



Rapport Projet 6 :

Qualité de Service dans les

réseaux Ad Hoc

Tuteur : Ralph El Khoury

Auteurs : Moussa Kane – Hafid Houd – Alexandre Lafarcinade

Remerciements

Nous tenons à remercier notre tuteur, Ralph Ek Khoury pour nous avoir permis de travailler sur un projet aussi intéressant que la qualité de services dans le cadre des réseaux Ad hoc. Ses orientations ainsi que sa patience ont été des éléments primordiaux de la réussite de ce projet.

Sommaire

I- Présentation	p4
1- Ad Hoc	
2-L'Accès au Média	
3-OLSR	p6
4-QoS	p8
5-Problématique	
II- Cadre du projet	p9
1-Le simulateur NS	
a-Présentation	
b-Architecture	
2-Intégration de OLSR à NS	p12
III- Constat	p12
1-Pérfomances actuelles	p13
a-Congestion	
b-Gestion statique des retransmission	
c-Qualité de service	
2-Simulations	
3-Résultats	p18
IV- Solution proposée	p18
1-Principe général	
2-Développement et implémentation	p19
a- Matlab/c++	
b- Intégration dans l'architecture de ns	
c- Compilation	
3-Simulations	p24
V-Analyse & performances des résultats	p28
1- Comparaison	
2- Evaluation des performances de notre solution	p32
VI- Contraintes & Difficultés	p33
VII- Bilan	p34
Annexes	p35

I- Présentation

1- Ad Hoc

La norme IEEE 802.11 (ISO/IEC 8802-11) est un standard international décrivant les caractéristiques d'un réseau local sans fil (WLAN).

Par abus de langage et pour des raisons de marketing le nom Wifi se confond aujourd'hui avec le nom de la certification.

Ainsi un réseau Wifi est en réalité un réseau répondant à la norme 802.11 qui fonctionne comme les réseaux Ethernet que nous connaissons à la différence fondamentale que l'accès au Média se fait par voie radioélectrique.

Par contre les réseaux wifi tels que nous les connaissons actuellement souffrent d'une contrainte relativement préoccupante: la mobilité. D'ou l'introduction d'une nouvelle solution sans infrastructure centrale c'est à dire que l'on fasse abstraction du concept de point d'accès indispensable dans le Wifi. Ainsi on voit naitre en 1998 MANET acronyme de Mobile Ad-Hoc Networks, un groupe de travail de l' IETF (Internet Engineering task Force).

Les réseaux **Ad Hoc** se représentent comme un ensemble de noeuds, jouant le rôle de station et de routeur. Un réseau Ad Hoc est un réseau sans fil, à infrastructure changeant dans le temps et ne comportant pas d'infrastructure fixe (câble réseau, routeur, switch etc.). La topologie de ce type de réseau est dynamique, les noeuds peuvent se déplacer au cours du temps. Ce type de réseau ne fait pas appel à un administrateur ou un gestionnaire de réseau car les réseaux Ad Hoc ont pour but d'être autonomes et déployables à tout moment.

2-L'Accès au Média

La couche Liaison de données de la norme 802.11 est composée de deux sous-couches la couche de contrôle de la liaison logique (Logical Link Control, notée LLC) et la couche de contrôle d'accès au support (Media Access Control, ou MAC).

La couche MAC définit deux méthodes d'accès différentes:

- La méthode CSMA/CA utilisant la Distributed Coordination Function (DCF)
- La Point Coordination Function (PCF)

α-Le PCF

Point Coordination Fonction (PCF) est un mode dans lequel les stations de base ont la charge de la gestion de l'accès au canal dans leur zone de couverture pour les mobiles qui leur sont rattachés.

Il est fondé sur l'interrogation à tour de rôle des stations, ou polling, contrôlée par le point d'accès. Une station ne peut émettre que si elle est autorisée et elle ne peut recevoir que si elle est sélectionnée. Cette méthode est conçue pour les applications temps réel (vidéo, voix) nécessitant une gestion du délai lors des transmissions de données.

b-Le CSMA/CA et le DCF

Dans un réseau local Ethernet classique, la méthode d'accès utilisée par les machines est le CSMA/CD (Carrier Sense Multiple Access with Collision Detection), pour lequel chaque machine est libre de communiquer à n'importe quel moment. Chaque machine envoyant un message vérifie qu'aucun autre message n'a été envoyé en même temps par une autre machine. Si c'est le cas, les deux machines patientent pendant un temps aléatoire avant de recommencer à émettre.

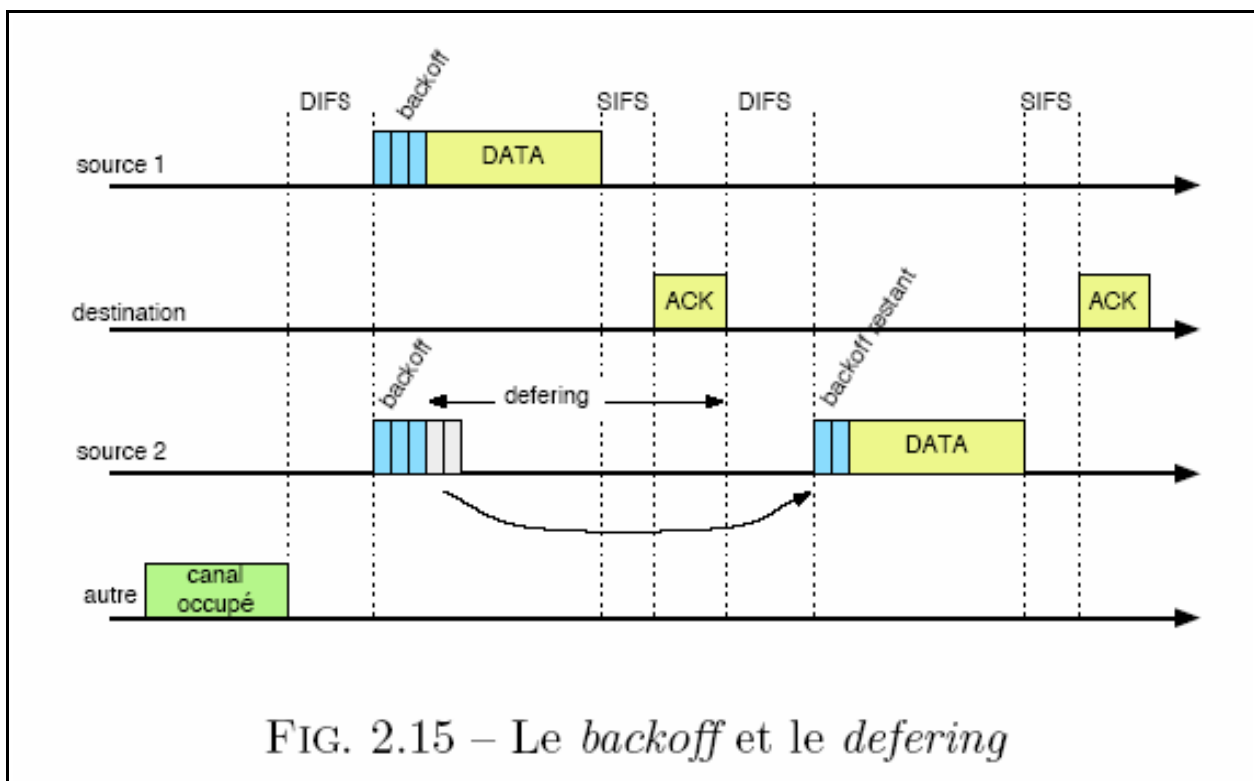
Dans un environnement sans fil ce procédé n'est pas possible dans la mesure où deux stations communiquant avec un récepteur ne s'entendent pas forcément mutuellement en raison de leur rayon de portée. Ainsi la norme 802.11 propose une méthode similaire appelé CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance).

La méthode d'accès au média CSMA/CA utilise un mécanisme d'esquive de collision basé sur un principe d'accusés de réception réciproques entre l'émetteur et le récepteur:

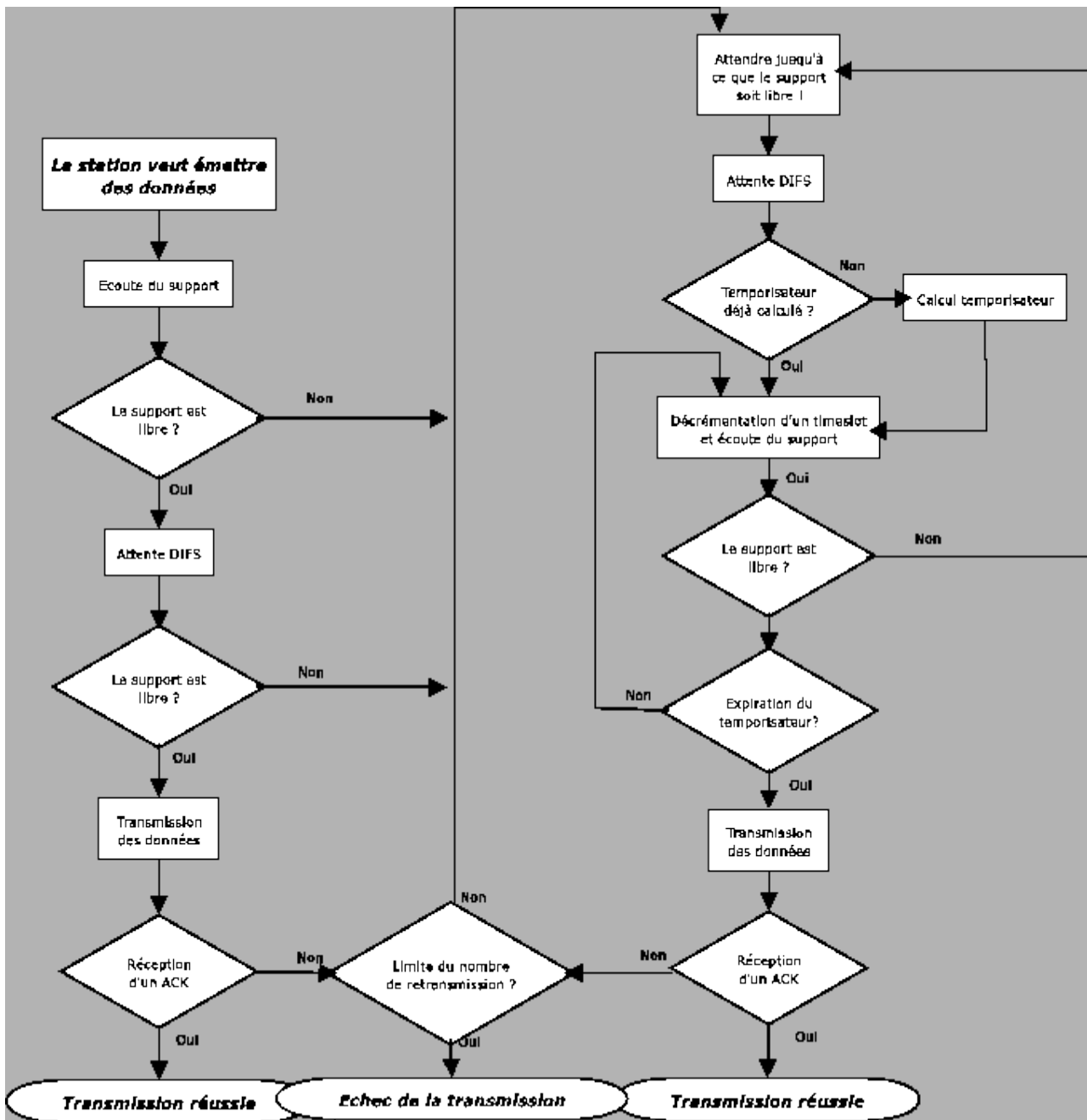
La station voulant émettre écoute le réseau. Si le réseau est encombré, la transmission est différée. Dans le cas contraire, si le média est libre pendant un temps donné (appelé DIFS pour Distributed Inter Frame Space), alors la station peut émettre.

La station transmet un message appelé Ready To Send (noté **RTS** signifiant prêt à émettre) contenant des informations sur le volume des données qu'elle souhaite émettre et sa vitesse de transmission. Le récepteur (généralement un point d'accès) répond un Clear To Send (**CTS**, signifiant Le champ est libre pour émettre), puis la station commence l'émission des données.

L'idée du 802.11 est de vérifier dans un premier temps lors de l'émission d'une donnée, si le canal est libre. Si c'est le cas, l'émetteur doit attendre une période de durée aléatoire appelée backoff avant d'émettre. Ce mécanisme s'applique lorsque le canal devient libre aussi bien après une de nos propres émissions qu'après toutes autres émissions. Ainsi, si plusieurs mobiles veulent émettre, il y a peu de chances pour qu'ils aient choisi la même durée. Celui qui a choisi le plus petit backoff va commencer à émettre, et les autres vont alors se rendre compte qu'il y a à nouveau de l'activité sur le canal et vont ré-attendre. La figure suivante schématise ce qui se passe lorsque deux mobiles veulent émettre et met en avant les mécanismes de temporisation.



c- L'Algorithme du Backoff



Dans le DCF, le calcul du temporisateur se fait par le biais de l'algorithme du back-off. Il est utilisé de la même manière que dans le CSMA/CD . La seule chose qui change, c'est qu'on ne détecte pas la collision, mais, on déduit qu'il s'est produit une collision lorsqu'on ne reçoit pas d'ACK.

Le Back-off permet de tirer un nombre aléatoire entre 0 et X, où la valeur de X croit exponentiellement par rapport au nombre de tentatives de transmission. Le nombre tiré est multiplié par le timeslot. La station devra attendre le temps correspondant au résultat de l'algorithme avant de refaire une nouvelle tentative de réémission sur le support, tout en vérifiant qu'il soit libre. Après un certain nombre d'échec, on considère que l'émission a échoué.

Cette technique permet d'éviter au maximum les collisions en laissant, pour chaque station, la même probabilité d'accès au support.

3-OLSR

Dans un premier temps, la source doit trouver le chemin jusqu'au destinataire. Elle peut s'appuyer sur une connaissance préalable du chemin ou demander à d'autres entités un chemin partiel ou complet. Si la source utilise une information incomplète, une chaîne de relais peut se créer jusqu'à joindre le destinataire. Ce dernier s'appuie alors sur les informations reçues pour retrouver le chemin vers la source et ainsi construire le chemin.

Dans le cas d'un réseau Ad Hoc, l'opération de routage se heurte à de nombreuses difficultés car la recherche de routes s'appuie sur des informations dynamiques. Des mécanismes (réguliers ou utilisés seulement lors de la recherche de routes) doivent exister pour obtenir une route valable. En clair, les noeuds ne peuvent s'appuyer sur une information statique, et doivent obtenir dynamiquement les informations sur les routes réactualisées.

Trouver un chemin n'est qu'une partie du problème, il faut pouvoir assurer la stabilité des communications car la mobilité des noeuds peut entraîner de nombreuses reconfigurations des chemins. Ainsi, durant la communication, l'ensemble des relais d'une communication va changer plus ou moins fréquemment. De plus, par la nature même des réseaux Ad Hoc, les déconnexions peuvent être un événement fréquent en effet, un noeud peut se retrouver sans possibilité de joindre le destinataire, simplement par le fait qu'il ne possède pas de voisins ou que le graphe du réseau joignable n'inclut pas le destinataire. Cette erreur oblige la source et/ou certains relais à temporiser des envois et/ou à informer les applications de la source de cet événement.

Nous ne nous intéresserons qu'à un protocole de routage pro-actif car comme le précisait notre Cahier des Charges, la solution envisagée s'appuie sur des informations de routage les plus complètes possibles. Ainsi on préférera un protocole qui dresse une «carte» complète de la topologie du réseau plutôt qu'un qui établit des tables trop souvent.

