2008

UFR Ingénieurs 2000

Vivien Boistuaud
Julien Herr

TP DE MODELISATION DE RESEAUX AVEC LE LOGICIEL NS2

Ce document a été réalisé par V. Boistuaud et J. Herr dans le cadre des travaux pratiques de Modélisation des Réseaux coordonnés par Hakim Badis, au sein de l'UFR Ingénieurs 2000 de l'Université de Marne-la-vallée.

Table des matières

Introduction

Dans le cadre de nos travaux pratiques de modélisation des réseaux au sein de l'UFR Ingénieurs 2000 de l'Université de Paris-Est-Marne-la-vallée, nous avons simulé plusieurs configurations de réseaux, essentiellement dans le but d'étudier les modalités de fonctionnement des différents types de files d'attentes.

En utilisant le logiciel de simulation NS2 (Network Simulator 2), dont les simulations se programment en TCL/TK, nous avons simulé plusieurs types de réseaux de différentes topologies. Ce rapport présente les résultats de nos expérimentations.

Après une description plus approfondie des objectifs du TP et des outils mis en œuvre, ce rapport présente quelques éléments de prise en main rapide du langage TCL, indispensable à la simulation de réseaux avec NS-2. Il présente ensuite l'étude de différents réseaux, les algorithmes spécifiques testés, ainsi que les conclusions de nos expérimentations.

1. Présentation générale du TP

Dans le cadre de notre formation d'ingénieur en informatique et réseaux, nous avons suivi une formation en modélisation des réseaux, dispensée par Pierre Delanoy, Hervé Picard, et Hakim Badis. La problématique principale que nous avons étudiée était orientée autour de la gestion de files d'attentes, permettant de modéliser les flux dans un réseau.

En effet, un réseau est composé d'un ensemble de files d'attentes qui interviennent à chaque étape du traitement : commutation, routage, réception de paquets… Ce sont elles, entre autre, qui permettent de gérer les congestions sur un lien en limitant la quantité de paquets qui peuvent être mis en attente pendant que la liaison est occupée.

1. Objectifs des Travaux Pratiques

Le premier objectif des travaux pratiques dirigés par Hakim Badis était de nous présenter les logiciels de simulation réseau NS-2 et NAM, qui permettent de simuler des réseaux complexes et de visualiser leur fonctionnement.

L'objectif sous-jacent était de nous faire simuler avec ces outils diverses situations couramment rencontrés sur un réseau comme la congestion, la coupure de lien sur un réseau en anneau, ou encore la simulation de systèmes M/M/1 et M/M/1/2.

Conjointement à ces TP, un projet a également été proposé par M. Badis. Celui-ci a fait l'objet d'un second rapport disponible conjointement à celui-ci.

2. Le langage de script TCL

Le langage TCL est un langage de script puissant qui permet d'utiliser éventuellement une approche de programmation orienté objet. Il est facilement extensible par un certain nombre de modules.

Dans notre cas, il est indispensable d'utiliser le langage TCL pour pouvoir travailler avec les objets fournis par NS-2.

3. L'outil de simulation NS-2

L'outil NS-2 fournit un ensemble d'objets TCL spécialement adaptés à la simulation de réseaux. Avec les objets proposés par ce moteur de simulation, on peut représenter des réseaux avec liens filaires ou sans fils, des machines et routeurs (Node), des flux TCP et UDP (par exemple pour simuler un flux CBR[1]), et sélectionner les politiques et règles régissant les files d'attentes mises en œuvre dans chacun des nœuds.

Dans tous nos exemples, nous avons toujours simulé des flux UDP constants, car ils permettent d'étudier les phénomènes de rejet de paquets sans encombrer le réseau.

NS-2 ne permet pas de visualiser le résultat des expérimentations. Il permet uniquement de stocker une trace de la simulation, de sorte qu'elle puisse être exploitée par un autre logiciel, comme NAM.

4. L'outil de visualisation NAM

NAM est un outil de visualisation qui présente deux intérêts principaux : représenter la topologie d'un réseau décrit avec NS-2, et afficher temporellement les résultats d'une trace d'exécution NS-2. Par exemple, il est capable de représenter des paquets TCP ou UDP, la rupture d'un lien entre nœuds, ou encore de représenter les paquets rejetés d'une file d'attente pleine.

Ce logiciel est souvent appelé directement depuis les scripts TCL pour NS-2, de sorte à visualiser directement le résultat de la simulation

[1] CBR : Constant Bitrate

2. Prise en main de TCL pour NS

Afin de prendre en main le langage TCL, nécessaire à la conception de simulations de réseau avec NS-2, nous avons réalisé un certain nombre d'entrainements : comprendre un script TCL, traduire un programme C en TCL, et réaliser une première applications pour NS-2 et NAM.

1. Exemple de script TCL commenté

Afin de prendre un premier contact efficace avec le langage de script TCL, qui permet de manipuler les objets de simulation de NS2, nous avons étudié et commenté le script suivant :

```
# -- Définition d'une méthode appelée "test"
proc test {} {
    # -- a vaut 43
    set a 43
    # -- b vaut 27
    set b 27
    # -- c vaut 27 + 43 = 70
    set c [expr $a + $b]
    # -- d vaut (43 – 27) * 70 = 16 * 70 = 1120
    set d [expr [expr $a - $b] * $c]

    # -- pour k allant de 0 à 10 exclu, on incrémente k à chaque boucle
    for {set k 0} {$k < 10} {incr k} {
      if {$k < 5} {
          # -- si k vaut moins de 5 exclus, on affiche val. de d puissance k
          puts "k < 5, pow = [expr pow($d, $k)]"
      } else {
          # -- si k est sup. ou égal à 5, on affiche val. de d modulo k
          puts "k >= 5, mod = [expr $d % $k]"
      }
    }
}

# -- Appel de la méthode appellée "test"
test
```

L'exécution de ce script se fait, grâce à ns, en utilisant simplement la commande :

```
$ ns test.tcl
```

Le résultat commenté de l'exécution est le suivant, les lignes en rouge correspondant à des commentaires :

```
[vivien@andLinux mdr]# ns test.tcl
# d = 1120 puissance 0 vaut 1
k < 5, pow = 1.0
# d = 1120 puissance 1 vaut 1120
k < 5, pow = 1120.0
# d = 1120 puissance 2 vaut 1254400
k < 5, pow = 1254400.0
# d = 1120 puissance 3 vaut 1404928000
k < 5, pow = 1404928000.0
# d = 1120 puissance 4 vaut 1573519360000
k < 5, pow = 1573519360000.0
# 1120 modulo 5 vaut 0 (1120/5 = 224 reste 0)
k >= 5, mod = 0
# 1120 modulo 6 vaut 4 (1120/6 = 186 reste 4)
k >= 5, mod = 4
# 1120 modulo 7 vaut 0 (1120/7 = 160 reste 0)
k >= 5, mod = 0
# 1120 modulo 8 vaut 0 (1120/8 = 140 reste 0)
k >= 5, mod = 0
# 1120 modulo 9 vaut 4 (1120/9 = 124 reste 4)
k >= 5, mod = 4
```

Ceci correspond bien au résultat attendu : la compréhension d'un script TCL est donc assez intuitive et simple.

2. Entraînement au TCL par traduction d'un script C

Afin de nous familiariser plus avant avec le langage TCL, et pour découvrir sa simplicité d'utilisation, nous avons réécrit le programme C suivant en TCL :

```
#include <stdio.h>

int main( int argc, char *argv[] ) {
  int count = 20;             /* The number of Fibonacci numbers to print */
  int num[] = { 0, 1 };       /* First and second seed values */
  int i;

  printf( "The first %d Fibonacci numbers:\n", count );
  for ( i = 1; i < count; i++ ) {
    printf( "%d:\t%d\n", i, num[0] );
    num[1] = num[0] + num[1];
    i++;
    printf( "%d:\t%d\n", i, num[1] );
    num[0] = num[0] + num[1];
  }
  return 0;
}
```

Ce que nous avons traduit par le script TCL suivant. La traduction est faite ligne à ligne : l'algorithme n'a pas été modifié ni optimisé :

```
proc main {} {
    set count 20
    array set num {
          0 0
          1 1
    }
    puts "The first $count Finonacci numbers:\n"
    for {set i 1} {$i < $count} {incr i} {
          puts "$i:\t$num(0)"
          set num(1) [expr ($num(0) + $num(1))]
          incr i
          puts "$i:\t$num(1)"
          set num(0) [expr ($num(0) + $num(1))]
    }
}

main
```

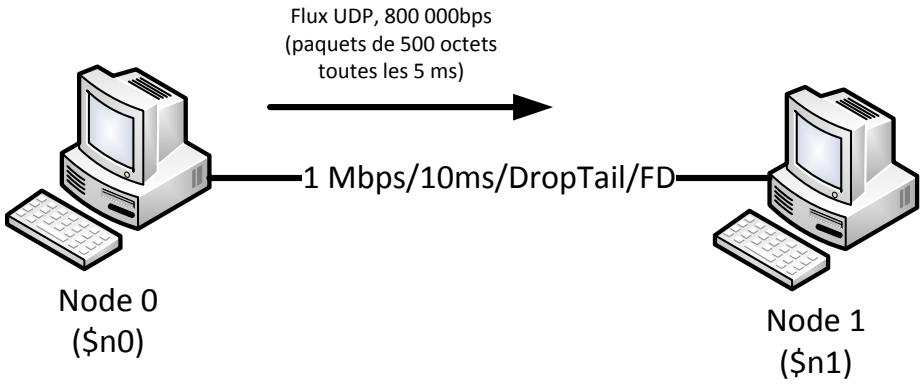
On notera, entre autre, qu'en TCL pour simuler un tableau il faut utiliser un dictionnaire (une table de hachage), que la commande puts permet d'insérer directement la valeur une variable dans la chaîne qu'elle affiche sur la sortie standard, et qu'il existe une instruction spécifique pour l'assignation d'une valeur à une variable (set) et pour l'incrément d'une variable (incr).

De plus, on remarque que les crochets sont utilisés pour extraire la valeur de retour d'une instruction et pouvoir la passer directement comme paramètre d'une méthode.

Après ces quelques notions de TCL, nous avons découvert le fonctionnement de NS-2 et NAM, qui permettent respectivement de fournir des objets TCL permettant de manipuler des objets réseaux (ordinateur/routeur = Node, trafic réseau, files d'attente…) et de visualiser les résultats d'une simulation NS-2 en montrant la topologie du réseau étudié, et les résultats des calculs (transmission de paquets, sous forme de dessins.

3. Première simulation avec NS-2

Pour observer le fonctionnement de NS-2 et de son outil de simulation NAM, nous avons simulé un montage réseau simple, correspondant au diagramme suivant :



Pour cela, nous avons créé deux nœuds NS-2 (`node`), reliés par un lien full duplex (`duplex-link`) supportant un débit de 1Mbps, un temps d'accès au medium de 10 ms, et qui utilise l'algorithme de file d'attente DropTail[2] pour la gestion des congestions.

Nous avons ensuite crée un agent UDP (`Agent/UDP`) attaché au nœud n0 (`attach-agent`) qui permet à n0 de transmettre des paquets UDP sur le réseau. Puis, nous avons créé un envoi de paquets à débit constant (Constant Bit Rate –CBR- `Application/Traffic/CBR`) attaché à l'agent UDP définit précédemment (`attach-agent`).

