# Conception d'un outil d'analyse et analyse de la trace d'un réseau en C

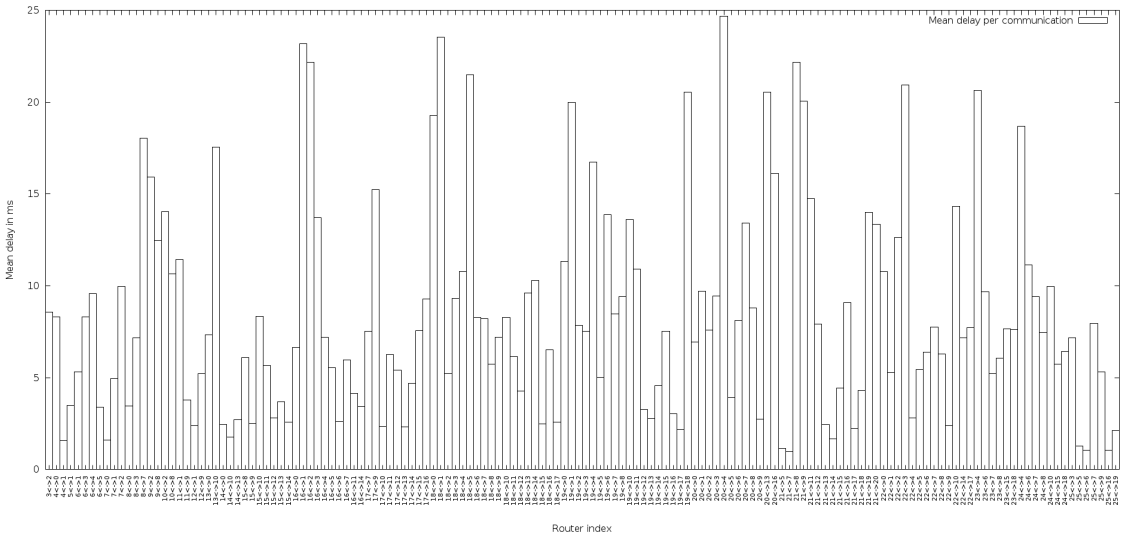
THIBAUT EHLINGER

Vendredi 27 novembre 2015

## ABSTRACT

Ce rapport présente la conception en C d'un utilitaire permettant l'analyse d'un fichier trace d'un réseau fonctionnant pour un certain format de fichier indépendemment de sa taille.

Il y figure aussi quelques pistes concernant les résultats trouvés lors de l'analyse d'un fichier de trafique spécifique fourni par notre encadrant.



# 1 FONCTIONNALITÉS IMPLÉMENTÉES – MANUEL D'UTILISATION

Le sujet du projet était volontairement assez vague, nous laissant le choix dans les fonctionnalités à implémenter. Il était probablement très dur d'implémenter toutes les fonctionnalités demandées, j'ai donc choisi d'implémenter celles "de base" bien sûr.

Puis j'ai implémenté celles qui m'intéressaient, dans le sens où elles me donneraient une plus grande visibilité sur le réseau ayant produit la trace. Il est en effet impossible de visualiser un réseau de cette ampleur à partir d'une simple trace.

## 1.1 Utilisation générale

Pour utiliser l'exécutable, il faut forcément l'appeler avec l'option *-f* suivie du nom du fichier à parser, suivies des éventuelles options d'utilisation.

Exemple :

```
$./trace -f trace2650.txt -p all
```

## 1.2 Options

Voici les statistiques globales qui sont traitées dans ce projet ainsi qu'un manuel d'utilisation. Toutes les options sont compatibles. Mais ne peuvent-être appelées qu'une fois par exécution. Il n'est donc pas possible de tracer un seul flux et de compter les flux lors d'une seule exécution, par exemple.

### 1.2.1 *-f <Nom de fichier>*

Indispensable pour préciser le fichier à parser.

