	H	H	1

Table de Routage H

seq	dest	next	hop
1	A	G	3
1	D	G	2
1	G	G	1
1	J	J	1

Table de Routage G

seq	dest	next	hop
1	D	D	1
1	A	D	2
1	H	H	1
1	J	H	2

Table de Routage A

seq	dest	next	hop
1	D	D	1
1	G	D	2
1	H	D	3
1	J	D	4

Table de Routage D

seq	dest	next	hop
1	A	A	1
1	G	G	1
1	H	G	2
1	J	G	3

Fig 4 - [AODV] - Découverte de route entre le nœud A et J

2.3.3. ZRP – Protocole Hybride

[ZRP] (*Zone Routing Protocol*) est un protocole de routage hybride utilisant deux protocoles : un protocole proactif nommé IARP (*IntrAzone Routing Protocol*) et un protocole réactif nommé IERP (*IntErzone Routing Protocol*).

Il fonctionne en découpant le réseau en plusieurs zones. Chaque nœud définie une zone limitée en nombre de saut autour de lui. L'aspect proactif de IARP permet d'identifier les routes locales à chaque zone. Et l'aspect réactif de IERP permet de localiser les destinations lointaines (hors zone locale).

Lorsque qu'un nœud souhaite envoyer un paquet, il vérifie si la destination est dans sa zone. Si c'est le cas, la destination se trouve donc dans sa table de routage, et le paquet est envoyé. Sinon le protocole IERP est lancé. Une demande de route est alors envoyée à chaque nœud périphérique de la zone. Si un nœud périphérique connaît la destination, une réponse sera reçue par le nœud demandeur. Et sinon, la recherche se poursuit récursivement sur les nœuds périphériques des nouvelles zones découvertes.

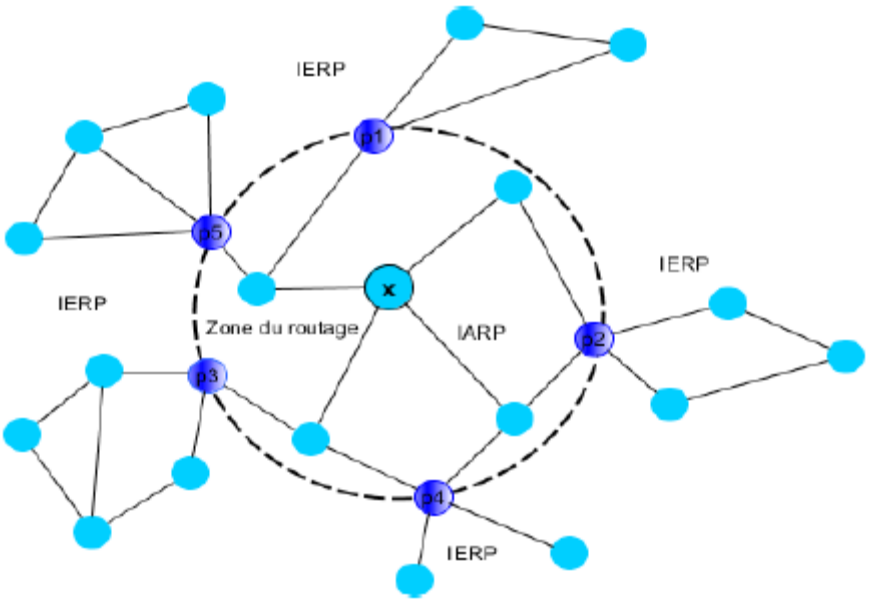

Fig 5 - [ZRP] - Principe de fonctionnement

2.4. La sécurité dans les réseaux ad-hoc

Comme la majorité des réseaux informatiques, les réseaux ad-hoc sont menacés par plusieurs types d'attaquants qui tentent d'effectuer des actions malveillantes, celles-ci pouvant impacter le bon fonctionnement du réseau autant au niveau matérielles que logicielles des entités connectés. Bien que les réseaux hiérarchisés traditionnels bénéficies de diverses couches de protection classique (e.g. firewall, IDS, IPS, proxy, ...), les réseaux ad-hoc sont considérés sensibles face à la liberté de mises en place de très nombreuses attaques différentes.

2.4.1. Types d'attaquant

2.4.1.1. Nœuds malicieux

Dans un réseau ad-hoc autonomes les nœuds peuvent apparaître ou disparaître volontairement. Parmi ces entités mobiles qui constituent le réseau, il existe parfois des nœuds malveillants dont le principal moyen est l'analyse des flux. Ils peuvent par exemple, chercher à écouter secrètement les trafics circulants entre les nœuds, ou parfois modifier les paquets qui transitent par leurs nœuds. C'est ainsi que l'analyse du trafic leurs permet par exemple de déterminer la source et l'identité des nœuds en cours de communication ou la nature des communications échangées.

2.4.1.2. Nœuds égoïstes

Un nœud peut avoir un mauvais comportement qui lui permet de ne pas effectuer correctement sa part de gestion du réseau (tel que le routage par exemple). Ce comportement non autorisé peut dégrader sévèrement les performances d'un réseau ad-hoc ou causer involontairement des dégâts aux autres nœuds.