Nous avons enfin créé un agent vide, destiné à recevoir des paquets UDP ou TCP sans les traiter, comme le fichier /dev/null sous Unix (`Agent/Null`). On l'attache au nœud n1 (`attach-agent`) puis on connecte l'agent UDP et l'agent vide (`connect`).

Enfin, on indique que les traces de simulation doivent être loguées, qu'on souhaite lancer NAM à la fin de la simulation, que celle-ci va durer 5 secondes et que le CBR ne sera émis qu'à partir de la 0.5ème seconde de simulation, et jusqu'à la 4.5ème seconde.

Le code TCL de la simulation, commenté en détail, est fournit sur la page suivante.

[2] DropTail est un algorithme de file d'attente FIFO (First In First Out – premier arrivé, premier sorti)

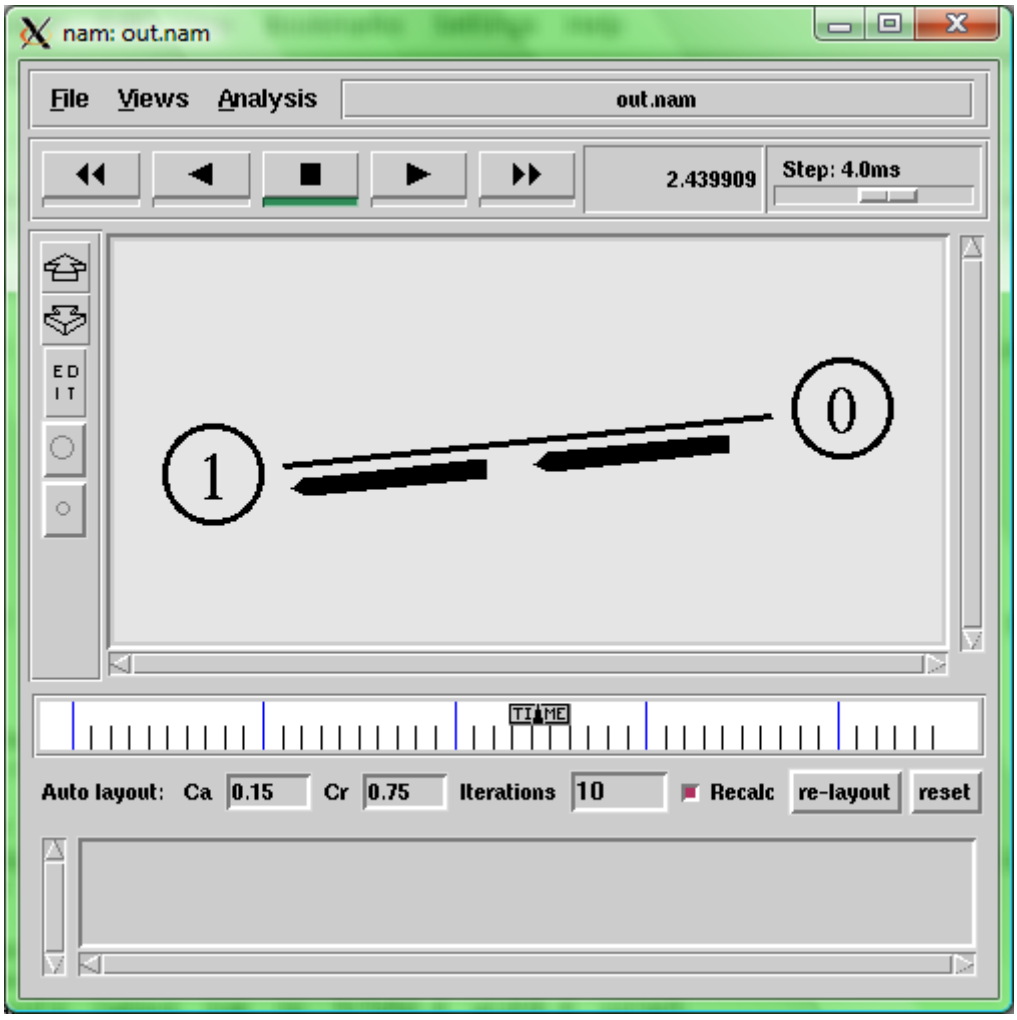
```
# Création d'un simulateur
set ns [new Simulator]

# Création du fichier de trace utilisé par le visualisateur
set nf [open out.nam w]

# Indique à NS de logguer ses traces dans le fichier $nf (out.nam)
$ns namtrace-all $nf

# Lorsque la simulation sera terminée, cette procédure sera appelée
# pour lancer automatiquement le visualisateur (NAM)
proc finish {} {
  # Force l'écriture dans le fichier des infos de trace
  global ns nf
  $ns flush-trace
  close $nf

  # Lance l'outil de visualisation nam
  exec nam out.nam &

  # Quitte le script TCL
  exit 0
}

# création de deux noeuds
set n0 [$ns node]
set n1 [$ns node]

# Création d'une liaison de communication full duplex entre les noeuds n0 & n1
# Fonctionne à 1Mbps, 10ms de délai, et utilise l'algorithme de file DropTail
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

# création d'un agent UDP implanté dans n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

# Création d'un traffic CBR pour le noeud 0 générateur de paquets à vitesse
constante
# Paquets de 500 octets (4000 bits), générés toutes les 5 ms.
# ---> Ceci représente un trafic de 800 000 bps (inférieur à la capacité du
lien)
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005

# Ce traffic est attaché à l'agent UDP udp0
$cbr0 attach-agent $udp0

# Création d'un agent vide, destiné à recevoir les paquets dans le noeud n1
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0

# Le trafic issu de l'agent udp0 est envoyé vers null0
$ns connect $udp0 $null0

# Début de l'envoi du CBR à 0.5s après le début de la simulation
$ns at 0.5 "$cbr0 start"

# Fin de l'envoi du CBR à 4.5s après la fin de la simulation
```

```
$ns at 4.5 "$cbr0 stop"

# La simulation s'arrête après 5 secondes, et appelle la procédure
# TCL nommée "finish" définie précédemment
$ns at 5.0 "finish"

# Démarrage du moteur de simulation
$ns run
```

On lance ensuite la simulation et la visualisation du résultat à l'aide de la commande :

```
$ ns 2nodes-udp-cbr.tcl
```

Ce qui donne le résultat suivant[3] :



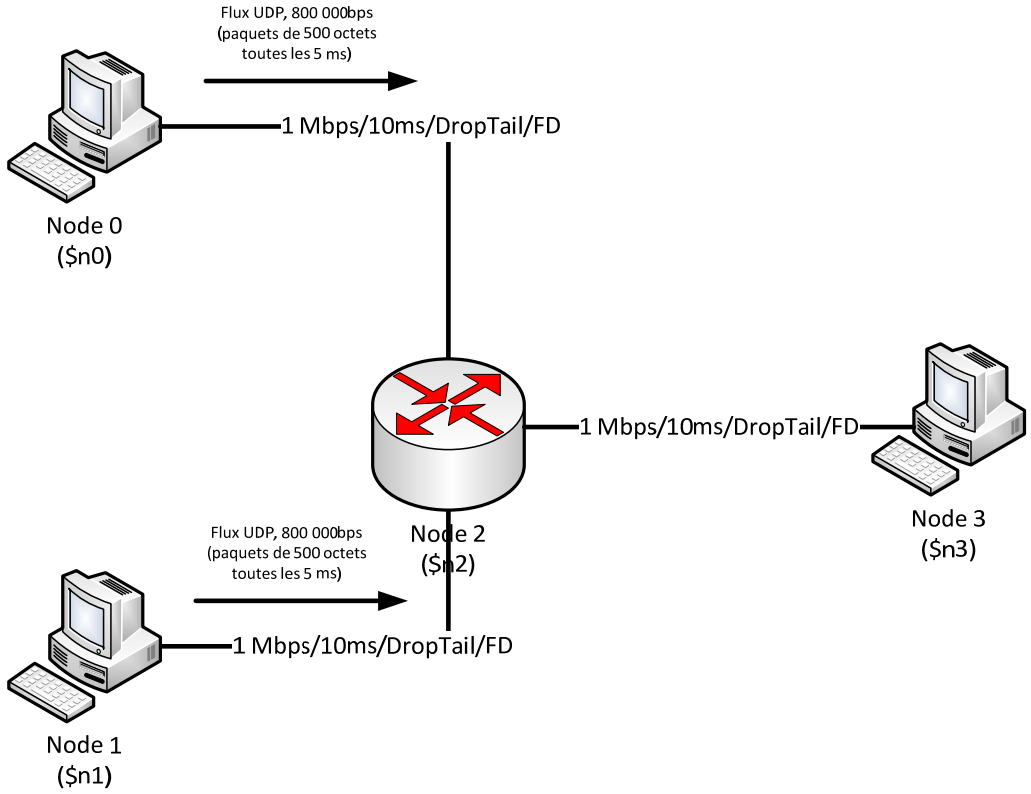
[3] Ici, la simulation a été réalisée sur une distribution andLinux, qui est une émulation Linux d'une Kubuntu sous Windows Vista.

3. Simulations avec NS-2

A la suite de notre apprentissage de l'utilisation basique de NS-2, nous avons réalisé xx simulations, dont les modes opératoires et les résultats sont expliqués ci-après.

1. Simulation de congestion dans un réseau

Pour simuler une congestion dans un réseau, nous avons réalisé un montage correspondant au diagramme suivant :



Soit deux nœuds émettant un CBR de 800 000 bps, à destination d'une machine écoutant ces deux CBR, en passant par un nœud (routeur ou commutateur) auquel les trois nœuds sont reliés. La congestion va se faire entre le routeur/commutateur et le nœud destination, car le lien est limité au mieux à 1 Mbps alors que la somme des CBR émis à sa destination est de 1 600 Kbps, donc plus élevé.

