Évaluation de performance TCP avec NS-2

Projet ÉvalPerf 2025–2026

Mehdi JAFARIZADEH
21920385

25 novembre 2025

Résumé

Ce projet étudie, par simulation ns-2, les performances de plusieurs variantes de TCP (NewReno, Sack1, Vegas) dans une topologie en haltère soumise à différents scénarios de congestion. Les traces sont analysées par des scripts AWK et Python afin de mesurer le débit par flux, l'équité (indice de Jain), la taille moyenne et maximale de la file au goulet, le nombre de paquets perdus ainsi que le *goodput* et le surcoût des retransmissions.

Les premières expériences montrent l'impact du RTT et de la concurrence entre flux sur le débit moyen, puis comparent le comportement des variantes TCP : NewReno et Sack1 remplissent fortement la file (plus de 700 paquets en moyenne) et génèrent de nombreuses pertes, alors que Vegas maintient une file très courte au prix d'un débit légèrement plus faible. L'étude des politiques de file met en évidence un compromis important entre DropTail et RED : un DropTail correctement dimensionné (50–200 paquets) permet d'atteindre presque 100 % de la capacité du goulet avec un nombre de pertes modéré, tandis qu'un RED mal paramétré limite effectivement la taille de la file mais provoque davantage de drops et une sous-utilisation du lien.

Enfin, sur un lien cœur de 10 Gb/s, Sack1 parvient à exploiter environ 97 % de la capacité, contre 76–81 % pour NewReno et Vegas, ce qui illustre la meilleure scalabilité de TCP avec SACK sur les liens très haut débit. L'ajout de trafic de fond UDP et de flux TCP supplémentaires (partie 1.5B) réduit progressivement le débit total TCP (de 10 à 6 Mb/s) et augmente la taille de la file et le nombre de pertes, tout en maintenant un surcoût de retransmission inférieur à 1 %. Ces résultats soulignent l'importance du choix de la variante TCP et du dimensionnement des files pour garantir de bonnes performances dans les réseaux congestionnés.

Table des matières

1 Introduction **3**

2 Topologie et paramètres de simulation **3**
2.1 Topologie générale 3
2.2 Trafic généré 4
2.3 Mesures et post-traitement 4
2.4 Paramètres communs 4

3 Partie 1.1 – Partage du goulet entre 6 flux TCP **4**
3.1 Objectif 4
3.2 Scénario et méthodologie 5
3.3 Résultats 5
3.4 Analyse 6

4 Partie 1.2 – Effet de la congestion et du RTT **7**
4.1 Objectif . . . 7
4.2 Scénarios et méthodologie . . . 7
4.3 Résultats . . . 8
4.4 Analyse . . . 9

5 Partie 1.3 – Comparaison des versions TCP **10**
5.1 Objectif . . . 10
5.2 Scénarios et méthodologie . . . 10
5.3 Résultats . . . 11
5.3.1 Débit par flux et débit total . . . 11
5.3.2 Files d'attente et pertes au goulet . . . 11
5.3.3 Équité entre flux . . . 11
5.4 Analyse . . . 12

6 Partie 1.4 – Influence de la taille de file et de l'algorithme RED **13**
6.1 Objectif . . . 13
6.2 Scénarios et méthodologie . . . 14
6.3 Résultats . . . 15
6.3.1 Débit total et équité . . . 15
6.3.2 Pertes au goulet . . . 16
6.3.3 Taille moyenne et maximale de la file . . . 17
6.4 Analyse . . . 17

7 Partie 1.5A – Scalabilité des variantes TCP sur un lien 10 Gbps **19**
7.1 Objectif . . . 19
7.2 Scénario et méthodologie . . . 19
7.3 Résultats . . . 20
7.4 Analyse . . . 20

8 Partie 1.5B – Impact du trafic de fond et de flux supplémentaires **21**
8.1 Objectif . . . 21
8.2 Scénarios et méthodologie . . . 22
8.3 Résultats . . . 22
8.3.1 Débit brut et goodput . . . 22
8.3.2 Taille de la file et pertes au goulet . . . 23
8.3.3 Surcoût des retransmissions . . . 24
8.4 Analyse . . . 25

A Scripts Tcl **26**

B Scripts AWK et Python **27**
B.1 Exemple AWK : débit moyen par flux TCP . . . 27
B.2 Exemple AWK : statistiques de file au goulet . . . 27
B.3 Exemple Python : génération d une figure de débit . . . 28

1 Introduction

Les performances des protocoles de transport, et en particulier de TCP, jouent un rôle central dans le dimensionnement et l'exploitation des réseaux modernes. Le comportement de contrôle de congestion (congestion control) conditionne non seulement le débit obtenu sur un lien donné, mais aussi le partage de la bande passante entre flux concurrents, la taille des files d'attente dans les routeurs et, par conséquent, la latence perçue par les applications. L'objectif de ce projet est d'étudier expérimentalement, à l'aide d'un simulateur, plusieurs aspects de ce comportement : équité entre flux, impact de l'hétérogénéité des RTT, influence de la variante TCP utilisée (NewReno, Sack1, Vegas), effet de la taille de file et des algorithmes de gestion de file (DropTail, RED), ainsi que la capacité de ces variantes à exploiter des liens à très haut débit.

Pour cela, nous utilisons le simulateur à événements discrets NS-2, qui permet de décrire précisément une topologie, d'instancier différents agents TCP/UDP et de tracer l'ensemble des événements (émission, réception, pertes de paquets, etc.). Les fichiers de trace produits par NS-2 sont ensuite post-traités en deux étapes : (i) des scripts AWK extraient des métriques agrégées telles que le débit moyen par flux, l'occupation moyenne et maximale de la file au goulet, le nombre de pertes, ou encore l'indice d'équité de Jain ; (ii) des scripts PYTHON lisent ces fichiers texte et construisent les graphiques présentés dans ce rapport (à l'aide de `matplotlib`). Cette séparation entre extraction numérique et visualisation permet de garder un pipeline reproductible et de vérifier facilement la cohérence des résultats.

Le rapport est structuré de la manière suivante. La Section 1 décrit la topologie de base utilisée et les paramètres de simulation communs à l'ensemble des scénarios. Les sections suivantes présentent ensuite les résultats détaillés des différentes parties du projet : partage du goulet entre six flux TCP, effet de la congestion et du RTT, comparaison des variantes TCP, impact de la taille de file et de RED, puis étude de la scalabilité sur un lien à 10 Gbps avec et sans trafic de fond. Chaque partie est organisée selon le même schéma : rappel de l'objectif, description du scénario NS-2, présentation des métriques et des figures obtenues, puis analyse et interprétation des résultats.

2 Topologie et paramètres de simulation

2.1 Topologie générale

Sauf mention contraire, l'ensemble des expériences repose sur une topologie en « haltère » (dumbbell) comportant huit nœuds. Trois nœuds sources n_0, n_1 et n_2 sont connectés à un routeur cœur n_3 via des liens d'accès à haut débit, et trois nœuds destinations n_5, n_6 et n_7 sont connectés à un second routeur cœur n_4. Le lien entre n_3 et n_4 constitue le *goulet d'étranglement* partagé par tous les flux TCP étudiés.

Dans le scénario de base (Parties 1.1 à 1.4), les liens d'accès (n_0, n_3), (n_1, n_3) et (n_2, n_3), ainsi que (n_4, n_5), (n_4, n_6) et (n_4, n_7), ont un débit de 100 Mb/s et des délais de propagation respectifs de 5, 15 et 30 ms. Les mêmes valeurs sont utilisées symétriquement de l'autre côté du goulet, de sorte que les flux correspondants $(n_0 \to n_5)$, $(n_1 \to n_6)$ et $(n_2 \to n_7)$ présentent des RTT hétérogènes. Le lien cœur (n_3, n_4) est configuré à 10 Mb/s avec un délai de 10 ms et une file d'attente `DropTail` de taille limitée (`queue-limit`) dont la valeur est ajustée selon la partie (par exemple 10, 50, 200 ou 1000 paquets en Partie 1.4).

Dans la Partie 1.5, dédiée aux liens à très haut débit, la même topologie logique est réutilisée mais la capacité du lien goulet est portée à 10 Gb/s et les paramètres des agents TCP (taille de fenêtre, taille de segment) sont adaptés afin d'explorer la scalabilité des différentes variantes.

2.2 Trafic généré

Dans la plupart des scénarios, nous considérons six flux TCP longue durée, mis en place entre les paires (n_0, n_5), (n_1, n_6) et (n_2, n_7), avec deux flux par paire. Chaque flux est modélisé par un agent `TCP` associé à une application `FTP` générant un trafic de type « saturation » (always-on). Les flux démarrent simultanément à $t = 0{,}5$ s et restent actifs jusqu'à la fin de la simulation, de façon à observer un régime permanent sur le goulet. Selon la partie, nous utilisons successivement différentes variantes TCP (`NewReno`, `Sack1`, `Vegas`) en gardant la même topologie.

Pour étudier l'impact d'un trafic de fond non contrôlé, certaines expériences introduisent en plus un flux `CBR/UDP` sur le goulet, avec un débit constant (par exemple 2 ou 4 Mb/s en Partie 1.5B). Enfin, dans quelques scénarios, le nombre de flux TCP est augmenté (jusqu'à 9 flux) afin d'observer l'effet de la concurrence accrue sur l'équité et sur l'occupation de la file.

2.3 Mesures et post-traitement

NS-2 produit deux types de traces utilisés dans ce travail : (i) un fichier de trace global (`.tr`) contenant l'ensemble des événements de chaque paquet (émission « + », arrivée « - », réception « r », perte « d »), et (ii) pour le lien goulet, un fichier de trace de file (`trace-queue`) permettant de suivre précisément le nombre de paquets en attente.

À partir de ces traces, plusieurs scripts AWK calculent les métriques suivantes :

— le débit moyen de chaque flux TCP sur l'intervalle de mesure, ainsi que le débit total agrégé ;
— la taille moyenne et maximale de la file au goulet, et le nombre de paquets perdus (*drops*) ;
— l'indice d'équité de Jain appliqué aux débits des flux ;
— dans les parties haut débit, la distinction entre débit brut et *goodput*, ainsi que le pourcentage de débit consacré aux retransmissions.

Les valeurs obtenues sont enregistrées dans des fichiers texte synthétiques (par exemple `thr_*.txt`, `queue_*.txt`, `worst_*.txt`). Ces fichiers servent ensuite d'entrée à des scripts PYTHON qui génèrent l'ensemble des figures de ce rapport (diagrammes en barres, courbes débit–RTT, etc.) à l'aide de la bibliothèque `matplotlib`. Ce flux de travail automatisé (NS-2 → AWK → Python) facilite la reproductibilité des expériences et permet de vérifier rapidement la cohérence entre les différentes parties du projet.

2.4 Paramètres communs

3 Partie 1.1 – Partage du goulet entre 6 flux TCP

3.1 Objectif

L'objectif de cette première partie est de mettre en place une topologie simple mais suffisamment riche pour observer le partage d'un goulet d'étranglement entre plusieurs flux TCP de longue durée. On souhaite en particulier :

— construire une topologie à 8 nœuds avec un lien cœur jouant le rôle de goulet (*bottleneck*) partagé ;
— créer six flux `TCP`/`FTP` concurrents, deux par paire source–destination ;
— introduire des RTT hétérogènes entre les différentes paires afin de préparer l'étude de l'impact du RTT sur l'équité TCP dans les parties suivantes ;
— valider la chaîne de mesure (traces NS-2, scripts AWK, visualisation sous Python) avant de passer à des scénarios plus complexes. :contentReference[oaicite :0]index=0

3.2 Scénario et méthodologie

Topologie. La topologie retenue est une structure en haltère à huit nœuds. Les nœuds n_0, n_1 et n_2 jouent le rôle de sources à gauche, les nœuds n_5, n_6 et n_7 sont les destinations à droite, et deux routeurs cœur n_3 et n_4 sont interconnectés par un lien commun qui constitue le goulet d'étranglement. Chaque source est connectée à n_3 et chaque destination à n_4 par un lien d'accès à haut débit. :contentReference[oaicite :1]index=1

Les paramètres des liens sont les suivants :

— liens d'accès « feuilles → cœur » à 100 Mb/s avec des délais de propagation différents pour créer des RTT hétérogènes :
  — n_0–n_3 et n_4–n_5 : 100 Mb/s, 5 ms ;
  — n_1–n_3 et n_4–n_6 : 100 Mb/s, 15 ms ;
  — n_2–n_3 et n_4–n_7 : 100 Mb/s, 30 ms ;
— lien cœur n_3–n_4 à 10 Mb/s, délai 10 ms, avec une file `DropTail` limitée à 50 paquets dans chaque sens (`queue-limit`). :contentReference[oaicite :2]index=2

Ainsi, les RTT approximatifs des trois paires (n_0, n_5), (n_1, n_6) et (n_2, n_7) sont respectivement faibles, moyens et élevés (environ $5 + 10 + 5$, $15 + 10 + 15$ et $30 + 10 + 30$ ms), ce qui permet de tester le comportement de TCP dans un environnement hétérogène.

