Projet 1 - Rapport

Simulation de traffic UDP et TCP avec `ns2`

Jérémie Dumas

2 novembre 2010

Introduction

Composition du projet Les fichiers présents dans cette archive :

`rapport.pdf`	Le présent rapport.
`exo*.tcl`	Script OTcl correspondant aux différents exercices.
`debit.sh`	Script calculant le débit moyen et le débit moyen utile d'un flot.
`loss.sh`	Script calculant le taux de paquet perdus pour un flot donné.
`delay.sh`	Calcule le délai moyen de bout en bout pour un flot donné.
`rtt.sh`	Script bash calculant le RTT pour un flot TCP donné.

Les différents scripts sont un peu commentés, on peut taper par exemple `./debit.sh` pour que le script nous explique comment on se sert de lui.

Simulation et `ns2` Le présent sujet porte sur une modélisation de traffic internet grâce au logiciel `ns2`. L'intérêt étant de pouvoir mesurer de manière simple les performances de divers protocoles, sur des topologies particulières. C'est l'avantage de la simulation : pouvoir étudier aisément des topolgies arbitraires sans avoir à connecter plein d'ordinateurs pour la reproduire.

Il y a cependant des limitations, du fait que le logiciel ait un comportement parfaitement déterministe (même si l'on peut introduire un modèle d'erreurs), et que la qualité des mesures effectuées dépendent évidemment de la qualité du modèle implémenté par `ns2`.

1 Premier exemple avec un traffic UDP

Question 1.3. On a une topologie minimaliste constituée de deux nœuds n0 et n1, reliés par un lien duplex. La file d'attente du lien (n0,n1) représente en fait la file d'attente du nœud n0 pour les paquets qui sont à destinations de n1. Ca permet d'avoir facilement une file d'attente par connexion sortante pour n0.

2 Concurrence de flots TCP

Question 2.1. On observe un débit moyen de 0.82208 Mb/s pour le flot 1 (du nœud n0 vers le nœud n3), ainsi qu'un débit moyen de 1.21146 Mb/s pour le flot 2 (du nœud n1 vers le nœud n2).

Question 2.2. C'est sans surprise le flot 2 qui a le plus grand débit moyen. On observe aussi que les résultats sont cohérents : la somme des deux débits observés est inférieure à la capacité du lien entre n1 et n2. Finalement, on voit qu'avec des tailles de paquets identiques le flot 2 ait un débit 1.5 fois plus élevé que le flot 1.

En revanche si l'on introduit une dissymétrie dans la taille des paquets, donnant par exemple un `packetSize_ 600` au flot 2, les débits observés sont rééquilibrés. Cela peut s'interpréter de la manière suivante : plus les paquets du flot 2 sont petits, plus il est facile pour un paquet du flot 1 arrivant sur le nœud n1 de s'intercaler entre deux paquets du flot 2.

3 Mix TCP et UDP

Question 3.2. On observe qu'il y a un perte de paquets au niveau du nœud n2, et que le flot TCP est modifié en conséquence. On observe plusieurs phénomènes : le *slow-start* du protocol TCP au début, puis le *congestion avoidance* qui amène le flot TCP à envoyer moins de paquets non acquittés à travers le réseau.

Question 3.3. Dans le fichier de trace, on voit apparaître un nouveau type d'événement, le `d` pour *droped* (paquet perdu). Ici le drop a lieu après un *enqueue*, signifiant que le paquet a été mis dans la file mais jeté pour cause de dépassement de capacité (type de file `DropTail`).

Question 3.4. On remarque bien l'augmentation du nombre de paquet dans la file d'attente au niveau du lien (n2,n3) à partir de $t = 2s$, quand n0 se met à émettre. De plus on voit que la file d'attente est souvent remplie de paquets UDP, et que les paquets TCP qui arrivent de manière plus sporadique sont alors jetés plus facilement (type de file *DropTail*).

