Jady RAMANANDRAY
Thomas VIOLA
3TC3
Projet PRS – groupe **explorers**

Introduction

Dans le cadre de la Programmation Réseaux et Système, chaque groupe est amené à programmer trois serveurs distincts, capables de s'adapter en fonction du comportement du client qui lui est connecté. Le seul critère de performance évalué est le débit: il faut donc à tout prix maximiser ce paramètre par tous les moyens possibles, quitte à modifier les valeurs des paramètres dans les RFC de TCP. L'objectif principal est d'avoir un débit maximal en s'inspirant des protocoles connus de TCP. Pour mieux travailler en équipe, un partage git a été nécessaire.

Implémentation

- On utilise une communication par transport UDP, à partir de laquelle on intègre le protocole *Three-way handshake* de TCP. Le serveur global attend constamment de nouveaux clients: il s'occupe de la première mise en connexion de tout client qui désire accéder au réseau, puis passe la main à un serveur-fils disponible que pour ce client en question. Pour bien distinguer les 2 processus, un **affichage du pid du processus en cours d'exécution** a été fortement utile.

- On a choisi d'implémenter un serveur-fils **très agressif**, étant donné qu'une socket serveur ne communique qu'avec un seul et unique client: nous avions choisi une **cwnd initiale très élevée**, notamment entre 100 et 200 selon la taille du fichier demandé. En effet, si on suit les normes RFC qui stipulent une cwnd initiale de 1, 2, 4 ou 10, le débit est très affaibli.

- Les fragments envoyés sont de taille 1500, dont les 6 premiers octets représentent le numéro de fragment. C'est d'ailleurs la **taille maximale que peut recevoir le client1.** Des tests ont été mis en place pour déduire cette capacité du buffer client.

- Lorsque le dernier RTT est trop grand (>100msec, valeur expérimentale), on n'attend pas 3*RTT pour recevoir un ACK: on configure le **timeout par sa valeur par défaut qui est de 300msec.**

Difficultés rencontrées et solutions

- La première connexion: le client se connectait au port du serveur-fils sans même qu'il soit encore créé -> il a fallu mettre un sleep(1) entre le fork et l'envoi du port du serveur-fils dédié au client (temps à considérer pour le calcul à la main du débit!).

- **Les fichiers binaires:** les fichiers *copy_** n'étaient jamais identiques à l'original quand c'était un fichier binaire (.png, .pdf, …): mauvaise utilisation de sprintf au lieu de memcpy quand on duplique les fragments dans le buffer d'envoi + souci de pointeur (+7 au lieu de +6…)

- La lecture du fichier: l'accès au fichier par fread à chaque boucle diminuait considérablement le débit -> on a créé un buffer dynamique dans lequel est dupliqué le fichier demandé et dont la taille est identique à ce dernier. On peut alors fermer le fichier pendant tout l'envoi.

- **La fin du fichier:** quand le pointeur qui accède au buffer du fichier excède les bornes du buffer, il y avait une boucle infinie: rien est acquitté, et le serveur envoie des fragments non-existants -> faire un break dès que la fin de fichier est atteinte.

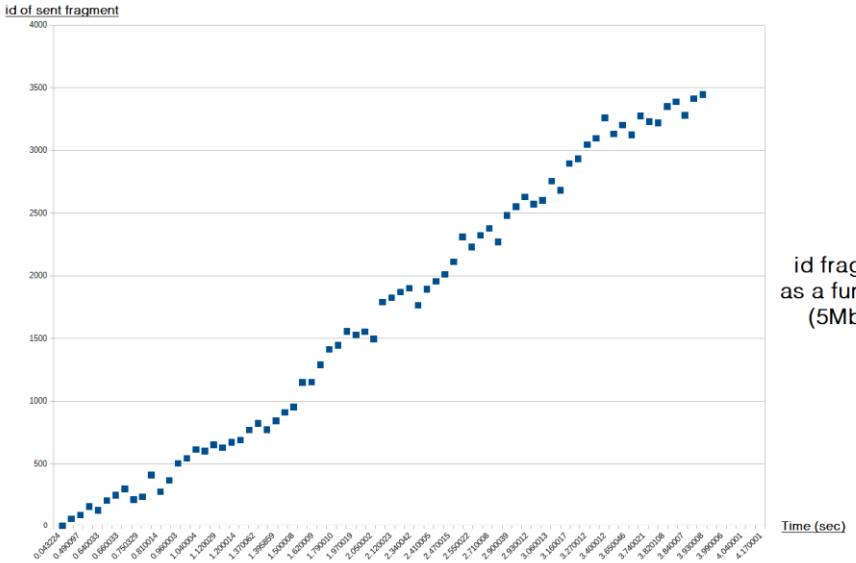
- Le RTT: d'abord implémenté en millisecondes (variable float/double), cela faisait bloquer le programme au bout d'environs 15000 fragments envoyés (impossible d'afficher la variable, impossible de dupliquer sa valeur, impossible de comprendre pourquoi,…) -> solution: passer en µsecondes donc en variable int, donc pas de virgules et tout est bien qui finit bien.