Flux TCP. Six flux TCP de longue durée sont configurés, deux par paire source–destination :

— flux fid 0 et fid 1 : $n_0 \to n_5$ (RTT faible) ;
— flux fid 2 et fid 3 : $n_1 \to n_6$ (RTT moyen) ;
— flux fid 4 et fid 5 : $n_2 \to n_7$ (RTT élevé).

Chaque flux est un agent `TCP/NewReno` attaché à une application `FTP` de type *bulk transfer*, démarrée à $t = 0{,}5$ s et active jusqu'à la fin de la simulation (100 s). Pour faciliter l'analyse a posteriori, le champ `fid_` de chaque agent TCP est initialisé avec un identifiant unique (0 à 5), ce qui permet de filtrer les événements par flux dans le fichier de trace NS-2. :contentReference[oaicite :3]index=3

Mesure du débit et tracés. Le simulateur génère un fichier de trace global `part1_1.tr`. Un script AWK comptabilise, pour chaque `fid`, le volume total de données utiles reçues pendant la fenêtre de mesure et en déduit le débit moyen en Mb/s. Les valeurs résultantes sont rassemblées dans un fichier synthétique `thr_part1_1.txt`. :contentReference[oaicite :4]index=4 Ensuite, un script Python lit ce fichier et produit le diagramme en barres présenté à la Figure 1, en utilisant la bibliothèque `matplotlib`.

3.3 Résultats

Le tableau 1 résume les débits moyens obtenus pour chaque flux TCP, tels qu'extraits de `thr_part1_1.txt`, et la Figure 1 en donne une représentation graphique. :contentReference[oaicite :5]index=5

TABLE 1 – Débit moyen par flux TCP (Partie 1.1).

Flux (fid)	Débit moyen [Mb/s]
0	2,459
1	2,458
2	1,532
3	1,524
4	0,980
5	0,979

On observe que les deux flux associés au RTT le plus faible (fid 0 et 1) atteignent chacun un débit d'environ 2,46 Mb/s, ceux au RTT intermédiaire (fid 2 et 3) environ 1,53 Mb/s, et ceux au RTT le plus élevé (fid 4 et 5) environ 0,98 Mb/s. La somme des six débits est d'environ 9,94 Mb/s, très proche de la capacité théorique du goulet à 10 Mb/s, ce qui confirme que le lien cœur est bien le facteur limitant et qu'il est pratiquement saturé pendant la phase stationnaire. :contentReference[oaicite :6]index=6

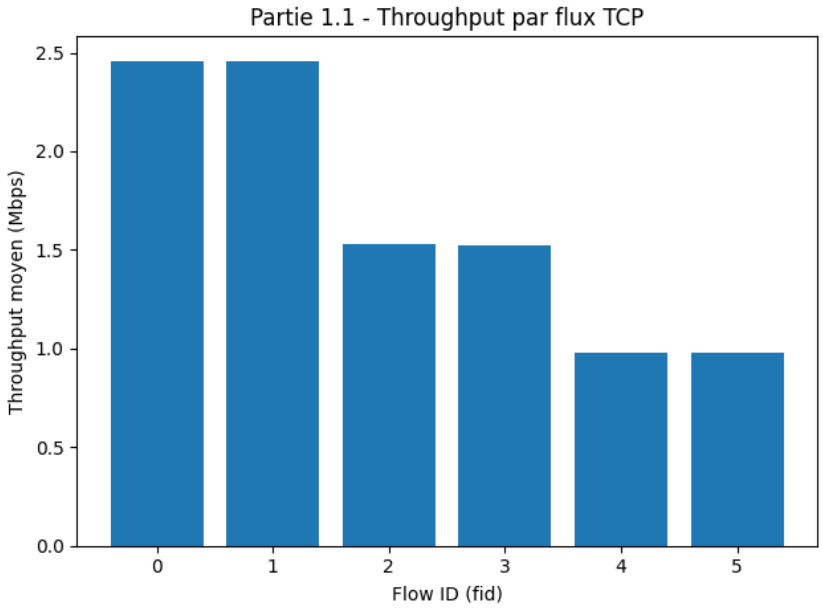

FIGURE 1 – Débit moyen par flux TCP (Partie 1.1).

3.4 Analyse

Cette première expérience remplit deux rôles. D'une part, elle valide la mise en place de la topologie et de la chaîne de post-traitement : la présence de six `fid` distincts dans les traces, la saturation du lien cœur et la cohérence des débits obtenus montrent que le scénario est correctement configuré. D'autre part, elle met déjà en évidence l'influence du RTT sur le partage de la bande passante entre flux TCP concurrents.

En effet, les deux flux empruntant le chemin au RTT le plus faible obtiennent un débit significativement plus élevé que ceux avec RTT moyen, eux-mêmes avantagés par rapport aux flux au RTT le plus long. Ce comportement est cohérent avec l'algorithme de contrôle de congestion de TCP : à perte donnée sur le goulet, un flux à RTT court parcourt son cycle *additive increase/-multiplicative decrease* plus rapidement et augmente donc plus souvent sa fenêtre de congestion, ce qui lui permet d'envoyer davantage de données sur l'intervalle de temps considéré.

On remarque également que, pour une même paire source–destination, les deux flux associés (par exemple fid 0 et 1) obtiennent quasiment le même débit, ce qui indique une bonne équité entre flux partageant le même RTT et la même route. Les différences observées entre paires seront étudiées plus en détail dans la Partie 1.2, qui se concentre spécifiquement sur le lien entre RTT, congestion et équité.

4 Partie 1.2 – Effet de la congestion et du RTT

4.1 Objectif

Cette partie vise à analyser plus finement l'impact du temps de trajet aller–retour (RTT) sur le partage de la bande passante entre flux TCP, en distinguant deux situations :

- un cas où le lien cœur reste un goulet d'étranglement saturé pendant toute la simulation (*cas avec congestion*) ;
- un cas où la capacité du lien cœur est suffisamment élevée pour qu'il n'y ait plus de congestion ni de limitation par la fenêtre de réception TCP (*cas sans congestion*). :contentReference[oaicite :0]index=0

L'idée est de montrer que le biais RTT bien connu de TCP NewReno est particulièrement marqué lorsque les flux se partagent un goulet commun, et qu'il devient beaucoup plus faible lorsque la congestion disparaît.

4.2 Scénarios et méthodologie

Topologie et RTT. Les deux scénarios reprennent la même topologie en haltère que celle de la Partie 1.1 (nœuds n_0 à n_7 et lien cœur n_3–n_4). Les liens d'accès sont configurés à 100 Mb/s avec des délais de propagation différents : 5 ms pour n_0 et n_5, 15 ms pour n_1 et n_6, 30 ms pour n_2 et n_7. Le lien cœur a un délai de 10 ms. On en déduit trois RTT approximatifs pour les paires (n_0, n_5), (n_1, n_6) et (n_2, n_7) :

$$\mathrm{RTT}_1 \approx 40 \text{ ms}, \quad \mathrm{RTT}_2 \approx 80 \text{ ms}, \quad \mathrm{RTT}_3 \approx 140 \text{ ms}.$$

Comme en Partie 1.1, six flux `TCP/Newreno/FTP` de longue durée sont créés, deux par paire, avec des identifiants `fid` de 0 à 5.

Cas avec congestion. Dans le premier script (`part1_2_congestion.tcl`), le lien cœur n_3–n_4 est configuré à 10 Mb/s, avec une file `DropTail` de 50 paquets dans chaque sens, ce qui en fait le goulet d'étranglement de la topologie. :contentReference[oaicite :3]index=3 Tous les flux FTP démarrent simultanément à $t = 0{,}5$ s et restent actifs jusqu'à $t = 200$ s afin d'observer une congestion stable.

Cas sans congestion. Dans le second script (`part1_2_no_congestion.tcl`), l'objectif est au contraire de supprimer tout goulet sur le lien cœur. Après quelques essais intermédiaires (capacité trop faible conduisant encore à une saturation, ou trop élevée provoquant un temps de simulation excessif), la capacité du lien n_3–n_4 a été fixée à 300 Mb/s, soit strictement supérieure à la somme des trois liens d'accès (3×100 Mb/s). Pour éviter que la fenêtre TCP elle-même ne devienne limitante, les paramètres suivants sont utilisés pour tous les agents TCP :

```
$tcp set window_ 2000
$tcp set maxcwnd_ 2000
```

Les autres éléments de la topologie et du trafic (six flux FTP de longue durée, RTT hétérogènes) restent identiques au cas précédent. :contentReference[oaicite :5]index=5

Mesure du débit et équité. Dans les deux scénarios, un fichier de trace global est produit et un script AWK calcule, pour chaque `fid`, le volume total de données reçues au niveau des nœuds n_5, n_6 et n_7. Le débit moyen *goodput* est obtenu en divisant ce volume par la durée de la simulation (200 s) et en le convertissant en Mb/s ; les résultats sont stockés dans `thr_congestion.txt` et `thr_no_congestion.txt`. L'indice d'équité de Jain est ensuite calculé à partir de ces six débits :

$$J = \frac{\left(\sum_{i=0}^{5} x_i\right)^2}{6 \times \sum_{i=0}^{5} x_i^2},$$

où x_i désigne le débit du flux i.

Les valeurs numériques sont enfin visualisées à l'aide de scripts Python, sous forme d'un diagramme en barres comparant les débits par `fid` (avec/sans congestion) et d'un nuage de points représentant le débit moyen en fonction du RTT pour les deux scénarios.

4.3 Résultats

Débit par flux. Le Tableau 2 présente les débits obtenus dans les deux cas.

TABLE 2 – Débit moyen par flux TCP avec et sans congestion (Partie 1.2).

Flux (fid)	Cas congestion [Mb/s]	Cas sans congestion [Mb/s]
0	2,464	39,852
1	2,463	40,210
2	1,537	38,332
3	1,533	34,965
4	0,985	29,309
5	0,984	30,260

Dans le cas avec congestion, la somme des six débits est d'environ 9,97 Mb/s, soit pratiquement la capacité du lien cœur à 10 Mb/s : le goulet est donc saturé en permanence. :contentReference[oaicite :8]index=8 Dans le cas sans congestion, la somme des débits atteint environ 213 Mb/s, bien en deçà des 300 Mb/s du lien cœur, ce qui confirme que ce dernier n'est plus limitant. :contentReference[oaicite :9]index=9

La Figure 2 illustre ces valeurs sous forme de diagramme en barres, tandis que la Figure 3 montre le débit moyen en fonction du RTT pour les deux scénarios.