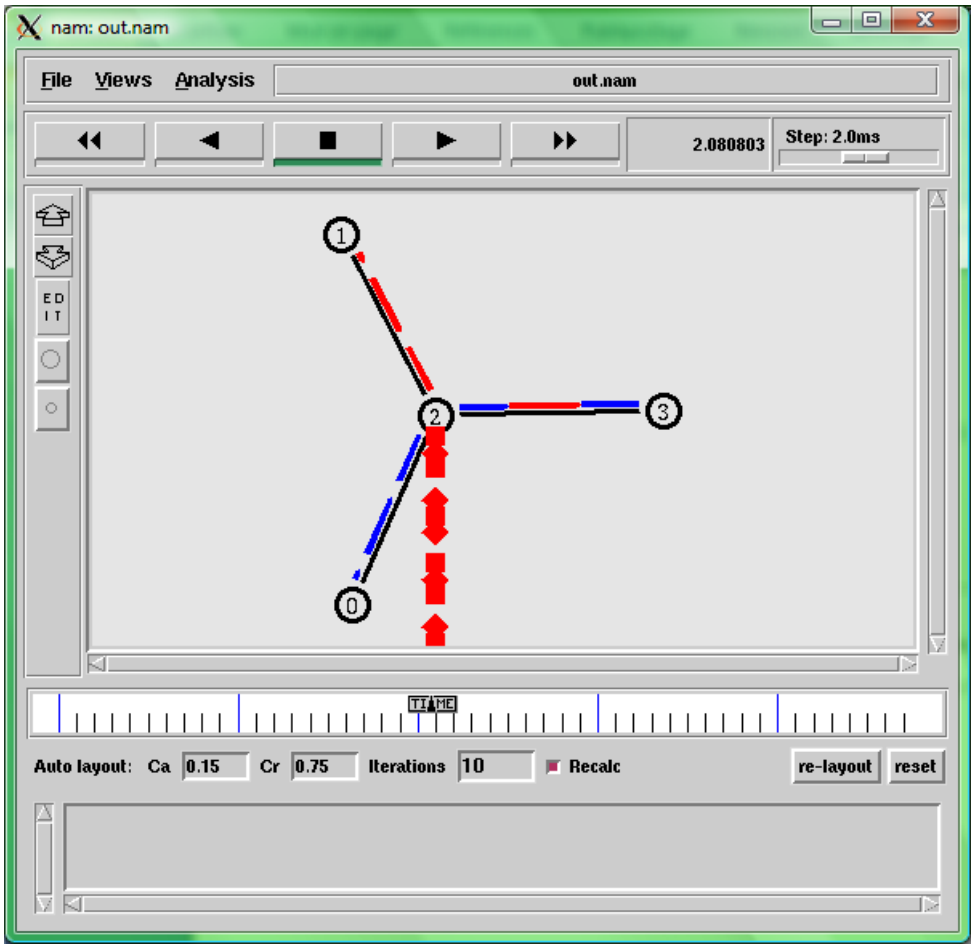
Les deux nœuds commencent l'émission de leurs données en CBR en même temps, et nous avons observé le comportement du nœud de commutation à ce moment.

Le script que nous avons mis en place est décrit et commenté sur la page suivante.

```
# Création du simulateur
set ns [new Simulator]

# Création du fichier de traces NS-2
set nf [open out.nam w]
$ns namtrace-all $nf

# Procédure de fin de simulation, qui écrit les données dans le fichier
# et lance NAM pour la visualisation
proc finish {} {
  global ns nf
  $ns flush-trace
  close $nf

  exec nam out.nam &
  exit 0
}

# Création des noeuds
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

# Création des liens, tous en 1Mbps/10ms de TR/file d'attente DropTail
$ns duplex-link $n0 $n2 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n3 $n2 1Mb 10ms DropTail


# Création de deux agents implantés dans n0 et n1
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1


# Traffic CBR de 500 octets toutes les 5 ms pour UDP0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

# Traffic CBR de 500 octets toutes les 5 ms pour UDP1
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005
$cbr1 attach-agent $udp1

# Création d'un agent vide, destiné à recevoir les paquets implanté dans n1
set null0 [new Agent/Null]
$ns attach-agent $n3 $null0

# Le trafic issus des agents udp0 et udp1 est envoyé vers null0
$ns connect $udp0 $null0
$ns connect $udp1 $null0

# Scénario de début et de fin de génération des paquets par cbr0
```

```
$ns at 0.5 "$cbr0 start"
$ns at 0.5 "$cbr1 start"
$ns at 4.5 "$cbr0 stop"
$ns at 4.5 "$cbr1 stop"

# La simulation va durer 5 secondes et appeller la proc finish
$ns at 5.0 "finish"

# Definition de classes pour la coloration
$udp0 set class_  1
$udp1 set class_  2

# Coloration des classes : bleu pour udp0 (classe 1) et rouge pour udp1
(classe 2)
$ns color 1 Blue
$ns color 2 Red

# début de la simulation
$ns run
```

L'ensemble des files d'attente des liens sont des FIFO, ce qu'il est important de noter à ce moment. Lorsque nous exécutons la simulation, nous obtenons un résultat proche du suivant :



En effet, une saturation du lien reliant les nœuds 2 et 3 se produit à partir du moment où la quantité de paquets à transmettre est trop importante pour la taille du lien, et que la file d'attente du lien est saturée. Comme la discipline de la file est FIFO, ce sont les premiers paquets acceptés qui sont transmis.

Avant saturation, un paquet sur deux transmis du nœud 2 vers le nœud 3 provient du flux UDP du nœud 0 et un paquet sur deux provient du flux du nœud 1. A partir du moment où la file d'attente est pleine, 37,5% des paquets[4] sont rejetés. Etant donné l'ordre d'arrivé des paquets, ce sont toujours les paquets provenant du nœud 1 ($n1) qui sont rejetés, à hauteur de 75% des paquets reçu.

Avec la discipline FIFO, tous les paquets provenant du nœud 0sont transmis au nœud 3, et un quart des paquets du nœud 1 sont transmis au nœud 3, tandis que trois quart des paquets émis depuis le nœud 1 à destination du nœud 3 sont perdus.

Afin d'améliorer l'équité des pertes entre les flux, nous avons ensuite étudié l'influence de la discipline de file d'attente mise en œuvre sur les liens.

Nous avons donc remplacé la discipline de file d'attente des trois liens par une SFQ (Stochastic Fairness Queueing),qui est supposé être un algorithme de répartition équitable. Les lignes à modifier sont les suivantes :

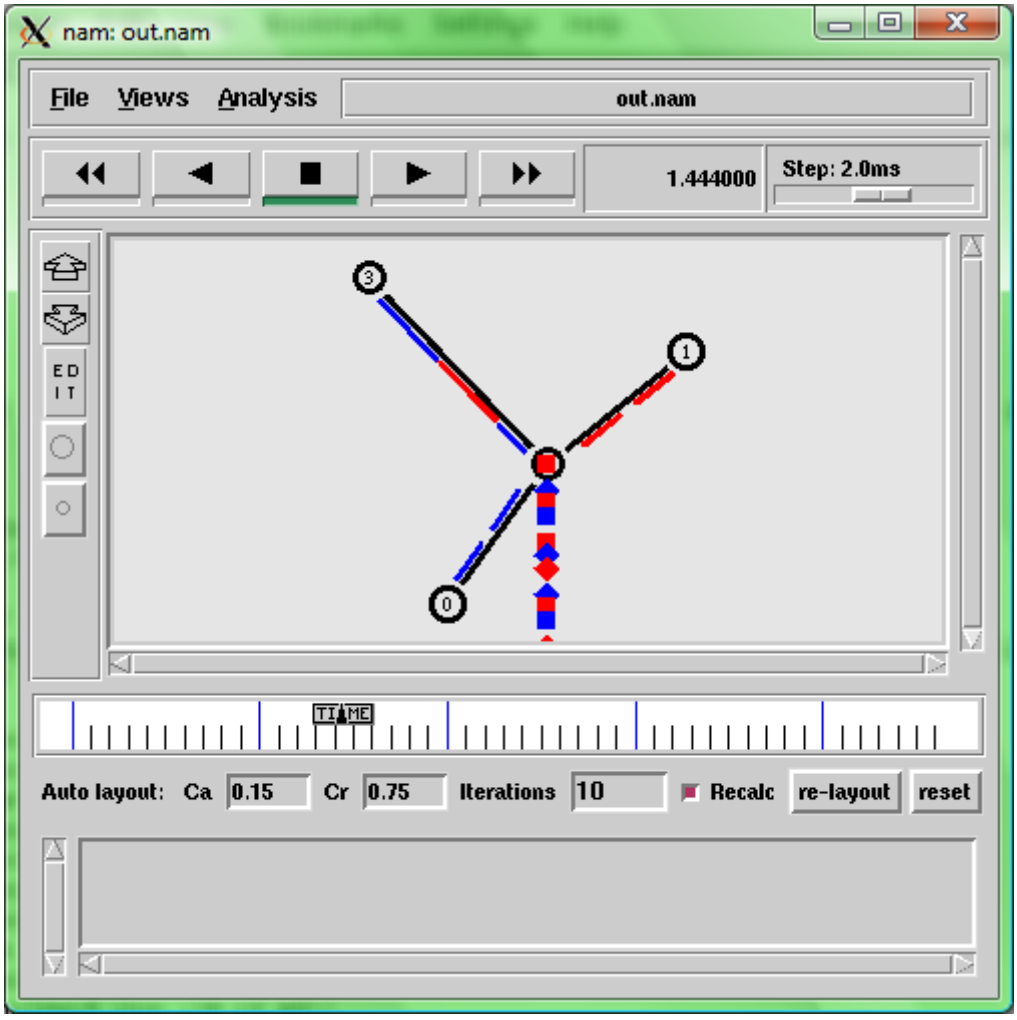
```
# Création des liens, tous en 1Mbps/10ms de TR/file d'attente DropTail
$ns duplex-link $n0 $n2 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n3 $n2 1Mb 10ms DropTail
```

Qu'on remplace par les lignes suivantes :

```
# Création des liens, tous en 1Mbps/10ms de TR/file d'attente DropTail
$ns duplex-link $n0 $n2 1Mb 10ms SFQ
$ns duplex-link $n1 $n2 1Mb 10ms SFQ
$ns duplex-link $n3 $n2 1Mb 10ms SFQ
```

[4] 600 Kbps rejetés sur les 1600Kbps transmis soit 600 / 1600 = 37,5 %

Le résultat visuel de la simulation est le suivant :



Avec la discipline SFQ, un paquet sur deux qui est rejeté, provient du flux UDP 0 et un sur deux provient du flux UDP 1. Comme 37,5% des paquets sont toujours perdu, environ 2 paquets sur 3 sont reçus par le nœud 3 pour chaque flux.

Cette discipline présente l'avantage d'être en effet équitable : les deux flux ont la même priorité et sont acheminés dans les mêmes conditions, ce qui permet d'avoir une transmission harmonieuse des données sur le réseau.

Cependant, on remarque que les disciplines de file d'attente utilisées précédemment ne permettent toujours que de détecter la congestion au niveau du nœud 2 (routeur ou commutateur). L'inconvénient de cette solution est qu'une partie de la bande passante utilisée sur les liens n0-n2 et n1-n2 est perdue inutilement puisqu'un tiers des paquets sont jetés au niveau du nœud n2.

Par conséquent, nous avons souhaité identifier les pertes au niveau des émetteurs, pour optimiser la consommation de bande passante globale sur le réseau.

Pour cela, nous avons tenté d'utiliser l'algorithme de discipline de file d'attente RED (Random Early Detection) sur le lien reliant le nœud 2 au nœud 3. Toutefois, ceci n'a pas été concluant dans la mesure où les pertes sont toujours identifiées seulement au niveau du nœud 2 : les nœuds 0 et 1 consomment toujours inutilement de la bande passante.

Pour pouvoir gérer les pertes au niveau de la source, il conviendrait d'utiliser plutôt un algorithme de gestion de la QoS, externe à la discipline de files d'attentes. Toutefois, après maintes recherches, aucune solution acceptable n'a été trouvée comme implantée en standard dans NS-2 pour remplir cette fonction pour un flux UDP.

2. Gestion de files d'attente

Nous avons précédemment étudié comment nous pouvions moduler la discipline de la file d'attente à travers un réseau en étoile composé de quatre nodes. Cependant, si on souhaite étudier plus en profondeur les files d'attente situés sur les différents liens du réseau (ou plutôt, sur les nœuds lors de l'accès au lien), il est possible d'utiliser des commandes spécifiques dans NS-2.