Le protocole OLSR (Optimized Link State Routing Protocol) est un algorithme proactif conçu pour minimiser la taille des messages de contrôle. Chaque noeud calcule régulièrement le sous-ensemble MPR (MultiPoint Relaying) de ses voisins, permettant de joindre l'ensemble des mobiles à deux sauts. Ensuite, un noeud diffuse régulièrement un message TC (Topology Control) à l'ensemble du réseau. Ce paquet contient l'ensemble de ses voisins l'ayant choisi pour faire partie de leur MPR. Ainsi, un noeud peut associer le voisin à joindre du réseau au noeud voisin permettant de le joindre. Pour réduire les problèmes de diffusion, chaque noeud réémet le message s'il appartient à l'ensemble MPR de la source locale.

Un des avantages des protocoles proactifs, sous réserve d'une mise à jour correcte de l'ensemble des tables, est le fait de trouver rapidement le destinataire sans devoir au préalable effectuer une recherche dans le réseau complet (l'information pour router les messages est immédiatement disponible).

De plus, les pertes de chemin sont relativement peu fréquentes, car les changements de topologie sont diffusés automatiquement. Enfin, chaque noeud possède des informations sur l'ensemble du réseau, ce qui peut se révéler pratique pour la couche application.

Mais les défauts de ces protocoles se révèlent catastrophiques dans le cas de réseaux Ad Hoc avec une forte mobilité.

Premièrement, il existe un coût constant dans le réseau pour diffuser les informations de routage. Même si certains algorithmes réduisent la quantité d'information, cette occupation d'une partie de la ressource radio réduit la quantité disponible pour les communications entre mobiles. Deuxièmement, ce type de protocole n'est pas adapté pour de grands réseaux. En effet, la quantité d'information à diffuser pour une maintenance cohérente augmente proportionnellement au nombre de noeuds.

Finalement, une mobilité importante dans le réseau peut entraîner des risques de perturbations importantes. En effet, la quantité d'information devient trop importante pour le réseau car le nombre de messages est fonction des changements topologiques.

Le choix d'OLSR s'est vite imposé car l'autre protocole de routage disponible avec ns, AODV, est un protocole que l'on caractérise de ré-actif, c'est à dire qu'il ne recherche la route pour acheminer un paquet que lorsque le besoin s'en fait sentir. Dans une application telle qu'une communication ad hoc, cela pose problème, et ce plus encore dans le cas de notre projet car nous avons besoin d'informations de routage pour calculer le nombre de retransmissions en évitant d'ajouter de la charge au réseau comme le ferait AODV.

4-QoS

Généralement, dans un réseau, le routage permet d'établir une route de plus court chemin en terme de distance ou de délai entre deux noeuds source et destination.

Dans le cadre d'une qualité de service, le but du protocole de routage est de trouver la meilleure route selon les critères précis de la qualité de service souhaitée (délai, taux de perte, quantité de bande passante, ...), et reposant sur des liens fiables.

Ce dernier point est à la fois important et difficile à assurer dans le cas des réseaux ad hoc en raison de leur topologie dynamique. Les noeuds constituant le réseau ad hoc doivent stocker et mettre à jour les états des liens dans un environnement qui est mobile. Ce processus est donc très complexe et coûteux car des ruptures de liens peuvent intervenir à tout moment beaucoup plus fréquemment que dans des réseaux classiques.

Plusieurs solutions de protocoles de routage pour les Manets ont été proposées.

La "qualité de service" pour les réseaux est un terme très largement employé mais pour lequel le principe même d'une définition n'est pas logique. Il est donc difficile de la définir exactement.

En effet, la qualité de service demandée pour un réseau dépend des caractéristiques de ce réseau et des besoins de ses utilisateurs.

De manière générale, il s'agit de réduire la congestion, les délais, les files d'attente et la perte de paquets d'informations. Dans le cas d'un réseau ad hoc, il va également s'agir d'adapter ces critères de qualité à la mobilité des noeuds constituant ce réseau, c'est à dire assurer une qualité de service malgré le caractère dynamique de la topologie du réseau.

Il existe des modèles de qualité de service pour les réseaux, impliquant un certain nombre de normes, protocoles et bonnes pratiques pour assurer de la qualité de service auprès de ces réseaux. Des modèles de QoS ont été élaborés pour les réseaux ad hoc.

5-Problématique

Les études montrent que l'enjeu de la qualité de la transmission sans fil réside dans le nombre de retransmissions de trames dans l'Algorithme du Backoff parce que c'est le mécanisme de la couche liaison qui évite la congestion sur la couche plus haute du TCP.

Etant dans un environnement où les noeuds sont supposés être très mobiles, le nombre limite de retransmissions ne pourrait être distribué équitablement entre les noeuds du réseaux suivant leur connaissance des routes, le nombre de sauts à faire pour atteindre les destinations. Ainsi l'enjeu

central de ce projet demeure en l'affectation d'un nombre limite de retransmissions dynamique suivant les données récoltées dans la couche supérieure réseau du modèle OSI en vue d'améliorer la Qualité de Service, terme qui signifie pour nous beaucoup de choses incluant l'augmentation des débits, la réduction des délais et la limitation des collisions de façon notoire.

Le principe dont nous allons partir est que, pour permettre à des paquets d'avoir une plus grande chance d'arriver à destination, on va chercher à faire évoluer le nombre des retransmissions limite au niveau de chaque noeud. Plusieurs paramètres doivent être pris en compte, la position du noeud dans la communication, la probabilité de réussite de la transmission. Nous travaillerons uniquement sur la position du noeud dans le réseau.

La solution sera décrite plus en détails par la suite.

L'étude se déroulera en utilisant l'outil logiciel de simulation ns, la partie qui suit a pour but de définir l'environnement dans lequel nous avons travaillé et la mise en place des différents outils nécessaires à la réalisation.

II- Cadre du projet

Lorsque nous avons rédigé notre Cahier des Charges, nous avons prévu l'installation d'outils nécessaires à nos travaux. Premièrement une distribution de Linux qui nous permettrait d'y installer le simulateur ns, ensuite la gestion du protocole OLSR par le simulateur. Nous avons choisi d'installer deux machines l'une sous Debian, l'autre sous Gentoo, et ce pour une plus grande souplesse dans l'avancement mais également afin de pouvoir travailler simultanément sur un même objectif.

Nous allons à présent détailler les outils logiciels ainsi que les outils complémentaires installés afin de réaliser notre projet.

1-Le simulateur NS

a-Présentation

NS est outil logiciel de simulation de réseaux informatiques. Il est principalement bâti avec les idées de la conception objet, de réutilisation du code et de modularité. Le simulateur Ns est devenu un standard de référence en ce domaine. L'avantage de ce logiciel, c'est qu'il est disponible sur Internet, son utilisation est gratuite et il est compatible avec les systèmes Windows et Unix. Toutefois, l'installation et l'utilisation sous Windows ne semblaient pas offrir suffisamment de garanties au vu de la difficulté d'installation et d'accès aux codes source, ce qui était une condition primordiale de notre travail, c'est pourquoi nous avons opté pour l'univers Linux.

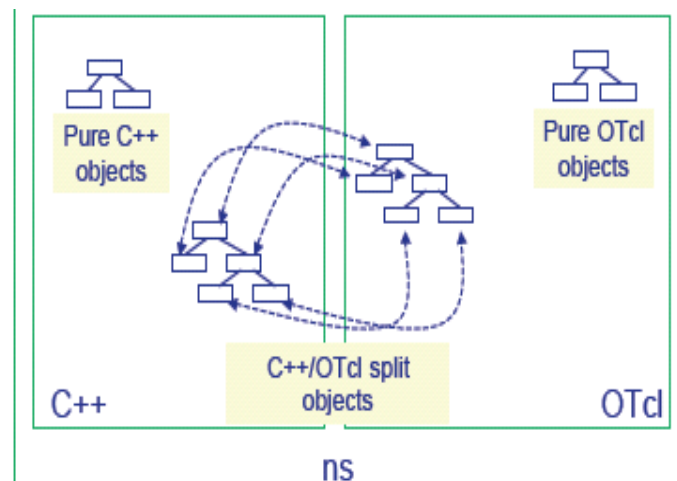
Nous avons choisi ce logiciel car c'est un très bon outil dans le domaine de la recherche, et du développement de nouveaux protocoles pour différents types de réseaux. Pour exemple, le protocole OLSR qui est à présent largement répandu a été développé, testé et validé à l'aide de ns. Ce simulateur a la caractéristique d'être en accord avec le réel pour la plupart des points suivant : délai, bande passante, durée de vie ; et pour faire de la qualité de service. Cet outil nous permettra d'implémenter notre solution sans avoir à déboursier pour du matériel et par conséquent à simuler si l'on le souhaite, un réseau de 100 noeuds. Ce qui était impossible dans une expérimentation réelle car on serait très vite limité en budget.

b-Architecture

L'architecture de ns est très complexe et nous ne présenterons que les parties sur lesquelles nous avons travaillé et qui nous ont permis de mener à bien notre projet.

Ans, il existe un hiérarchie des classes qui doit être appréhendée avant de commencer à exploiter le code du simulateur. Une grande partie du travail a été de se documenter, de chercher et d'analyser le logiciel afin de mieux comprendre son fonctionnement. D'abord le principe général et l'interaction entre les langages c++ et tcl, puis plus en profondeur, l'organisation en classes c++ et finalement les structures de données et les paramètres correspondant à ce que nous voulions faire, à savoir changer le nombre de retransmissions de la couche MAC en utilisant des informations de routage.

NS est un outil de simulation de réseaux bâti autour d'un langage de programmation appelé TCL «Tool Command Language ». Pour avoir une vue globale sur le logiciel, il est intéressant de détailler les composants essentiels au développement.



Architecture ns

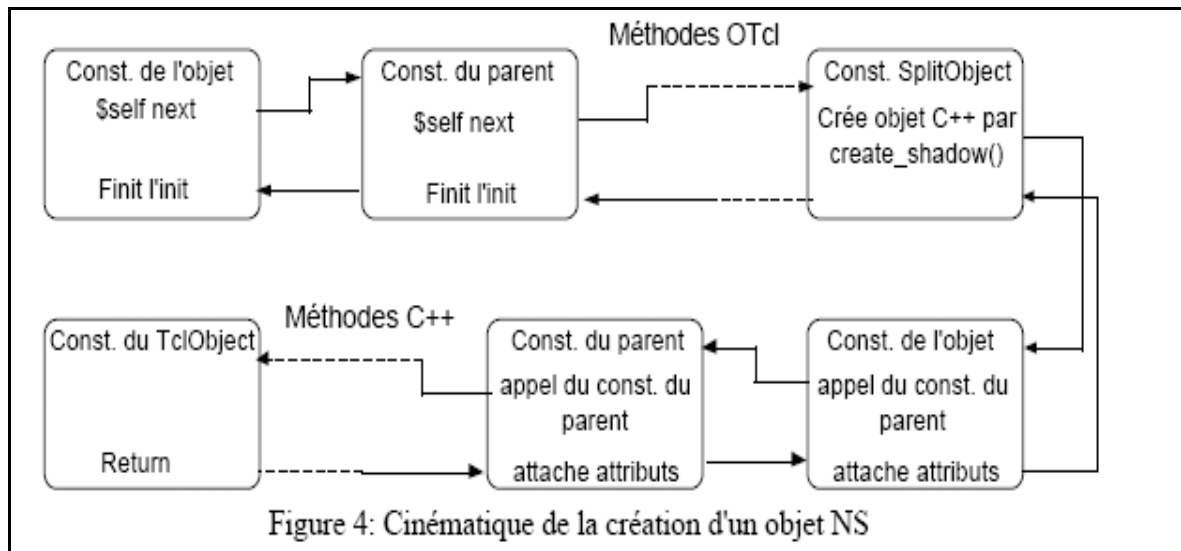
Le cœur du logiciel est un ensemble de classes c++ qu'il est nécessaire de compiler pour construire l'exécutable. C'est grâce à ces classes que sont créés les objets qui sont après utilisés pour simuler le fonctionnement d'une application réseau. Par exemple, les composants d'un réseau, un noeud, une couche du modèle OSI ou même un paquets sont en fait des objets c++ qui sont définis par des classes. Il est possible d'accéder au code c++ afin d'apporter des modifications ou même d'ajouter des classes. Cela nous intéresse fortement comme nous le verrons plus en détails dans la partie correspondant à l'implémentation de la solution.

Les classes c++ sont organisées de la manière suivante dans le répertoire du simulateur:

ns / «nom du protocole, ou contexte»/fichiers.h et .cc .

Par contexte, nous entendons par exemple pour tous les objets permettant de mettre en place la couche MAC, le contexte est MAC. Pour le protocole OLSR, il y a plusieurs classes dont packet_olsr.cc et olsr_rtable.cc.

Les simulations sont réalisées en écrivant des scripts en langage TCL qui sont ensuite interprétés par le simulateur afin de créer les objets et de les placer dans le «scénario».



Il existe une partie de l'architecture ou les paramètres des protocoles, les paramètres par défaut ainsi que les bibliothèques nécessaires à la réalisation de simulations sont définis (répertoire tcl/lib). Ns permet de tracer les événements observés lors des simulations, c'est une fonction très importante qui permet de récupérer des résultats en fonction du temps, du noeud, ces informations peuvent relatives à la topologie, aux paquets...

Nam : c'est l'outil de visualisation des simulations. L'interface permet de visualiser le déroulement des communications en plaçant la topologie dans l'espace et en simulant les échanges dans le temps.

Nous avons détaillé l'outil logiciel de simulation, mais le plus important dans une étude telle que la notre est l'analyse des résultats recueillis. Deux programmes ont été utilisés, l'un pour extraire les données des fichiers trace, awk, l'autre pour mettre en forme ces données sous forme de graphiques, gnuplot.

Procédure d'installation de ns-allinone versions 2.29 et 2.30

- Téléchargement de l'archive de la version
- Extraction dans le répertoire souhaité (/var/local pour la version 2.30 sous Debian, /home/ns pour la version 2.29, sous Gentoo)
- Lancement du script d'installation ./install dans le répertoire ns-allinone"version"
 - Si erreur, faire ./configure avant pour installer les paquets de l'archive nécessaires à l'installation du simulateur.
- Une fois l'install terminé, il faut compiler ns
 - Commande make (le Makefile contient la suite d'opérations, les dépendances et les bibliothèques nécessaires à la compilation)
 - la compilation est longue, plus de 30 min.
- L'installation est alors terminée, pour la vérifier, lancer ./validate, un script qui permet de tester l'installation est lancé et validera ou non l'installation.
- Pour une utilisation plus commode, exporter les variables d'environnement pour lancer les commandes nécessaires en tapant uniquement la commande "ns" par exemple à partir de n'importe quel répertoire.

2-Intégration de OLSR dans NS

Dans le Cahier des Charges, nous avons choisi de simuler en utilisant le protocole OLSR dans nos communications.

OLSR n'est pas inclus dans "ns" par défaut, en tout cas pas dans les versions stables disponibles, pour l'installer, il faut "patcher" certains fichiers de "ns", ajouter les bibliothèques et classes OLSR et recompiler le tout.

- Téléchargement de l'archive um-olsr (d'autres existent mais celle-ci semblait la plus complète et le mieux documentée)
- Extraction de l'archive (renommer le répertoire "um-olsr" en "olsr" car le patch le nécessite)
- Déplacer le fichier "um-olsr_ns-"version"_v0.8.8.patch" dans le répertoire "rep_ns/ns"version""
- Déplacer le répertoire olsr dans ce même répertoire
- Exécuter le patch avec la commande


```
#patch -p1 < um-olsr_ns-"version"_v0.8.8.patch
```

Remarque: le patch doit être modifié en fonction de la version de ns utilisée (remplacer ns2.29 par ns2.30 partout dans le fichier .patch)

- De plus, il peut être nécessaire d'effectuer certains ajouts ou modifications de code directement sans le patch si ce dernier ne le fait pas pour diverses raisons. Par exemple, nous avons dû ajouter la ligne correspondant aux modules olsr (les .o) dans le Makefile car le patch les avait omis.
- Ensuite, lorsque le patch est appliqué, il faut recompiler ns qui inclura maintenant olsr
 - Exécution de la commande make. (ce doit être bien moins long que la compilation initiale car très peu de choses sont modifiées)

Remarque: ne surtout pas exécuter la commande ./validate qui reconfigurerait tous les fichiers pour l'installation du simulateur seul!
- L'installation est alors complète et le simulateur prêt à l'emploi avec la prise en charge du protocole OLSR nécessaire dans nos travaux.
- Pour tester, un scripte mettant en œuvre OLSR est disponible avec l'archive du protocole, lancer la simulation avec ns et s'il n'y a pas d'erreurs c'est que l'intégration a réussi.