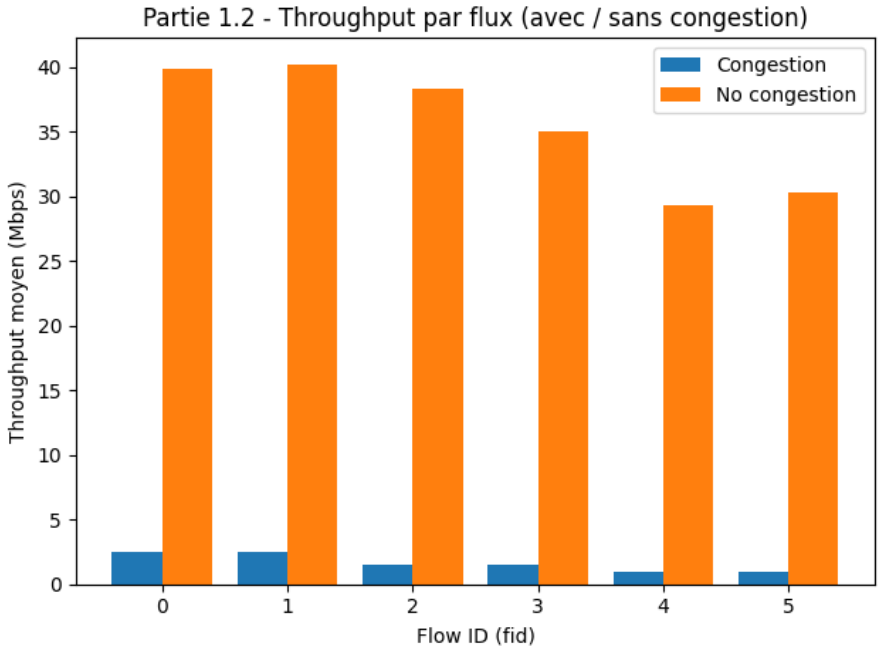

FIGURE 2 – Débit moyen par flux avec et sans congestion (Partie 1.2).

Équité. À partir des débits de la Table 2, on obtient un indice de Jain d'environ

$$J_{\text{congestion}} \approx 0{,}881, \qquad J_{\text{sans cong.}} \approx 0{,}985.$$

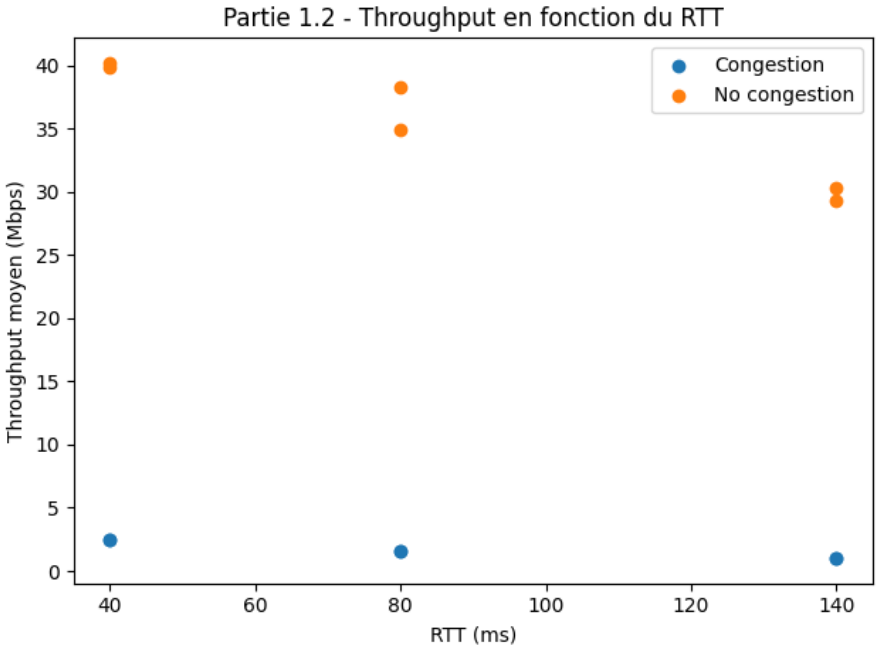

FIGURE 3 – Débit moyen en fonction du RTT, avec et sans congestion.

Le premier cas révèle donc une équité imparfaite entre flux, tandis que le second est très proche de l'équité parfaite (valeur 1). :contentReference[oaicite :10]index=10

4.4 Analyse

Biais RTT en présence de congestion. Dans le cas où le lien cœur est un goulet à 10 Mb/s, les flux associés au RTT le plus court (fid 0 et 1, RTT $\approx$ 40 ms) obtiennent chacun un débit d'environ 2,46 Mb/s, contre seulement $\approx$ 0,98 Mb/s pour les flux au RTT le plus long (fid 4 et 5, RTT $\approx$ 140 ms). :contentReference[oaicite :11]index=11 Autrement dit, un flux à RTT court reçoit plus de 2,5 fois la bande passante d'un flux à RTT long, ce qui se traduit par un indice d'équité de Jain limité à 0,88.

Ce comportement est cohérent avec la dynamique du contrôle de congestion TCP NewReno : la phase d'augmentation additive de la fenêtre de congestion dépend du nombre de RTT écoulés. Un flux à RTT court parcourt plus de cycles *additive increase/multiplicative decrease* par unité de temps, ce qui lui permet d'augmenter plus fréquemment sa fenêtre et donc d'occuper une fraction plus importante du goulet lorsqu'il est saturé.

Atténuation du biais en l'absence de goulet. Dans le cas sans congestion, le lien cœur n'est plus saturé et les pertes y sont négligeables ; de même, la fenêtre TCP n'est plus limitante grâce à la valeur élevée de `window_` et `maxcwnd_`. Les flux disposent donc de ressources suffisantes pour atteindre un débit compatible avec leur RTT. Les flux à RTT court restent avantagés (en moyenne $\approx$ 39,9 Mb/s contre $\approx$ 29,7 Mb/s pour ceux à RTT long), mais l'écart relatif est nettement plus faible (facteur $\approx$ 1,3 au lieu de 2,5) et l'indice d'équité grimpe à 0,985.

On peut interpréter ces résultats de la façon suivante : en l'absence de goulet commun, le débit de chaque flux est essentiellement déterminé par la relation de type $x \approx$ cwnd/RTT, avec une fenêtre congestion similaire pour tous. Les différences de RTT induisent encore des différences de débit, mais celles-ci restent modérées et n'engendrent plus une forte iniquité globale.

Conclusion de la Partie 1.2. La comparaison des deux scénarios montre que le manque d'équité entre flux TCP de RTT différents apparaît principalement lorsqu'ils se partagent un

lien congested commun. Lorsque le goulet est supprimé et que la fenêtre TCP est suffisamment grande, le biais RTT persiste mais devient beaucoup plus faible et l'équité globale du système est presque parfaite. Cette observation est importante pour la conception des réseaux : dans les environnements fortement congestionnés, on peut s'attendre à ce que les flux à RTT court dominent la bande passante, tandis que dans les réseaux bien dimensionnés, l'influence du RTT sur le débit reste limitée.

5 Partie 1.3 – Comparaison des versions TCP

5.1 Objectif

Après avoir étudié l'impact du RTT et de la congestion, cette partie vise à comparer différentes variantes de TCP dans un environnement où tous les flux voient exactement les mêmes conditions réseau. Les objectifs sont les suivants :

- homogénéiser les RTT de tous les chemins pour éliminer l'effet du RTT ;
- comparer le débit, l'occupation de la file du goulet et le nombre de pertes pour plusieurs versions de TCP (`NewReno`, `Sack1`, `Vegas`) ;
- analyser l'équité entre flux à l'aide de l'indice de Jain ;
- étudier un scénario mixte où des flux `NewReno` et `Vegas` coexistent et se partagent le même goulet.

Initialement, le sujet proposait également `TCP/Cubic`, mais cette variante n'est pas disponible dans la version de NS-2 utilisée. Nous l'avons donc remplacée par `TCP/Sack1`, qui est supportée nativement. :contentReference[oaicite :0]index=0

5.2 Scénarios et méthodologie

Topologie et paramètres. La topologie reste une structure en haltère à huit nœuds, comme dans les parties précédentes, mais les RTT sont rendus homogènes : tous les liens d'accès entre les sources (n_0, n_1, n_2) et le routeur n_3, ainsi qu'entre le routeur n_4 et les destinations (n_5, n_6, n_7), ont un débit de 100 Mb/s et un délai de 10 ms. Le lien cœur n_3–n_4 est configuré à 10 Mb/s avec un délai de 10 ms, et une file `DropTail` de taille limitée à 50 paquets (`queue-limit 50`) dans chaque sens. Ainsi, tous les flux TCP traversent le même goulet, avec un RTT identique.

Comme dans les parties précédentes, six flux `TCP`/`FTP` de longue durée sont créés :

- deux flux entre n_0 et n_5 (fid 0 et 1) ;
- deux flux entre n_1 et n_6 (fid 2 et 3) ;
- deux flux entre n_2 et n_7 (fid 4 et 5).

Tous les flux démarrent à $t = 0{,}5$ s et restent actifs jusqu'à la fin de la simulation ($t = 200$ s). La fenêtre TCP est dimensionnée de sorte qu'elle ne soit pas limitante (`window_ = maxcwnd_ = 2000`).

Quatre scénarios étudiés. Quatre fichiers Tcl distincts sont utilisés pour faciliter l'analyse :

1. *NewReno* : les six flux sont des agents `TCP/Newreno` ;
2. *Sack1* : les six flux sont des agents `TCP/Sack1`, avec des récepteurs `TCPSink/Sack1` afin d'exploiter correctement l'option SACK ;
3. *Vegas* : les six flux sont des agents `TCP/Vegas` ;
4. *Mix* : trois flux `NewReno` et trois flux `Vegas` coexistent sur le même goulet (par exemple fid 0–2 en NewReno, fid 3–5 en Vegas).

Chaque scénario produit un fichier de trace global (`.tr`) pour le calcul du débit, et un fichier de trace de file pour le lien $n_3 \to n_4$ (`_queue.tr`).

Mesures. À partir des traces globales, des scripts AWK calculent le débit moyen de chaque flux (en Mb/s) sur la fenêtre de mesure, puis la somme des six débits. Les valeurs sont stockées dans les fichiers `thr_newreno.txt`, `thr_sack.txt`, `thr_vegas.txt` et `thr_mix_newreno_vegas.txt`. L'indice de Jain est ensuite calculé pour chaque scénario à partir de ces six débits.

À partir des traces de file du goulet, un autre script AWK reconstruit la longueur de la file au cours du temps et en déduit :

— la taille moyenne de la file (queue moyenne) ;
— la taille maximale observée (queue max) ;
— le nombre de paquets perdus sur le goulet (événements `d`).

Les résultats sont regroupés dans quatre fichiers de synthèse : `queue_summary_newreno.txt`, `queue_summary_sack.txt`, `queue_summary_vegas.txt` et `queue_summary_mix.txt`, dont les valeurs principales sont rappelées ci-dessous.

5.3 Résultats

5.3.1 Débit par flux et débit total

La Figure 4 montre le débit moyen de chacun des six flux pour les quatre scénarios, et la Figure 5 le débit total agrégé.

Dans le scénario **NewReno**, les six flux obtiennent des débits très proches, compris entre 1,59 et 1,67 Mb/s, pour une somme d'environ 9,86 Mb/s, soit quasi la capacité du goulet à 10 Mb/s. L'indice de Jain est très proche de 1 ($J \approx 0{,}999$), ce qui traduit une excellente équité.

Dans le scénario **Sack1**, les débits par flux sont également très homogènes (entre 1,30 et 1,35 Mb/s) avec un indice de Jain pratiquement égal à 1 ($J \approx 0{,}9999$), mais le débit total n'atteint qu'environ 7,96 Mb/s : dans cette configuration, SACK exploite légèrement moins la capacité du goulet que NewReno.

Pour **Vegas**, le débit total remonte à environ 9,96 Mb/s, mais la répartition entre flux devient moins équitable : deux flux atteignent environ 2,43 Mb/s, tandis que les quatre autres restent autour de 1,27 Mb/s ; l'indice de Jain tombe à $J \approx 0{,}90$.