### 1.2.2 *-h*

Affiche une aide d'utilisation

### 1.2.3 *-F all/N(entier)*

- **`all :`** compte le nombre de flux donnés dans le fichier.
  **`attention : compter 3 minutes d'exécution`**
- **`N`** (entier) : trace le flux appelé ayant N pour identifiant.
- Exemples :
  `$./trace -f trace2650.txt -F all`
  `$./trace -f trace2650.txt -F 2342`

### 1.2.4 *-p all/N(entier)*

- **`all :`** réalise un échantillonnage de l'ensemble des paquets injectés dans le réseau toutes les n secondes.
- **`N`** (entier) : trace le paquet appelé ayant N pour identifiant.
- Exemple :
  `$./trace -f trace2650.txt -p 2342`

#### 1.2.5 *-r all/N(entier)*

- **all** : affiche le nombre de paquets reçus par chacun des noeuds du réseau.
- **N** (entier) : trace un échantillonnage du nombre de paquets reçus chaque seconde par un routeur. Cette trace est placée dans le dossier `traces/` et porte le nom de trace_routeur*N*.tr. On peut ensuite modifier le fichier `one_router.gp` pour générer un graphique de cette trace.
- Exemples :
  ```
  $./trace -r trace2650.txt -F all
  $./trace -r trace2650.txt -F 4
  ```

#### 1.2.6 *-l all*

- **all** : Affiche un bilan des communication de bout en bout, pour les paires de routeurs ayant communiqué.
  De plus, créé une trace du nombre de paquets ayant circulé avec succès entre ces routeurs dans `traces/trace_delay.tr`
- Exemple :
  ```
  $./trace -r trace2650.txt -l all
  ```

## 2 MÉMOIRE ET ALGORITHMIQUE

Le fichier fourni est volumineux puisqu'il pèse plus de 140Mo. De plus, l'utilitaire compatible avec des fichiers bien plus volumineux que celui-ci, pouvant dépasser le Go.

Aussi chaque choix de conception influe directement sur la vitesse d'éxécution ainsi que par le volume de mémoire vive monopolisé par celui-ci.

Nous n'allons pas détailler l'ensemble du code fourni, mais simplement commenter les points qui ont nécéssité de faire des choix. Commençons par préciser que le fichier n'est parcouru qu'une seule fois et qu'à part pour la fonctionnalité de comptabilisation de tous les flux, le temps d'exécution est inférieur à 5 secondes.

### 2.1 Statistiques globales et traçage de flux et paquets

Il était trivial de comptabiliser les paquets générés et détruits dans l'ensemble du réseau, il suffisait de compter le nombre de code 0 et 4 du fichier. Pour comptabiliser le nombre de flux c'était différent puisque la mort et la naissance d'un flux n'étaient jamais clairement marqués. La seule solution était de répertorier tous les identifiants déjà rencontrés. J'ai commis l'erreur de les stocker simplement dans une liste chaînée croissante sans doublon, ce qui rend le temps d'exécution du programme quadratique : en effet, pour chaque ligne du fichier, on parcourt en moyenne toute la liste des flux déjà répertoriés. Il eût été plus judicieux d'utiliser une hashtable. Je n'ai plus reproduit cette erreur pour les autres fonctionnalités.

Concernant le traçage d'un flux ou d'un paquet, il suffisait là aussi de parcourir une seule fois tout le fichier (sauf pour les paquets où l'on pouvait même s'arrêter après la mort du paquet), et de répertorier toutes les informations dans une structure de données adaptée.

### 2.2 Statistiques de circulation intra-routeur

Celle-ci est aussi simple puisqu'il suffit de crééer un tableau de la taille du nombre de routeurs (constante dans `global.h`)et d'incrémenter le nombre de destructions et de passage y ayant lieu. Un tableau de deux entiers par routeur faisait l'affaire.