Ces nœuds qui ont un mauvais comportement sont généralement égoïstes et non collaboratifs, et souhaitent obtenir des avantages par rapport aux autres nœuds. Ils visent généralement à économiser leurs batteries d'énergie, ou bien élargir leur bande passante.

2.4.1.3. Attaquant interne-externe

Un **attaquant interne** est un nœud malveillant qui peut contrôler d'autres nœuds membres d'un réseau. Il peut ainsi disposer des informations relatives à un nœud ciblé pour effectuer une attaque. Par exemple, plusieurs attaques peuvent être lancées en se basant sur la connaissance des informations contenues dans les tables de routage ou bien usurper l'identité d'autres nœuds.

Au contraire, un **attaquant externe** n'est pas en possession de ces informations, et par conséquent les actions possibles pour ce type d'attaquant vont se limiter à détecter la présence ou non de communications, et éventuellement à supprimer des paquets passant par eux en ne les relayant pas. Ce type de contrainte limite fortement l'éventail des attaques possiblement réalisables.

2.4.2. Principales attaques en milieu ad-hoc

2.4.2.1. Déni de service

Dans les réseaux ad-hoc, les attaques de type déni de service peuvent avoir plusieurs formes. Elles peuvent viser la bande passante du réseau, le trafic circulant entre les nœuds mobiles, les ressources énergétiques des nœuds ou la qualité de la transmission.

Les attaquants sont capables par exemple de faire du brouillage sur le canal pour gêner les communications dans le réseau. Ils peuvent aussi altérer les tables de routages en jouant sur les mécanismes de routage. Toujours dans le but d'anéantir la qualité de service du réseau.

2.4.2.2. Privation de sommeil

Ces attaques sont fréquemment rencontrées dans les réseaux ad-hoc où les nœuds disposent de ressource énergétique limitée.

L'attaque consiste à demander de façon permanente des services à un nœud victime, ce qui influence négativement sur sa puissance et ses ressources d'une part, et d'autre part, ça empêche d'autres nœuds du réseau d'échanger avec le nœud trop sollicité.

2.4.2.3. Usurpation d'identité

Vu que les nœuds dans un réseau ad hoc sont identifiés de façon unique, par leur adresse MAC ou IP, l'attaquant peut ainsi usurper facilement l'identité d'un autre nœud du réseau en modifiant sa propre adresse dans les champs des messages qu'il délivre. Il peut ainsi recevoir les paquets destinés aux nœuds légitimes, ou former une boucle de routage entre les nœuds. En somme altérer le routage.

2.4.2.4. Trou de ver

Ce type d'attaque est particulièrement difficile à contrer. Elle peut être lancée par une collaboration d'au moins deux attaquants externes situés sur des zones géographiques distinctes.

L'objectif de cette attaque est de former un tunnel entre deux nœuds corrompus, afin de téléporter des messages de routage. Vu que la plupart des algorithmes de routage prennent les routes les plus courtes, si un message passant par le trou de ver voit son TTL diminuer (ou simplement non incrémenté) par les nœuds corrompus, alors l'algo choisira sûrement la route corrompue. Cette mécanique d'attaque vise à s'approprier la route de communication entre des nœuds cibles. Cela permet de mettre en place diverses choses telles que du MITM ou bien du filtrage de trafic pour du DoS ou de l'espionnage...

3. Simulation sous NS2

Nous avons choisi **[NS2]** comme simulateur car il possède déjà beaucoup d'implémentations de différents protocoles disponibles librement sur le web. Nous avons décidé de choisir les 4 protocoles de routage suivants : **AODV**, **DSDV**, **DSR** et **OLSR**.

La procédure d'installation des outils, NS et ses compagnons (tel que NAM et XGraph), est détaillée en annexe (Annexe 1). Est détaillée y compris, la procédure pour *patcher* NS2 afin d'inclure le support du protocole OLSR. L'initiative à été prise suite aux nombreux problèmes que nous avons rencontré lors de ces manipulations.

3.1. Métriques choisies

Nous avons choisi une seule métrique caractéristique de la robustesse d'un protocole : le **Taux de Perte des Paquets** (TPP) dans le réseau, selon divers paramètres de simulation (expliqués dans le paragraphe qui suit). Cette métrique a été ramené à un pourcentage avec la formule suivante :

$$TPP = \frac{Total\ paquets\ perdus}{(Total\ paquets\ délivrés + Total\ paquets\ perdus)} x\,100$$

3.2. Paramètres de simulations choisis

Les paramètres de simulation sont les variables choisies qui permettent d'effectuer un jeu de simulations. Chaque paramètre est fixé à une valeur constante. De plus, à chaque paramètre est associé un triplet de valeurs correspondant à une valeur min et max ainsi qu'un pas. Ce triplet a le mérite de définir un jeu de simulations.

Les paramètres retenus sont les suivants :

- **Nombre de nœud** dans le réseaux
- **Nombre de paquet** envoyé par canal de communication
- **Vitesse de déplacement** des nœuds

Ces paramètres permettent de mesurer les comportements et l'impact qu'ils ont sur les performances des protocoles observés. Notamment, le nombre de nœud qui est un facteur de densité du réseau, le nombre de paquet envoyé qui représente un facteur de charge sur le réseau, et enfin la vitesse de déplacement des nœuds qui est un facteur de mobilité.