Enfin, dans le scénario **Mix** (3 NewReno + 3 Vegas), le débit total reste voisin de la capacité du goulet ($\approx 9{,}94$ Mb/s), mais les flux NewReno dominent largement : ils obtiennent entre 2,17 et 3,02 Mb/s, alors que les flux Vegas sont limités à 0,53–0,72 Mb/s. L'indice de Jain chute à $J \approx 0{,}71$, ce qui révèle une forte iniquité.

5.3.2 Files d'attente et pertes au goulet

Les résumés de file obtenus sur le lien $n_3 \to n_4$ sont les suivants :

— **NewReno** : queue moyenne = 26,26 paquets, queue max = 50 paquets, drops = 1257 ;
— **Sack1** : queue moyenne = 26,32 paquets, queue max = 50 paquets, drops = 1254 ;
— **Vegas** : queue moyenne = 36,13 paquets, queue max = 51 paquets, drops = 862 ;
— **Mix** : queue moyenne = 36,13 paquets, queue max = 51 paquets, drops = 862.

Les Figures 6 et 7 représentent les files maximales et moyennes, tandis que la Figure 8 donne le nombre de paquets perdus.

On constate que la taille maximale de la file reste, pour tous les scénarios, très proche de la limite imposée (50–51 paquets). En revanche, la *queue moyenne* et le nombre de pertes varient sensiblement : NewReno et Sack1 maintiennent en moyenne une file d'environ 26 paquets, avec plus de 1200 pertes, alors que Vegas et le scénario Mix présentent une file moyenne plus élevée (environ 36 paquets) mais un nombre de pertes nettement plus faible (≈ 860).

5.3.3 Équité entre flux

La Figure 9 représente l'indice de Jain calculé sur les six flux pour chaque scénario. On retrouve numériquement les observations précédentes :

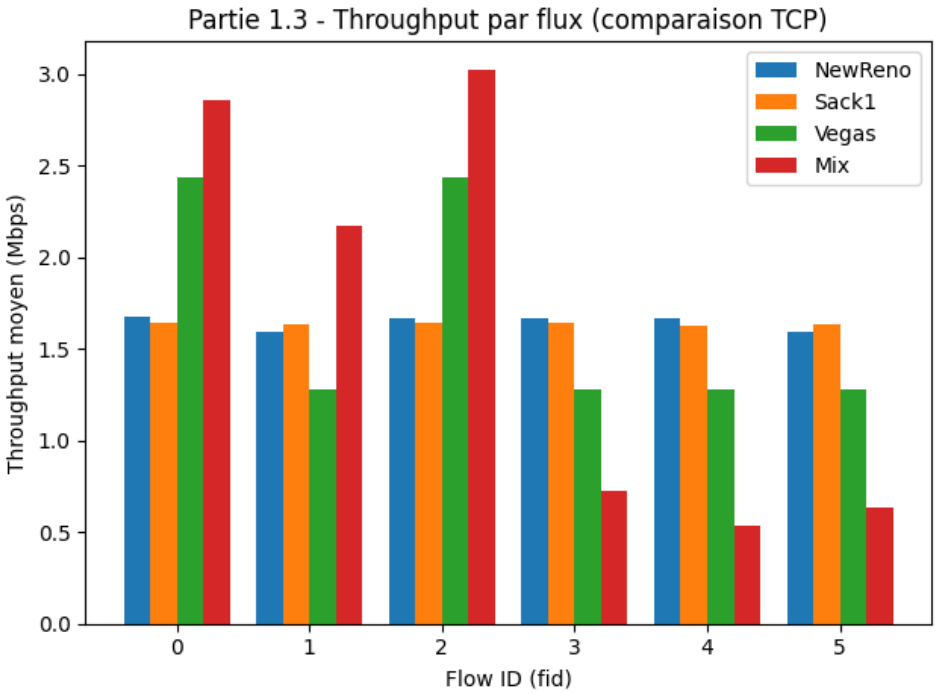

FIGURE 4 – Débit moyen par flux pour chaque version de TCP (Partie 1.3).

— **NewReno** et **Sack1** présentent une équité presque parfaite ($J \approx 1$) : tous les flux obtiennent un débit similaire ;
— **Vegas** reste relativement équitable ($J \approx 0{,}90$), mais avec une disparité visible entre flux ;
— le scénario **Mix** est nettement inéquitable ($J \approx 0{,}71$), les flux NewReno dominant les flux Vegas.

5.4 Analyse

Lorsque tous les flux utilisent la même variante de TCP et que les RTT sont homogènes, les scénarios **NewReno** et **Sack1** présentent un comportement très proche : le lien goulet est bien saturé (surtout pour NewReno), l'équité entre flux est excellente, mais au prix d'une file d'attente relativement chargée et de nombreuses pertes. Ces variantes *loss-based* remplissent la file jusqu'à la limite, puis réagissent aux pertes pour réduire la fenêtre de congestion, ce qui explique les oscillations autour d'une queue moyenne d'environ 25 paquets et les plus de 1200 drops.

Le scénario **Vegas**, de type *delay-based*, adopte un comportement différent : il maintient la file autour d'un niveau cible (queue moyenne plus élevée, mais plus stable), et réduit ainsi significativement le nombre de pertes sur le goulet. En revanche, la répartition de la bande passante entre les six flux est moins homogène : certains flux gardent un débit élevé tandis que d'autres restent durablement en dessous, ce qui se traduit par un indice de Jain plus faible.

Le cas **Mix** met en évidence la difficulté de faire coexister des variantes loss-based et delay-based : les flux NewReno, insensibles au délai, continuent à pousser la file vers sa limite et prennent l'essentiel de la capacité, tandis que les flux Vegas, plus prudents, réduisent leur fenêtre dès que le délai augmente et se retrouvent fortement pénalisés. On obtient ainsi un débit total proche de la capacité du goulet, mais au prix d'une équité très faible (Jain $\approx 0{,}71$) et d'un comportement que l'on peut juger inacceptable dans un environnement multipartage.

En résumé, cette partie montre que, dans un contexte de RTT homogènes, NewReno et Sack1 assurent un partage équitable mais génèrent beaucoup de pertes, tandis que Vegas réduit

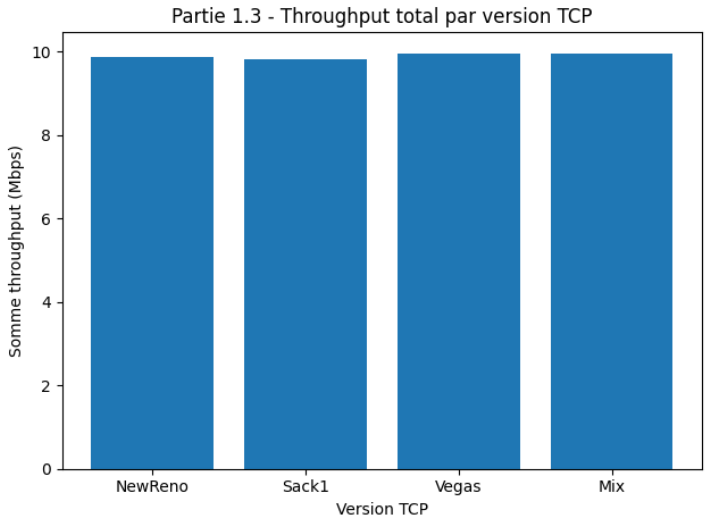

FIGURE 5 – Débit total au goulet pour chaque version de TCP.

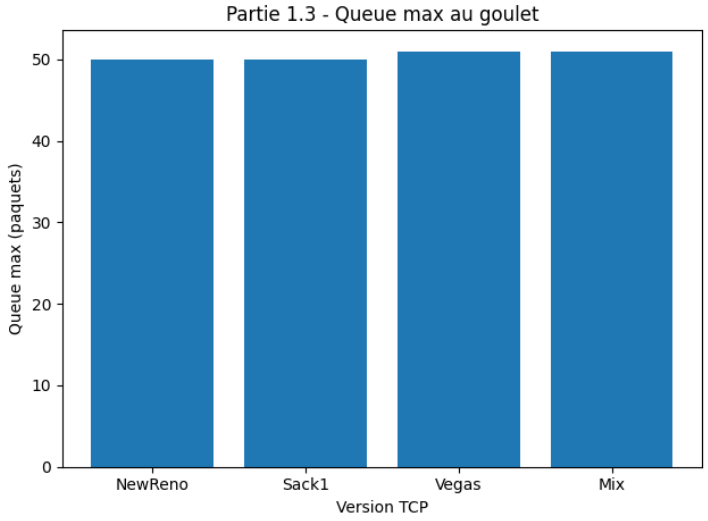

FIGURE 6 – Taille maximale de la file au goulet.

les pertes au prix d'une équité moindre. Lorsque différentes variantes coexistent, les flux delay-based comme Vegas sont désavantagés face aux variantes loss-based, ce qui est un point important à garder en tête pour le déploiement de nouveaux algorithmes TCP dans l'Internet.

6 Partie 1.4 – Influence de la taille de file et de l'algorithme RED

6.1 Objectif

Dans cette partie, nous nous intéressons à l'impact de la *taille de la file d'attente* et du *mécanisme de gestion de file* sur les performances des flux TCP au niveau du lien goulet. Nous évaluons plus précisément :

— le débit total obtenu sur le goulet et l'équité entre les flux (indice de Jain) ;
— la taille moyenne et maximale de la file au goulet ;
— le nombre de paquets perdus (*drops*) sur ce lien.

Deux types de files sont comparés :

— **DropTail** avec différentes valeurs de `queue-limit` (10, 50, 200, 1000 paquets) ;

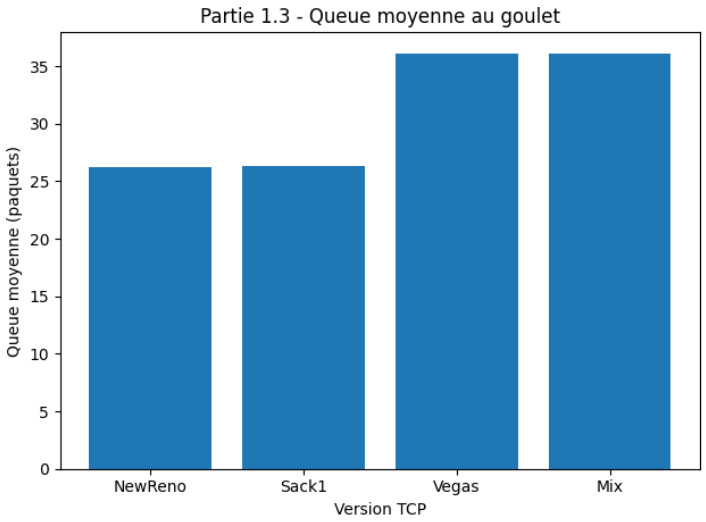

FIGURE 7 – Taille moyenne de la file au goulet.

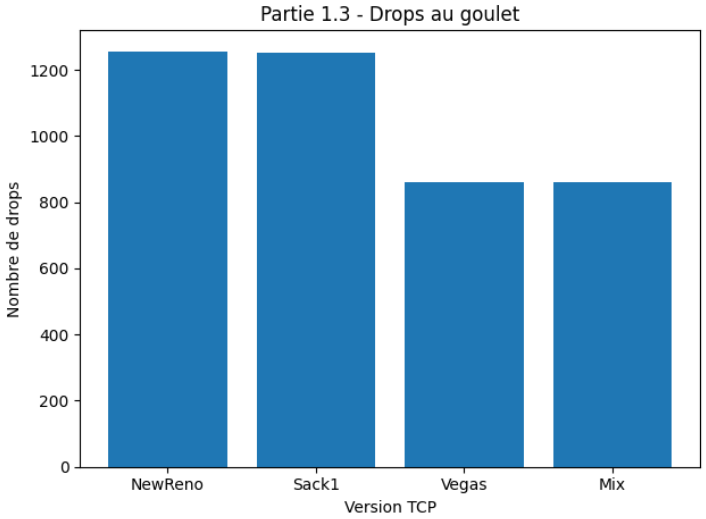

FIGURE 8 – Nombre de paquets perdus au goulet pour chaque scénario.