Nous disposons à présent d'un environnement de travail qui nous permettra de poursuivre dans nos travaux. Intéressons nous donc à l'analyse de la QoS dans le cadre des réseaux ad hoc mettant en œuvre le protocole OLSR.

Nous commencerons par une analyse du comportement du réseau avec les mécanismes de QoS par défaut, à savoir la gestion des retransmissions mais également l'utilisation ou non des RTS/CTS dans la gestion des collisions.

III- Constat

Le but initial de notre étude est de faire apparaître les faiblesses de la gestion des retransmissions avec un K statique dans les communications d'un réseau ad hoc. Pour cela, nous avons mené des recherches préalables afin de savoir quelles seraient les orientations à suivre afin de caractériser le comportement du réseau avec OLSR.

1-Performances actuelles

a-Congestion

L'état actuel des performances nous amène à dire que la situation en terme de congestion peut être améliorée afin d'éviter la saturation du réseau. En l'état actuel des choses plus une destination est loin moins les paquets risquent d'arriver à destination. Ainsi, rien ne garantit qu'une communication nécessitant une régularité du débit pourra avoir lieu sans encombre. C'est pourquoi il existe des mécanismes que nous allons expérimenter permettant de réserver une partie de la bande passante pour un service sans encombre.

b-Gestion statique des retransmissions

Dans le réseau Ad Hoc basé sur 802.11 le nombre limite de retransmission est fixe pour tous les nœuds du réseau donc la probabilité pour qu'une machine puisse être sélectionnée pour émettre est théoriquement la même. Le nœud qui tire le minimum dans l'algorithme du backoff doit émettre pendant que les autres attendent avant de tenter à réémettre jusqu'à l'expiration de leur possibilité d'émission. Ce procédé pose un problème du fait que les nœuds soient très mobiles et ont une connaissance différente des destinations et de la topologie du réseau, une autre méthode pourrait appliquer le nombre limite de retransmissions suivant que les nœuds veulent atteindre des voisins loins ou proches. Ainsi si une trame veut atteindre sa destination en 6 sauts qu'elle ait les mêmes chances d'arriver à destination que celles qui font juste un saut.

c-Qualité de service

Appliquée aux réseaux à commutation de paquets (réseaux basés sur l'utilisation de routeurs) la QoS désigne l'aptitude à pouvoir garantir un niveau acceptable de perte de paquets, défini contractuellement, pour un usage donné (voix sur IP, vidéo-conférence, etc.). Dans notre cas avant implémentation de nos solutions la QoS demeure relativement peu performante en terme de délais de congestion et de débits ce que nous allons observer dans les schémas suivants.

2-Simulations

Pour se faire nous avons simulé 4 topologies différentes :

- un réseau, de 6 nœuds, linéaire
- un réseau, de 10 nœuds, linéaire
- un réseau, de 14 nœuds, linéaire
- un réseau, de 10 nœuds, maillé

Pour chacune des simulations précédentes nous avons établi une communication basée sur le protocole UDP dans laquelle nous avons fait varier le débit du CBR (Constant Bit Rate) de 512 kb/s à 1024 kb/s,

- nombre de retransmissions de la couche MAC, qui est par défaut dans NS-2 de 4 pour les paquets de données et 7 pour les paquets RTS/CTS que nous avons laissé tel quel.

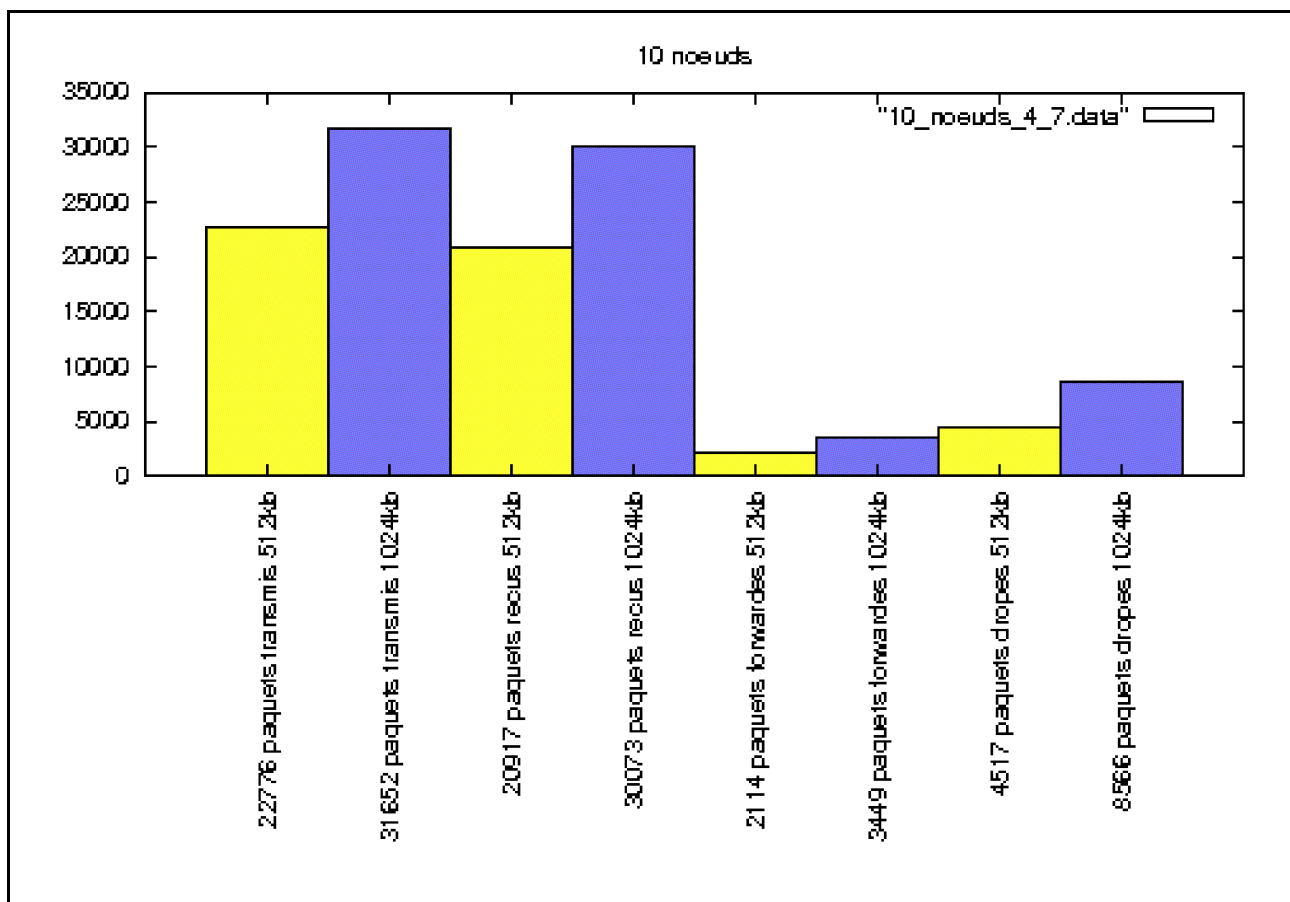
Chacune de ces 24 simulations nous a généré un fichier trace qui contient tous les messages échangés lors de la simulation. Ce dernier est un fichier lourd qui nécessite des scripts awk pour en extraire des données telles que le nombre paquets transmis, reçus, forwardés... ainsi que le délai et le débit de transmission des paquets.

Les scripts de calcul de délai de transmission nécessitent beaucoup de ressources, la totalité nous a pris environ 5h, au moins 30 minutes par calcul...

Par manque de temps jusqu'à présent, nous n'avons encore pas tracé les résultats observés dans le cas où les RTS/CTS ne sont pas utilisés.

Pour illustrer de telles simulations, nous avons utilisé Gnuplot.

Les simulations dans le cas existant de ns, à savoir le nombre de retransmissions par défaut à 4/7, une communication CBR entre les noeuds 0 et 9 dans une topologie linéaire de 10 noeuds, utilisation des RTS/CTS nous donnent:



On y observe que le nombre de paquets dropés est très important, plus d'un quart!

Intéressons nous donc à ces paquets dropés en essayant de définir la raison de ces pertes. Pour cela, nous utilisons un scripte awk différent qui va détailler la provenance des paquets perdus, et le résultat obtenu pour un débit de 512 est :

```
les noeuds ont drope 3337.000000 paquets NRTE
les noeuds ont drope 40.000000 paquets IFQ
les noeuds ont drope 1140.000000 paquets COL
```

Les paquets NRTE sont des paquets perdus car le noeud ne connaît pas encore la topologie et donc ne peut pas définir vers quel noeud envoyer. Les marques IFQ correspondent aux files d'attente

trop pleines. Enfin, ce qui nous intéresse le plus ce sont les paquets marqués COL qui sont les pertes à cause de l'échec des retransmissions. On notera ici que leur nombre est assez important.

Pour compléter l'étude de cette simulation, nous appliquons un scripte awk qui mesure les débits et délais:

Pour un débit CBR de 512Kbps:

le delai moyen est de 1.210873 s

le debit est de 5949.809063 octets/s

Pour 1024Kbps:

le delai moyen est de 2.293805 s

le debit est de 5546.203262 octets/s

Nous avons à présent un aperçu de ce qui se passe dans une réseau dont les paramètres sont ceux par défaut d'une communication ad hoc dans le simulateur ns.

Enlevons maintenant les RTS/CTS qui permettent normalement d'améliorer l'accès au canal:

les noeuds ont transmis 30592.000000 paquets

les noeuds ont reçu 26495.000000 paquets

les noeuds ont forwardé 7113.000000 paquets

les noeuds ont dropé 5044.000000 paquets

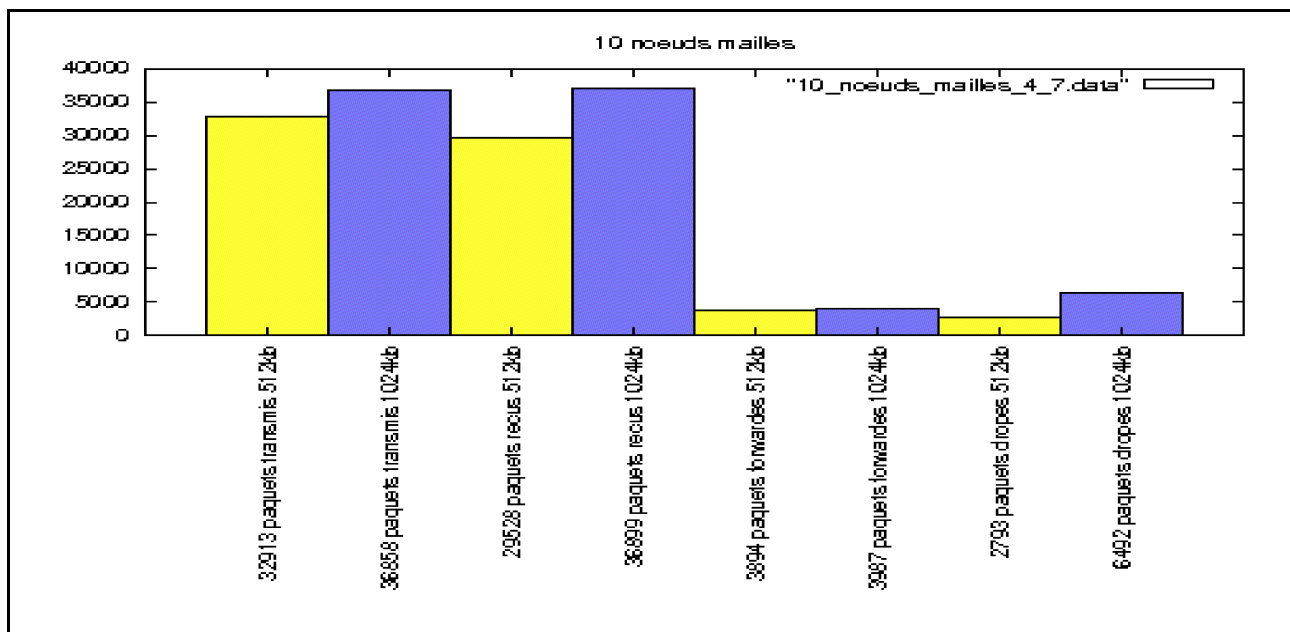
les noeuds ont dropé 1389.000000 paquets NRTE

les noeuds ont dropé 1604.000000 paquets IFQ

les noeuds ont dropé 2051.000000 paquets COL

On note que quand les stations n'utilisent pas les messages Clear to Send et Ready to Send, le nombre de collisions augmente sensiblement (le double) et ce car on envoie avec moins de précautions.

Nous allons faire varier la topologie afin d'observer ce qui se passe dans une topologie moins linéaire, avec des noeuds maillés.



Les résultats sont sensiblement les mêmes, beaucoup de pertes malgré l'utilisation des RTS/CTS, pour y voir plus clair, nous allons détailler ces paquets dropés pour un débit de 512:

les noeuds ont drope 361.000000 paquets NRTE
les noeuds ont drope 2178.000000 paquets IFQ
les noeuds ont drope 254.000000 paquets COL

Pour les débits et délais CBR 512:

le delai moyen est de 1.745726 s

le debit est de 19523.369714 octets/s

CBR 1024Kbps:

le delai moyen est de 1.909825 s

le debit est de 20245.769272 octets/s

Les collisions génèrent beaucoup moins de pertes, de même que les NRTE, cela s'explique par le fait que les noeuds possédant plus de voisins, la connaissance de la topologie survient plus rapidement dans le scénario de simulation. Pour les collisions, le fait que certains noeuds «centralisent» un grand nombre d'informations RTS/CTS fait que ces messages ont un effet bien meilleur que dans le cas d'une topologie linéaire dans laquelle ils ne valent que pour des couples de deux noeuds en général. Là, un message RTS/CTS sera reçu par un plus grand nombre de noeuds, ces derniers n'essaieront donc pas d'envoyer avant que le canal soit libre, il en résulte une meilleur fluidité dans la communication.