Le tableau ci-dessous présente les valeurs fixés par nos simulations.

Variables de simulation		
CST_NbNode	**50**	**Nombre de nœud par défaut**
NbNodeMin	10	Nombre de nœud minimum

Variables de simulation		
NbNodeMax	100	Nombre de nœud maximum
NbNodeInc	10	Pas d'incrémentation de la variable
CST_NbPacketSent	**1000**	**Nombre de paquet envoyé par défaut**
NbPacketSentMin	200	Nombre de paquet minimum
NbPacketSentMax	3800	Nombre de paquet maximum
NbPacketSentInc	400	Pas d'incrémentation de la variable
CST_NodeSpeed	**25**	**Vitesse des nœuds par défaut (m/s)**
NodeSpeedMin	5	Vitesse de déplacement minimum
NodeSpeedMax	55	Vitesse de déplacement maximum
NodeSpeedInc	5	Pas d'incrémentation de la variable

Tableau 1 - Variables de simulation NS2

3.3. Jeux de simulations effectués

Un jeu de simulation représente une suite consécutive de plusieurs simulations où un unique paramètre varie successivement. Un jeu de simulation va se dérouler de la manière suivante : 1. On choisit quel paramètre va varier. 2. On fixe tous les autres paramètres à leurs valeurs constantes respectives. 3. On exécute toutes les simulations en faisant varier le paramètre voulu de sa borne min à max, et selon son pas d'incrémentation. 4. On récupère et compile les résultats des simulations.

Typiquement les résultats sont traités à la volée entre chaque simulation du jeu, par le script python *"script_extract1.py"* disponible en annexe (Annexe 3). Un résultat de simulation est composé de plusieurs fichiers contenant des graphiques temporaires (au format **[XGraph]**). Il y a 12 graphiques générés temporairement (1 métrique x 4 protocoles x 3 paramètres). Ce script python extrait toutes les valeurs de ces graphes en fin de chaque simulation, afin d'en calculer la moyenne pour chacun. À partir d'un graphique au format métrique x temps, le script python en extrait le point moyen de cette métrique pour la valeur de la variable de simulation. Ce point résultat est écrit en mode ajout dans 12 graphiques de sortie résumant toutes les simulations du jeu lancé. Une fois le jeu de simulation terminé, nous avons donc ces 12 graphiques entièrement complétés. Les courbes suivent le format : métrique moyenne x paramètre du jeu de simulation.

Ces 12 graphes de sortie sont réduits à 3 (1 métrique x 3 paramètres) en fusionnant les courbes des différents protocoles en un. C'est ce que fait le script python *"script_merge2.py"* disponible en annexe (Annexe 3). Les résultats de chaque protocole sont ainsi visualisables et comparables en un coup d'œil sur un même graphe, selon le paramètre observé.

Donc si vous avez bien suivi, nous avons lancé 3 jeux de simulations différents (où 4 protocoles ont été simulé), un pour chaque paramètre de simulation, qui nous fournit 3 graphes exprimant la métrique selon les 3 paramètres de simulation, et où chacun des 3 graphes contiennent toutes les courbes des 4 protocoles (cf. le script tcl en Annexe 2).

3.4. Scénario de simulation choisi

Le scénario de simulation arrêté correspond à l'établissement de **N communications** entre des paires de nœud du réseau choisies aléatoirement. Une paire de nœud est composée d'un nœud source et d'un nœud puits. Typiquement la source émet des données vers le puits qui les réceptionne. Chaque communication consiste donc à envoyer un nombre de paquet prédéfini du nœud source vers le nœud puits. Les paquets de données envoyés respectent tous le même format qui est l'envoi en **UDP** de paquet de **512Ko** à débit constant. Le débit est calculé en fonction du nombre de paquet total à envoyer au cours de la simulation, ainsi que le temps total de simulation afin d'avoir un taux de transmission constant tout du long.

Remarque : Une contrainte sur N (i.e. nombre de communication) est de devoir être toujours inférieure ou égale au nombre de nœud total dans le réseau. C'est pourquoi, le script de simulation complet "script_ns.tcl", prendra pour nombre de connexion (N) la valeur minimale entre le nombre de connexion et le nombre de nœud. Donc pour un réseau de 10 nœuds, même si l'on souhaite établir 35 communications aléatoirement, seul 10 pourront être établies. Et sinon dans le cas favorable, si le réseau contient 50 nœuds, alors là, les 35 communications seront parfaitement établies.

3.5. Configuration NS2

La configuration complète du contexte et du scénario de simulation est disponible au début du script tcl (cf. Annexe 2). Ci-dessous, deux tableaux présentant les valeurs fixées par nos simulations.

Pour le premier : les paramètres généraux qui peuvent être modifiés aisément pour influer sur la charge ou la densité du réseau, ainsi que la durée totale de simulation en seconde.

Le deuxième tableau présente quant à lui la configuration précise utilisé par NS2 pour simuler toute la pile protocolaire sous-jacente à utiliser, ainsi que d'autres constantes liés à la génération du trafic entre les nœuds du réseau pendant la simulation.