— **RED** (Random Early Detection) avec deux tailles de file (50 et 200 paquets).

L'objectif est de mettre en évidence le compromis entre utilisation de la capacité, taux de pertes et délai de mise en file, et de montrer qu'un mauvais paramétrage de RED peut conduire à des performances inférieures à celles de DropTail. :contentReference[oaicite :0]index=0

6.2 Scénarios et méthodologie

Topologie et trafic. La topologie est identique à celle de la Partie 1.3, avec des RTT homogènes pour tous les flux. Les liens d'accès entre les sources et le routeur n_3, ainsi qu'entre le routeur n_4 et les destinations, sont configurés à 100 Mb/s avec un délai de 10 ms. Le lien cœur n_3–n_4 constitue le goulet d'étranglement : il est configuré à 10 Mb/s avec un délai de 10 ms.

Six flux `TCP/NewReno/FTP` de longue durée traversent le goulet, comme dans les parties précédentes. Ils démarrent à $t = 0{,}5$ s et restent actifs jusqu'à $t = 200$ s, de sorte que l'on observe un régime permanent bien établi. :contentReference[oaicite :1]index=1

Configurations de file. Deux scripts Tcl distincts sont utilisés :

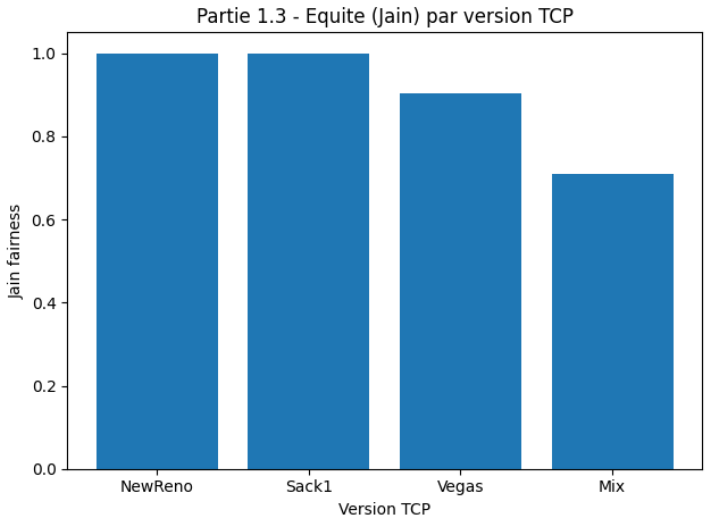

FIGURE 9 – Indice d'équité de Jain pour chaque version de TCP.

— `part1_4_droptail.tcl` : le lien n_3–n_4 utilise une file `DropTail` et seul le paramètre `queue-limit` est modifié entre les simulations (10, 50, 200, 1000 paquets) ;
— `part1_4_red.tcl` : le lien n_3–n_4 utilise une file `RED` ; la valeur de `queue-limit` est fixée à 50 ou 200 paquets, et les seuils `minth` et `maxth` sont choisis de manière simple et proportionnelle à `qlim` (sans réglage fin des paramètres de RED). :contentReference[oaicite :2]index=2

Mesures et post-traitement. Pour chaque simulation, un fichier de trace global permet de calculer, à l'aide d'un script AWK, le débit moyen de chaque flux (en Mb/s) et le débit total au niveau du goulet. L'indice d'équité de Jain est ensuite calculé à partir des six débits.

Par ailleurs, un traceur de file (`trace-queue`) est activé sur le lien $n_3 \to n_4$. Un second script AWK reconstruit la longueur instantanée de la file à partir des événements « + » (entrée en file), « - » (sortie de file) et « d » (drop) et en déduit :

— la taille moyenne de la file (*queue moyenne*) ;
— la taille maximale observée (*queue max*) ;
— le nombre de paquets perdus sur le goulet (*drops*).

Les valeurs agrégées sont stockées dans les fichiers `queue_droptail_*.txt` et `queue_red_*.txt` et utilisées par un script Python pour générer les figures de cette partie (total throughput, drops, fairness, queue moyenne et queue max). :contentReference[oaicite :3]index=3

6.3 Résultats

6.3.1 Débit total et équité

La Figure 10 présente le débit total obtenu sur le goulet pour les six configurations. Pour DropTail, on observe :

— **DT-10** : débit total ≈ 8,69 Mb/s (sous-utilisation du lien) ;
— **DT-50** : débit total ≈ 9,86 Mb/s ;
— **DT-200** : débit total ≈ 9,95 Mb/s ;
— **DT-1000** : débit total ≈ 9,97 Mb/s.

Pour RED :

— **RED-50** : débit total ≈ 8,87 Mb/s ;
— **RED-200** : débit total ≈ 8,84 Mb/s.

Dans tous les scénarios, l'indice de Jain est très proche de 1, ce qui indique un partage très équitable de la bande passante entre les six flux (les RTT et les routes étant identiques). Seul le cas DropTail avec `qlim` très grand (1000 paquets) montre une légère dégradation de l'équité (Jain $\approx 0{,}97$). :contentReference[oaicite :4]index=4

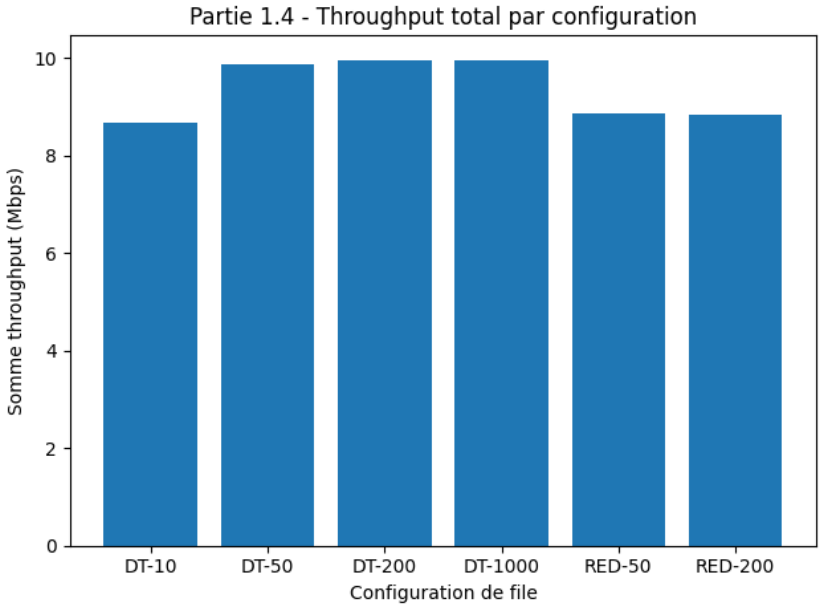

FIGURE 10 – Débit total au goulet en fonction de la configuration de file (Partie 1.4).

La Figure 11 montre l'indice de Jain pour chaque configuration.

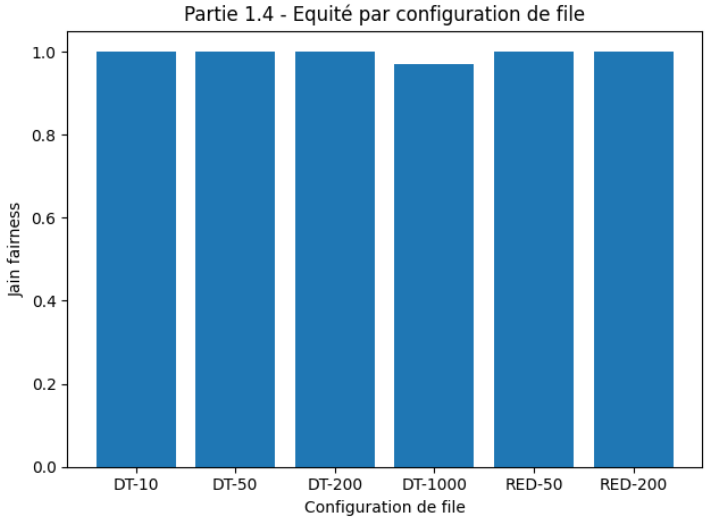

FIGURE 11 – Indice d'équité de Jain selon la configuration de file.

6.3.2 Pertes au goulet

Les valeurs de *drops* extraites des fichiers de synthèse sont les suivantes :

— **DropTail** :
  — DT-10 : 1954 paquets perdus ;
  — DT-50 : 1257 paquets perdus ;

— DT-200 : 537 paquets perdus ;
— DT-1000 : 813 paquets perdus.

— **RED** :
— RED-50 : 2574 paquets perdus ;
— RED-200 : 2606 paquets perdus.

Ces valeurs sont représentées à la Figure 12. On voit que, pour DropTail, l'augmentation de la taille de file de 10 à 200 paquets réduit fortement le nombre de pertes, puis que les pertes remontent légèrement pour `qlim = 1000`. Pour RED, le nombre de drops reste très élevé, de l'ordre de 2600 paquets, et ce de manière quasi indépendante de `queue-limit`.

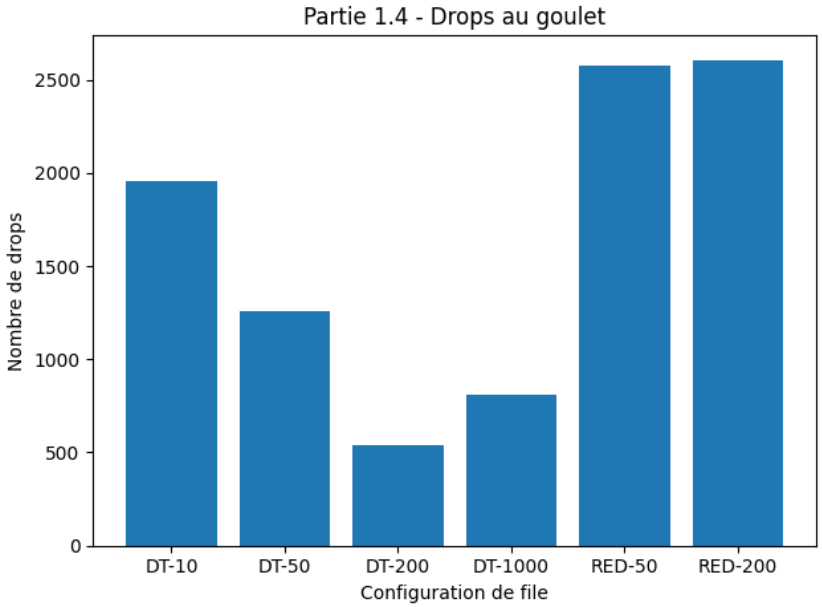

FIGURE 12 – Nombre de paquets perdus au goulet pour chaque configuration.

6.3.3 Taille moyenne et maximale de la file

Les mesures de file au goulet $n_3 \to n_4$ donnent les résultats suivants.

Pour **DropTail** :
— DT-10 : queue moyenne = 3,15 paquets, queue max = 10 paquets ;
— DT-50 : queue moyenne = 26,26 paquets, queue max = 50 paquets ;
— DT-200 : queue moyenne = 141,82 paquets, queue max = 200 paquets ;
— DT-1000 : queue moyenne = 714,69 paquets, queue max = 1000 paquets.

Pour **RED** :
— RED-50 : queue moyenne = 4,82 paquets, queue max = 26 paquets ;
— RED-200 : queue moyenne = 4,84 paquets, queue max = 54 paquets.

Les Figures 13 et 14 montrent respectivement la taille maximale et la taille moyenne de la file pour chaque configuration.