### 2.3 Traçage des délais moyens de communications de bout en bout

Cette fonctionnalité présentait le plus de défis. Toutes les informations concernant les communications de bout en bout d'un noeud à un autre étaient stockées dans une structure `links_charge`. Il est claire que je ne pouvais pas m'y prendre comme je l'avais fait avec les identifiants de flux, puisqu'une liste de listes chaînées aurait immensément accru le temps d'exécution. En effet à chaque réception en un point d'un paquet, il fallait trouver l'évènement correspondant à son émission. Or celui-ci lui étant antérieur, il fallait le stocker. J'ai donc fait deux optimisations :

- Les structures `links_charge` contenant les informations pertinentes sur les différents liens, étaient stocké dans un tableau de la taille de $\frac{N \times (N-1)}{2}$ éléments. Une fonction `get_hash(i,j)` permettant de renvoyer l'indice de l'élément correspondant à la communication entre i et j. Cette fonction de hash se base sur la linéarisation d'une demi-matrice.
- Il n'est donc pas nécessaire de parcourir tous les `links_charge` pour trouver lequel correspond à la ligne du fichier analysé. L'autre optimisation concerne les paquets : une fois qu'on trouve une réception/destruction de paquet, on stocke la durée du voyage puis on supprime le paquet du buffer de paquets dans `links_charge`. Ainsi on ne fait que très peu de comparaisons pour trouver où stocker les paquets émis par un noeud.