Paramètres généraux de simulation		
SimTimeLapse	100	Durée de la simulation (s)
NetworkWidth	1000	Largeur du terrain simulé (m)
NetworkHeigth	1000	Longueur du terrain simulé (m)
NbConnections	35	Nombre de connexion établies par le trafic dans le réseau entre paires de nœud
RecordLapseTime	1,0	Intervalle de temps auquel les données sont récupérées (s)

Tableau 2 - Paramètres généraux de simulation

Configuration de simulation pour NS2		
AgentType	UDP	Transport UDP
TrafficType	CBR	Trafic à débit constant (ConstantBitRate)
PacketsSize	512	Taille des paquets de donnée (octet)
Debit	2Mb	Débit supporté
MobilityModel	RandomWaypoint	Modèle de mobilité
CommunicationRange	250m	Portée des nœuds (m)
LinkLayer	LL	Couche liaison
MacType	Mac/802_11	Protocole d'accès au support
QueueType	Queue/DropTailQueue	File d'attente
MaxSizeQueue	50	Taille de la file d'attente (paquet)
AntModel	Entenna/OmniAntenna	Type d'antenne (ici multi- directionnelle)
ReflectionModel	Propagation/TwoRayGround	Modèle de réflexion
NetworkInterfaceType	Phy/WirelessPhy	Interface WiFi
ChannelType	Channel/WirelessChannel	Canal de communication WiFi

Tableau 3 - Configuration de simulation pour NS2

3.6. Les scripts (TCL & Python)

Les scripts qui ont été rédigés sont au nombre de 4. Un script Tcl et 3 scripts python. Le script Tcl s'occupe de **lancer tous les jeux de simulations**. Tandis que les scripts pythons s'occupent respectivement, d'**extraire**, **fusionner** les données de simulation, et **supprimer** les fichiers de travail temporaire.

Ils sont tous disponible en annexe (Annexe 2 & 3), et ont en partie déjà étés discutés précédemment. Ils sont commentés et leur lecture devrait suffire pour leur compréhension.

3.7. Résultats

Pour information, la configuration machine utilisée pour nos simulations est la suivante :

- *Processeur* : Intel Core i5 [i.e. 2x 2,5GHz physique x 2 threads]
- *RAM* : 6 Go
- *Architecture* : 64 bits
- *VM :* Debian Stretch 9.4 Cinnamon i686 2Go RAM

Remarque : pour information, les graphiques XGraph présentés dans les six figures de ce paragraphe ont été lu et généré sous la version Windows.

Pour obtenir les résultats présentés plus loin, si suffit de lancer la simulation sous linux avec NS d'installé (ce référer à l'Annexe 1 pour plus de détail) par la commande suivante. Au passage,

python3 doit également être présent sur la machine. Prévoyez du temps pour la simulation.

```
cd simulation/
ns script_ns.tcl
```

3.7.1. Cas moyen

Les résultats sont visualisables dans les figures 6, 7 et 8 à la page suivante. La métrique observée en ordonnée correspond à la mesure du taux moyen de perte des paquets par l'ensemble des nœuds du réseau. C'est un pourcentage. Elle montre les performances du réseau (un taux proche de 0 est meilleur). Les variables observées en abscisse correspondent, respectivement aux figures énoncées précédemment, à :

1. la variation du nombre de nœud dans le réseau [de 10 à 100 nœuds] : indicateur de la densité du réseau ;
2. la variation du nombre de paquet total envoyés par les nœuds sources vers le nœud puits (i.e. débit) [de 200 à 3800 paquets] : indicateur de résilience face à la charge ;
3. la variation de vitesse de déplacement des nœuds [de 2 à 55 m/s] : indicateur de résilience face à la mobilité des nœuds.

La **figure 6** nous montre que suivant l'augmentation du nombre de nœud (donc d'une augmentation de la densité du réseau) : le protocole AODV ne le supporte pas très bien, contrairement au protocole DSR qui semble plutôt apprécier, tandis que DSDV et OLSR s'adaptent bien. À noter que les performances restent honnêtes vu qu'elles ne sont jamais dégradées au-delà de 15 % pour l'ensemble des protocoles (avec nos paramètres de test).

La **figure 7**, quant à elle, nous montre comment les protocoles se comportent face à l'augmentation de la charge sur le réseau. Ils ont tous un comportement assez similaire, leurs pertes de paquets augmentent en fonction de l'augmentation de la charge. DSDV semble être celui qui supporte le mieux cette évolution.

La **figure 8** nous montre les capacités d'adaptation des topologies réseaux des différents protocoles face à la mobilité des nœuds (en somme : la résilience). On constate que DSR est le pire avec des performances minorées par 10 % de perte, puis on constate que les trois autres se bornent entre 5 % et 10 % de taux de perte de paquets (malgré des variations).

Nous remarquerons qu'aucun des 4 protocoles observés ne se démarque du lot. Ils ont tous leurs avantages et leurs inconvénients, malgré des performances plutôt acceptables. Vu qu'ils ne dépassent que rarement les 20 % de perte de paquets.