Question 3.5. Pour calculer le taux de paquets UDP perdu, on procède comme suit : tout d'abord, isoler les différents événements pour un paquet donné (tri en fonction du champ $12 du fichier de trace). On ne garde que les événements du type *receive* ou *drop* (si un paquet a été *received* puis *droped*, on ne va garder que l'événement *drop*).

Enfin, il suffit de calculer le nombre de *drop* obtenu par rapport au nombre total de paquets considérés (*drop* + *receive*).

Question 3.6. On recense bien 7 paquets UDP perdus sur les 500 transmis. Soit un taux de perte de 0.014. C'est bien cohérent avec ce que l'on peut observer via `nam`.

Question 3.7. La définition du délai de bout en bout que l'on utilise est la suivante : c'est le délai entre le premier événement relatif à un paquet (un `+`, mise en file), et le dernier événement de ce paquet (un *receive* `r`, vu que l'on ignore les paquets *droped*).

De fait, le délai moyen de bout en bout est calculé comme suit : on isole les événements relatifs à chaque paquet, que l'on tri par ordre chronologique. On fait ensuite la différence entre les événements extrémaux (s'il n'y a pas de *drop* dans le lot). Enfin on calcule la moyenne des délais obtenus.

Au final cela nous donne un délai moyen de 0.0502724 s pour le flot UDP. Au vu des temps de propagation sur les liens (10 + 20 ms), ça implique un temps d'attent en gros de 20 ms dans la file d'attente de n2 vers n3. Pourquoi pas après tout.

Question 3.8. On change de type de file d'attente, on utilise cette fois une file de type *SFQ*. Cette fois on observe qu'il y a 34 paquets UDP perdus, soit un taux de perte de 0.068. Le délai moyen de bout en bout est alors estimé à 0.0737383 s.

Cela paraît raisonnable : le flot CBR ne prennant pas en compte la présence du traffic FTP, il perd régulièrement des paquets car ceux-ci sont émis trop régulièrement. C'est lui qui fait caguer, donc la file *SFQ* moins gentille avec ses paquets UDP. Le délai moyen en prend aussi un coup, car les paquets UDP passent plus de temps dans la file d'attente au profit des paquets TCP.

4 TCP et erreurs

Question 4.1. Au bout de 0.31 s (soit à la date $t = 1.31$ s), on voit avec `nam` que le régime se stabilise. On rappelle que la fenêtre de congestion désigne le nombre de paquet non encore acquittés qui peuvent être envoyés sur le canal. Ici quand on atteint un régime stationnaire, on voit que le lien est utilisé à son maximum : cela signifie qu'il n'y a pas de contraintes imposées par la fenêtre de congestion.

Question 4.2. On observe en passant la capacité du lien (n0,n1) à 10 Mb/s qu'il y a au plus 20 paquets non acquittés qui peuvent transiter à la suite sur le canal : c'est la taille max par défaut que donne `ns2` à une fenêtre de convestion de TCP.

Question 4.3. Pour calculer le débit moyen utile, on compte chaque paquet qui a été transmis 1 fois, et on retranche à sa taille la taille d'un header TCP (40 octets). On peut reprendre le script de calcul de débit moyen de la partie 1, en filtrant en fonction du numéro de segment (champ \$11) plutôt que du numéro de paquet (champ \$12) pour éviter de recompter les paquets qui pourraient être retransmis.

Au final on calcule un débit utile de 0.887 Mb/s. C'est rassurant de voir que c'est $\leqslant 1$ Mb/s ! Et puis proportionnellement, la taille du header faisant presque $\frac{1}{10}$ de la taille du paquet utile (500 octets ici), la mesure semble plutôt correct (le débit physique restant étant consacré au transport des headers).