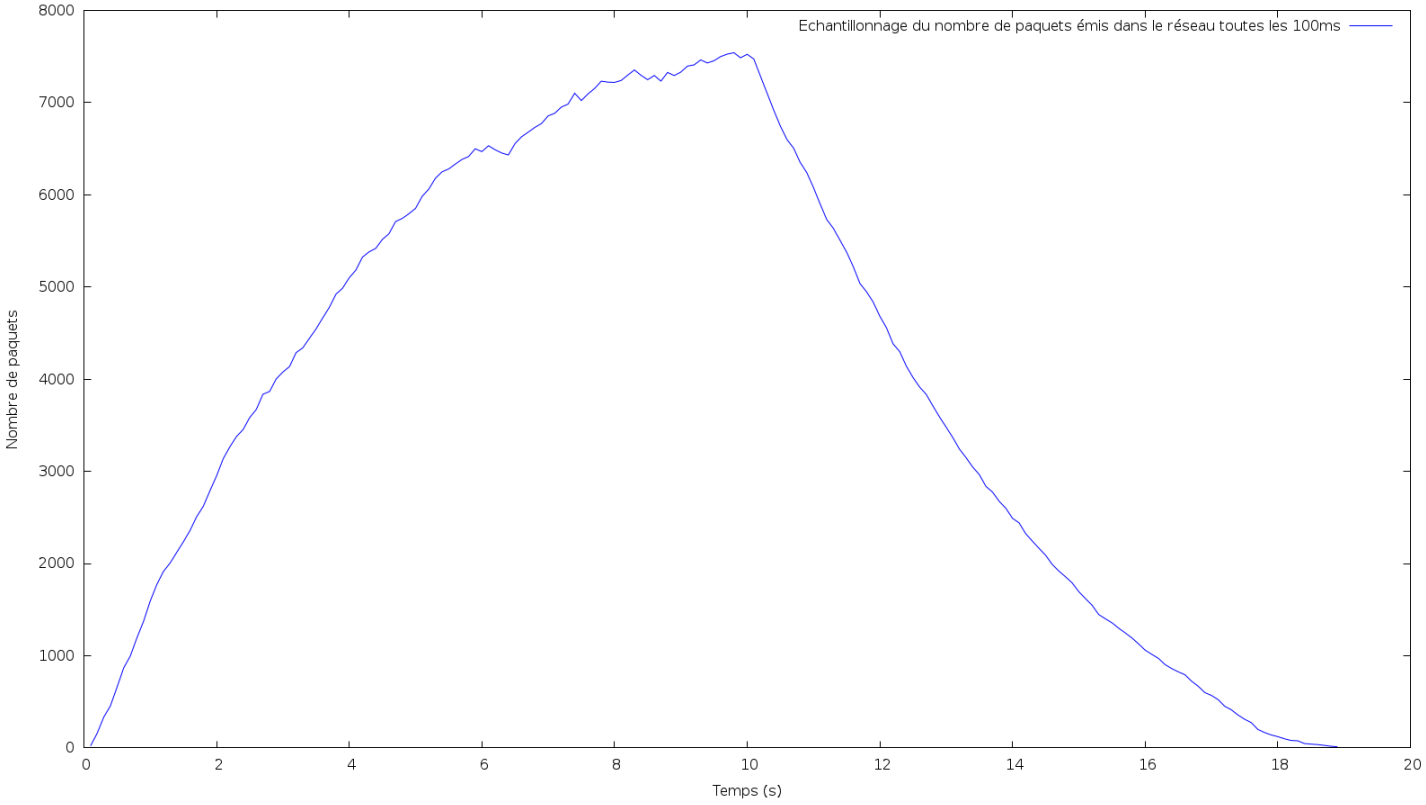



**Figure 1:** Echantillonnage du nombre de paquets émis toutes les 100ms

# 3 RÉSULTATS ET EXPLOITATION

Les données collectées concernent trois aspects du réseau :

- Ses données globales
- Les données concernant les routeurs (et non les liens)
- Les données concernant les communications de bout en bout

## 3.1 Données globales

L'exécutable indique que durant les 20 secondes d'exécution, **750 979** ont été émis dans le réseau, dont **35 164** ont été détruits. De plus la figure **??** nous montre la répartition de ces émissions au fil du temps. On constate que l'émission des paquets augmente plutôt linéairement au cours du temps, puis qu'elle s'écroule complètement aux alentours de la $11^{eme}$ seconde, formant une gigantesque dent de scie.

De cela on peut d'ores et déjà déduire une chose : ce réseau n'est pas un réseau uniforme régit par des *Constant Bit Rates* avec des liens et des files de routeurs surdimensionnées. En effet une telle topologie peut signifier plusieurs choses :

- soit d'une **congestion** d'un routeur du réseau, qui serait détectée par les émetteurs et qui arrêteraient alors d'émettre.