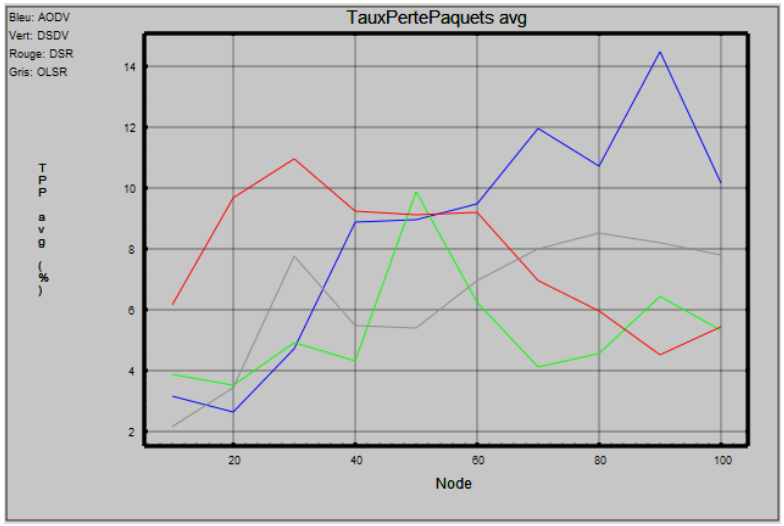

Fig 6 - Pourcentage moyen du Taux de Perte des Paquets de l'ensemble des nœuds du réseau, selon la variation du nombre de nœud total dans le réseau.

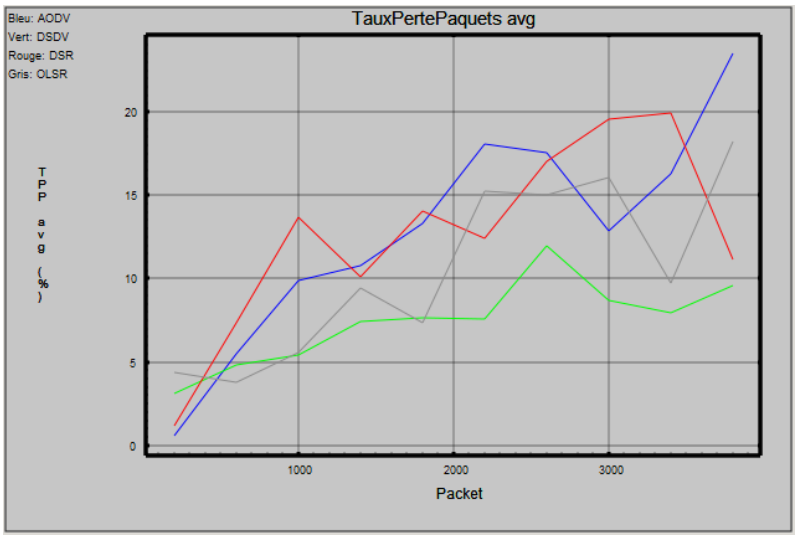

Fig 7 - Pourcentage moyen du Taux de Perte des Paquets de l'ensemble des nœuds du réseau, selon la variation du nombre de paquets total envoyés dans le réseau.

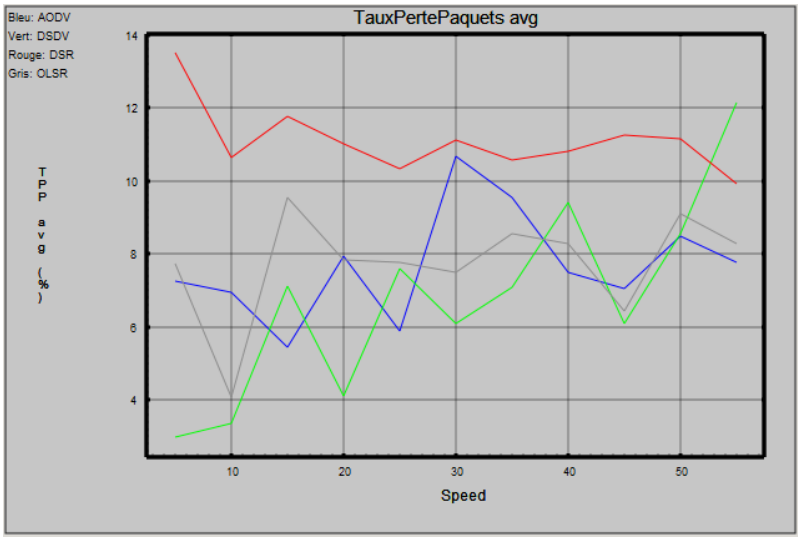

Fig 8 - Pourcentage moyen du Taux de Perte des Paquets de l'ensemble des nœuds du réseau, selon la variation de la mobilité des nœuds dans le réseau (en m/s).

3.7.2. Pire des cas

Nous présentons ici les mêmes types de résultats, sauf qu'ils ont été récoltés au pire des cas. Au lieu de prendre le cas moyen, le maximum est récupéré à chaque instant parmi les nœuds du réseau (soit le pire à chaque fois). En ainsi nous pourrons comparer les comportements de nos 4 protocoles, toujours face aux mêmes variations de paramètres. Les trois figures 9, 10 et 11 sont visibles page suivante.

Figure 9 nous observons que le protocole DSDV se comporte le plus mal face à l'augmentation du nombre de nœud (avec un pic à 95 % de perte, c'est vraiment un enlisement). OLSR est quant à lui le meilleur en restant constamment sous les 40 % de perte au pire des cas.