Lorsque l'on enlève les RTS/CTS dans une topologie comme celle-ci, on observe:

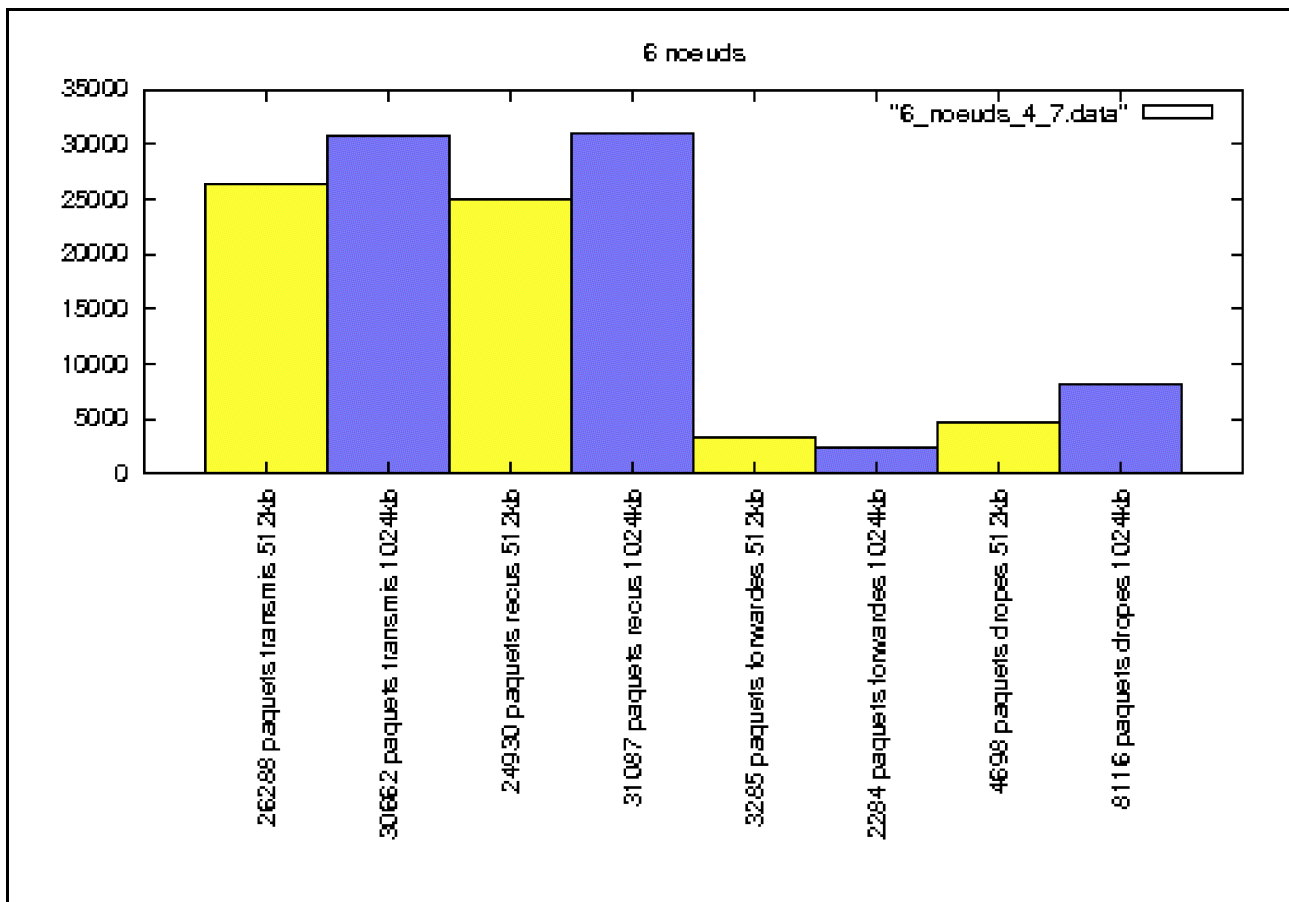
les noeuds ont transmis 30592.000000 paquets
les noeuds ont reçu 26495.000000 paquets
les noeuds ont forwardé 7113.000000 paquets
les noeuds ont drope 5044.000000 paquets
les noeuds ont drope 1389.000000 paquets NRTE
les noeuds ont drope 1604.000000 paquets IFQ
les noeuds ont drope 2051.000000 paquets COL

Là, nous observons sensiblement le même comportement que pour la topologie linéaire sans RTS/CTS, le nombre de collisions augmente, mais ici le changement est encore plus important, les paquets COL sont quasiment 10 fois plus nombreux!

On pourrait conclure sur une telle topologie que l'utilisation des RTS/CTS est primordiale dans le cadre de réseaux maillés, plus encore que dans un cadre linéaire.

Nous pouvons également observer deux autres topologies analogues à celle de 10 noeuds linéaire, nous ferons simplement varier la longueur.

Longueur de 6:



Les dropés en détail:

les noeuds ont drope 2731,000000 paquets NRTE
les noeuds ont drope 316.000000 paquets IFQ
les noeuds ont drope 1651.000000 paquets COL

Débits et délais:

CBR 512:

le delai moyen est de 0.996046 s
le debit est de 12316.603616 octets/s

CBR 1024:

le delai moyen est de 1.362084 s
le debit est de 14169.253370 octets/s

Pour la longueur de 14, il ne nous semble pas opportun de présenter de résultats car le comportement est le même pour toute topologie linéaire, qu'elle soit de 6, 10, 14, seules les valeurs changent sans pour autant permettre de tirer de conclusions supplémentaires.

3-Résultats

Nous pouvons d'ors et déjà tirer quelques conclusions sur le comportement des réseaux ad hoc avec les paramètres par défaut dans ns.

La gestion des retransmissions par la couche MAC avec les valeurs statiques communes à tous les noeuds entraîne un grand nombre de pertes dues aux echecs des retransmissions et au dépassement de ce nombre autorisé.

L'utilisation des RTS/CTS permet, dans des topologies linéaires de réduire les pertes par collision de moitié, ce qui est bien mais leur intérêt est encore plus frappant dans des topologies maillées où nous avons observé une réduction par 10 environ de ces pertes.

Nous nottons également que l'augmentation du débit crée une congestion dans le réseau quand la simulation avance, de là résulte un grand nombre de pertes dues au dépassement des files d'attente de paquets au niveau des noeuds.

Il est donc intéressant de chercher à résoudre ces différents problèmes afin d'améliorer la QoS dans des communications de type ad hoc. Pour ce faire, nous allons présenter une alternative à la gestion des retransmissions de la couche MAC issue d'une réflexion menée au LIA de l'IUP d'Avignon par notre tuteur, Ralph El Khoury. Nous n'aborderons pas la gestion de la fenetre de transmission de TCP ni les files d'attente.

Nous présenterons d'abord le principe de la solution, les étapes de son développement et de son implémentation pour aboutir aux tests du comportement du réseau en mettant en oeuvre ce mécanisme pour ensuite le comparer avec notre constat de l'existant en se plaçant dans un contexte similaire (topologie, RTS/CTS).

IV- Solution proposée

1-Principe général

Notre solution part du principe que plus on s'éloigne de la source, plus les chances de réussite de la transmission d'un paquet diminue. D'une part car les files d'attente sont remplies mais également car il y'a plus de chances de subir des collisions. Aussi, pour un noeud éloigné, il est couteux en terme de charge du réseau de redemander un paquet à la source lorsque le nombre de retransmissions vers la destination est excédé. Pour les noeuds voisins ou à 1 saut de la source, c'est moins problématique car le demande de retransmissions n'aura d'effet que sur des liens à 1 saut maximum contrairement à un noeud éloigné qui influera sur autant de lien qu'il est éloigné de la source.

Pour tenter d'y remédier, nous proposons d' adapter le nombre de retransmissions de la couche MAC en fonction de la position du noeud dans la chaine de communication. Ainsi, plus on s'éloigne de la source, plus on attribuera un nombre limite de retransmissions important.

Le calcul de ce nombre de retransmissions se fera en prenant en compte la topologie du réseau, par des informations de routage (couche 3) telles que le nombre de sauts vers la source et vers la destination, la position du noeud et les adresses source et destination. Ce calcul doit retourner une valeur de Kdynamique qui sera ensuite passée à la couche MAC (2) afin qu'elle adapte son nombre de retransmissions.

2-Développement et implémentation

Adaptation du nombre de retransmissions (K) de la couche MAC en fonction de la position du noeud dans la chaine de communication. Ainsi, on définira un nombre minimal de retransmissions (K_{min}) et un pas entre deux valeurs de K (K_{bis}).

Dans une topologie type d'un réseau linéaire de 10 noeuds, par exemple, on aura une évolution du nombre de retransmissions en fonction de la distance à la source et à la destination de chaque noeud.

Exemple:

10 noeuds et trafic entre noeud 1 et 9:

On choisit $K_{min}=2$, $K_{bis}=2$

N°noeud:	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0
Kdyn:	2	2	4	6	8	10	12	14	14

Les principales étapes de l'intégration de l'algorithme sont les suivantes:

- passage du code Matlab en code c++
- recherche de la position dans les classes du simulateur
- recherche des parametres nécessaires au calcul du K dynamique
- implémentation dans le code du simulateur
- compilation et lancement de ns pour vérifier

a- Matlab/c++

L'algorithme de calcul de K dynamique est passé de code Matlab en c++, voir annexes. On notera ici que la librairie math.h a été très utile car elle permet d'utiliser les mêmes fonctions que Matlab, seul un travail sur les types de variables et l'intégration dans une classe c++ ont nécessité un peu de recherche.

Une fois cet algorithme implémenté en c++ indépendamment du code du simulateur, restait à trouver sa place en vue de l'intégration dans le code du simulateur ns.

b- Intégration dans l'architecture de ns

Le mécanisme s'appuie sur des informations de la couche 3 du modèle OSI, on a besoin de données de routage telles que l'adresse de destination du paquet traité, de l'adresse source, de la position courante et des nombres de sauts à la source et à la destination.

Les adresses source et destination en c++ sont situées dans l'en tête des paquets.

Fichiers correspondants en c++: **common/packet.h** (pour connaître la structure et les méthodes d'accès).

Les informations de routage, correspondance entre adresses et nombre de sauts pour les atteindre sont dans la table de routage, il nous faut donc avoir accès aux structures de données de celle-ci.

Fichiers c++: **olsr/OLSR_rtable.h et .cc**

Nous travaillons avec le protocole de routage pro-actif OLSR, le mécanisme, pour récupérer et utiliser ces informations sera donc utilisé à partir de la classe OLSR.

Fichiers correspondants: **olsr/Olsr.h** et **olsr.cc**

Afin de faire tourner l'algo, deux valeurs doivent être passées à la fonction, il s'agit de Kinit (le K souhaité pour le noeud central) et Kbis (le pas entre deux valeurs de K). Elles doivent être définies dans un fichier tcl.

Fichier tcl: **tcl/lib/ns-default.tcl**

Nous avons souhaité rendre notre solution implémentable par tout autre utilisateur de ns qui voudrait utiliser la gestion des retransmissions sans que le simulateur soit irrémédiablement modifié. Pour qu'il fonctionne "normalement" par défaut ou utilise notre algorithme, nous avons défini une variable à modifier à partir du scripte de simulation tcl. Dans ns-default.tcl:

```
Mac/802_11 set Use_Kdyn_RetryLimit 0
```

On la met à un dans notre scripte de simulation tcl pour utiliser l'algo, reste à 0 par défaut et utilise les valeurs par défaut de ns (Short et Long RetryLimit).

Pour que les retransmissions dépendent de la valeur K que nous calculons, il faut à présent se placer au niveau de la couche MAC (niveau 2 du modèle OSI).

Le passage du paramètre entre couche 3 et 2 dans ns n'a pas été facile à effectuer car les différentes classes sont indépendantes. Toutefois, grâce à un aiguillage de la part de notre tuteur, nous avons vu que quelques années auparavant, un étudiant dont il était le tuteur travaillait sur un contrôle de puissance avec OLSR. En observant sa solution, nous avons remarqué qu'il passait une info entre les deux couches, et ce en entrant l'info sur la puissance calculée dans l'en tête du paquet. Nous avons décidé de tenter cette solution et cela nous permet effectivement de passer l'info de OLSR(niveau 3) à MAC(niveau 2).

Dès lors, restait à modifier les fonctions de retransmissions de la couche MAC afin qu'elles récupèrent cette valeur de K et qu'elles agissent en prenant notre solution en compte.

Fichiers c++: **mac/mac_802_11.h** et **.cc** et **common/packet.h** (pour le champ de l'en tête qu'on ajoute)

Il faut déclarer toute nouvelle structure de données que l'on crée dans un fichier en tête.

Fichier c++: **olsr/olsr_repositories.h**

Finalement, pour que l'intégration soit complète, on ajoute nos fichiers à compiler dans le Makefile de ns. Aussi, comme nous avons travaillé sur une deuxième installation de ns sur la même machine, nous avons ajouté l'emplacement des bibliothèques tcl dans ce même fichier afin de compiler seulement notre nouveau simulateur.

```
olsr/OLSR.o      olsr/OLSR_state.o      olsr/OLSR_rtable.o      olsr/OLSR_printer.o
olsr/OLSR_rtable_kdyn.o\
```

Compléments:

Nous avons créé une classe intermédiaire, qui s'inspire de la classe de la table de routage de OLSR et qui contient une structure correspondant à une table de routage adaptée à notre solution (position, @src,@dest, hop_src, hop_dest et kdyn).

Nous y avons également défini la fonction de calcul du K dynamique pour des raisons de praticité.

Cette fonction prend en paramètres:

- la longueur de la communication h
- la position du noeud dans la communication id
- le K initial pour le noeud central Kinit
- le pas entre deux valeurs de K Kbis.

Difficultés:

- Appréhension du code du simulateur: classes c++ et tcl, des structures de données (tables de routages, paquets, en tete des paquets, structures d'adresses...).
- Compréhension des paramètres utilisés, où les trouver et comment les exploiter (principalement infos de la table de routage, le nombre de sauts vers une destination).
- Trouver une place pour intégrer l'algorithme (en fonction des données exploitables selon la classe).
- Faire passer un paramètre entre deux couches (les classes de couches différentes n'ont pas de lien en code c++, il fallait trouver autre chose).

c- Implémentation

Préambule, les parties de code qui suivent servent à illustrer l'implémentation sans être complètes. Vous trouverez en annexe la totalité du code par fichier sur lequel nous avons travaillé.

La place de l'algorithme est donc dans le code du protocole OLSR. On définit une classe supplémentaire (possibilité de simplement ajouter la fonction à la classe de la table de routage de OLSR déjà existante), fichiers **OLSR_rtable_kdyn.h et .cc** où :

- dans un premier temps, nous avons pensé à dresser une table contenant toutes les informations utiles (adresses source et destination, nombre de sauts vers source et destination et Kdynamique) qui nous permettaient d'adapter le nombre de retransmissions en fonction de la longueur et de la position dans la communication.
- Dans cette classe, nous implémentons l'algorithme par la fonction:

```

/*****Parametres*****/
h, la longueur de la chaine de communication
id, la position du noeud dans ctte chaine
K, la valeur de Kinitial
Kbis, le pas entre deux valeurs de K
*****/
u_int32_t calc_kdyn(u_int32_t h, u_int32_t id, u_int32_t K, u_int32_t Kbis);

```

Pour l'utilisation de notre fonction, on se place dans le fichier **OLSR.cc**, dans la fonction `recv(Packet* p, Handler* h)` pour que le calcul du K dynamique se fasse pour chaque paquet et qu'on puisse ensuite le placer dans l'en tete.

```

/*****
Creation ou MAJ de la table kdyn
*****/
//Je vérifie si mon entrée existe
OLSR_rt_kdyn* chewam = rtable_kdyn_.lookup(ra_addr(),ih->saddr());
//Si la table existe, on rafraîchi sinon on ajoute

```

Cela s'est avéré inutile car, comme nous le montrerons par la suite, cette table n'était pas nécessaire. Par la suite, nous avons extrait les paramètres sans avoir besoin de cette table:

- les adresses à partir de l'en tete des paquets
- le nombre de sauts à partir de la table de routage déjà existante

La première version de calcul du K dynamique dans la classe OLSR a été effectuée avec une topologie supposée linéaire. Ainsi, comme nous n'arrivions pas encore à récupérer les informations complètes de routage, nous avons cherché une première solution pour mettre en oeuvre notre calcul:

- les adresses des noeuds de ns sont du type 0,1, 2 ... 10
- nous arrivions à lire les adresses source et destination dans l'en tete des paquets
- nous disposions également du numéro du noeud courant

Nous pouvions alors calculer le K dynamique de telle manière:

```

/*****Calcul de la longueur de la communication*****/
//ih->saddr(), l'adresse source du paquet
//ih->daddr(), l'adresse destination
u_int32_t dist_com=ih->saddr()+ih->daddr();
/*****
          Lancement de l'algo avec les parametres récupérés:
- dist_com, longueur communication
- ra_addr(), n° du noeud courant
- Kinit(), la valeur de Kinit définie dans ns-default.tcl
- Kbis(), valeur de Kbis dans ns-default.tcl
*****/
//u_int32_t kdyn = rtable_kdyn_.calc_kdyn(dist_com, ra_addr(),Kinit(), Kbis());