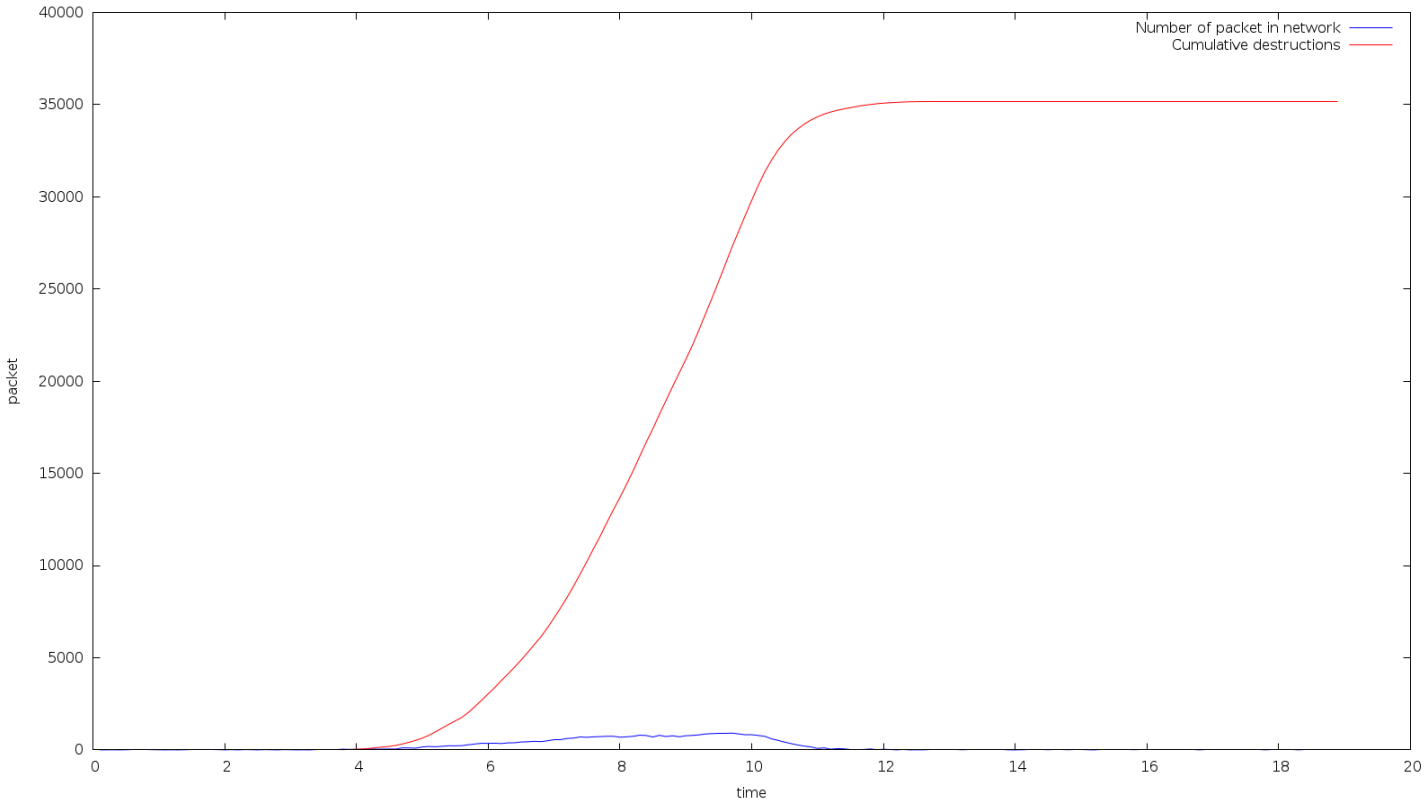



**Figure 2:** Echantillonnage du cumul des destructions au cours du temps superposé à la courbe de la Figure **??**

- soit d'une syncronisation de tous les émetteurs. Peut-être parce qu'ils émettent tous en fonction d'un même algorithme TCP synchronisé par exemple.
- soit d'un trafic généré par une loi "en dent de scie". On en observerait alors simplement une dent dans son intégralité, ce qui serait pertinent comme trace de réseau.

La figure **??** nous permet néanmoins d'avancer sur la question. En effet on constate qu'à partit de la sixième seconde et jusqu'à la onzième, le nombre de destructions cumulé passe de moins de 5 000 à plus de 35 000 (soit multiplié par sept environ!), puis il stagne. On peut comprendre que quelque chose se passe sur le réseau. S'il y a destruction, cela signifie qu'un routeur au moins reçoit plus de paquets que ce qu'il pourrait en supporter. La mission consistera donc à déceler le potentiel coupable.