La **figure 10** nous montre leurs résiliences face à la charge sur le réseau. On constate que le schéma au pire des cas est similaire au schéma du cas moyen. Ils ont des performances dégradées selon l'augmentation de la charge.

La **figure 11** nous montre leurs résiliences face la mobilité des nœuds dans le réseau. Nous voyons un DSDV avec des performances inférieures à 50 % au pire des cas, dès une mobilité dépassant les 15 m/s. Le pire ici. Puis nous apercevons un banc composé des 3 autres protocoles qui sont plutôt résistant vu qu'ils bornent leurs performances au dessus des 60 % de manière assez stable.

Nous constatons donc que les comportements au pire des cas respectent les schémas du cas moyen, bien qu'avec des dégradations proportionnellement plus importantes. Et nous pouvons en conclure : bien qu'au cas moyen les protocoles répondent bien, au pire des cas nous observons des fois des abandons (car, à 50 %, voire 70 % de perte de paquet, ça dégrade les performances drastiquement).

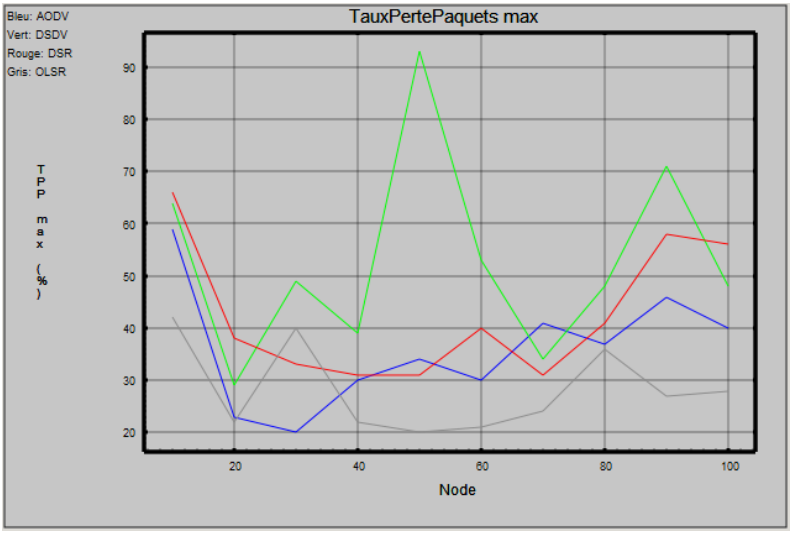

Fig 9 - Pourcentage du Taux de Perte des Paquets du pire des nœuds du réseau, selon la variation du nombre de nœud total dans le réseau.

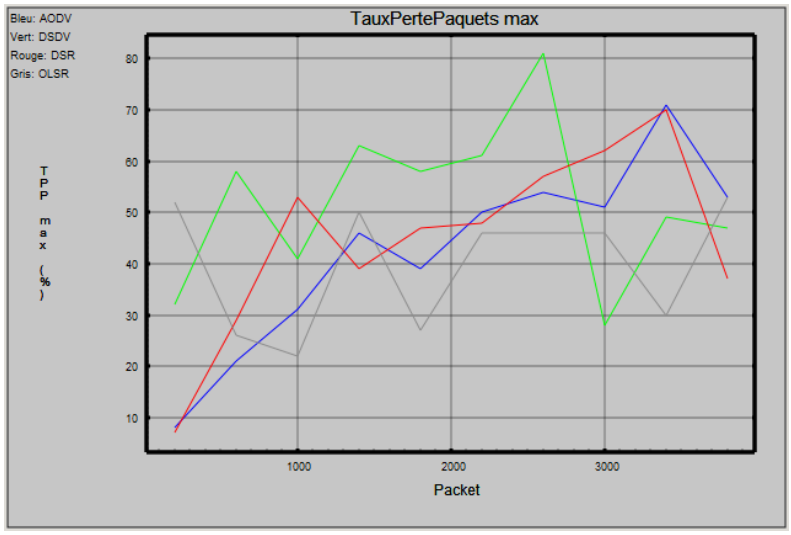

Fig 10 - Pourcentage du Taux de Perte des Paquets du pire des nœuds du réseau, selon la variation du nombre de paquets total envoyés dans le réseau.

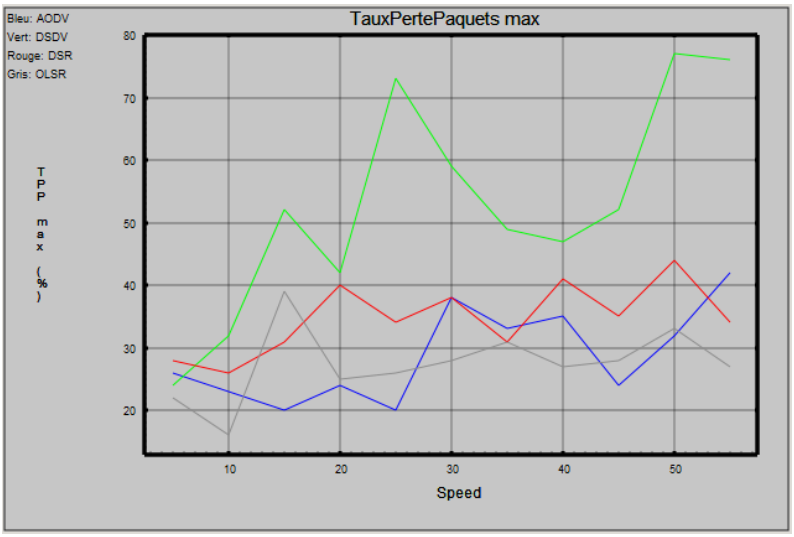

Fig 11 - Pourcentage du Taux de Perte des Paquets du pire des nœuds du réseau, selon la variation de la mobilité des nœuds dans le réseau (en m/s).