Jady RAMANANDRAY
Thomas VIOLA
3TC3
Projet PRS – groupe **explorers**

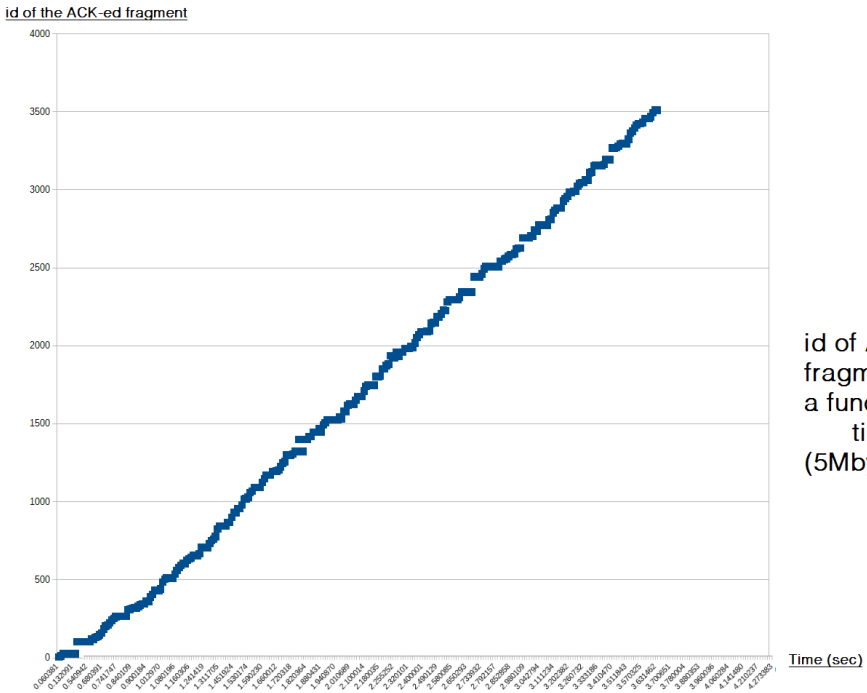
- **La réception des ACK:** on pensait que les ACK étaient envoyés dans un ordre, donc ne récupérait que le dernier reçu, donc on renvoyait inutilement des fragments (d'ailleurs rejetés par le client) -> solution: récupérer le dernier plus grand ACK reçu et le comparer au dernier fragment envoyé.

- Les graphes de performances: des tableaux dynamiques dans lesquels on stocke l'id des fragments envoyés et acquittés (le temps en abscisses). Deux problèmes majeurs: une écriture au-delà de la taille du tableau et tout est bloqué sans avertissement (il a fallu du temps pour voir que c'était ça…) ; l'autre problème, c'est la fréquence à la quelle on stocke les données : si trop rapide alors on stocke que les 2 premières secondes donc c'est inutile, et si trop lent on ne comprend pas la continuité des envois/ACK -> la solution était de prendre les id concernés toutes les 50 millisecondes.

Résultats et performances



id fragments sent as a function of time (5Mbytes file)



id of ACK-ed fragments as a function of time (5Mbytes file)

Limites et améliorations

Après l'envoi du n^{ème} fragment, si l'ACK reçu est k < n, alors on renvoie cwnd nouveaux fragments à partir du fragment k+1 **: pas de gestion des ACK dupliqués, ni de congestion avoidance, ni de fast retransmit** -> beaucoup de va-et-vient pour le renvoi de fragments, toujours accompagné de l'envoi de nouveaux fragments.

Conclusion

Objectifs atteints: débit relativement élevé (1200Ko/s en VM et 4000Ko/s en boot), utilisation de la fenêtre de congestion et vérification des ACK comme en TCP.

Comportement linéaire, stable pour le client1, à voir pour le client2.