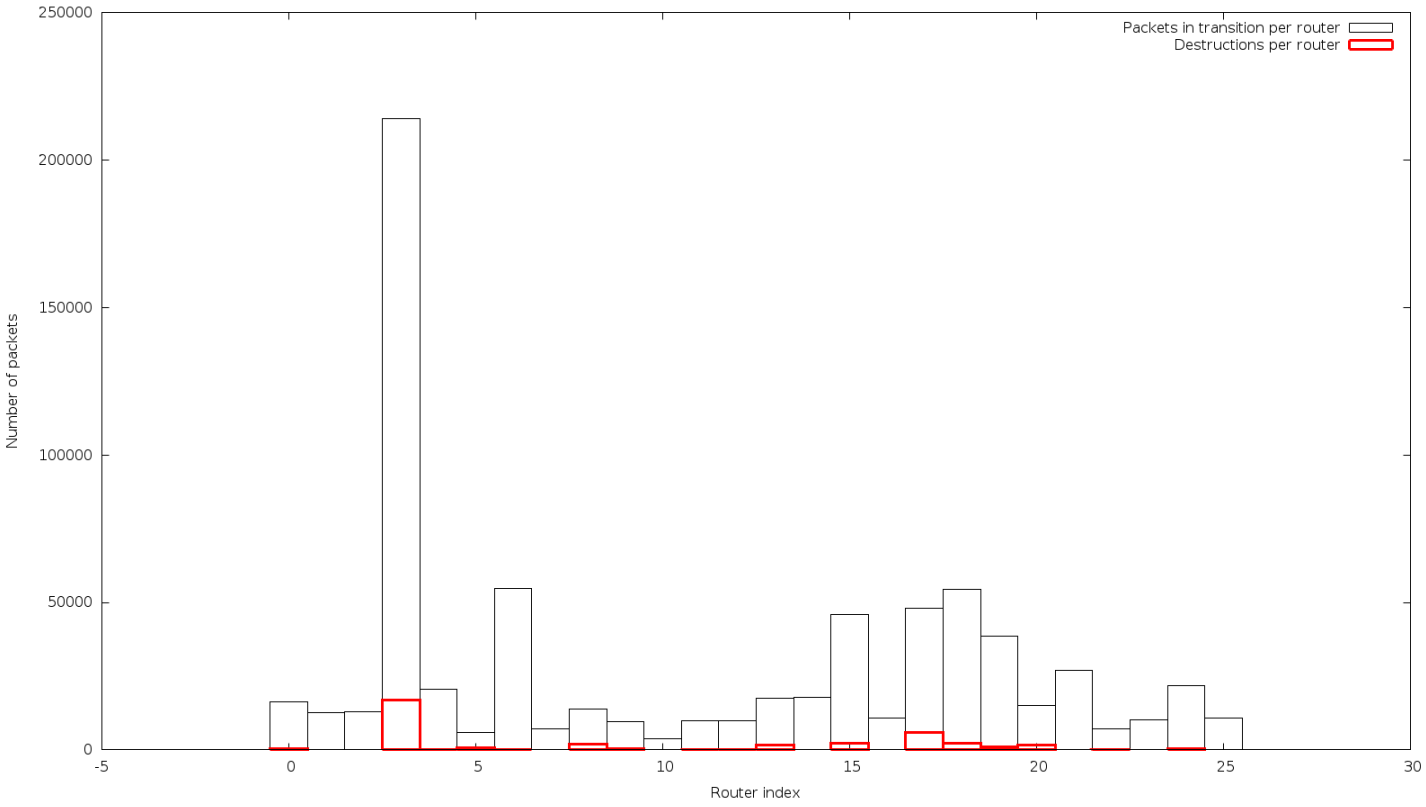


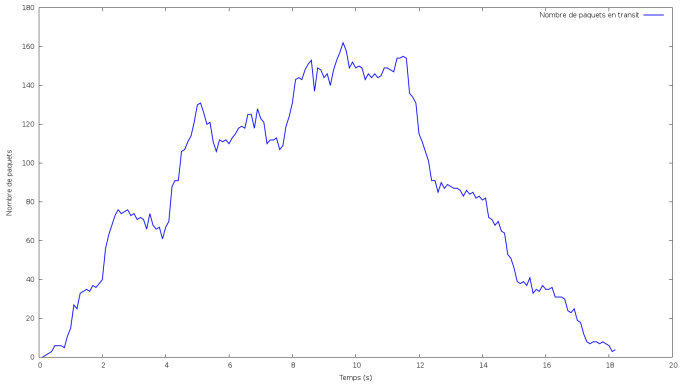

**Figure 3:** Diagramme du nombre de paquets en transit (et non reçus) dans chacun des routeurs du réseau

## 3.2 Données des noeuds du réseau

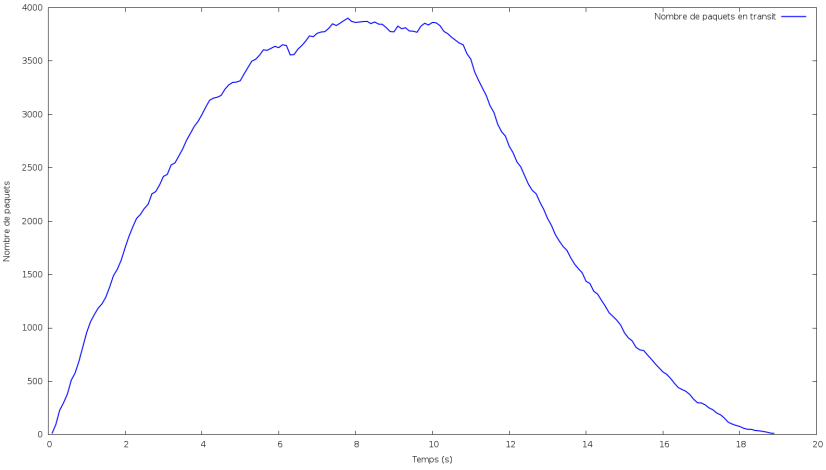
La réponse au problème paraît évidente dans la figure **??** : le routeur numéro 3 voit passer 213 889 paquets à lui seul, et en reçoit 419 524. En tapant la commande :

```
./read -f trace2650.txt -m res26.txt -r all