4. Conclusion

Dans ce travail nous avons commencé par une étude de l'existant dans les deux domaines que sont : le routage en environnement ad-hoc, et les systèmes distribués pair à pair pur.

Nous avons également travaillé sur NS2 afin de simuler les protocoles existants que sont : AODV, DSDV, DSR, et OLSR. Et nous avons produit et observé différents résultats graphiques de part nos simulations.

De cela, nous pouvons conclure qu'aucun protocole de routage ad hoc existants n'est meilleur dans tous les domaines qu'un autre. Il faut choisir selon le contexte.

Liste des figures

Liste des tableaux

Liste des références

[XGraph]

Site officiel de l'outil de visualisation de graphe XGraph

http://www.xgraph.org/

[TutoOLSR]

Tutoriel pour l'ajout d'OLSR à NS2 (**ns2.chennai@gmail.com**)

https://cloudns2.wordpress.com/um-olsr-patch/

[SrcOLSR]

Sources pour NS2 pour le protocole OLSR

https://sourceforge.net/projects/um-olsr/

[NS2]

1. Sources officielles de NS : **https://sourceforge.net/projects/nsnam/**
2. Tuto de Marc greis : **http://www.isi.edu/nsnam/ns/tutorial/**

[NSallinone]

Sources tout-en-un pour NS2.35

https://sourceforge.net/projects/nsnam/files/allinone/ns-allinone-2.35/

Annexes

Nos tests ont été fait sous une Debian Stretch 9.4 (dernièrement en mars 2018).

Annexe 1 - Procédure d'installation des outils

Les principaux outils intéressés par cette annexe sont : NS2 (simulateur réseau), NAM (visualiseur réseau), XGraph (visualiseur de graphe), et enfin les protocoles à simuler sous NS.

Tout d'abord il faut savoir que ces outils peuvent être installer directement depuis les répertoires de la distribution via le gestionnaire de paquet (tel que *apt* sous debian). Seulement, tous les protocoles ne font pas partie de la distribution standard de NS2, tel que le protocole OLSR qui demande un patch de NS et sa recompilation.

Il existe une distribution «toute en un» pour la suite d'outil relative à NS2 trouvable en ligne sous le nom d'archive *"ns-allinone-2.35.tar"* **[NSallinone]**. Cette archive contient toutes les sources des outils nécessaire à NS (et plus d'ailleurs). Nous allons donc présenter la procédure de compilation des outils depuis cette archive, ainsi que la procédure pour incorporer le protocole OLSR à NS.

1. Installation et compilation de NS2.35

Premièrement il faut extraire l'archive.

```
tar -xvzf ns-allinone-2.35.tar.gz
```

Ensuite il faut installer (au besoin) les dépendances nécessaires à la compilation des outils.

```
sudo apt install build-essential autoconf automake libxmu-dev
sudo apt install gcc
```

Maintenant il faut préparer l'étape de compilation en corrigeant trois fichiers. Le premier est "ls.h" qui se trouve dans le dossier "linkstate" de ns. La correction à appliquer se trouve à la ligne 137, il faut rajouter le pointeur de l'objet (i.e. "this ->") devant l'appel à la méthode erase.

```
cd ~/ns-allinone-2.35/ns-2.35/linkstate
gedit ls.h
   ligne 137 : void eraseAll() { this->erase(...
```

Deuxièmement, dans le fichier ./ns-2.35/mdart/mdart_adp.cc, il faut ajouter le namespace globale (i.e. ::) pour la fonction hash à deux endroit dans le fichier, car il y a ambiguïté avec std::hash et inline hash définit dans ./ns-2.35/mdart/mdart_function.h ligne 230.

```
cd ~/ns-allinone-2.35/
gedit ns-2.35/mdart/mdart_adp.cc
   ligne 108 : nsaddr_t dstAdd_ = ::hash(reqId);
   ligne 396 : nsaddr_t dstAdd_ = ::hash(mdart_->id_);
```

Et enfin dans le fichier ./ns-2.35/configure, il faut modifier la ligne comme suit pour désactiver les erreurs de compilation pour le cast implicite d'entier vers char qui apparaissent depuis le fichier ./ns-2.35/bitmap/play.xbm :

```
gedit ./ns-2.35/configure
   ligne 5128 : V_CCOPT ="$V_CCOPT -Wall -Wno-write-strings -Wno-narrowing"
```

Maintenant que les erreurs de code sont corrigé pour s'adapter à une compilation sous debian stretch avec les dernières versions de paquet installés, on peut donc continuer par indiquer en variable globale les compilateurs présents (comme demandé par le fichier ./install) :

```
export CC=gcc CXX=c++
```