Ainsi, on peut moduler la taille d'une file d'attente avec la fonction NS-2 `queue-limit` et on peut moduler la position de l'affichage du contenu d'une file avec la commande `duplex-link-op` en agissant sur la variable `queuePos`.

Il nous est également possible de simuler la perte et le rétablissement d'un lien via les commandes `down` et `up`. Afin d'analyser plus en profondeur ces possibilités, nous sommes repartis du modèle de la section précédente, avec des files d'attente en CBQ et des pertes détectées au niveau du nœud 2.

Juste avant les lignes :

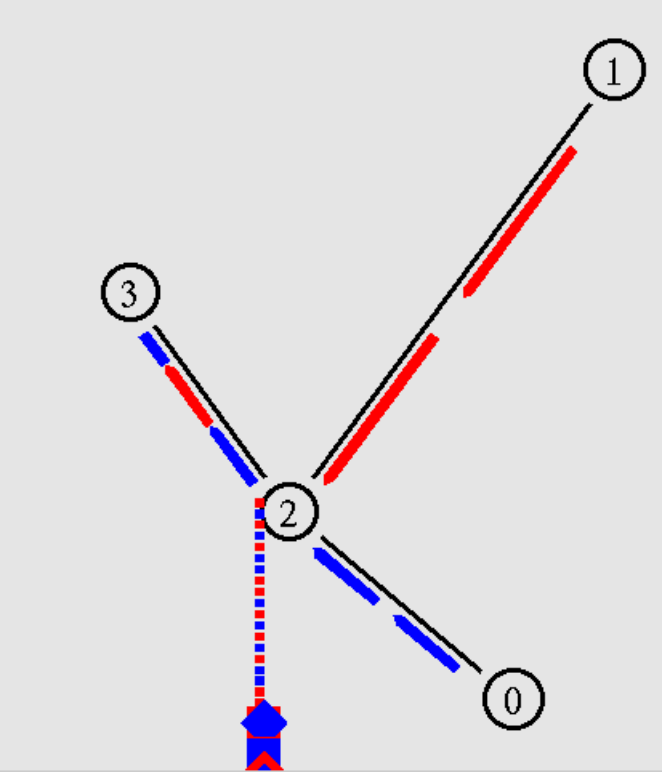
```
# début de la simulation
$ns run
```

On ajoute les lignes suivantes, qui servent à simuler que la file d'attente ne peut contenir que 10 paquets, qu'elle doit être positionnées visuellement à 28° (0.5

radians), et nous allons couper la liaison entre les nœuds 2 et 3 à compter de la 2ème seconde de simulation, pour 0.5 secondes :

```
$ns queue-limit $n2 $n3 10
$ns duplex-link-op $n2 $n3 queuePos 0.5

$ns rtmodel-at 2.0 down $n2 $n3
$ns rtmodel-at 2.5 up $n2 $n3
```

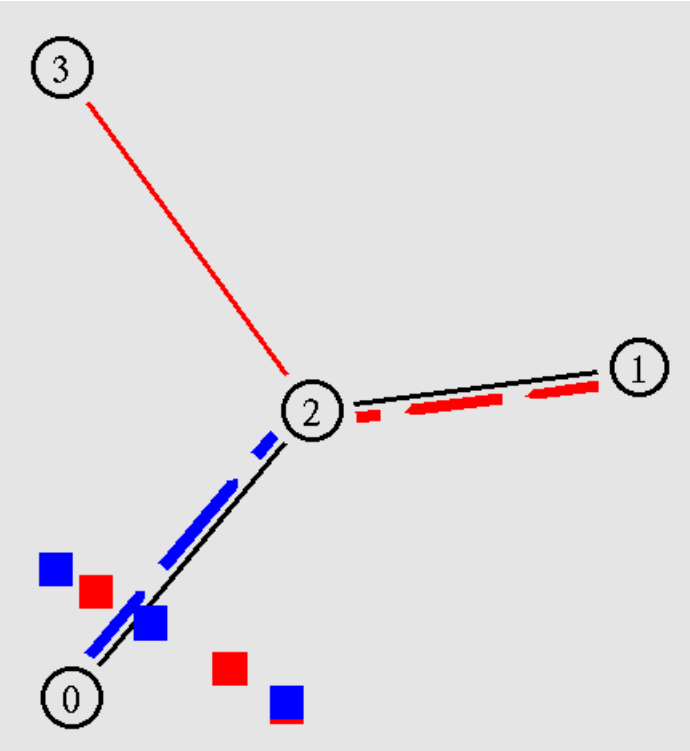
On lance alors la simulation. Les diagrammes ci-après montrent respectivement le comportement en fonctionnement avec une file d'attente de 10 Ko, et le comportement en fonctionnement lorsque la liaison entre les nœuds 2 et 3 est rompue.



On remarque que la file d'attente peut contenir visuellement 20 paquets, correspondant à nos 10 Ko (20 x 500 octets = 10 000 octets). On notera que, par défaut, notre version de NS-2 compte les tailles de files en octets, et non en nombre de paquets ; ce n'est pas le cas pour les versions de NS-2 inférieures à la 2.28 exclue.

On observe également mieux, grâce à cette modélisation, le comportement de la discipline SFQ : environ un paquet sur deux provient du nœud 0 et un sur deux du nœud 1, mais cela est valable sur la moyenne : on remarque que dans certains cas la

file comporte deux paquets du flux UDP0 ou deux paquets du flux UDP1 à la suite, et non une alternance exacte. Le principe de l'algorithme SFQ est, en effet, de fournir une discipline de file qui soit équitable en moyenne, mais rapidement calculable, donc pas toujours exact.



Lors de la rupture du lien entre les nœuds 2 et 3, la file d'attente est instantanément vidée car le destinataire (nœud 3) ne peut plus être contacté par le router/commutateur (nœud 2), et qu'il n'est donc pas nécessaire de stocker ces paquets qui ne pourront être transmis.

Par conséquent, le nœud 2 continue de recevoir les paquets des nœuds 0 et 1, mais les jette (drop) sans en tenir compte, dans la mesure où le destinataire n'est pas connu du nœud.

Une fois que le lien sera rétabli, la file d'attente se remplira de nouveau au fur et à mesure de la saturation du lien, et au bout d'environ une demie seconde, le comportement du système sera le même que dans le cas précédent : saturation de la file d'attente et perte des paquets reçus.

3. Etude approfondie des pertes de paquets

Pour visualiser et exploiter les résultats d'une simulation hors de NAM, il existe deux façons de procéder avec NS-2. Pour les exemples ci-après, on considère partir du script obtenu à partir des modifications réalisées dans la section précédente.

1. Générer des statistiques avec NS-2

Un moyen de mesurer les pertes et la bande passante au niveau du nœud 3 est de demander au moteur de simulation NS-2 de réaliser ces statistiques pour nous.

Pour cela, nous effectuons les modifications ci-après. Dans un premier temps, nous remplaçons les lignes :

```
set null0 [new Agent/Null]
$ns attach-agent $n3 $null0
```

Par ces lignes permettant de créer un moniteur d'écoute des pertes de paquets :

```
set sink0 [new Agent/LossMonitor]
$ns attach-agent $n3 $sink0
```

En conséquence, il faut aussi remplacer les lignes suivantes, qui référençaient $null0 :

```
$ns connect $udp0 $null0
$ns connect $udp1 $null0
```

Par les lignes suivantes, qui référencent sink0 :

```
$ns connect $udp0 $sink0
$ns connect $udp1 $sink0
```

Puis, à la suite des lignes :

```
set nf [open out.nam w]
$ns namtrace-all $nf
```

On ajoute la ligne permettant de créer le fichier de trace des pertes et de la bande passante :

```
set f0 [open trace.tr w]
```

En dessous des lignes :

```
$ns flush-trace
close $nf
```

On ajoute la ligne suivante, qui ferme le fichier contenant les calculs de pertes à la fin de la simulation :

```
close $f0
```

Remarque : il faut également penser à mettre f0 en tant que variable globale lors de l'utilisation du mot clef « global » au début de la procédure finish.

Au dessus de la ligne :

```
$ns at 5.0 "finish"
```

On ajoute la ligne suivante, qui appelle la méthode traceloss, que nous avons définit pour calculer périodiquement (toutes les 0,2 secondes) les pertes :

```
$ns at 0.0 "traceloss"
```

Enfin, on ajoute la procédure traceloss au fichier, en la plaçant juste après la création de sink0 :

```
proc traceloss {} {
  global sink0 f0
  set ns [Simulator instance]
  set time 0.1
  set now [$ns now]
  puts $f0 "$now [$sink0 set nlost_] [expr [$sink0 set bytes_] * 0.08]"
  $sink0 set bytes_ 0 # Remise à zero du compteur
  $ns at [expr $now+$time] "traceloss"
}
```

On obtient, à la fin de la simulation, un nouveau fichier nommé trace.tr. Celui-ci contient, sur chaque ligne, trois informations : le temps de la mesure, le nombre de paquets perdus mesurés, et le débit mesuré en Kbps (1 Kbps = 1000 bps).

On remarque d'emblée que, là où nous avons placé le LossMonitor, il n'est capable de détecter les pertes qu'en cas de rupture de lien, et non en cas de saturation du lien[5]. Par conséquent, la valeur obtenue est inexploitable, toujours à zéro.

Concernant la bande passante, on remarque que dans la mesure où NS-2 n'appelle pas précisément notre méthode toutes les 100 ms comme prévu, on observe quelques variations de débits liées à une erreur temporelle supérieure à 5 ms, ce qui provoque l'inclusion d'un paquet supplémentaire dans le calcul, soit une erreur de 4 Kbps dans la mesure.

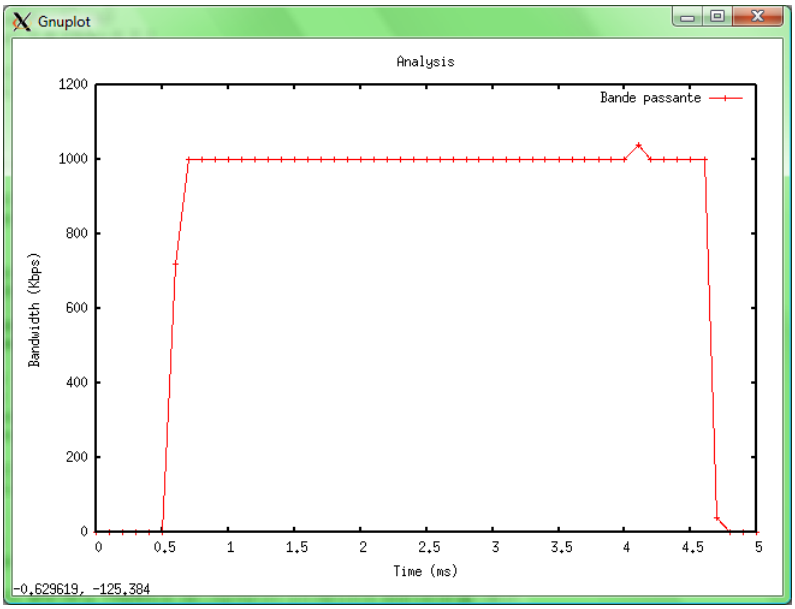
Pour visualiser les résultats avec gnuplot, nous avons créé le script suivant :

```
set title "Analysis"
set xlabel "Time (ms)"
set ylabel "Value"
set autoscale
plot "trace.tr" using 1:2 title 'Pertes' with linespoints
plot "trace.tr" using 1:3 title 'Bande passante' with linespoints
```

Qu'il suffit de charger en lançant gnuplot et en exécutant la commande :

```
load "bandwidthplot"
```

Ce qui donne le résultat suivant :



[5] Constaté sous Windows comme sous Linux, sous NS-2 version 2.32

Comme nous l'avions fait remarquer précédemment, on observe une erreur de calcul vers les 4.10 secondes de simulation. De plus, on remarque que le débit augmente progressivement et diminue progressivement : ceci est dû au temps d'accès au médium qui est de 10ms, et au fait que les paquets sont transmis à raison de 2x40 Kb toutes les 5ms, et au fait qu'à la fin de la transmission, la file d'attente se vide.