```

Après de plus amples recherches, nous avons réussi à extraire des informations plus complètes à partir de la table de routage et le code est devenu:

```

/*****
          Calcul de la distance
*****/
//lecture du nombre de hop dans la table routage
OLSR_rt_entry* entry = rtable_.find_hop(ih->saddr());
if(entry!=NULL){
    hop_src=entry->dist();
}
OLSR_rt_entry* entry1 = rtable_.find_hop(ih->daddr());
if(entry1!=NULL){
    hop_dest=entry1->dist();
}

//La longueur de la chaine est alors égale à:
u_int32_t dist_com=hop_src+hop_dest;

/*****
          Lancement de l'algo avec les parametres récupérés
- dist_com, longueur communication
- hop_src, position du noeud courant dans la communication
- Kinit(), la valeur de Kinit définie dans ns-default.tcl
*****/

```

```
- Kbis(), valeur de Kbis dans ns-default.tcl
*****/
u_int32_t kdyn=rtable_kdyn_.calc_kdyn(dist_com, hop_src,Kinit(), Kbis());
```

Pour conclure, nous plaçons la valeur de kdyn dans l'en tete du paquet, c'est ici que l'information est passée de la couche 3 (OLSR) vers la couche 2 (MAC):

```
ch->Pt_opti_ =kdyn;
//ch le pointeur sur l'en tete du paquet et Pt_Opti_ le champ réservé pour Kdyn.
```

Dans MAC, ce sont les fonctions **RetransmitRTS** et **RetransmitData** qui récupèrent cette valeur pour leur traitement. On se place alors dans le fichier **MAC_802_11.cc**, dans les fonctions correspondantes:

Dans RetransmitRTS:

```
//début de la fonction
u_int32_t retry_test;           //declaration de la variable qui dit si ns utilise notre
                                //algorithme ou pas

ssrc_ += 1;                      // STA Short Retry Count
hdr_cmn *ch = HDR_CMN(pktTx_);

//test pour savoir si on utilise kdyn ou pas (donné par le tcl)
if (macmib_.getUse_Kdyn_RetryLimit()==0){
    retry_test=macmib_.getShortRetryLimit();           //comportement par défaut
} else retry_test=ch->Pt_opti_;                         //sinon, le nombre de retransmissions
                                                         //devient égal à Pt_opti (Kdynamique)
if(ssrc_ >= retry_test) {                               //... routine de retransmission de MAC
```

Dans RetransmitData:

```
//... début de la fonction
if((u_int32_t) ch->size() <= macmib_.getRTSThreshold()) {
    rcount = &ssrc_;
    //test pour savoir si on utilise kdyn ou pas (donné par le tcl)
    if (macmib_.getUse_Kdyn_RetryLimit()==0){
        thresh=macmib_.getShortRetryLimit();           //par défaut
    } else thresh=ch->Pt_opti_;                         //avec Kdyn
    // cout<<"Test recup valeur en mac: "<<thresh<<"\n";
} else {
    rcount = &slrc_;
    if (macmib_.getUse_Kdyn_RetryLimit()==0){
        thresh=macmib_.getLongRetryLimit();
    } else thresh=ch->Pt_opti_;
    //cout<<"Test recup valeur en mac: "<<thresh<<"\n";
} //...routine de retransmission de MAC
```

Les autres ajouts ou modifications de code sont disponible en annexes, les morceaux de code ci-dessus sont les principaux, au centre de l'application que nous avons développé.

Afin de compléter l'implémentation, il faut recompiler ns en lançant la commande *make*, très peu de classes sont affectées par ces modifications, cette opération est donc assez rapide. Le compilateur donne les instructions en cas d'erreur, s'armer de patience est très important car nous avons modifié ou ajouté du code dans plusieurs fichiers et il nous a fallu un grand nombre de compilations...

Pour tester, bon nombre d'affichages sur la sortie standard ont été incorporés au code, pour vérifier que notre algorithme était bien utilisé, nous avons redirigé ces affichages vers un fichier.

```
#nsprojet TCL/Kdyn/simu_10_512.tcl >> test_10_512_kdyn.txt
```

Un exemple de résultat obtenu dans un tel fichier est:

```
noeud n°:0*****
Test calcul distance: hop_src:0 /hop_dest: 9 /longueur: 9
*****Test calcul Kdyn*****
H= 9 idnode ds chaine= 0 Kinit= 8 Kbis= 2
Kmax= 14
Kdyn dans pt_opti_noeud n°:0 =2
Kdyn calculé avec table routage: 2
noeud n°:7*****
Test calcul distance: hop_src:7 /hop_dest: 2 /longueur: 9
*****Test calcul Kdyn*****
H= 9 idnode ds chaine= 7 Kinit= 8 Kbis= 2
Kmax= 14
Kdyn dans pt_opti_noeud n°:7 =14
Kdyn calculé avec table routage: 14
```

Cela correspond à ce que nous espérions car la topologie est la même que lors de l'étude préalable avec le code Matlab.

L'implémentation est donc validée et permettra de réaliser les tests comparatifs avec les mêmes topologies que pour le constat de l'existant.

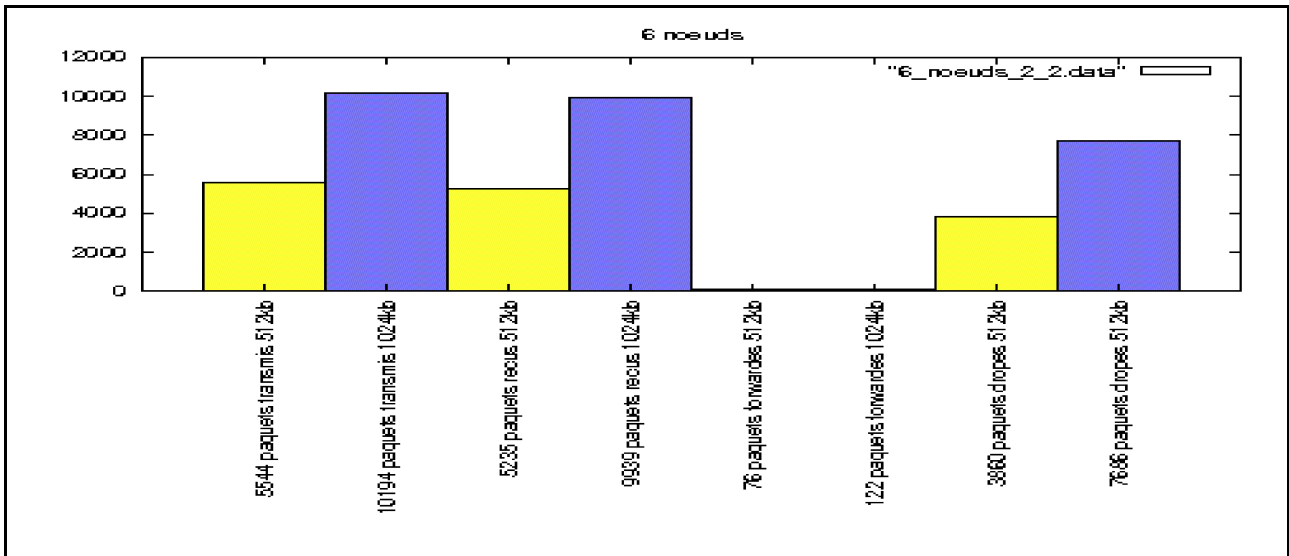
3-Simulations

Nous reprenons les topologies qui nous ont permis de dresser notre base de tests avec les valeurs par défaut.

Dans un premier temps, alors que la solution n'était pas encore implémentée, nous avons lancé une série de tests en faisant varier la valeur du nombre de retransmissions. C'était pour nous un bon moyen d'observer ce qu'il se passait quand le nombre de retransmissions limite pour chaque noeud augmentait ou diminuait. Toutefois, cette série de tests ne peut pas être considérée comme concluante car le K reste statique et commun à tous les noeuds du réseau, mais elle nous a permis de voir si le nombre des retransmissions avait un effet réel sur le comportement du réseau.

On commence avec la topologie linéaire de longueur 6:

Cas 1: Krts= 2 Kdata=2



On note qu'il y'a beaucoup de paquets perdus, environ 15%, voyons le détail:

les noeuds ont drope 3703.000000 paquets NRTE

les noeuds ont drope 3.000000 paquets IFQ

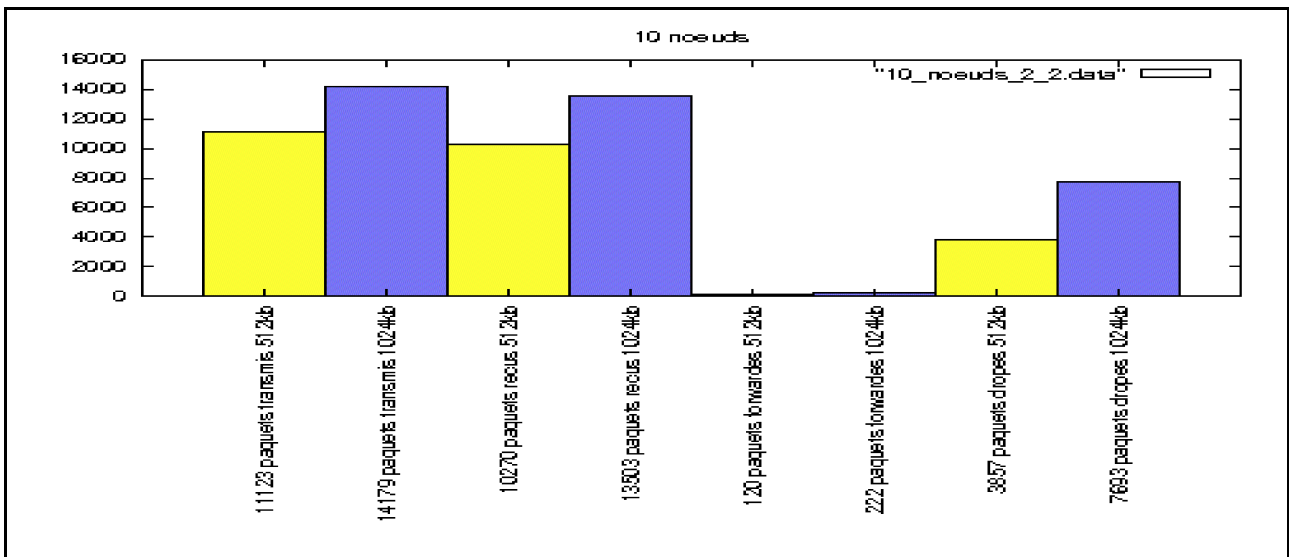
les noeuds ont drope 140.000000 paquets COL

Si l'on prend comme base de comparaison le graphique de la partie III, 2, on observe que le nombre de paquets perdus ne change pas significativement mais lorsque l'on s'intéresse au détail, les résultats sont plus parlant.

Ainsi pour la même topologie, en passant d'un nombre de retransmissions égal à 4/7 à 2/2, on obtient un nombre de pertes dues aux collisions divisé par plus de 10!

Nous pouvons d'ors et déjà prévoir qu'adapter le nombre des retransmissions en fonction de la longueur d'une topologie linéaire a un effet bénéfique sur le comportement du réseau en terme de limitation des pertes dues aux collisions.

Nous avons effectué le test sur une topologie de longueur 10 avec les mêmes valeurs de K:



Le détail pour le débit de 512:

les noeuds ont drope 3295.000000 paquets NRTE

les noeuds ont drope 1.000000 paquets IFQ

les noeuds ont drope 190.000000 paquets COL

Là encore, avoir modifié le nombre de retransmissions fait chuter le nombre de pertes dues aux collisions.

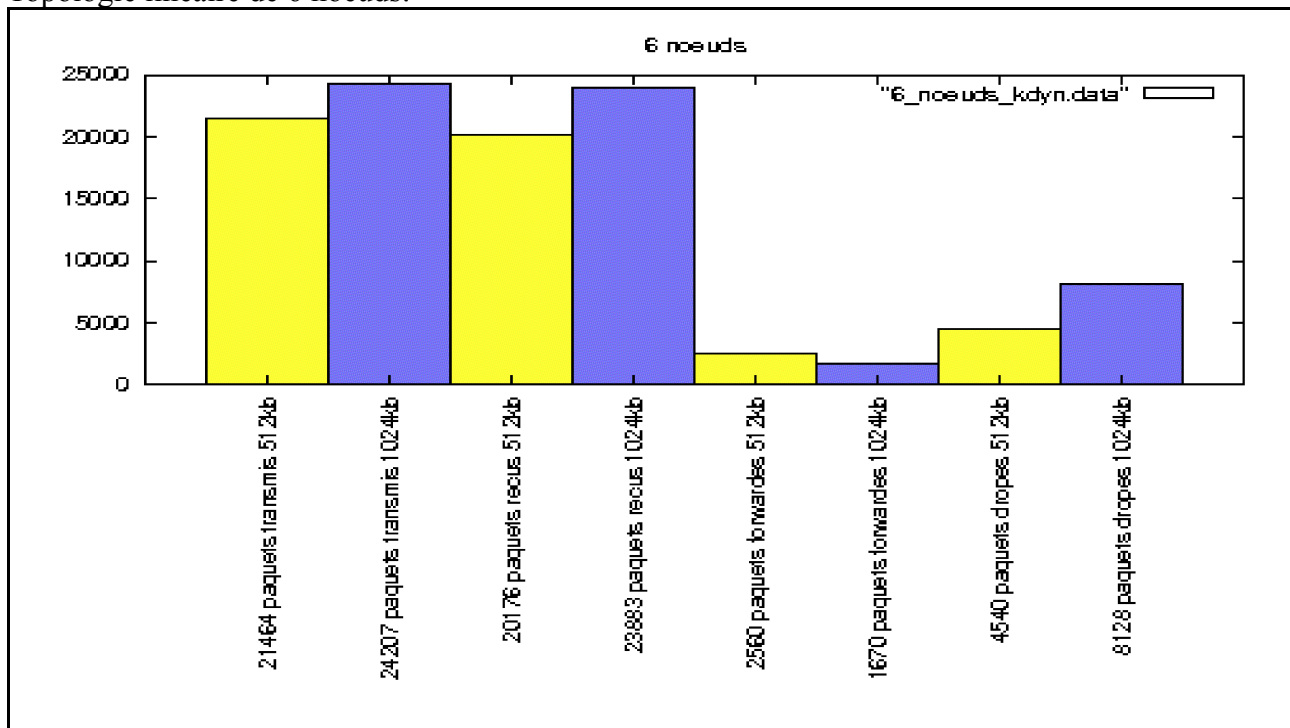
Nous avons réitéré avec une topologie de longueur 14 et nous avons observé les mêmes résultats à peu de choses près.

Par la suite, ces tests ont été repris avec un $K=8/16$, mais les résultats étaient tellement mauvais que nous ne les présenterons pas.