6.4 Analyse

Taille de file DropTail : compromis débit / pertes / délai. Les résultats montrent d'abord que la taille de file a un impact significatif sur les performances en DropTail :

— Avec un buffer très petit (**DT-10**), la file est souvent pleine, ce qui conduit à un grand nombre de drops (1954 paquets) et à un débit total nettement inférieur à la capacité du

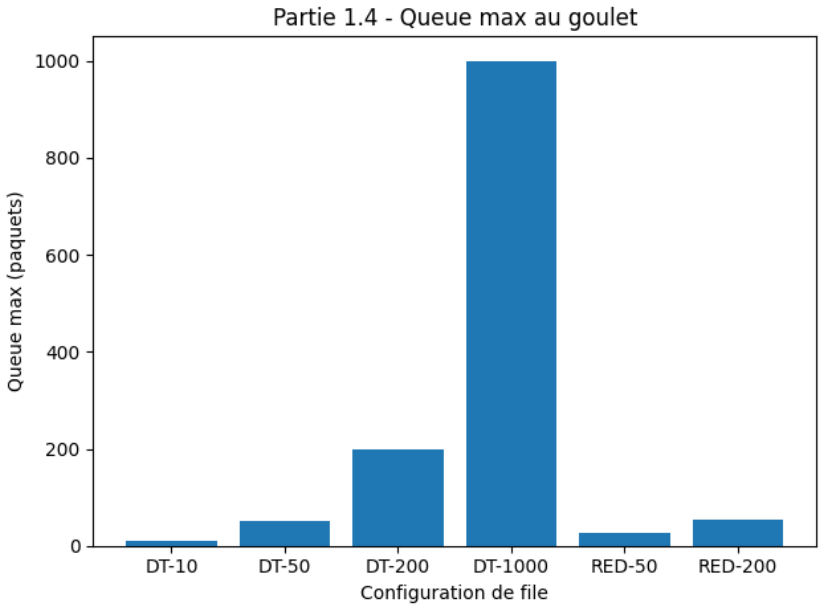

FIGURE 13 – Taille maximale de la file au goulet selon la configuration de file.

goulet (8,69 Mb/s). TCP réduit fréquemment sa fenêtre de congestion et le lien n'est pas pleinement exploité.

- En augmentant la taille de file à **50** puis **200** paquets, la file peut absorber davantage de rafales, ce qui réduit fortement les pertes (1257 puis 537 drops) et permet d'atteindre un débit quasi maximal ($\approx$ 9,95 Mb/s).
- Lorsque la file devient très grande (**DT-1000**), le débit reste excellent ($\approx$ 9,97 Mb/s), mais la queue moyenne dépasse 700 paquets. Cela correspond à un délai de mise en file très important, typique du phénomène de *bufferbloat* : le lien est bien utilisé, mais au prix d'une forte latence.

Ainsi, un buffer trop petit conduit à une sous-utilisation du lien et à beaucoup de pertes, tandis qu'un buffer trop grand engendre un délai excessif sans bénéfice significatif sur le débit.

Comparaison RED vs DropTail. Les scénarios RED présentent un comportement très différent :

- la taille moyenne de la file reste très faible (environ 5 paquets), quel que soit `qlim` (50 ou 200), et la taille maximale observée est nettement inférieure à la limite physique (26 ou 54 paquets) ;
- en contrepartie, le nombre total de drops est très élevé (près de 2600 paquets) et le débit total reste autour de 8,8 Mb/s, donc inférieur aux meilleures configurations DropTail.

Cela reflète la philosophie de RED : introduire des pertes précoces et probabilistes pour éviter que la file ne se remplisse complètement. Dans notre configuration, les paramètres de RED (minth, maxth, probabilité maximale) ont été choisis de façon simple et non optimisée ; le résultat est que RED commence à dropper trop tôt et trop souvent, ce qui maintient la file très courte (faible délai) mais réduit l'utilisation de la capacité du lien.

Équité. Dans tous les scénarios, l'indice de Jain reste proche de 1 : la topologie symétrique et les RTT identiques pour tous les flux font que la nature de la file (DropTail ou RED) n'a que peu d'influence sur l'équité entre flux. Les différences observées portent donc essentiellement sur le compromis entre *throughput total*, *taux de pertes* et *délai de file*.

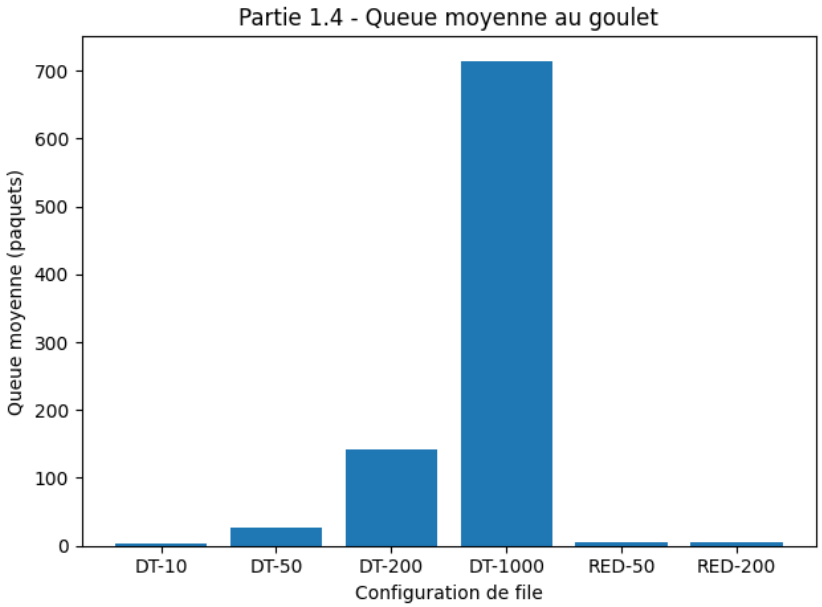

FIGURE 14 – Taille moyenne de la file au goulet selon la configuration de file.

Bilan de la Partie 1.4. Cette étude montre qu'il existe une taille de file « raisonnable » pour DropTail (ici de l'ordre de 50 à 200 paquets) qui permet de combiner :

— une utilisation quasi maximale du lien ;
— un nombre de pertes modéré ;
— un délai de mise en file acceptable.

À l'inverse, une file trop petite dégrade le débit et une file trop grande engendre un bufferbloat important. L'algorithme RED, dans la configuration simple utilisée ici, réussit à maintenir la file courte et à limiter le délai, mais au prix d'un très grand nombre de pertes et d'un débit total plus faible que celui obtenu avec un DropTail correctement dimensionné. Cela illustre le fait que, pour que RED soit réellement avantageux, un réglage fin de ses paramètres est indispensable.

7 Partie 1.5A – Scalabilité des variantes TCP sur un lien 10 Gbps

7.1 Objectif

Les parties précédentes se concentraient sur un goulet à 10 Mb/s. Dans cette sous-partie, nous cherchons à savoir dans quelle mesure différentes variantes de TCP sont capables d'exploiter un *lien cœur à très haut débit* (10 Gbps). Plus précisément, nous comparons trois versions de TCP – `NewReno`, `Sack1` et `Vegas` – en termes de débit total obtenu sur le lien et donc d'utilisation de la capacité disponible. :contentReference[oaicite :0]index=0

7.2 Scénario et méthodologie

La topologie logique reste celle en haltère déjà utilisée : six flux `TCP/FTP` de longue durée traversent un lien cœur partagé. Les liens d'accès sont configurés à un débit très élevé, de façon à ne pas constituer de goulet. Le lien cœur n_3–n_4 est porté à une capacité de 10 Gbps, avec un RTT de quelques millisecondes. :contentReference[oaicite :1]index=1

Pour chaque variante TCP, un script Tcl distinct est utilisé (`part1_5_newreno.tcl`, `part1_5_sack.tcl`, `part1_5_vegas.tcl`). Afin d'éviter que la fenêtre d'émission ne limite le débit, la taille de segment (`packetSize_`) et la fenêtre de congestion maximale (`window_`, `maxcwnd_`) sont fixées à des valeurs très élevées. Tous les flux démarrent à $t = 0{,}5$ s et restent actifs jusqu'à la fin de la

simulation, de sorte que l'on observe un régime stationnaire sur le lien 10 Gbps. :contentReference[oaicite :2]index=2

À la fin de chaque simulation, le volume total de données reçues par les six récepteurs est divisé par la durée de simulation, ce qui fournit le *débit global* sur le lien (exprimé en Mb/s). Ces valeurs sont enregistrées dans trois fichiers texte : `thr_part1_5_newreno.txt`, `thr_part1_5_sack.txt` et `thr_part1_5_vegas.txt`. Un script Python lit ensuite ces fichiers, convertit les débits en Gb/s et génère deux graphiques :

— le débit total par variante TCP (Figure 15) ;
— le pourcentage d'utilisation de la capacité du lien (Figure 16).

7.3 Résultats

Les débits globaux mesurés sont les suivants :
— **NewReno** : 7669,3 Mb/s ≈ 7,67 Gb/s ; :contentReference[oaicite :4]index=4
— **Sack1** : 9693,4 Mb/s ≈ 9,69 Gb/s ; :contentReference[oaicite :5]index=5
— **Vegas** : 8121,2 Mb/s ≈ 8,12 Gb/s. :contentReference[oaicite :6]index=6
Rapportés à la capacité de 10 Gbps du lien cœur, ces valeurs correspondent à :
— environ **76–77 %** de la capacité pour NewReno ;
— environ **96–97 %** de la capacité pour Sack1 ;
— environ **81 %** de la capacité pour Vegas.
Ces résultats sont représentés sur les Figures 15 et 16.

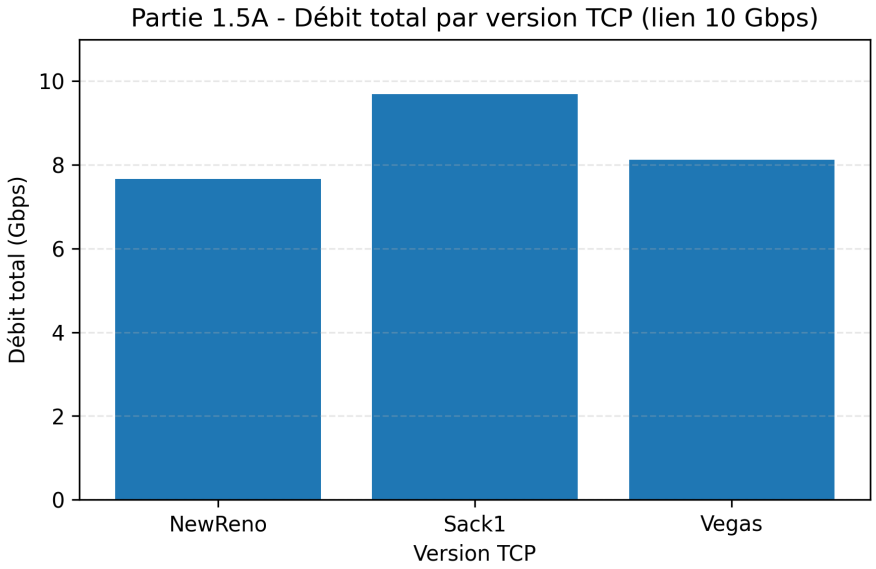

FIGURE 15 – Partie 1.5A – Débit total par version TCP sur un lien 10 Gbps.

7.4 Analyse

On observe d'abord que les trois variantes parviennent à des débits très élevés, mais avec des différences notables de scalabilité :

— **TCP Sack1** exploite presque pleinement la capacité du lien : avec près de 9,7 Gb/s, la capacité de 10 Gbps est utilisée à plus de 96 %. Cela montre que la combinaison d'une grande fenêtre de congestion et des accusés de réception sélectifs (SACK) permet de récupérer efficacement après plusieurs pertes et de maintenir un cwnd très élevé sur des liens à grand produit bande passante–délai.