2. Détermination par un calcul externe

On peut également déterminer ces valeurs en calculant dans un autre langage les valeurs recherchées. Pour cela, il faut pouvoir connaitre l'ensemble des paquets, le chemin qu'ils parcourent, et toutes autres informations nécessaires au calcul.

Pour obtenir ces informations, il suffit d'ajouter les lignes suivantes :

```
set f1 [open traceall.tr w]
$ns trace-all $f1
```

A la suite des lignes déjà existantes :

```
set nf [open out.nam w]
$ns namtrace-all $nf
```

De plus, il faut penser à fermer le descripteur de fichier dans la procédure finish, en ajoutant f1 en global, et en utilisant la commande `close $f1`.

Chaque ligne du fichier a le format suivant :

```
+ 0.5 0 2 cbr 500 ------- 1 0.0 3.0 0 0
```

De gauche à droite, séparés par des espaces, la signification est la suivante :

- + indique une entrée en file d'attente, - indique une sortie de file, r une réception de paquet, d que le paquet est jeté, et h qu'il s'agit d'un hop
- L'instant auquel l'évènement se produit (ici 0.5 ms après le début)
- Le nœud par lequel le paquet entre en file (ici, le nœud 0)
- Le nœud qui se trouve à l'autre extrémité du lien pour lequel on entre ou on sort de file d'attente (ici, le nœud 2)

- Le type de trafic (ici on est en bitrate constant – cbr)
- La taille du paquet de données, en octets (ici 500 octets)
- Les flags appliqués au paquet (ici aucun, matérialisé par plusieurs tirets)
- L'identifiant du flux (en IPv6) ou la class définie manuellement (ici 1)
- Le numéro du nœud source du paquet, un point, puis le numéro de l'agent émetteur (relatif à l'ordre des agents enregistrés sur le nœud)
- Le numéro du nœud destinataire du paquet, un point, puis le numéro de l'agent récepteur (relatif à l'ordre des agents enregistrés sur ce nœud)
- Le numéro de séquence du paquet, relatif au flux auquel il appartient
- Un numéro de paquet unique qui permet de repérer un paquet au sein de l'ensemble des paquets de la simulation

On peut, par exemple, traiter ces données à l'aide d'un script awk (ou PERL, ou Python, …). Dans notre cas, pour analyser le nombre de paquets perdus et la bande passante consommée au cours du temps, nous avons utilisé le script awk de la page suivante. Pour l'exécuter : `awk –f nomfichier.awk < traceall.out`

Celui-ci analyse chaque ligne du fichier de sortie, et place les statistiques sur les pertes de paquet et la bande passante dans le fichier total.dat. On peu ensuite visualiser ce fichier avec gnuplot grâce aux scripts suivants :

```
set title "Analysis"
set xlabel "Time (ms)"
set ylabel "Bandwidth (Mbps)"
set autoscale
plot "total.dat" using 1:3 title 'Bande passante' with linespoints
```

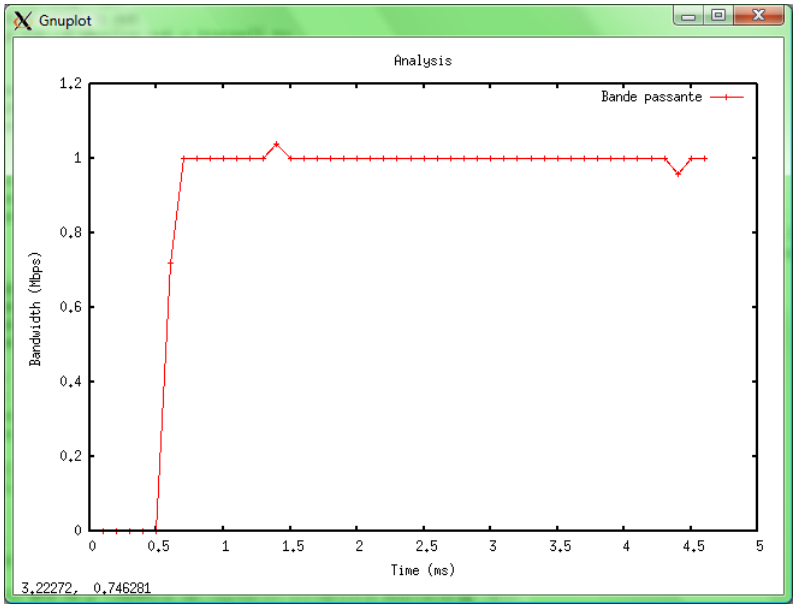
Et :

```
set title "Analysis"
set xlabel "Time (ms)"
set ylabel "Packets"
set autoscale
plot "total.dat" using 1:2 title 'Pertes' with linespoints
```

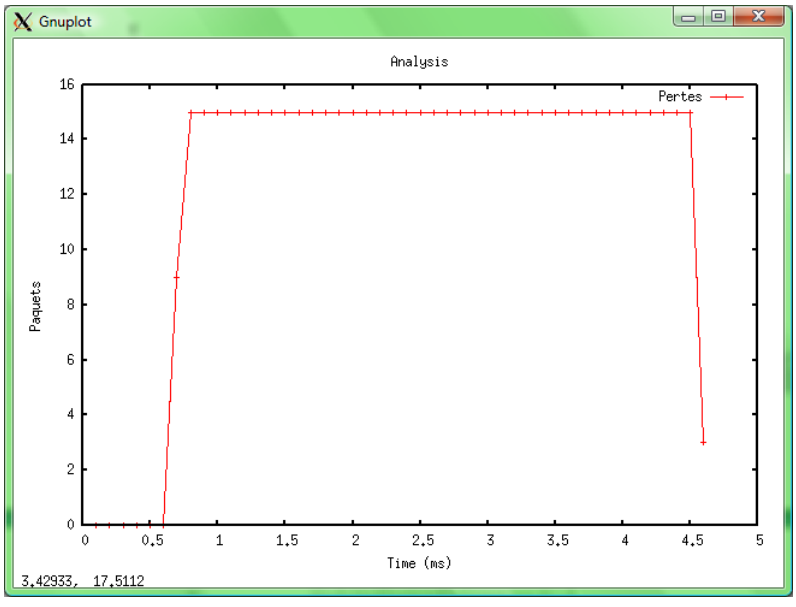
```
BEGIN {
  # Données de la simulation: début à 0, step de 100ms
  record_time = 0.0;
  time_interval = 0.100;

  received_bytes = 0;
  lost_packets = 0;
  total_received_bytes = 0;
  total_lost_packets = 0;
}

{
  action = $1;
  time = $2;
  src = $3;
  dst = $4;
  name = $5;
  size = $6
  flow_id = $8;
  src_address = $9;
  dst_address = $10;
  seq_no = $11;
  packet_id = $12;

  if ((action == "r" || action == "d") && src == "2" && dst == "3") {

    # if received time belong to a new interval
    if (time > (record_time + time_interval)) {
        total_received_bytes = total_received_bytes + received_bytes;
        total_lost_packets = total_lost_packets + lost_packets;
        record_time = record_time + time_interval;
        total = received_bytes * 8 / time_interval /1000000;
        total_lost = lost_packets;
        printf "%f %f %f\n", record_time, total_lost, total > "total.dat";

        while (time > (record_time + time_interval)) {
          record_time = record_time + time_interval;
          printf "%f %f %f\n", record_time, 0.0, 0.0 > "total.dat";
        }

        # this received packet belong to the next time interval
        if (action == "r") {
          received_bytes = size;
          lost_packets = 0;
        } else if (action == "d") {
          received_bytes = 0;
          lost_packets = 1;
        }
    } else {
      # if rcv_time still belong to this time_interval
      if (action == "r") {
        received_bytes = received_bytes + size;
      } else if (action == "d") {
        lost_packets = lost_packets + 1;
      }
    }
  }

}
```

```
END {
  average_throughput = total_received_bytes * 8 / record_time / 1000000;
  printf "Average_throughput: %1.3f (Mbits)\n", average_throughput;
  printf "Total lost packets: %1f\n", total_lost_packets;
}
```

Les tracés des courbes que nous avons obtenus sont les suivants :



Et :



On remarque que l'analyse des données avec un traitement externe est plus précis : même si on a quelques différences au niveau du débit (dépassement des 1Mbps

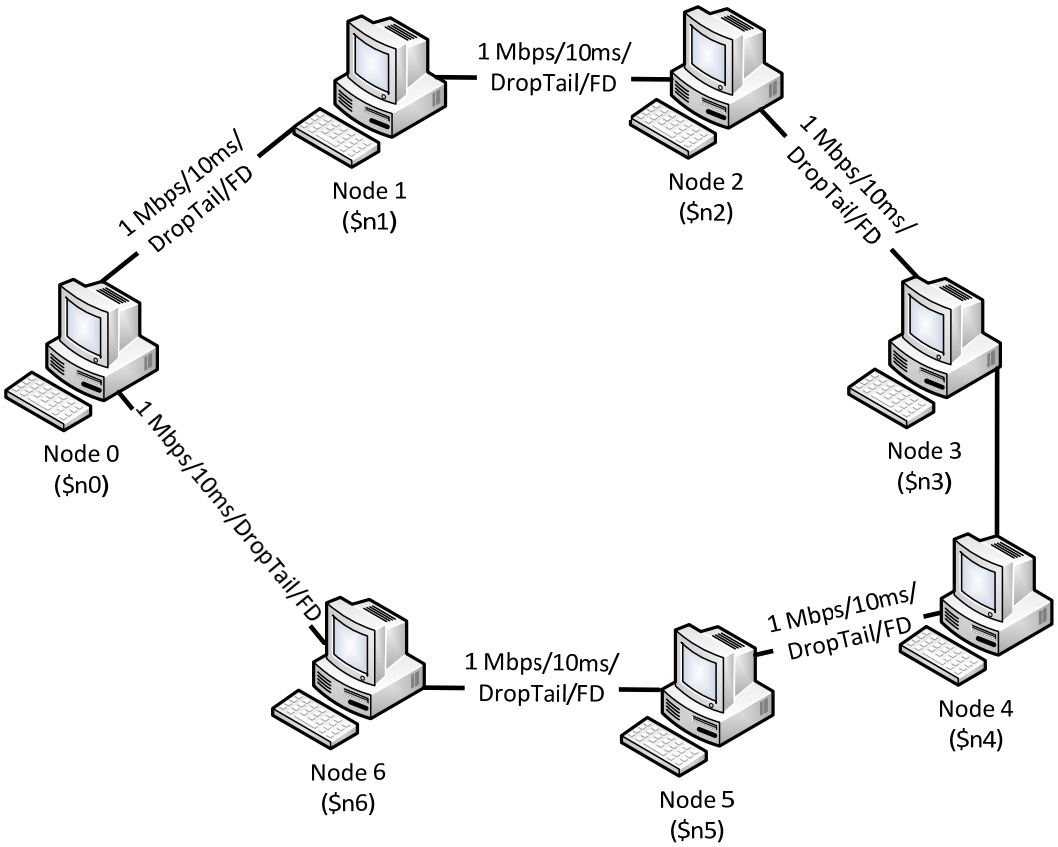
alloués), on peut analyser les pertes de paquets qui augmentent progressivement au début du transfert et se stabilisent à 15 paquets perdus par seconde par la suite.