Passons maintenant aux simulations mettant en oeuvre le mécanisme de gestion des retransmissions dynamique en fonction de la position du noeud. Nous activons donc l'option permettant d'utiliser l'algorithme dans nos simulations tcl:

Mac/802_11 set Use_Kdyn_RetryLimit_ 1

Topologie linéaire de 6 noeuds:



Détail des paquets dropés:

les noeuds ont drope 2891.000000 paquets NRTE

les noeuds ont drope 180.000000 paquets IFQ

les noeuds ont drope 1469.000000 paquets COL

Débits et délais:

CBR 512 Kbps:

le delai moyen est de 0.807796 s

le debit est de 10774.154126 octets/s

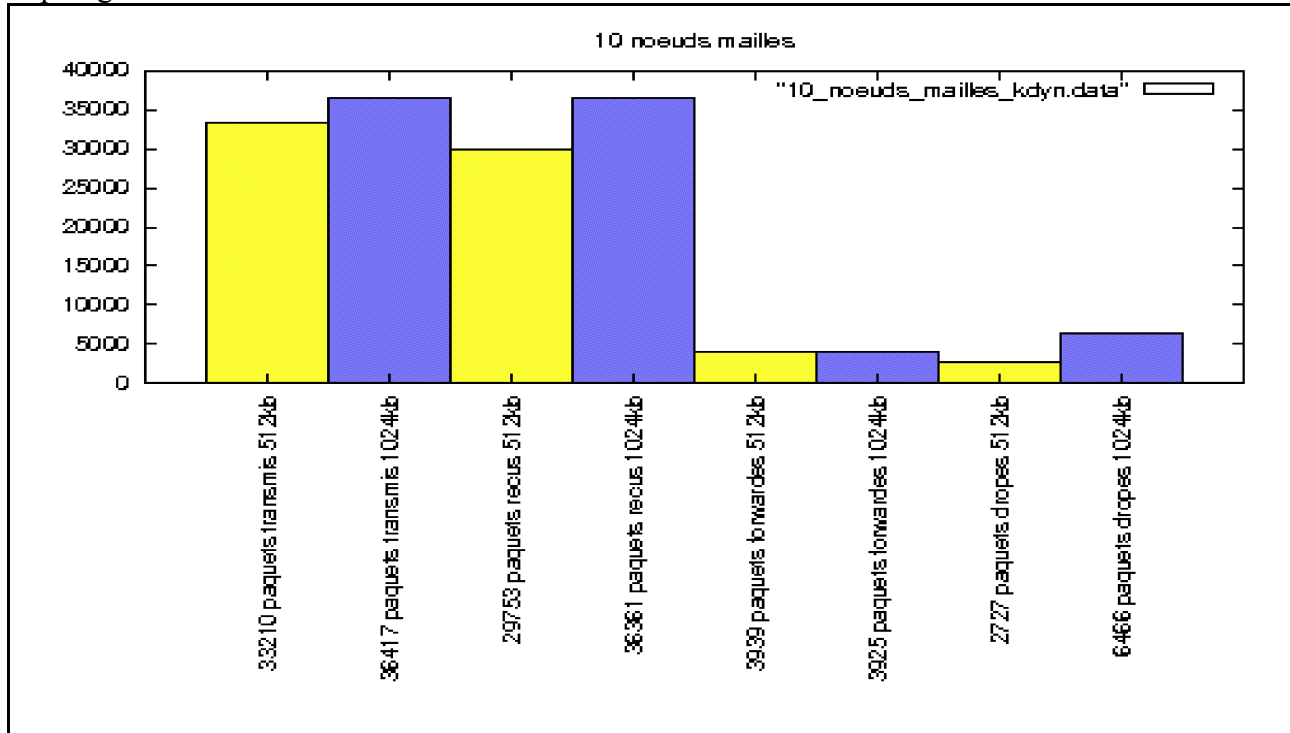
CBR 1024:

le delai moyen est de 0.943505 s

le debit est de 11603.322566 octets/s

Nous n'analyserons pas ici ces résultats, de même que pour la topologie suivante, cela fera l'objet de comparaisons entre les deux comportements dans une partie à venir.

Topologie étoile de 10 noeuds:



Détail des paquets dropés:

les noeuds ont drope 242.000000 paquets NRTE

les noeuds ont drope 2263.000000 paquets IFQ

les noeuds ont drope 222.000000 paquets COL

Débits et délais:

512 Kbps:

le delai moyen est de 1.781851 s

le debit est de 19822.452855 octets/s

1024 Kbps:

le delai moyen est de 1.847963 s

le debit est de 20293.788478 octets/s

Nous disposons à présent d'une base solide afin de dresser un comparatif entre notre solution et les mécanismes de gestion des retransmissions statiques vus dans la partie Constat.

V-Analyse & performances des résultats

1- Comparaison

TOPOLOGIE	Retransmissions	DEBITS CBR	DELAIS Moy	DEBITS	COLLISIONS
10 noeuds maillés	K=4/7	512 Kbps	1.745726 s	19523.3 o/s	254
10 noeuds maillés	K=4/7	1024 Kbps	1.909825 s	20245.7 o/s	183
6 noeuds linéaires	K=4/7	512 Kbps	0.996046 s	12316.6 o/s	1651
6 noeuds linéaires	K=4/7	1024 Kbps	1.362084 s	14169.2 o/s	1999
6 noeuds linéaires	Kdyn	512 Kbps	0.807796 s	10774.1 o/s	1469
6 noeuds linéaires	Kdyn	1024 Kbps	0.943505 s	11603.3 o/s	1063
10 noeuds maillés	Kdyn	512 Kbps	1.781851 s	19822.4 o/s	222
10 noeuds maillés	Kdyn	1024 Kbps	1.847963 s	20293.7 o/s	241

Ce tableau est un récapitulatif des résultats des simulations précédentes.

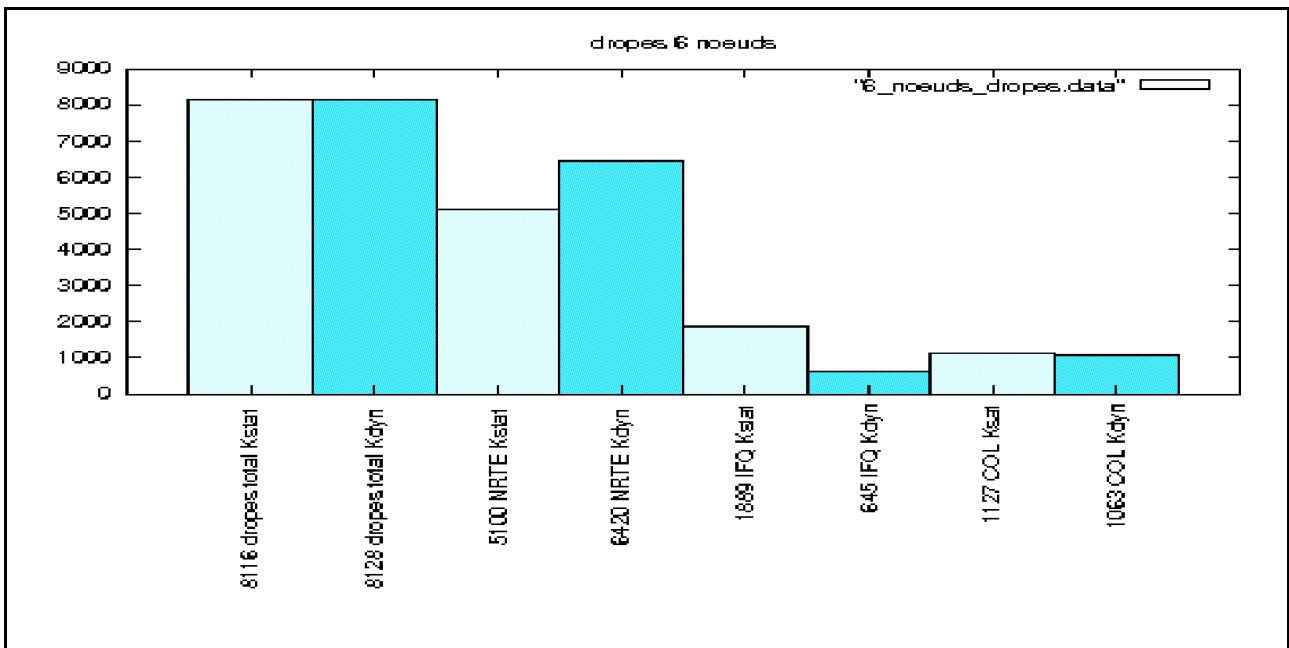
Ces résultats nous indiquent une légère amélioration au niveau du délai moyen de transmission des paquets CBR lorsque l'on utilise le K dynamique pour la gestion des retransmissions. Au niveau des débits, les valeurs sont similaires.

Nous pensons que parmi les raisons, les débits de CBR que nous avons choisis sont assez élevés pour de tels réseaux, toutefois ils correspondent à des cas réels de communication. Aussi, ils sont élevés en rapport avec la bande passante choisie.

Nous noterons également que certains résultats de simulations effectuées se sont avérés inexploitable car à cause de ce débit trop élevé, beaucoup de pertes sont constatées au niveau des paquets de contrôle du protocole OLSR. Il en résulte que les informations sur la topologie ont du mal à être récupérées par les noeuds et du coup beaucoup de pertes de type NoRouteEnable.

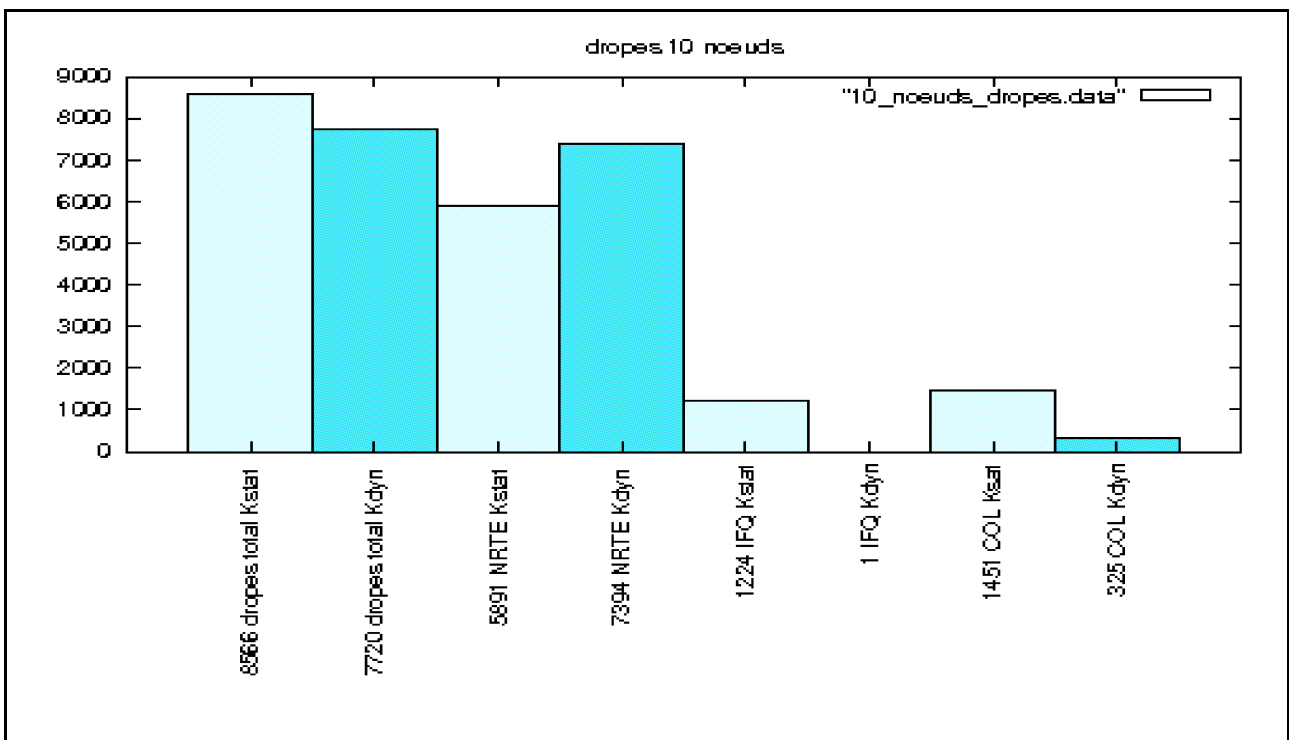
Bien que ces résultats ne soient pas probants et que notre analyse nécessite de reprendre une partie des simulations avec un débit CBR plus adapté à notre réseau, il existe un domaine sur lequel nous avons noté de réels progrès, il s'agit du nombre de collisions sur l'ensemble du réseau.

Les graphiques suivants vont nous permettre de démontrer l'apport en terme de QoS de notre solution dans le cadre de communications ad hoc mettant en oeuvre une gestion des retransmissions dynamique en fonction de la position dans la topologie.

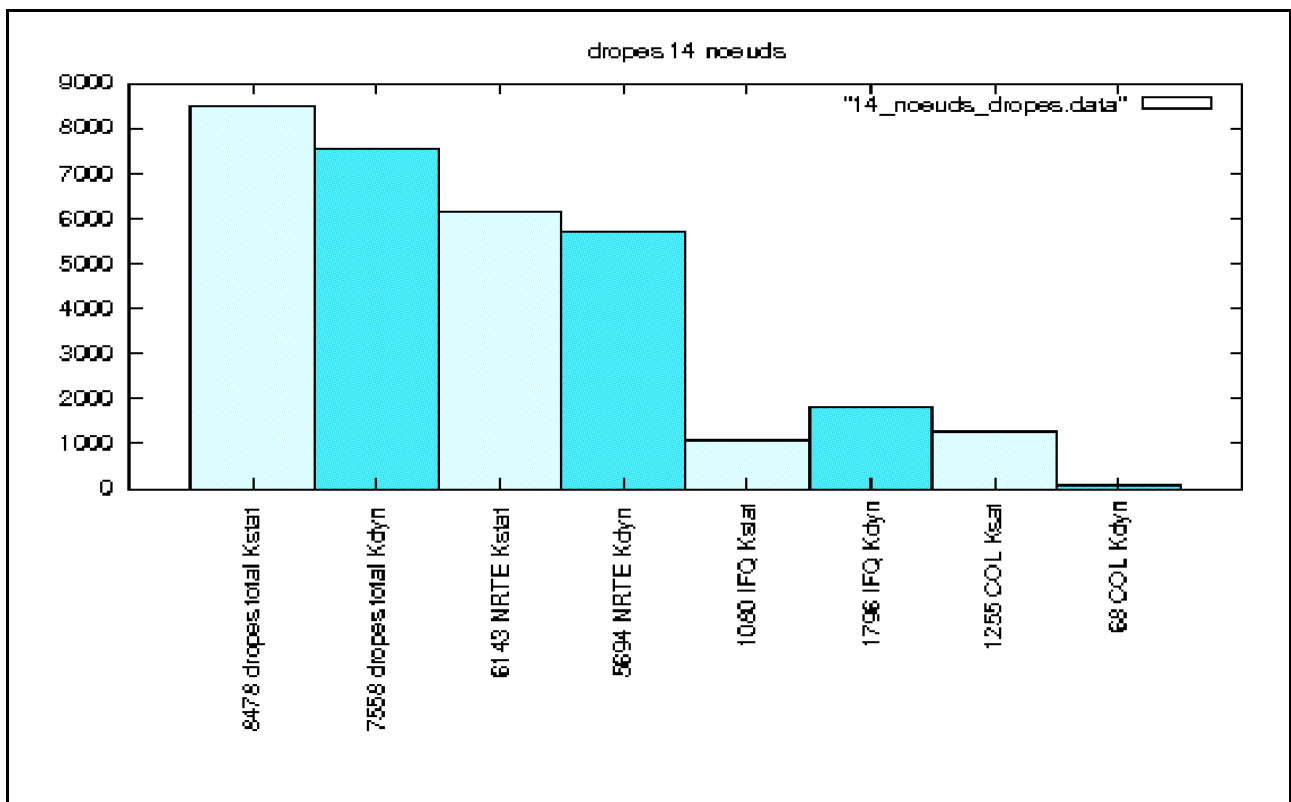
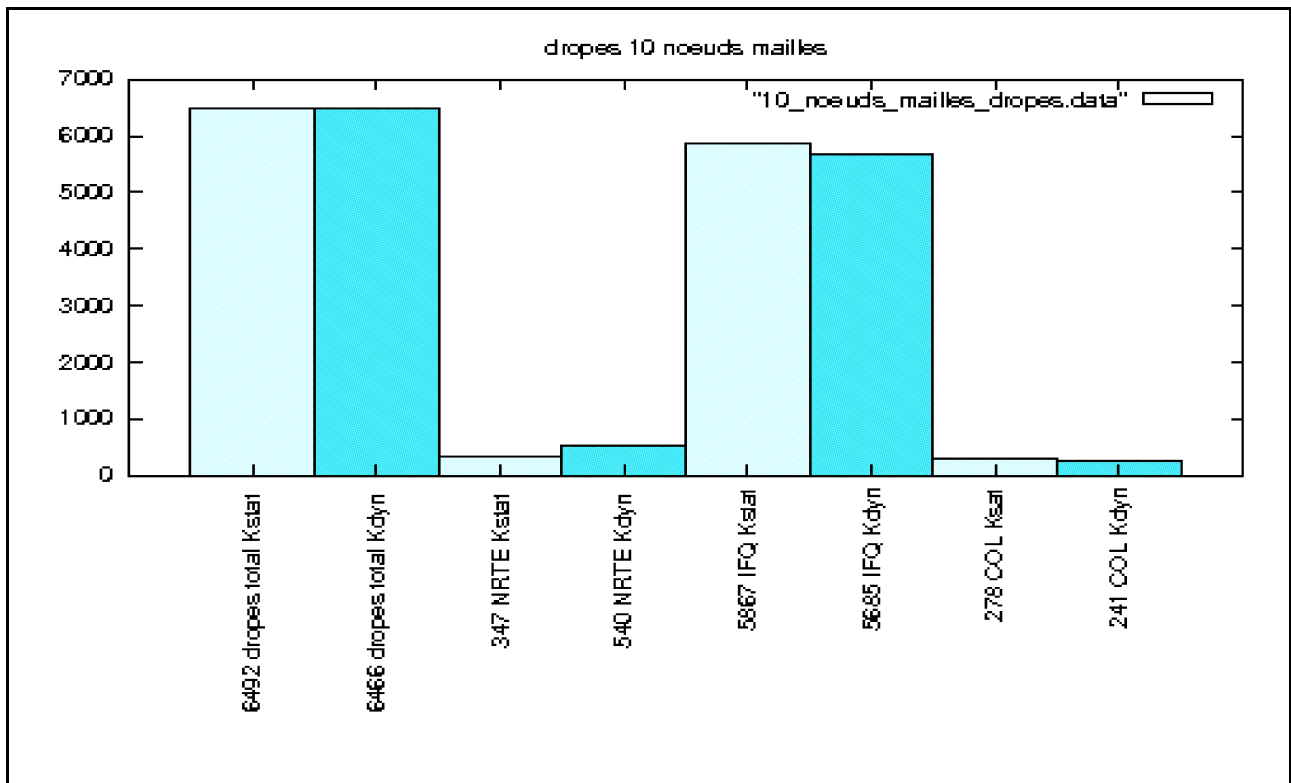