```

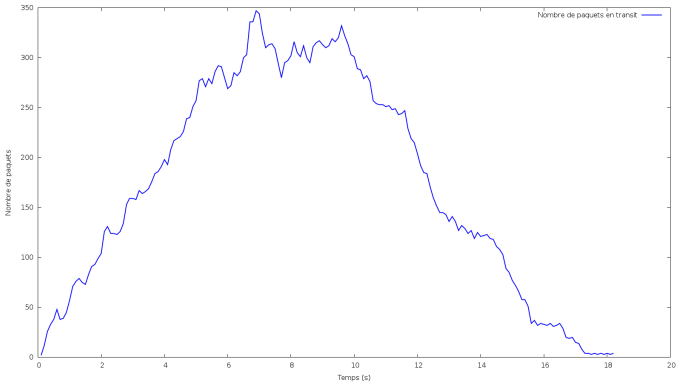
On voit qu'il est à lui seul responsable de 16 878 destructions, soit environ la **moitié** des destructions répertoriées dans le réseau! Ce graphique témoigne donc d'une grande disparité dans la charge des différents routeurs du réseau.



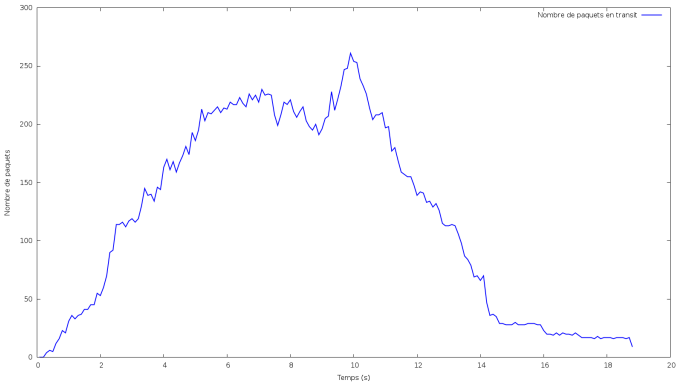

**(a)** Routeur 1




**(b)** Routeur 3




**(c)** Routeur 6




**(d)** Routeur 8

**Figure 4:** Add your own figures before compiling