Enfin, nous pouvons lancer l'installation.

```
cd ~/ns-allinone-2.35/
sudo ./install
```

Remarque : Il faut également penser à ajouter le chemin vers NS au fichier ".bashrc" pour plus d'ergonomie, ainsi que les autres dépendances. L'installation réussie le conseil une fois terminée. Donc, ajoutez au besoin les lignes suivantes en fin du fichier ~/.bashrc en prenant soin de remplacer les chemins selon votre configuration.

```
# LD_LIBRARY_PATH
OTCL_LIB=/<path>/ns-allinone-2.35/otcl-1.14
NS2_LIB=/<path>/ns-allinone-2.35/lib
[X11_LIB=/usr/X11R6/lib]
USR_LOCAL_LIB=/usr/local/lib
export            LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OTCL_LIB:$NS2_LIB:$X11_LIB:
$USR_LOCAL_LIB

# TCL_LIBRARY
TCL_LIB=/<path>/ns-allinone-2.35/tcl8.5.10/library
USR_LIB=/usr/lib
export TCL_LIBRARY=$TCL_LIB:$USR_LIB

# PATH
XGRAPH=/<path>/ns-allinone-2.35/bin:
   /<path>/ns-allinone-2.35/tcl8.5.10/unix:
   /<path>/ns-allinone-2.35/tk8.5.10/unix
NS=/<path>/ns-allinone-2.35/ns-2.35/
NAM=/<path>/ns-allinone-2.35/nam-1.15/
PATH=$PATH:$XGRAPH:$NS:$NAM
```

Puis lancer la commande suivante pour appliquer les changements :

```
source ~/.bashrc
```

On peut tester si ns fonctionne avec la commande suivante, et quitter avec exit.

```
ns
%exit
```

Remarque : Si le visualiseur réseau NAM n'est pas compilé par défaut, il est possible de le faire à la main avec les commandes qui suivent pour corriger, configurer, compiler puis tester :

```
cd ./nam-1.15/
gedit configure
   ligne 5047 : V_CCOPT ="$V_CCOPT -Wall -Wno-write-strings -Wno-narrowing"
./configure --with-tcl=/<path_ns>/tcl-8.5.10 --with-tcl-ver=8.5
make
cd ../../
nam
```

2. Intégration du protocole OLSR à NS2.35

Nous avons eu quelques problèmes pour réaliser correctement cette opération. Nous nous sommes principalement inspiré du tutoriel **[TutoOLSR]** pour réussir (bien qu'il n'ait pas suffit). Bref. Pour commencer, il faut récupérer les sources d'OLSR, qui sont trouvable sur SourceForge **[SrcOLSR]**.

Une fois l'archive extraite. Il faut copier l'ensemble dans le dossier "ns-allinone-2.35/ns-2.35/". ATTENTION : Il faut penser à renommer le dossier copié (i.e. "um-olsr") en "olsr" sinon la compilation ne passera pas.

Ensuite, il faut *patcher* les fichiers sources de NS avec le *patch* adéquat présent dans le dossier 'olsr'. Cette opération vise à mettre à jour les sources de NS pour qu'elles prennent en compte le nouveau protocole, ainsi qu'a modifier les fichiers d'installation (tel que le fichier Makefile.in). Il est possible de faire cette étape à la main, mais c'est fastidieux. Avec le patch clé en main, il suffit d'exécuter la commande suivante.

```
cd ~/ns-allinone-2.35/ns-2.35
patch -p1 < olsr/um-olsr_ns-2.35_v1.0.patch
```

Pour enfin finir par recompiler NS.

```
cd ~/ns-allinone-2.35/
sudo ./install
```

Toujours pour tester si ns fonctionne correctement :

```
ns
%exit
```

Voilà !

Annexe 2 - Script TCL pour NS

'script_ns.tcl'

Pour exécuter ce script il est conseillé de disposer d'au minimum 500Mo d'espace disque et d'au minimum 1,5Go de RAM.

Ce script à pour objectif de configurer et lancer les différentes simulations NS en faisant varier les métriques incrémentalement. Il fournit des logs en sortie.

Afin de ne pas alourdir ce rapport, le fichier se trouve sur le github.

Annexe 3 - Scripts Python de collecte de donnée

'script_extract1.py'

Ce script permet d'extraire les données de simulation afin de construire un graphe moyen exprimant la métrique par rapport à une variable de simulation.

Afin de ne pas alourdir ce rapport, le fichier se trouve sur le github.

'script_merge2.py'

Ce script permet de fusionner les courbes des différents protocoles en un unique graphe pour comparaison (et ce pour chaque paramètre de simulation).

Afin de ne pas alourdir ce rapport, le fichier se trouve sur le github.

'script_eraseWorkingFiles3.py'

Ce script permet de simplement supprimer les fichiers de travail créé par les simulations du script TCL, au cas où on relancerait une deuxième fois le script. Car le script d'extraction python fonctionne en ajout, ce qui provoque des erreurs si les fichiers ne sont pas, au préalable, supprimés (ou au moins vidés).

Afin de ne pas alourdir ce rapport, le fichier se trouve sur le github.