La topologie à 6 noeuds n'est pas le cadre idéal pour mettre en oeuvre notre solution. En effet, la communication a une longueur maximum de 5 sauts, quand on calcule notre K dynamique pour chaque noeud, l'algorithme nous retourne des valeurs autour de 8 (car 8 est la valeur définie par défaut pour le K du noeud central). Or la topologie exige des valeurs de K faibles car les noeuds source et destination sont assez proches. Il n'est pas très couteux de redemander un paquet et donc il ne semble pas utile d'utiliser un K dynamique.

On remarque quand même une amélioration dans les pertes dues aux collisions.



C'est à partir de topologies de longueur 10 que les résultats sont flagrants. Dans le graphe ci-dessus, on peut noter que le nombre de collisions a été divisé quasiment par 5 en utilisant le K dynamique.

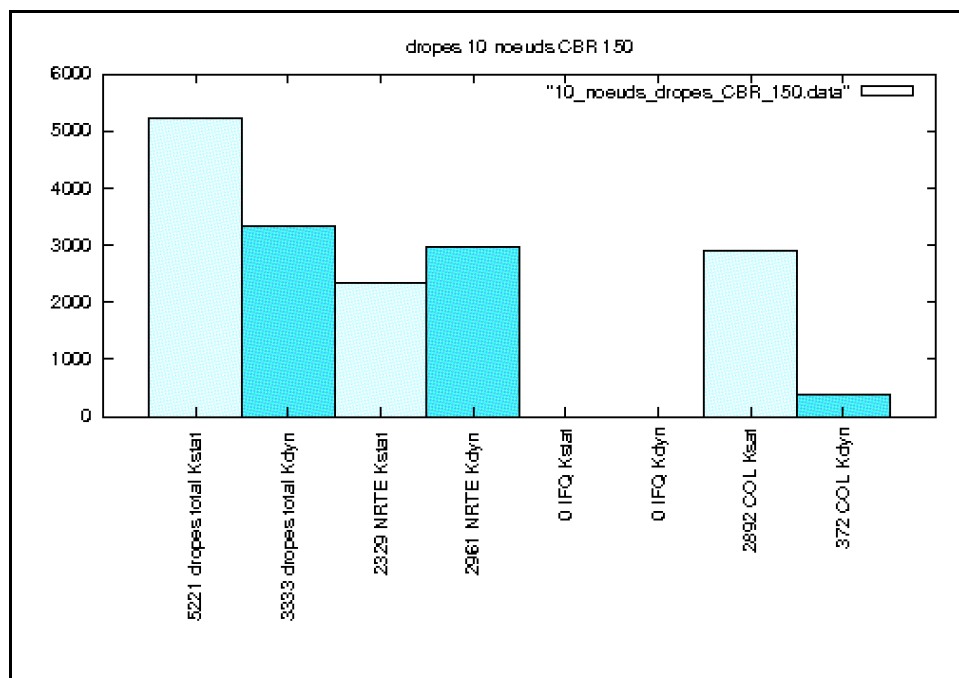


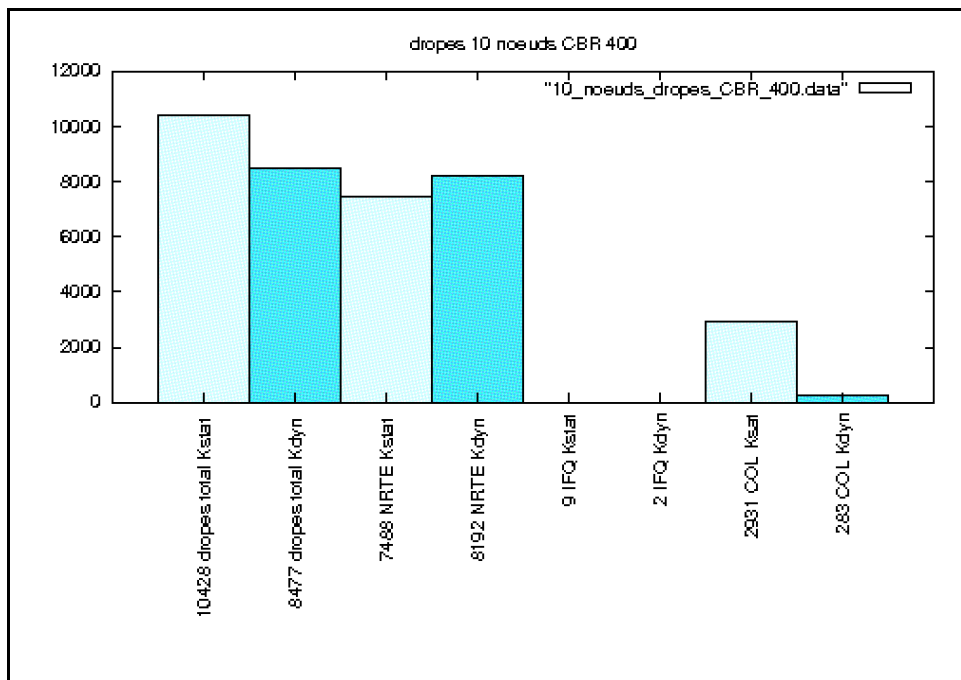
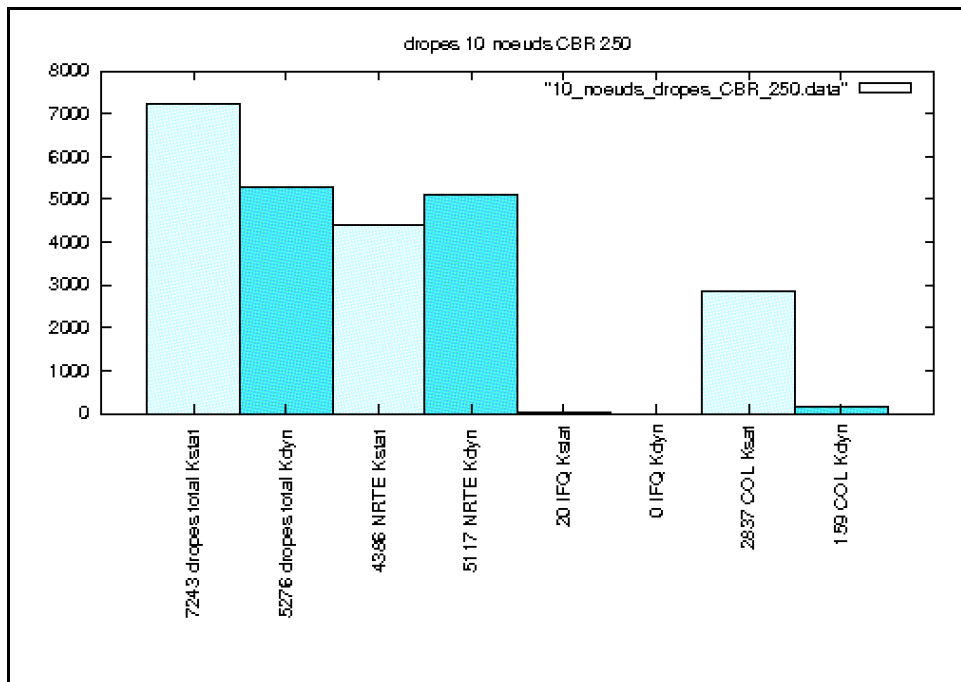
Cette simulation présente les résultats les plus spectaculaires au niveau de la réduction des collisions.

Nous concluons cette partie des simulations par le fait que l'implémentation de l'algorithme de gestion des retransmissions permet de réduire significativement le nombre de pertes dues aux collisions. Toutefois, le protocole OLSR nécessitant la connaissance de la topologie complète du réseau pour établir la communication, ajouté au débit important de notre trafic CBR font que beaucoup de paquets sont dropés à cause du manque de connaissance de la table.

Par conséquent, après conseil de notre tuteur nous avons baissé le débit du CBR en prenant 3 valeurs: 150Kbps, 250 Kbps, 400 Kbps. Nous avons simulé un réseau de 10 noeuds linéaire. Les résultats montrent une net augmentation des performances de l'algorithme de retransmission dynamique, mais cependant les paquets dropés a cause du NoRouteEnable reste important cela est du, comme il a été dit dans le paragraphe précédent, au manque de connaissance de la table de routage de OLSR.

Les histogrammes suivants montrent les différentes causes des paquets dropés(NRTE, IFQ, COL) pour les différentes simulations réalisées.





2- Evaluation des performances de notre solution

Notre étude touche à son terme. Nous avons essayer de faire apparaître les performance de notre solution de gestion des retransmissions en vue de réduire la congestion, de réduire les pertes tout en améliorant d'autres paramètres de QoS tels que les débits et délais.

Au niveau des pertes, il est indéniable que l'utilisation de notre mécanisme diminue sensiblement le nombre de pertes dues aux collisions. Pour ce qui est des autres drops, il faut chercher la faute au niveau de la façon dont OLSR route les paquets.

C'est un premier point qui nous permet d'affirmer que cette solution a un effet bénéfique dans le comportement de notre réseau.

Pour ce qui est de la congestion, on ne conclura pas en disant qu'on a amélioré quelque chose car le nombre total de pertes et plus particulièrement les files d'attente pleines ne bouge pratiquement pas avec le K dynamique.

Enfin, les délais et débits mesurés ne permettent pas non plus de dire si une amélioration se fait sentir, nous obtenons bien sûr des délais moyens inférieurs (et même des fois très largement) aux délais dans le cas du K statique. Cependant, les débits observés relèvent quasiment du surnaturel car en allongeant la durée de la simulation, en réduisant les débits du CBR jusqu'à 150 Kbps, on n'observe que quelques dizaines de paquets à la destination alors que des milliers ont été envoyés, cela se traduisait par des débits de quelques centaines d'octets pour notre communication!

VI- Contraintes & Difficultés

1-Contraintes

a- Matérielles et logicielles

Pour un avancement raisonnable de notre projet nous avons utilisé 2 machines Linux (Debian Gentoo). A l'IUP nous n'avons pas les droits pour les modifications du logiciel NS-2, nous avons donc effectué toutes les installations (Linux, NS-2,Gnuplot...) sur nos machines personnelles. Ensuite l'intégration du protocole OLSR a été réalisée sur les 2 postes qui ont chacun une version de NS-2 différente (NS-2.29 et NS-2.30). De plus pour l'algorithme de retransmission, les classes de NS-2 ont été modifiées, ces deux dernières étapes ont nécessité la recompilation du logiciel de simulation.

c-L'équipe de projet

Tout d'abord l'intégration au sein de l'équipe n'a pas posé de réels problèmes, la dynamique de groupe est venue naturellement.

Les problèmes sont apparus lors des rendez-vous pour mettre en commun les fruits de nos travaux, il a été difficile de trouver des créneaux où chacun d'entre nous, était disponible.

2- Difficultés

L' intégration de l'algo a été bien plus complexe que nous l'avions prévu dans le cahier des charges. D'abord à cause de la complexité de l'architecture et du code du simulateur mais également car il a été très difficile de faire passer le K dynamique de la couche 3 à la couche 2.

Nous noterons ici que nous avons eu du mal à démontrer les bonnes performances de notre solution à cause de résultats de simulations inexploitable.

L'analyse des résultats est incomplète et surtout nous avons eu beaucoup de mal à obtenir des résultats satisfaisant permettant de tirer des conclusions précises du comportement du réseau.

VII- Bilan

Afin de cloturer cette étude, nous allons dresser un bilan de tout ce qu'a comporté le déroulement de ce projet en faisant le lien avec les objectifs que nous nous étions fixé lors de la rédaction du Cahier des Charges.

En premier lieu, nous avons validé les points théoriques. Nous pouvons dire que nous avons une bien meilleure connaissance du monde des réseaux ad hoc, des méthodes d'accès au canal de 802.11, des protocoles de routage (pro actifs, OLSR) ainsi que des mécanismes de gestion des retransmissions de la couche MAC.

A propos de l'environnement logiciel, les installations des outils, OS, ns et son patch OLSR ont été effectuées avec succès.

Pour ce qui est des réalisations techniques de ce projet, nous soulignerons la réussite de l'implémentation de l'algorithme de gestion des retransmissions, de son codage en c++ à partir d'un code Matlab à son intégration dans le code du simulateur.

Enfin, nous terminerons en mettant en avant les bons résultats obtenus sur la réduction des pertes dues aux collisions.

Suites à donner au projet:

Nous sortons un peu frustrés de cette étude car nous aurions aimé apporter plus d'éléments permettant de complètement valider les performances de notre solution.

Une étude portée sur les débits et les délais pourrait être un très bon complément.