A priori, cette solution est la meilleure car elle se fait après la simulation : ceci évite de devoir relancer un processus de simulation lourd, et permet d'extraire n'importe quelle information puisque le scénario complet est facilement lisible.

Cette solution semble donc préférable à la première pour ces raisons.

4. Topologie dynamique en anneau

Afin d'étudier les protocoles de routage implantés dans NS-2, nous avons étudié le cas d'un réseau en anneau correspondant au diagramme suivant :

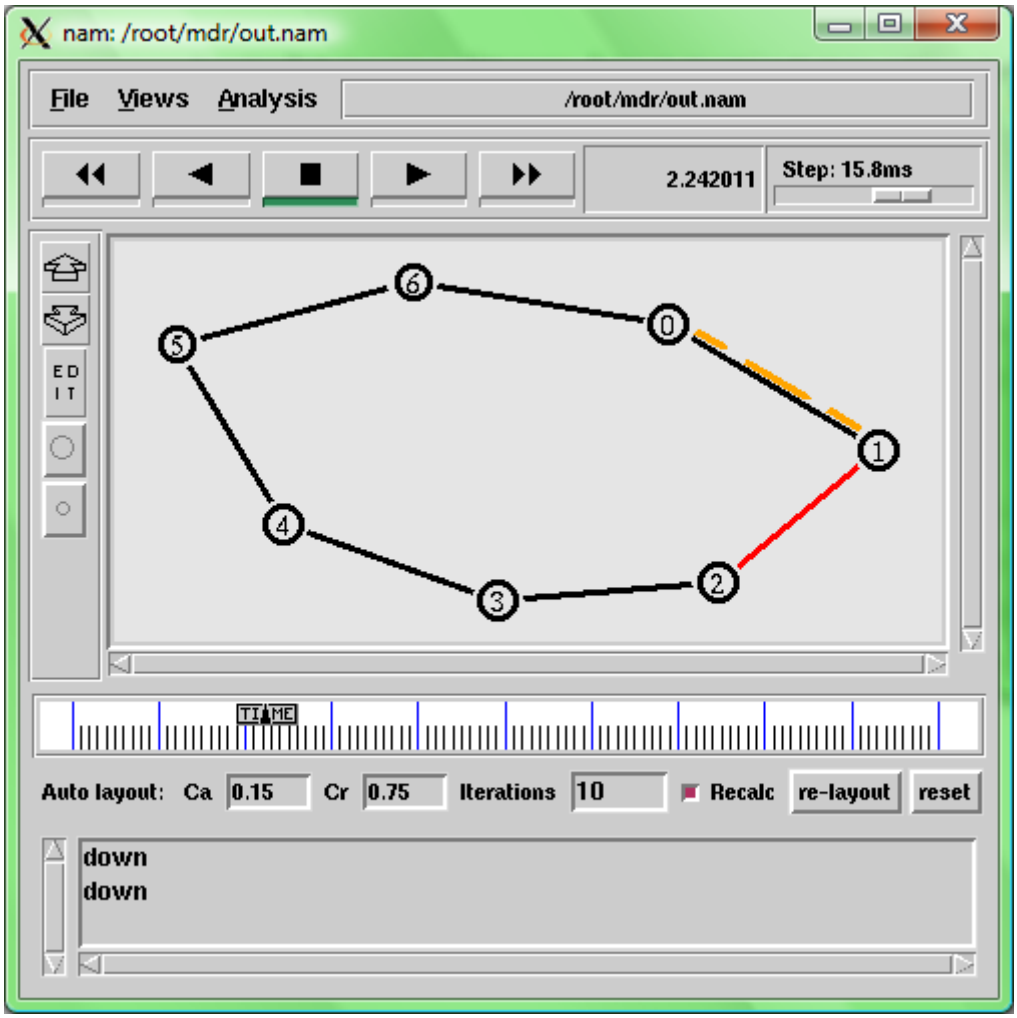


Tous les liens sont d'égale longueur, et possèdent des caractéristiques similaires (temps d'accès, vitesse de transmission, discipline de file d'attente).

L'objectif est de faire émettre un CBR depuis le nœud 0 et à destination du nœud 3, à compter de la 2ème seconde et jusqu'à la 10ème seconde de simulation, et en

coupant le lien entre les nœuds 1 et 2 après 2 secondes de simulation, pendant 2 secondes.

Le résultat que nous obtenons dans un premier temps est le suivant :



On remarque qu'en temps normal, les paquets émis par le nœud 0 à destination du nœud 3 sont routés en passant par les liens 0-1, 1-2 et 2-3. Lors de la rupture de lien entre les nœuds 1 et 2, la route n'est pas recalculée, ce qui provoque la perte de tous les paquets émis durant la rupture de lien, soit 2 secondes (400 paquets de 500 octets).

En effet, par défaut, NS-2 utilise un algorithme de routage utilisant la proximité pour déterminer le plus court chemin, mais avec des tables de routage statiques. En cas de rupture d'un lien, la route vers le nœud 3 n'est pas recalculée.

Le script conçu pour la simulation était le suivant :

```
# création d'un simulateur
set ns [new Simulator]
set nf [open out.nam w]
$ns namtrace-all $nf
# lorsque la simulation sera terminée, cette procédure est appelée pour lancer
automatiquement le visualisateur
proc finish {} {
  global ns nf
  $ns flush-trace
  close $nf
  exec nam out.nam &
  exit 0
}

# Creating network nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]
set n5 [$ns node]
set n6 [$ns node]

# Creating network links

$ns duplex-link $n0 $n1 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n2 $n3 1Mb 10ms DropTail
$ns duplex-link $n3 $n4 1Mb 10ms DropTail
$ns duplex-link $n4 $n5 1Mb 10ms DropTail
$ns duplex-link $n5 $n6 1Mb 10ms DropTail
$ns duplex-link $n6 $n0 1Mb 10ms DropTail

# On tente de réduire la taille de la queue

#$ns queue-limit $n2 $n3 100

$ns duplex-link-op $n0 $n1 queuePos 0.5
$ns duplex-link-op $n1 $n2 queuePos 0.5
$ns duplex-link-op $n2 $n3 queuePos 0.5
$ns duplex-link-op $n3 $n4 queuePos 0.5
$ns duplex-link-op $n4 $n5 queuePos 0.5
$ns duplex-link-op $n5 $n6 queuePos 0.5
$ns duplex-link-op $n6 $n0 queuePos 0.5

# Création de deux agents implantés dans n0 et n1

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

 # création d'un traffic CBR pour le n½ud 0 générateur de paquets à vitesse
constante paquets de 500 octets, générés toutes les 5 ms. Ce traffic est
attaché au udp0

set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

```
# création d'un agent vide, destiné à recevoir les paquets implanté dans n1

set sink0 [new Agent/LossMonitor]
$ns attach-agent $n3 $sink0

# le trafic issus de l'agent udp0 est envoyé vers sink0
$ns connect $udp0 $sink0

# scénario de début et de fin de génération des paquets par cbr0
$ns at 0.5 "$cbr0 start"
$ns at 10.0 "$cbr0 stop"

$ns rtmodel-at 2.0 down $n1 $n2
$ns rtmodel-at 4.0 up $n1 $n2

# la simulation va durer 10.5 secondes de temps simulé
$ns at 10.5 "finish"

$udp0 set class_  1
$ns color  1 Orange

# début de la simulation
$ns run
```

Pour palier à ce phénomène, il est possible de préciser l'algorithme de routage que l'on souhaite utiliser dans NS-2. Par exemple, on peut choisir l'algorithme de routage dynamique Distance Vector (vecteur de distance) qui permet de déterminer le plus court chemin parmi les liaisons disponibles.

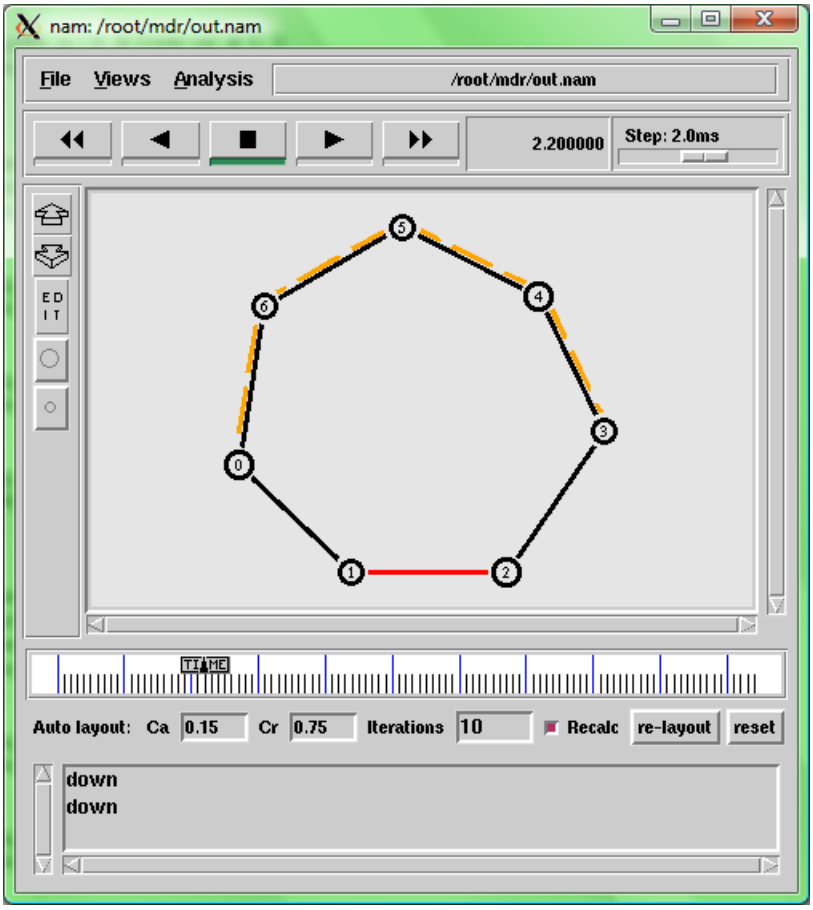
Pour préciser l'algorithme de routage, il suffit d'ajouter la ligne suivante juste après la création du simulateur :

```
$ns rtproto DV
```

Par conséquent, quand le lien 1-2 est rompu, les paquets transitent par le nouveau chemin le plus court : 0-6, 6-5, 5-4, 4-3 comme montré sur le diagramme ci-après.

Les paquets perdus sont ceux qui étaient déjà en train de transiter sur le lien 0-1, ou qui étaient déjà en file d'attente pour l'envoi sur le lien 1-2 au moment de la rupture du lien. Ce nombre est sensiblement négligeable comparé aux 400 paquets perdus précédemment.

De plus, une fois le lien rétabli, l'algorithme de routage DV est capable de recalculer la route la plus courte, et les paquets sont de nouveau transmis par les liens 0-1, 1-2 et 2-3.



5. Simulation d'un système M/M/1

Dans la notation de Kendal, M/M/1 signifie un système aux arrivées Markoviennes/Poissoniennes (M), à temps de traitement Markovien/Exponentiel (M), composé d'une file d'attente de taille illimitée et d'un seul serveur (1).

Nous avons, pour cela, utilisé un script fournit par M. Hakim Badis, que nous avons adapté et commenté pour plus de clarté. Le résultat se trouve sur la page suivante.