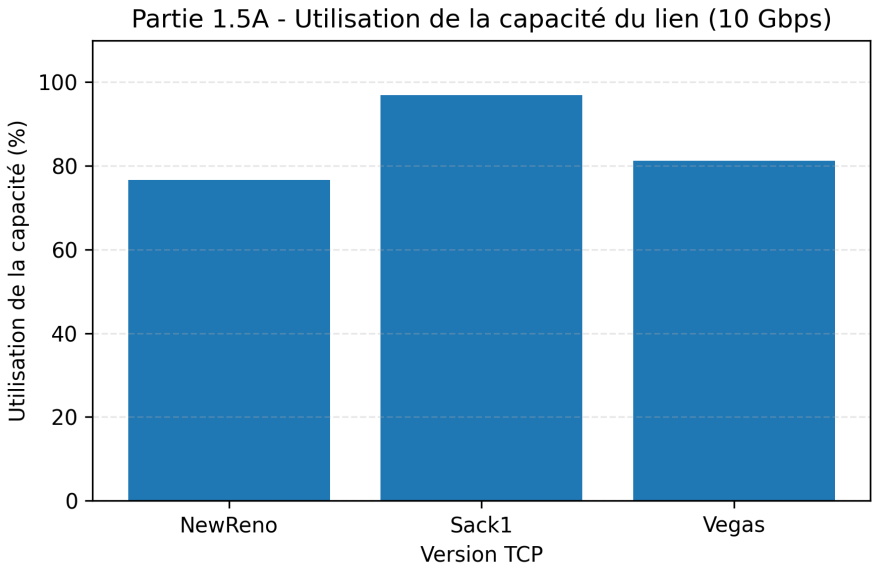

FIGURE 16 – Partie 1.5A – Utilisation de la capacité du lien (10 Gbps) selon la version TCP.

— **TCP NewReno** n'utilise qu'environ 77 % de la capacité. En l'absence de SACK, plusieurs pertes dans une même fenêtre conduisent à une phase de récupération plus longue et à une augmentation additive relativement lente. Sur un lien à 10 Gbps, chaque cycle de croissance de cwnd en fonction du RTT est coûteux, si bien que le débit reste durablement en dessous de la capacité maximale.
— **TCP Vegas**, de type *delay-based*, maintient un débit intermédiaire (environ 8,1 Gb/s). Son principe est d'ajuster la fenêtre en fonction du délai mesuré pour éviter que la file ne se remplisse complètement. Vegas laisse donc volontairement un peu de marge dans la file du goulet, ce qui limite la congestion et les pertes mais empêche d'atteindre la pleine saturation du lien.

Du point de vue de la scalabilité sur des liens très haut débit, **Sack1 apparaît comme la variante la plus efficace**, puisqu'elle combine un contrôle de congestion loss-based classique avec un mécanisme de récupération rapide grâce aux accusés sélectifs. Cependant, cette agressivité peut devenir un inconvénient lorsqu'elle coexiste avec des variantes plus prudentes comme Vegas : dans ce cas, Sack1 risque de monopoliser la bande passante au détriment des flux delay-based, ce que nous avons effectivement observé dans la Partie 1.3 avec le scénario mixte.

En résumé, ces expériences montrent que :

— NewReno est simple mais peu extensible sur des liens de 10 Gbps ;
— Sack1 exploite très bien la capacité mais peut être trop agressif vis-à-vis d'autres variantes ;
— Vegas privilégie des files courtes et un délai réduit, au prix d'une légère sous-utilisation de la capacité disponible.

Cette comparaison illustre l'importance de choisir (et de déployer) des variantes TCP adaptées aux environnements haut débit, en tenant compte non seulement du débit maximal, mais aussi de la coexistence avec d'autres algorithmes de contrôle de congestion.

8 Partie 1.5B – Impact du trafic de fond et de flux supplémentaires

8.1 Objectif

Dans cette sous-partie, nous étudions l'impact :

— d'un **trafic de fond** (UDP) sur le lien goulet,
— et de l'augmentation du nombre de flux TCP,
sur les performances globales du système.

Nous comparons trois scénarios en termes de :
— *débit brut* TCP (somme de tous les débits des flux),
— *goodput TCP* (débit utile, sans les retransmissions),
— taille moyenne de la file au goulet,
— nombre de paquets perdus,
— surcoût des retransmissions (taux moyen `retx_perte`).

8.2 Scénarios et méthodologie

Topologie et trafic. La topologie est identique à celle des parties précédentes, avec un lien goulet à 10 Mb/s entre deux routeurs. Sur ce lien, nous faisons toujours passer des flux `TCP/NewReno/FTP` de longue durée. Trois scénarios sont définis :

1. **bg0_extra0** : 6 flux TCP traversent le goulet, aucun trafic de fond (UDP) n'est présent.
2. **bg2_extra0** : mêmes 6 flux TCP, plus un trafic de fond UDP de 2 Mb/s sur le lien goulet.
3. **bg4_extra1** : 9 flux TCP au total (6 flux de base + 3 flux supplémentaires démarrant plus tard), et un trafic de fond UDP de 4 Mb/s.

Les flux TCP démarrent à $t = 0{,}5$ s (et, pour le scénario `extra1`, trois flux supplémentaires démarrent plus tard) et restent actifs jusqu'à la fin de la simulation.

Mesure des débits brut et utile. Pour chaque scénario, un script AWK calcule pour chaque flux TCP :
— le **débit brut** (colonne `brut`, en Mb/s), c'est-à-dire le débit incluant retransmissions,
— le **débit utile** (colonne `utile`, en Mb/s), c'est-à-dire le débit réellement délivré au récepteur,
— le `retx_perte`, qui représente la fraction de trafic retransmis ou perdu (en %).
Les résultats détaillés par flux sont consignés dans trois fichiers : `worst_bg0_extra0.txt`, `worst_bg2_extra0.tx` et `worst_bg4_extra1.txt`. Un script Python additionne ensuite les débits de tous les flux d'un scénario pour obtenir le *débit total brut* et le *goodput total* (Figure 17). :contentReference[oaicite :0]index=0

Mesure de la file et des pertes. En parallèle, une trace de file (`trace-queue`) est activée sur le lien goulet. Un script AWK, similaire à celui utilisé dans la partie 1.4, reconstruit la taille instantanée de la file à partir des événements « + » (entrée en file), « - » (sortie) et « d » (drop). Pour chaque scénario, ce script fournit :
— la **queue moyenne** (en paquets),
— la **queue max** (taille maximale observée),
— le **nombre total de drops** au goulet.
Les valeurs agrégées sont stockées dans les fichiers `queue_bg0_extr0.txt`, `queue_bg2_extr0.txt` et `queue_bg4_extr1.txt`, puis représentées graphiquement (Figures 18 et 19).

8.3 Résultats

8.3.1 Débit brut et goodput

Les débits individuels par flux (fichiers `worst_*.txt`) donnent les valeurs suivantes :
— **bg0_extra0** (6 flux TCP, pas de trafic de fond) :
  — débits bruts par flux entre 1,64 et 1,68 Mb/s ;

— somme des débits bruts ≈ 9,98 Mb/s ;
— somme des débits utiles ≈ 9,95 Mb/s.

— **bg2_extra0** (6 flux TCP + UDP 2 Mb/s) :
— débits bruts par flux entre 1,27 et 1,42 Mb/s ;
— somme des débits bruts ≈ 8,00 Mb/s ;
— somme des débits utiles ≈ 7,97 Mb/s.

— **bg4_extra1** (9 flux TCP + UDP 4 Mb/s) :
— débits bruts par flux entre 0,61 et 0,73 Mb/s ;
— somme des débits bruts ≈ 6,07 Mb/s ;
— somme des débits utiles ≈ 6,01 Mb/s.

Ces valeurs sont illustrées sur la Figure 17. On constate que :

— le goodput est toujours très proche du débit brut (différences de l'ordre de 0,3 à 0,6 %), ce qui signifie que le coût des retransmissions reste faible ;
— en revanche, le *débit total TCP* décroît nettement lorsque l'on ajoute du trafic de fond et des flux TCP supplémentaires : ≈ 10 Mb/s → 8 Mb/s → 6 Mb/s.

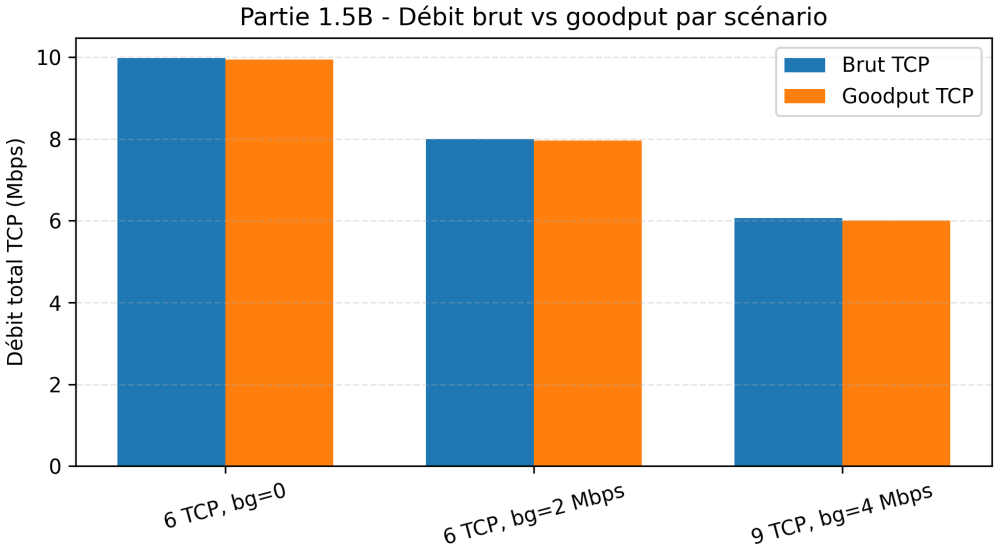

FIGURE 17 – Partie 1.5B – Débit brut vs goodput total pour chaque scénario.

8.3.2 Taille de la file et pertes au goulet

Les mesures de file issues des fichiers `queue_bg*.txt` sont :

— **bg0_extr0** :
— queue moyenne = 141,75 paquets,
— queue max = 200 paquets,
— drops = 537 paquets.

— **bg2_extr0** :
— queue moyenne = 146,82 paquets,
— queue max = 201 paquets,
— drops = 783 paquets.

— **bg4_extr1** :
— queue moyenne = 156,38 paquets,
— queue max = 201 paquets,
— drops = 1855 paquets.

La Figure 18 montre que la queue moyenne augmente progressivement avec la charge, tandis

que la Figure 19 met en évidence une hausse très marquée du nombre de drops dans le scénario le plus chargé.

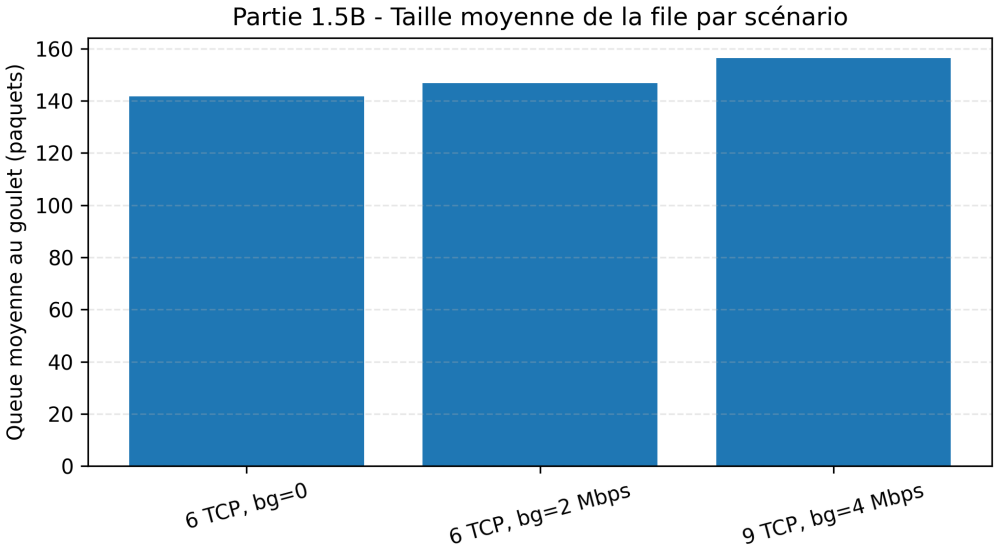

FIGURE 18 – Partie 1.5B – Taille moyenne de la file au goulet selon le scénario.