Annexes

Sources de documentation:

<http://hipercom.inria.fr/olsr/documentation.html>

<http://ftp.isi.edu/nsnam/dist/>

<http://masimum.dif.um.es/um-olsr/html/>

<http://www-sop.inria.fr/mistral/personnel/Eitan.Altman/ns.htm>

<http://www.isi.edu/nsnam/ns/ns-build.html>

Scriptes de simulation tcl:

simu_lineaire_10_cbr250.tcl

#

=====
=====

Define options

#

=====
=====

```

set opt(chan)      Channel/WirelessChannel      ;# channel type
set opt(prop)      Propagation/TwoRayGround     ;# radio-propagation model
set opt(netif)     Phy/WirelessPhy             ;# network interface type
set opt(mac)       Mac/802_11                  ;# MAC type
set opt(ifq)       Queue/DropTail/PriQueue     ;# interface queue type
set opt(ll)        LL                          ;# link layer type
set opt(ant)       Antenna/OmniAntenna         ;# antenna model
set opt(ifqlen)    50                          ;# max packet in ifq
set opt(nn)        10                          ;# number of mobilenodes
set opt(adhocRouting) OLSR                    ;# routing protocol

set opt(x)         1770                        ;# x coordinate of topology
set opt(y)         500                         ;# y coordinate of topology
set opt(stop)      100                        ;# time to stop simulation

```

```

set opt(cbr-start) 10.0

```

```

set opt(cbr-stop)  95.0

```

#

=====
=====

```

#Mac/802_11 set ShortRetryLimit_ 7 ;# retransmissions

```

```

#Mac/802_11 set LongRetryLimit_ 4 ;# retransmissions

```

```

Mac/802_11 set Use_Kdyn_RetryLimit_ 1 ;# on utilise l'algo de gestion des
;#retransmissions (0 sinon)

```

```

set ns_ [new Simulator ]

Phy/WirelessPhy set bandwidth_ 1.5e9

set tracefd [open simu_10_250.tr w]
set namtrace [open simu_10_250.nam w]
$ns_ trace-all $tracefd
$ns_ namtrace-all-wireless $namtrace $opt(x) $opt(y)

set topo [new Topography]

$topo load_flatgrid $opt(x) $opt(y)

create-god $opt(nn)

# configure mobile nodes
$ns_ node-config -adhocRouting $opt(adhocRouting) \
    -llType $opt(ll) \
    -macType $opt(mac) \
    -ifqType $opt(ifq) \
    -ifqLen $opt(ifqlen) \
    -antType $opt(ant) \
    -propType $opt(prop) \
    -phyType $opt(netif) \
    -channelType $opt(chan) \
    -topoInstance $topo \
    -wiredRouting OFF \
    -agentTrace ON \
    -routerTrace ON \
    -macTrace ON

for {set i 0} {$i < $opt(nn)} {incr i} {
    set node_($i) [$ns_ node]
}

# positions
#
$node_(0) set X_ 20
$node_(0) set Y_ 250
$node_(0) set Z_ 0.0
$node_(1) set X_ 220
$node_(1) set Y_ 250
$node_(1) set Z_ 0.0
$node_(2) set X_ 420
$node_(2) set Y_ 250
$node_(2) set Z_ 0.0
$node_(3) set X_ 620
$node_(3) set Y_ 250
$node_(3) set Z_ 0.0
$node_(4) set X_ 800

```

```

$node_(4) set Y_ 250
$node_(4) set Z_ 0.0
$node_(5) set X_ 980
$node_(5) set Y_ 250
$node_(5) set Z_ 0.0
$node_(6) set X_ 1100
$node_(6) set Y_ 250
$node_(6) set Z_ 0.0
$node_(7) set X_ 1280
$node_(7) set Y_ 250
$node_(7) set Z_ 0.0
$node_(8) set X_ 1460
$node_(8) set Y_ 250
$node_(8) set Z_ 0.0
$node_(9) set X_ 1640
$node_(9) set Y_ 250
$node_(9) set Z_ 0.0
# setup UDP connection
#
set udp [new Agent/UDP]
set null [new Agent/Null]
$ns_ attach-agent $node_(0) $udp
$ns_ attach-agent $node_(9) $null
$ns_ connect $udp $null
set cbr [new Application/Traffic/CBR]
$cbr set packetSize_ 512
$cbr set rate_ 250Kb
$cbr set app_fid_ 1

$cbr attach-agent $udp

$ns_ at 0.0 "$node_(0) label emetteur_cbr"
$ns_ at 0.0 "$node_(9) label recepteur_cbr"

#start & ends traffics
$ns_ at $opt(cbr-start) "$cbr start"

$ns_ at $opt(cbr-stop) "$cbr stop"

$ns_ at 10.0 "[$node_(0) agent 255] print_rtable"
$ns_ at 15.0 "[$node_(0) agent 255] print_linkset"
$ns_ at 20.0 "[$node_(0) agent 255] print_nbset"
$ns_ at 25.0 "[$node_(0) agent 255] print_nb2hopset"
$ns_ at 30.0 "[$node_(0) agent 255] print_mprset"
$ns_ at 35.0 "[$node_(0) agent 255] print_mprselset"
$ns_ at 40.0 "[$node_(0) agent 255] print_topologyset"#

for {set i 0} {$i < $opt(nn)} {incr i} {
    $ns_ initial_node_pos $node_($i) 20
}

# tell all nodes when the simulation ends

```

```
#
for {set i 0} {$i < $opt(nn)} {incr i} {
    $ns_ at $opt(stop).0 "$node_($i) reset";
}

$ns_ at $opt(stop).0002 "puts \"NS EXITING...\" ; $ns_ halt"
$ns_ at $opt(stop).0001 "stop"

#define color index
$ns_ color 0 blue
$ns_ color 1 red

proc stop {} {
    global ns_ tracefd namtrace
    $ns_ flush-trace
    close $tracefd
    close $namtrace}
# begin simulation
#
puts "Starting Simulation..."

$ns_ run
```

Scriptes awk d'exploitation des traces:**paquets.awk**

```

BEGIN {
  nombres_de_paquets_transmis = 0 ;
  nombres_de_paquets_recus = 0 ;
  nombres_de_paquets_forwardes = 0 ;
  nombres_de_paquets_dropes = 0 ;
  nombres_de_paquets_dropes_NRTE = 0;
  nombres_de_paquets_dropes_IFQ = 0;
  nombres_de_paquets_dropes_COL = 0;
}
{
  action = $1;
  temps = $2;
  src = $3;
  niveau = $4;
  numpaquet = $6;
  typetraffic = $7
  taillepaquet = $8;
  src_address = $9;
  dst_address = $11;

  if (action == "r"){
    nombres_de_paquets_transmis = nombres_de_paquets_transmis + 1 ;
  }

  if (action == "s"){
    nombres_de_paquets_recus = nombres_de_paquets_recus + 1 ;
  }

  if (action == "f"){
    nombres_de_paquets_forwardes = nombres_de_paquets_forwardes + 1 ;
  }

  if (action == "D"){
    nombres_de_paquets_dropes = nombres_de_paquets_dropes + 1 ;

    if ($5=="NRTE"){
      nombres_de_paquets_dropes_NRTE++;
    }
    else if($4=="IFQ") {
      nombres_de_paquets_dropes_IFQ++;
    }
  }
}
}
END {
  printf "les noeuds ont transmis %f paquets\n", nombres_de_paquets_transmis;
  printf "les noeuds ont recu %f paquets\n", nombres_de_paquets_recus;
  printf "les noeuds ont forwardes %f paquets\n", nombres_de_paquets_forwardes;
  printf "les noeuds ont dropes %f paquets\n", nombres_de_paquets_dropes;
}

```

```

printf "les noeuds ont drope %f paquets NRTE\n", nombres_de_paquets_dropes_NRTE;
printf "les noeuds ont drope %f paquets IFQ\n", nombres_de_paquets_dropes_IFQ;
printf "les noeuds ont drope %f paquets COL\n", (nombres_de_paquets_dropes-
(nombres_de_paquets_dropes_NRTE+nombres_de_paquets_dropes_IFQ));
}

```

delai.awk

```

BEGIN {
time=0;
l=94;
z=4053;
n=z-1;
while(j<z){ timeS[j]=0; timeR[j]=0 ; j++;};
}
{
for (j=l ; j<z ; j++){
if ($3 == "_0_" && $1 == "s" && $7 == "cbr" && $6 == j && $4 == "AGT"){
timeS[j]=$2;
m++;
}
if ($3 == "_9_" && $1 == "r" && $7 == "cbr" && $6 == j && $4 == "AGT"){
timeR[j]=$2;
k++;
}
}
}
}
END {
for (j=l ; j<z ; j++){
if(timeS[j] != 0 && timeR[j] != 0){
time=time + (timeR[j]-timeS[j]);
}
}
while (timeR[n] ==0){
n--;
}
printf "\nDernier paquet CBR reçu et premier envoyé:\n"
printf "timeR[%d]\t %f\n",z-1,timeR[n];
printf "timeS[%d]\t %f\n",l,timeS[l];

printf"\n\nnombre de paquets cbr envoyes\t %d\n",m;
printf"nombre de paquets cbr recus\t %d\n",k;
printf"nombre de paquets cbr perdu\t %d\n",m-k;
printf "\n\nle delai moyen est de %f\n",time/k;
printf "\nle debit est de %f octets/s\n",(k*512)/(timeR[n]-timeS[l]);
}

```

Scriptes de tracés pour gnuplot:**paquets.gp**

```
set title "14 noeuds"  
set xtics (" " 0, "20736 paquets transmis 512kb" 1, "27526 paquets transmis 1024kb" 2, "18831  
paquets recus 512kb" 3, "27681 paquets recus 1024kb" 4, "581 paquets forwardes 512kb" 5, "272  
paquets forwardes 1024kb" 6, "3806 paquets dropes 512kb" 7, "7558 paquets dropes 1024kb" 8, ""  
9)  
set xtics rotate by 90  
set style data boxes  
set terminal postscript eps  
set output '14_noeuds_Kdyn.eps'  
plot "14_noeuds_kdyn.data"
```

dropes.gp

```
set title "dropes 10 noeuds CBR 150"  
set xtics (" " 0, "5221 dropes total Kstat " 1, "3333 dropes total Kdyn " 2, "2329 NRTE Kstat " 3,  
"2961 NRTE Kdyn " 4, "0 IFQ Kstat" 5, "0 IFQ Kdyn" 6, "2892 COL Ksat" 7, "372 COL Kdyn" 8,  
"" 9)  
set xtics rotate by 90  
set style data boxes  
set terminal postscript eps  
set output '10_noeuds_dropes_CBR_150.eps'  
plot "10_noeuds_dropes_CBR_150.data"
```

Codes c++ de l'implémentation:**common/packet.h**

```

struct hdr_cmn {

    //code original

    /*****
    ajout du champ kdyn dans le common header du paquet
    *****/
    u_int32_t Pt_opti_;           //kdyn optimal a utiliser
    inline u_int32_t& opt() { return (Pt_opti_); }

} //fin struct hdr_cmn

```

olsr/OLSR_rtable_kdyn.h

```

#ifndef __OLSR_rtable_kdyn_h__
#define __OLSR_rtable_kdyn_h__
#include <olsr/OLSR_repositories.h>
#include <trace.h>
#include <map>
///
/// \brief Defines rtable_t as a map of OLSR_rt_entry, whose key is the destination address.
///
/// The routing table is thus defined as pairs: [dest address, entry]. Each element
/// of the pair can be accesed via "first" and "second" members.

typedef std::map<nsaddr_t, OLSR_rt_kdyn*> rtable_kdyn_t;

/*****
Classe table_Kdyn, notre table de routage avec les infos suivantes:
(@src / @dest / node_id / nb_sauts_src / nb_sauts_dest / Kdyn
*****/

class OLSR_rtable_kdyn {
rtable_kdyn_t pt_;           ///Data structure for the kdyn table
public:
    OLSR_rtable_kdyn();
    ~OLSR_rtable_kdyn();

    void clear();
    void rm_kdyn(nsaddr_t src);
    OLSR_rt_kdyn* add_kdyn(nsaddr_t node_id, nsaddr_t src, nsaddr_t dest, u_int32_t
        dist_src, u_int32_t dist_dest, u_int32_t Kdyn);
    OLSR_rt_kdyn* lookup(nsaddr_t node_id, nsaddr_t src);
    u_int32_t size();
    void print(Trace*);

```

```
//Notre fonction de calcul du K dynamique
u_int32_t calc_kdyn(u_int32_t h, u_int32_t id, u_int32_t K, u_int32_t Kbis);

};
#endif
```

olsr/OLSR_rtable_kdyn.cc

Ici, seul le code utile est implémenté

```
/******algo de calcul de kdyn******/

u_int32_t OLSR_rtable_kdyn::calc_kdyn(u_int32_t h, u_int32_t id, u_int32_t K, u_int32_t
Kbis){
    //h l'absc max, id la position du noeud, K la valeur de K initiale, Kbis le pas entre
    //deux valeurs de Kdyn
    u_int32_t jmin, jmax, Kdyn, xm, jinit=3, xinit=2, Kinit=K, Kmax, xmax;

    //verification de la parité de h (
    if ((h%2)==0){ //si le nb de noeuds est pair
        xm=h/2;

        if ((K-(xm-1)*Kbis) > Kbis){
            xinit=1;
            jinit=xm-xinit;
        }else { //on cherche le rayon jinit de Kdyn
            jmin=ceil(K/Kbis)-1; //l'entier superieur ou egal
            jmax=floor((K-1)/Kbis); //l'entier inf?rieur ou egal

            if (jmin==jmax)
                jinit=jmin;
            //else //le jinit n'est pas bon
            xinit = xm-jinit;
        }

        Kinit=K-jinit*Kbis;
        xmax=xmax+1+jinit;
        Kmax=K+jinit*Kbis;
        //cout<<"Kmax= "<<Kmax<<"\n";

        if (id<xinit)
            Kdyn=Kinit;
        else if ((id>=xinit) && (id<xm))
            Kdyn=K-(xm-id)*Kbis;
        else if ((id==xm) || (id==xm+1))
            Kdyn=K;
        else if ((id>xm+1) && (id<=xmax))
            Kdyn=K+(id-xm-1)*Kbis;
        else if ((id>xmax) && (id<=h)){
```

```

        Kdyn=Kmax;
    }
    //else erreur car id d?passe le nb de hop
}else {
    xm=(u_int32_t)(h/2);
    //si le nb de noeuds n'est pas pair
    //l'entier au dessus

    if ((K-(xm-1)*Kbis) > Kbis){
        xinit=1;
        jinit=xm-xinit;
    }else {
        //on cherche le rayon jinit de Kdyn
        jmin=ceil(K/Kbis)-1;
        //l'entier superieur ou egal
        jmax=floor((K-1)/Kbis);
        //l'entier inferieur ou egal

        if (jmin==jmax)
            jinit=jmin;
        //else //le jinit n'est pas bon
        xinit = xm-jinit;
    }

    Kinit=K-jinit*Kbis;
    xmax=xm+jinit;
    Kmax=K+jinit*Kbis;

    if (id<xinit)
        Kdyn=Kinit;
    else if ((id>=xinit) && (id<xm))
        Kdyn=K-(xm-id)*Kbis;
    else if (id==xm)
        Kdyn=K;
    else if ((id>xm) && (id<=xmax))
        Kdyn=K+(id-xm)*Kbis;
    else if ((id>xmax) && (id<=h))
        Kdyn=Kmax;
    //else erreur car id dépasse le nb de hop
}
return Kdyn;
} // fin fonction calc_kdyn

```

tcl/lib/ns-default.tcl

Ici, ce sont des ajouts:

```

#Instauration des variables Kinit et Kbis pour le Kdynamique
Agent/OLSR set Kinit_ 8
Agent/OLSR set Kbis_ 2

#variable pour utiliser l'algo ou non
Mac/802_11 set Use_Kdyn_RetryLimit_ 0; # retransmissions dynamiques 0/non,
# 1/oui

```

olsr/olsr_repositories.h

```

/*****
                                     Structure de la table kdyn
*****/

// An OLSR's kdyn table entry.
typedef struct OLSR_rt_kdyn {
    nsaddr_t    node_id_;
    nsaddr_t    src_addr_;
    nsaddr_t    dest_addr_; ///< Address of the next hop.
    u_int32_t   hop_src_;
    u_int32_t   hop_dest_;
    u_int32_t   kdyn_;      ///< kdyn optimal
    inline nsaddr_t& node_id() { return node_id_; }
    inline nsaddr_t& src_addr() { return src_addr_; }
    inline nsaddr_t& dest_addr() { return dest_addr_; }
    inline u_int32_t& hop_src() { return hop_src_; }
    inline u_int32_t& hop_dest() { return hop_dest_; }
    inline u_int32_t& kdynOpt() { return kdyn_; }
} OLSR_rt_kdyn;

```