```
# Création du simulateur
set ns [new Simulator]

# Lambda représente le nombre moyen de paquets par seconde
set lambda [exec 1.0 * [lindex $argv 0]]
# Mu représente la taille moyenne d'un paquet
set mu     [exec 1.0 * [lindex $argv 1]]

# Enregistrement des traces complètes de paquets dans out.tr
set tf [open out-$lambda-$mu.tr w]
$ns trace-all $tf

# On crée deux noeuds
set n1 [$ns node]
set n2 [$ns node]

# On crée un lien de 100Kbps entre les deux (FIFO, pas de temps d'accès)
set link [$ns simplex-link $n1 $n2 100kb 0ms DropTail]
# On limite la file à 100000 octets, ce qui dans notre cas tend vers l'infini
$ns queue-limit $n1 $n2 100000

# On crée des lois d'arrivée exponentielles pour le temps...
set InterArrivalTime [new RandomVariable/Exponential]
$InterArrivalTime set avg_ [expr 1/$lambda]
# ... Et pour la taille des paquets
set pktSize [new RandomVariable/Exponential]
$pktSize set avg_ [expr 100000.0/(8*$mu)]

set src [new Agent/UDP]
$src set packetSize_ 100000
$ns attach-agent $n1 $src

# On monitore la file d'attente au cours du temps
set qmon [$ns monitor-queue $n1 $n2 [open qm-$lambda-$mu.out w] 0.1]
$link queue-sample-timeout

proc finish {} {
    global ns tf
    $ns flush-trace
    close $tf
    exit 0
}

# Procédure d'envoi d'un paquet selon les loi exponentielles
proc sendpacket {} {
    global ns src InterArrivalTime pktSize
    set time [$ns now]
    # Programme l'envoi d'un nouveau paquet après temps de loi exponentielle
    $ns at [expr $time + [$InterArrivalTime value]] "sendpacket"
    set bytes [expr round ([$pktSize value])]
    $src send $bytes
}

# On place un récepteur pour les paquets transmis
set sink [new Agent/Null]
$ns attach-agent $n2 $sink

# Et on relie la source et la destination ensemble (en udp, non connecté)
$ns connect $src $sink
```

```
$ns at 0.0001 "sendpacket"
$ns at 1000.0 "finish"

$ns run
```

Dans le fichier de résultat qm.out, le 5ème élément de chaque ligne correspond respond à la taille instantanée moyenne de la file d'attente. Pour calculer la moyenne sur l'ensemble de la simulation, on peut donc utiliser le script awk suivant :

```
BEGIN {
  total_queue_size = 0;
  total_lines_count = 0;
}

{
  mean_size = $5;
  total_queue_size = total_queue_size + mean_size;
  total_lines_count = total_lines_count + 1;
}

END {
  average = total_queue_size / total_lines_count
  printf "%1f\n", average;
}
```

De plus, pour calculer le temps moyen d'attente en file, il faut utiliser le fichier de trace out-lambda-mu.tr qui contient le listing des paquets avec leur date d'entrée et de sortie de la file d'attente. Pour déterminer le temps moyen d'attente dans la file, il faut donc mesurer le temps moyen entre l'entrée dans la file d'attente et la sortie de la file d'attente. Pour cela, nous avons réalisé le script awk suivant :

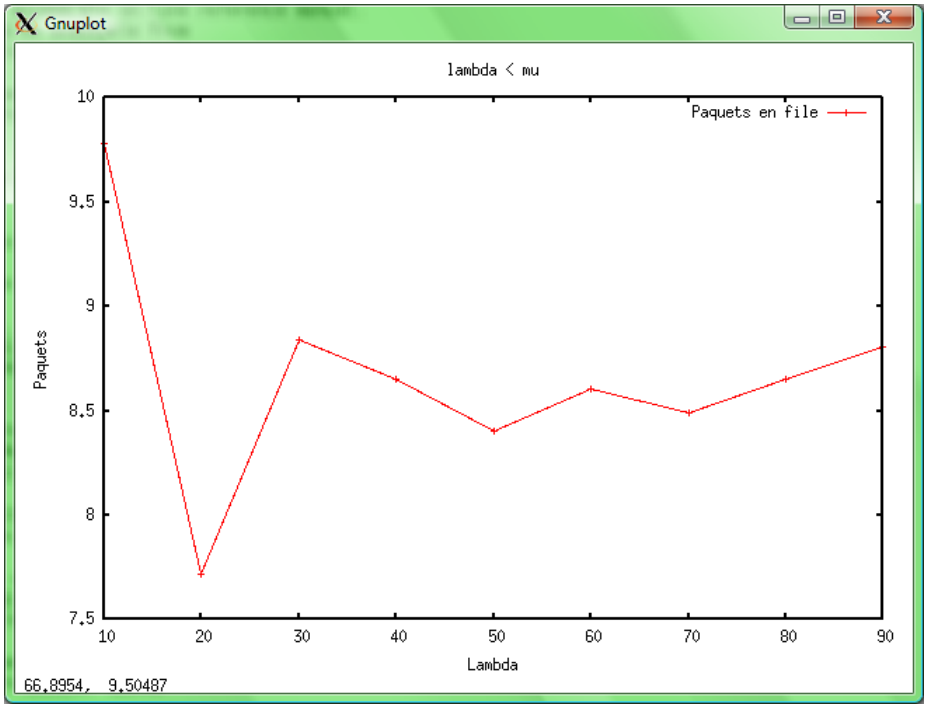
```
BEGIN {
  total_time = 0;
  total_packets = 0;
}

{
  action = $1;
  time = $2;
  src = $3;
  dst = $4;
  name = $5;
  size = $6
  flow_id = $8;
  src_address = $9;
  dst_address = $10;
  seq_no = $11;
  packet_id = $12;

  if (action == "+") {
    packets[packet_id] = time;
  } else if (action == "-") {
    total_packets = total_packets + 1;
    total_time = total_time + (time - packets[packet_id]);
  }
}

END {
  mean = total_time / total_packets * 1000;
  printf "%1f", mean;
}
```

Puis nous avons réalisé un script shell permettant d'automatiser les tests et la visualisation des résultats :

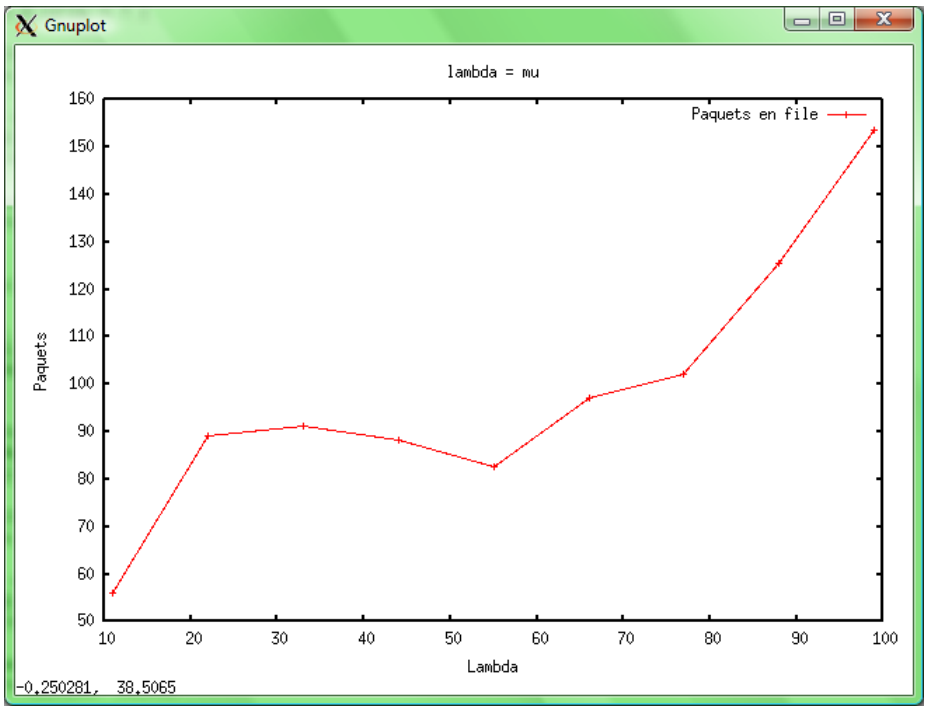
```
#!/bin/bash

simulate() {
  echo "Simulating with lambda=$1 and mu=$2 and writing to file $3-queue.dat
and $3-time.dat"
  ns MM1.tcl $1 $2
  res=`awk -f queue.awk < qm-$1-$2.out`
  echo "$1 $res" >> $3-queue.dat
  res2=`awk -f queue-time.awk < out-$1-$2.tr`
  echo "$1 $res" >> $3-time.dat
}

simulate 10.0 11.0 l-inf-mu
simulate 20.0 22.0 l-inf-mu
simulate 30.0 33.0 l-inf-mu
simulate 40.0 44.0 l-inf-mu
simulate 50.0 55.0 l-inf-mu
simulate 60.0 66.0 l-inf-mu
simulate 70.0 77.0 l-inf-mu
simulate 80.0 88.0 l-inf-mu
simulate 90.0 99.0 l-inf-mu

simulate 11.0 11.0 l-eq-mu
simulate 22.0 22.0 l-eq-mu
simulate 33.0 33.0 l-eq-mu
simulate 44.0 44.0 l-eq-mu
simulate 55.0 55.0 l-eq-mu
simulate 66.0 66.0 l-eq-mu
simulate 77.0 77.0 l-eq-mu
simulate 88.0 88.0 l-eq-mu
simulate 99.0 99.0 l-eq-mu