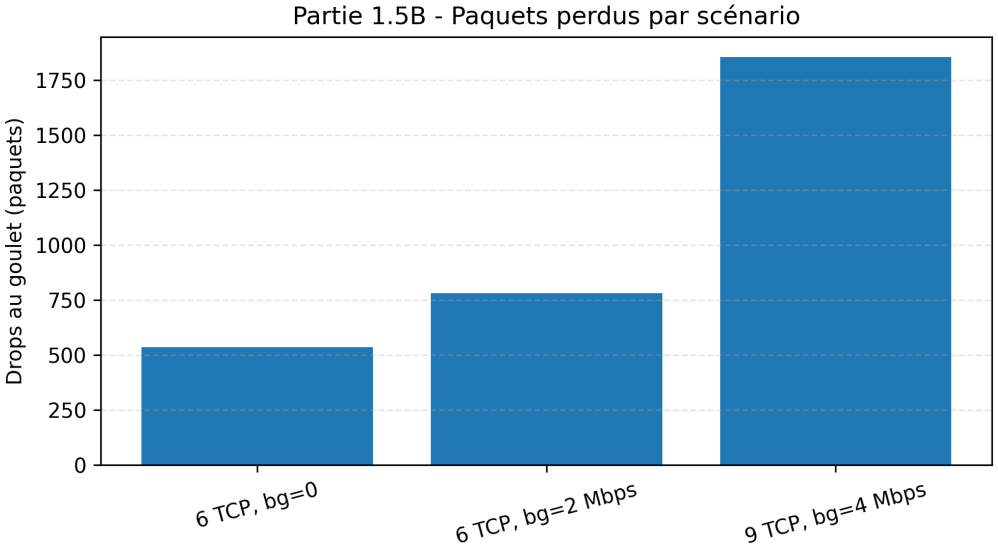

FIGURE 19 – Partie 1.5B – Nombre de paquets perdus au goulet selon le scénario.

8.3.3 Surcoût des retransmissions

À partir des fichiers `worst_*.txt`, nous avons calculé la valeur moyenne de `retx_perte` sur l'ensemble des flux TCP de chaque scénario :

— **bg0_extra0** : retx_perte moyen ≈ 0,31 % ;
— **bg2_extra0** : retx_perte moyen ≈ 0,43 % ;
— **bg4_extra1** : retx_perte moyen ≈ 0,91 %.

Ces valeurs sont représentées sur la Figure 20. On voit que la fraction de trafic consacrée aux retransmissions reste globalement faible (moins de 1 %), mais augmente avec la charge et le nombre de flux.

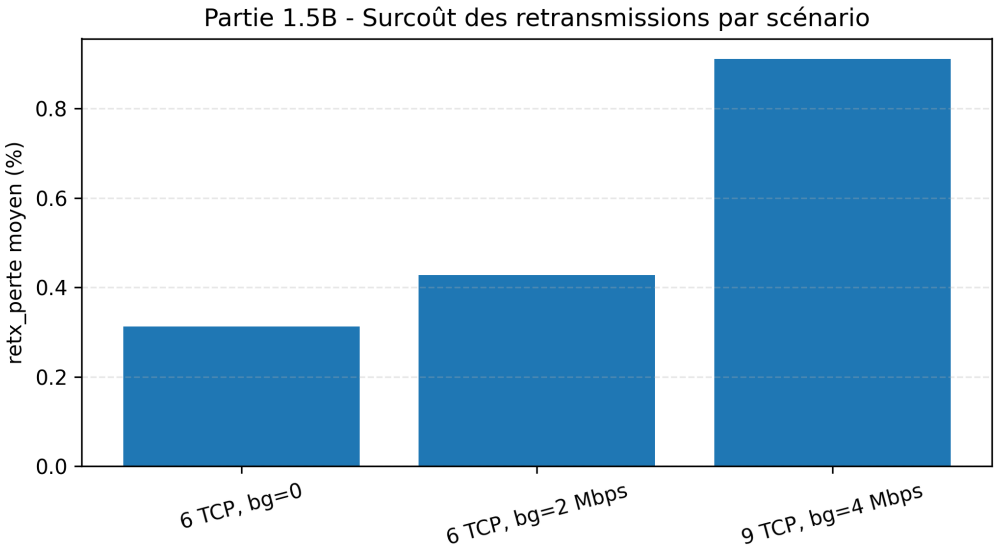

FIGURE 20 – Partie 1.5B – Surcoût des retransmissions (retx_perte moyen) pour chaque scénario.

8.4 Analyse

Les résultats de cette partie mettent en évidence plusieurs phénomènes importants :

Compétition entre TCP et trafic de fond. Lorsque l'on introduit un trafic de fond UDP sur le goulet, **le débit total TCP diminue presque linéairement** :

- sans fond (bg=0), les 6 flux TCP se partagent quasiment toute la capacité du lien (10 Mb/s) ;
- avec 2 Mb/s de fond (bg=2), la somme des débits TCP tombe à ≈ 8 Mb/s ;
- avec 4 Mb/s de fond et 3 flux TCP supplémentaires (bg=4, extra1), la part TCP se réduit à ≈ 6 Mb/s.

Le trafic UDP, non contrôlé par un mécanisme de congestion, prend sa part de la bande passante et force TCP à s'adapter en réduisant sa fenêtre.

Effet sur la file et les pertes. En parallèle, la **taille moyenne de la file** augmente avec la charge (de 142 à 156 paquets) et le **nombre de drops** est multiplié par plus de trois entre le premier et le dernier scénario (de 537 à 1855 paquets). Cela signifie que le goulet fonctionne de plus en plus proche de sa saturation, avec une file souvent remplie, ce qui entraîne davantage de pertes et donc une latence plus élevée.

Goodput vs débit brut et surcoût des retransmissions. Malgré l'augmentation des drops, la différence entre débit brut et goodput reste faible (quelques dizaines de kb/s sur une dizaine de Mb/s), ce qui se traduit par un `retx_perte` moyen inférieur à 1 % même dans le scénario le plus chargé. Autrement dit, **la majorité des pertes est absorbée par TCP avec un surcoût de retransmission modéré**, mais *au prix d'une diminution du débit total TCP* et d'une file plus longue.

Bilan. Cette étude montre que :

- l'ajout de trafic de fond UDP et de flux TCP supplémentaires réduit la part de bande passante disponible pour chaque flux TCP et augmente la taille de la file au goulet ;

— le système reste relativement efficace du point de vue du goodput (peu de retransmissions inutiles), mais la congestion supplémentaire se traduit par plus de pertes et un délai de file plus important ;
— dans un réseau partagé, **la présence de trafic non réactif (UDP)** peut donc dégrader sensiblement les performances perçues par les flux TCP, même si elle n'entraîne pas un surcoût massif en termes de retransmissions.

Ces observations complètent les conclusions de la Partie 1.5A sur la scalabilité des différentes variantes de TCP en montrant l'impact d'un environnement concurrent (trafic de fond + flux supplémentaires) sur le débit utile et la stabilité de la file.

A Scripts Tcl

Cette annexe ne présente que quelques extraits représentatifs des scripts Tcl utilisés dans le projet. Les fichiers complets (pour toutes les parties) sont fournis séparément avec le rapport.

Exemple : topologie de base et 6 flux TCP (Partie 1.1)

```
# Création du simulateur et des noeuds
set ns [new Simulator]

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 [$ns node]
set n7 [$ns node]

# Liens feuilles -> coeur (RTT hétérogènes)
$ns duplex-link $n0 $n3 100Mb 5ms  DropTail
$ns duplex-link $n1 $n3 100Mb 15ms DropTail
$ns duplex-link $n2 $n3 100Mb 30ms DropTail

$ns duplex-link $n4 $n5 100Mb 5ms  DropTail
$ns duplex-link $n4 $n6 100Mb 15ms DropTail
$ns duplex-link $n4 $n7 100Mb 30ms DropTail

# Lien goulet
$ns duplex-link $n3 $n4 10Mb 10ms DropTail
$ns queue-limit $n3 $n4 50

# Exemple de création d'un flux TCP NewReno
set tcp0  [new Agent/TCP/Newreno]
$tcp0 set fid_ 0
$ns attach-agent $n0 $tcp0

set sink0 [new Agent/TCPSink]
$ns attach-agent $n5 $sink0
$ns connect $tcp0 $sink0

set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp0
$ns at 0.5 "$ftp0␣start"

# ... (5 autres flux TCP configurés de facon similaire)
```

Exemple : configuration RED (Partie 1.4)

```
# Lien goulet avec file RED (extrait simplifié)
set ns [new Simulator]

# Paramètres RED (valeurs indicatives)
Queue/RED set thresh_ 5
Queue/RED set maxthresh_ 15
Queue/RED set maxqueue_ 50
Queue/RED set q_weight_ 0.002

$ns duplex-link $n3 $n4 10Mb 10ms RED
$ns queue-limit $n3 $n4 50

# ... (configuration des flux TCP identique aux autres scénarios)
```

B Scripts AWK et Python

Les scripts de cette section illustrent la logique de post-traitement utilisée pour obtenir les métriques (débit, taille de file, drops, etc.). Les versions complètes sont fournies séparément.

B.1 Exemple AWK : débit moyen par flux TCP

```
# throughput_par_flux.awk (extrait)
# Calcule le débit moyen (Mb/s) pour chaque flux TCP (fid)
# à partir des paquets TCP recus aux noeuds de destination (5,6,7).

($1 == "r" && $5 == "tcp" && ($4 == 5 || $4 == 6 || $4 == 7)) {
    bytes[$8] += $6
}

END {
    # Durée de la simulation (à adapter selon le scénario)
    sim = 200.0
    for (fid in bytes) {
        thr = bytes[fid] * 8 / sim / 1000000
        printf("fid %s : %.3f Mbps\n", fid, thr)
    }
}
```

B.2 Exemple AWK : statistiques de file au goulet

```
# queue_stats.awk (extrait)
# Reconstruit la taille de file et compte les pertes au goulet.

BEGIN {
  q = 0; qmax = 0;
  sumq = 0; n = 0; drops = 0;
}

{
  ev = $1
```

```
  if      (ev == "+") q++
  else if (ev == "-") q--
  else if (ev == "d") {
    drops++
    if (q > 0) q--
  }

  if (q > qmax) qmax = q
  sumq += q
  n++
}

END {
  if (n > 0) {
    printf("queue moyenne = %.2f paquets\n", sumq/n)
    printf("queue max = %d paquets\n", qmax)
    printf("drops = %d\n", drops)
  }
}
```

B.3 Exemple Python : génération d une figure de débit

```
# Exemple simplifié inspiré de plot_1_1.py
# Lecture d'un fichier thr_*.txt et tracé d'un barplot.

import matplotlib.pyplot as plt

fids = []
throughputs = []

with open("thr_part1_1.txt") as f:
    for line in f:
        parts = line.split()
        fids.append(int(parts[1]))   # fid
        throughputs.append(float(parts[3]))  # débit en Mb/s

plt.figure()
plt.bar(fids, throughputs)
plt.xlabel("Flux␣(fid)")
plt.ylabel("Débit␣moyen␣(Mb/s)")
plt.title("Débit␣moyen␣par␣flux␣TCP␣(exemple)")
plt.tight_layout()
plt.savefig("exemple_throughput_bar.png")
plt.close()
```

Les autres scripts Python (`plot_1_2.py`, `plot_1_3.py`, `plot_1_4.py`, `plot_1_5A.py`, `part_1_5B.py`) suivent la même structure : lecture de fichiers texte de synthèse, construction de vecteurs de métriques, puis génération de figures avec `matplotlib`.