Question 4.4. Le calcul des RTT est simple dans le cas où il n'y a pas d'erreur : il s'agit du délai entre la date d'envoi d'un paquet (événement `-`), et la date de réception (événement `r`) de l'acknowledge correspondant, qui porte donc le même numéro. Dans le cas où il peut y avoir des erreurs, il se passe plusieurs choses :

1. On va avoir des paquets qui vont être retransmis. On gardera donc la dernière occurence de l'envoi pour un paquet donné (dernier événement de type `-`).
2. Si le paquet 88 par exemple se perd, alors l'arrivée des paquets $89, 90, \ldots$ vont générer un acknowledge `ack(87)`. Il faudra donc considérer seulement le premier `ack` d'un numéro donné (premier événement de type `r`).
3. Lorsque le paquet 88 sera réémis et bien reçu, le récepteur va alors renvoyer un `ack(95)` s'il a bien reçu les paquets 89 à 95 entre temps. Il n'y aura donc pas de génération d'un `ack(92)` par exemple. *Ainsi, lorsque l'on envoie un paquet numéroté p il faudra considérer le premier* `ack` *de numéro de segment $\geqslant p$.*

Remarque : Le premier point, qui consiste à considérer l'`ack` d'un paquet comme un acquittement correspondant à la dernière retransmission d'un paquet, et sujet à discussion. Par exemple, on pourrait croire qu'un paquet a transmis soit considéré comme perdu, que l'on retransmette donc un paquet a', puis que peut après le paquet a arrive à destination. Il y a donc envoi d'un `ack(p)`, que l'on prendra pour un acquittement du paquet a', alors qu'il s'agit en fait d'un acquittement du paquet a.

Un tel biais dans la mesure peut conduire à une sous-estimation du véritable RTT. Notons donc que dans le cadre de notre simulation, un tel scénario ne se produira pas, tout simplement parce que les paquets mettent le même temps pour aller de n0 à n1 (et donc un paquet supposé perdu ne réapparaîtra pas plus tard dans la vie).

Valeur obtenue La méthode d'évaluation du RTT décrite ci-dessus a été implémentée dans le script `rttt.sh` présent avec ce rapport. On mesure alors un RTT moyen de 0.0646355 s dans le modèle sans erreurs.

Question 4.5. On introduit maintenant un modèle d'erreurs. Le débit moyen utile et le calcul du RTT présentés précédemment prennent en compte ces possibilités d'erreurs et de retransmissions. En faisant le calcul, on trouve un nouveau RTT moyen de 0.0704135 s, ainsi qu'un débit utile évalué à 0.593 Mb/s.

Question 4.6. La valeur du RTT moyen a augmenté : c'est normal, étant donné qu'à cause des erreurs, l'`ack` pour un segment donné peut arriver beaucoup plus tard (reprendre l'exemple précédent, où l'`ack` du paquet 92 n'est envoyé qu'après que le paquet 88 ait été retransmis et soit bien arrivé).

Quant à la valeur du débit moyen utile, elle a effectivement baissée. Là aussi, c'est normal, étant donné qu'à chaque erreur la fenêtre de congestion va être réduite à 1 *maximum segment size* (MSS), et que l'on bascule à nouveau en état de *slow-start* (où l'on augmente de 1 la MSS à chaque `ack` reçu).

Conclusion

`ns2`, c'est cool. Par contre, le fait de connaître exactement la topologie du réseau que l'on veut analyser peut nous donner envie de faire des trucs pas très propres dans les scripts de calcul de débit et de RTT, plutôt que de chercher à faire des scripts plus général. De plus, il est assez difficile d'être sûr de la validité de ses scripts, étant donné qu'il y a énormément de façons de parser le fichier de sortie pour en déduire les informations utiles.

Dans l'ensemble `n2` présente quand même l'avantage d'offrir des abstractions d'assez haut niveau pour modéliser des réseaux de communication. De fait, même si le manuel de référence n'est pas toujours très clair quand on début, il s'agit clairement d'un outils intéressant pour faire des mesures de performances.