simulate 12.1 11.0 l-sup-mu
simulate 24.2 22.0 l-sup-mu
simulate 36.3 33.0 l-sup-mu
simulate 48.4 44.0 l-sup-mu
simulate 60.5 55.0 l-sup-mu
simulate 72.6 66.0 l-sup-mu
simulate 84.7 77.0 l-sup-mu
simulate 96.8 88.0 l-sup-mu
simulate 108.9 99.0 l-sup-mu
```

Ce script permet de générer les données nécessaires à l'affichage des graphiques ci-après.

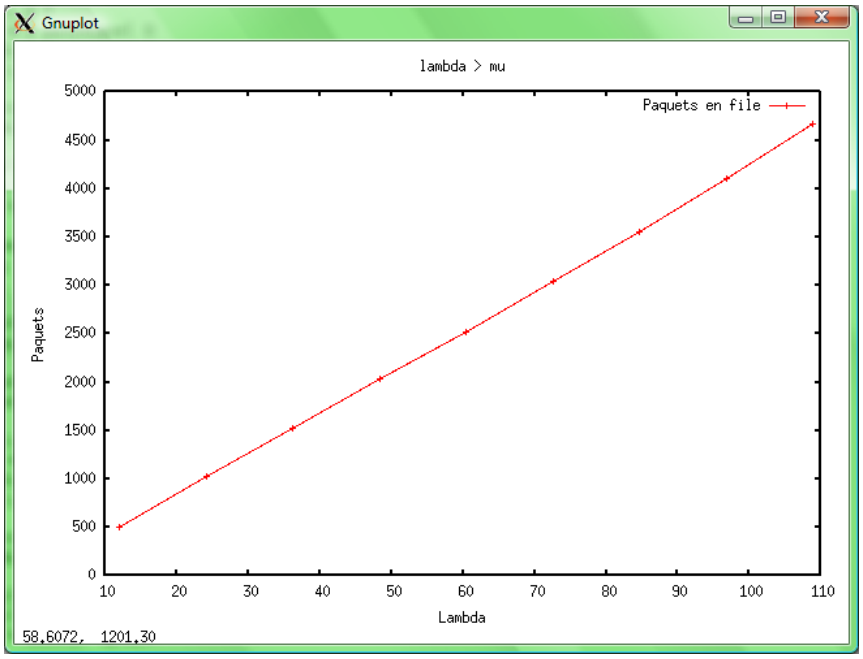
Paquets en file d'attente pour les trois cas de figure, en fonction de la valeur de lambda :



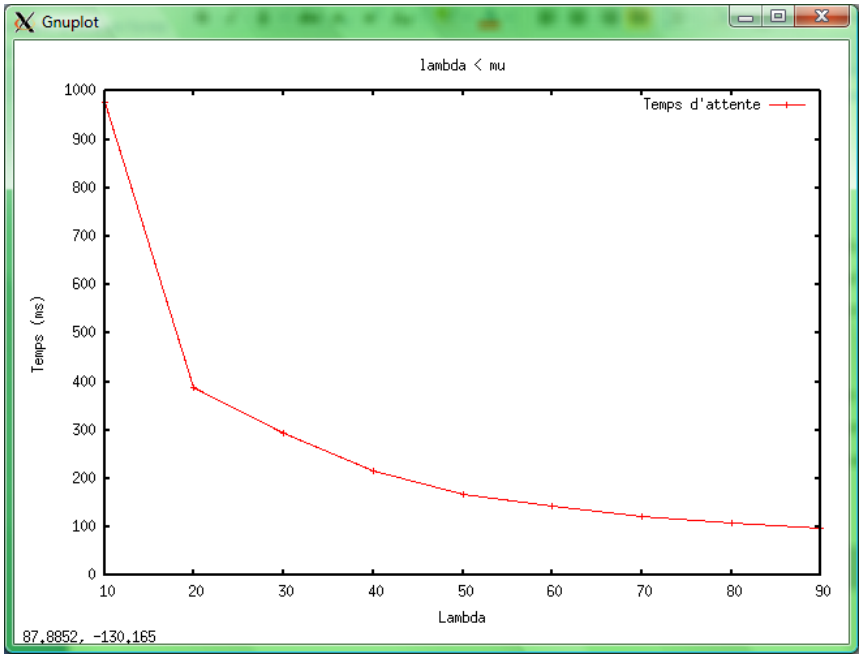
Dans le cas où lambda < mu, il y a très peu de paquets en file d'attente en moyenne, car le traitement par la file d'attente est assez rapide et que les paquets arrivent moins rapidement qu'ils ne partent. Globalement, quelle que soit la valeur de lambda, le nombre de paquets en file est compris entre 8 et 9.



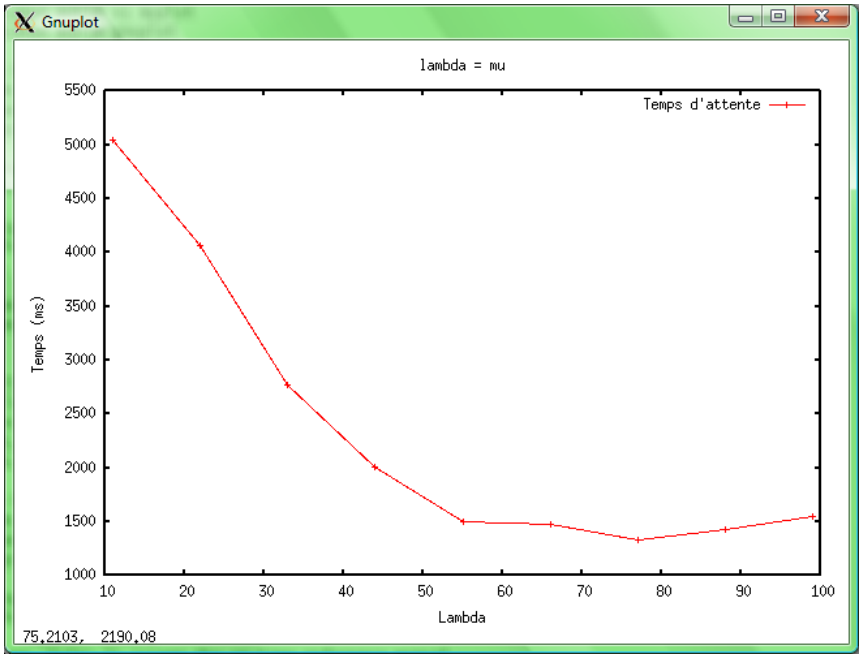
Lorsque lambda et mu sont égaux, il y a autant de paquets qui quittent le système que de paquets qui arrivent en file d'attente. Aussi, la taille de la file d'attente augmente consécutivement, mais reste assez stable car les paquets sont évacués à peu près aussi vite qu'ils n'arrivent dans le système.



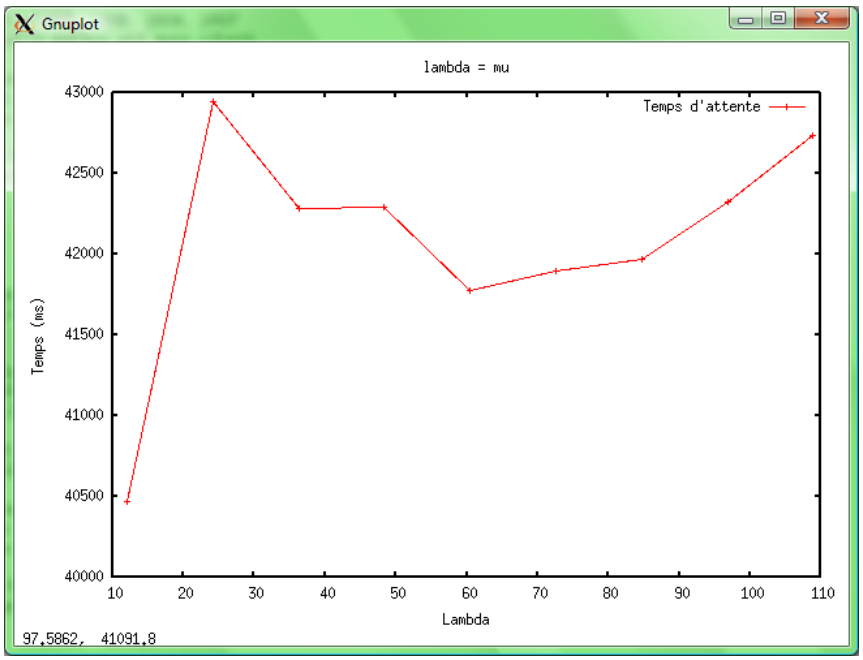
En revanche, dans le cas où lambda est supérieur à mu, les paquets entrent en file d'attente plus rapidement qu'ils ne sortent du système. Aussi, le nombre de paquets en file d'attente augmente de façon constante en fonction de la valeur de lambda.



Lorsque lambda est inférieur à mu, plus lambda croît, plus le temps de traitement est réduit car les paquets mettent moins de temps lors de leur traitement. Le temps d'attente moyen est donc exponentiellement décroissant.



Lorsque lambda est égal à mu, la même loi que précédemment s'applique, mais les paquets mettent plus de temps à être traités car la taille des files d'attente est plus important que précédemment, comme nous l'avons vu ci-avant.



En revanche, lorsque lambda est supérieur à mu, on observe l'effet inverse : comme la taille des files d'attente augmente de façon constante en fonction de lambda, le temps de traitement réduit ne suffit pas à compenser la taille importante des files d'attente. Par conséquent, cette courbe est sensiblement logarithmique.

6. Simulation d'un système M/M/1/2

Un système M/M/1/2 est un système dont la loi d'arrivée des clients est Markovienne (Poissonienne), donc la loi de traitement est Markovienne (Exponentielle), qui possède un seul serveur, et dont la file d'attente (y compris un slot de traitement par serveur) est limité à 2.

Par défaut, sous NS-2, la limitation de files d'attentes se fait en octet et non en nombre de paquets (ici, en nombre de clients). Il faut donc tout d'abord configurer la file d'attente DropTail pour qu'elle soit limitée en nombre de paquets. Pour cela, il faut ajouter au fichier de simulation la ligne :

```
Queue/DropTail set queue-in-bytes_ false
```

De plus, il faut changer la limite de la file d'attente, qui était précédemment de 100000 octets en remplaçant la ligne définissant la taille de la file par :

```
$ns queue-limit $n1 $n2 2
```

Les scripts utilisés pour la simulation sont, autrement, les mêmes que précédemment.

Avec cette limitation, le constat immédiat que nous pouvons faire est que la file d'attente sature rapidement : si lambda est inférieur à mu, il y a très peu de paquets perdus (de l'ordre de 0.1%) ; si lambda est égal à mu, la quantité de paquets perdus est d'environ 2%. En revanches, si lambda est supérieur à mu, alors le temps de traitement est trop important pour que tous les paquets puissent être acceptés dans la file d'attente. En théorie, les pertes sont de l'ordre de 37,5%. En pratique, il s'agit de 30% environ.

Conclusion

Le logiciel NS-2 et son outil de visualisation NAM sont particulièrement adaptés à l'étude de réseaux complexes (filiaires, sans fils) mettant en œuvre de nombreuses files d'attente, procédés de routage, types de trafics…

A travers ce TP, nous avons pu cerner la problématique du choix d'un type d'implémentation de file d'attente par rapport à un autre pour les routeurs et commutateurs d'un réseau. De nombreux systèmes mettent encore en œuvre aujourd'hui des algorithmes FIFO alors qu'ils ne sont pas équitables, mais sont un peu plus simple à calculer que les algorithmes comme SFQ ou CBQ.

NS-2 semble être un bon outil de simulation, et fournir une réelle assistance dans le cas d'élaboration de réseaux complexes en entreprise, pour pouvoir simuler les comportements théoriques des différents éléments du réseau en fonction de leurs propriétés comme la vitesse d'un lien, ou la discipline de file d'attente d'un routeur.