TP n° 11: Réseau : la théorie par l'expérimentation

Ce TP a pour but d'observer certains principes théoriques à propos du réseau. Le premier exercice porte sur les protocole TCP et UDP : vous avez normalement vu leurs différences, avantages et inconvénients... nous allons les tester par la pratique ! Le deuxième exercice porte sur la simulation des interactions entre client et serveur : comment un serveur fait-il pour traiter toutes ses requêtes malgré la taille limitée de son buffer ? Selon quelles distributions les paquets arrivent-ils ? Si la charge est trop importante, rajouter d'autres serveurs est-il un procédé efficace/rentable ? Vous aurez normalement vos réponses à ces questions en réalisant le TP.

Ce TP est noté. Le rendu est à faire pour le dimanche 13 mai (avant 23h59) par courrier électronique aux **trois** TD-men [1]. L'objet du mail devra être `[ASR2] Rendu TP11.` [2] **Le rapport devra faire au maximum 5 pages.** Il doit être en PDF, et s'intituler `rapport-TP-11-login.pdf`, par exemple `rapport-TP-11-atalon.pdf`. Les codes sources sont à rendre dans une archive jointe au mail de rendu, nommée `src-TP11-login.tar`, contenant un répertoire `src-TP11-login` qui lui-même contiendra les fichiers demandés.
Tout rendu en retard ou ne respectant pas les consignes (sujet du mail, destinataires, format du rapport...) coûtera jusqu'à 1,5 points sur 10.
Nous rappelons que le rendu est **individuel**. Cela signifie que, même si vous êtes encouragé.es à réfléchir à plusieurs, les codes et le rapport sont **individuels** et ne doivent par exemple pas être partagés/copiés/communs.

Exercice 1. *Do you want to hea a DUP joke ?*

Le but de cet exercice est de se familiariser avec UDP en codant un petit client qui envoie des fichiers vers un serveur simple. Le serveur en question possède comme adresse IPv4 l'adresse 148.251.48.246, et écoute en UDP sur le port 8888. Pour lui envoyer un fichier, il faut lui communiquer la taille en octets du fichier sous la forme d'un entier non signé de 64 bits, dans l'ordre "big-endian", puis le contenu du fichier (dans l'ordre).
Plusieurs limites sont mises en place par le serveur :

- La taille de chaque fichier est limitée à 10MO ;
- Le temps de transfert autorisé pour chaque fichier est limité à 100 secondes (passée cette limite, les données déjà envoyées sont supprimées et tout contact ultérieur est considéré comme l'envoi d'un nouveau fichier) ;
- Si le serveur ne reçoit pas de données pendant 2 secondes, le serveur considère que l'envoi des données est terminé (de même que précédant pour les contacts ultérieurs).

Le serveur enregistre les données qu'il reçoit et les met à disposition à l'adresse `https://vrac.grunblatt.org/`, sous la forme suivante :

- Si la taille annoncée du fichier correspond à la taille des données reçues, alors le fichier est nommé par son sha256sum ;
- Sinon, si la taille annoncée n'est pas la taille des données reçues, alors le fichier est nommé par son sha256sum préfixé de la chaine de caractères FAIL-.

1. Codez un client qui envoie un fichier vers le serveur. Ce client devra prendre en arguments sur la ligne de commande l'adresse IP du serveur (IPv4), le port du serveur, la taille maximum de chaque datagramme à envoyer, le temps entre l'envoi de chaque datagramme en microsecondes, et le chemin vers le fichier. Pour vérifier si vos fichiers arrivent, vous pouvez vérifier les fichiers présents sur `https://vrac.grunblatt.org`.
2. Vérifiez que vous arrivez à envoyer 100 octets sans erreurs, en utilisant une pause de 100000 microsecondes. Est-ce toujours le cas quand vous envoyez 2000 octets ? Et 10000 octets ? Pourquoi ?
3. Déterminez expérimentalement la taille maximum que chaque datagramme doit avoir pour que le fichier soit bien reçu, ainsi que le délai entre chaque envoi de datagramme. Écrivez pour cela un script (bash, fish, python3, . . .) permettant d'effectuer automatiquement ces tests et de déterminer ces valeurs.

1. `{valentin.lorentz,remy.grunblatt, alexandre.talon}@ens-lyon.fr`
2. sans le point, qui terminait juste la phrase.

4. Utilisez la commande Linux "ping" pour déterminer la taille maximale des datagrammes pouvant être acheminé entre votre machine et le serveur situé à l'adresse IPv4 l'adresse 148.251.48.246.

Exercice 2. *Je prendrai le saumon tartare s'il vous plaît.*

Dans cet exercice, on a un (ou plusieurs) client(s) et un (ou plusieurs) serveur(s). Le but est de simuler l'envoi des paquets par le client, et leur traitement par le serveur. Le client enverra n paquets selon un **processus** de Poisson de paramètre λ_c à fixer : ça signifie que les temps entre les paquets suivent une **loi exponentielle**. Le serveur dispose d'un buffer de taille B pour stocker les paquets qu'il n'a pas encore traité : si le buffer est plein et qu'un paquet arrive, ce paquet est **perdu** [3]. Le temps de traitement d'un paquet suit également un processus de Poisson de paramètre λ_s (potentiellement différente de λ_c) : les durées de traitement suivent une loi exponentielle. Pour plus de simplicité, on générera d'abord les temps d'envoi des paquets, qui seront stockés en mémoire, puis les durées de traitement des paquets (on en générera autant que de paquets, si on perd des paquets on en aura trop, mais ce n'est pas grave). Enfin, on simulera le serveur qui stockera les paquets et les traitera selon le schéma décrit ci-dessus.

1. Programmez une fonction qui prend un entier n et un flottant λ en argument et renvoie une liste [4] de n nombres suivant une distribution de Poisson de paramètre λ (cherchez sur un moteur de recherche/Stack overflow/autre, en citant la page d'où vous récupérer le code)
2. Générez les deux listes : celle des temps d'envoi des paquets (processus de Poisson : les inter-arrivées suivent une loi exponentielle de paramètre λ_c) et celle des durées de traitement des paquets (suivant une loi exponentielle de paramètre λ_s)
3. Programmez une fonction qui simule le serveur : elle prend en argument la liste des temps d'envoi des paquets (on suppose que les paquets sont reçus en même temps qu'ils sont envoyés), la liste des durées de traitement et un entier B (la taille du buffer). Elle reçoit les paquets quand ils arrivent, et les met dans le buffer. Si le buffer est plein, le paquet est perdu. Quand le serveur reçoit son premier paquet, ou vient de finir de traiter un paquet, il prend le paquet du buffer arrivé en premier, libérant une place, etc.
4. On suppose d'abord que le buffer est infini, *ie* $B = n$. Calculez les temps de séjour des paquets (temps de fin de traitement - temps d'arrivée). Faire un graphique représentant les probabilités d'un paquet d'avoir un temps de séjour de t.
5. Le buffer est encore infini. Étudier et faire un graphique représentant la probabilité que le système ait k paquets dans le buffer à un instant donné. Faites un graphique pour chacun des cas $\lambda_s = \lambda_c$, $\lambda_s < \lambda_c$ et $\lambda_s > \lambda_c$. Que remarquez-vous ?
6. Toujours avec un buffer infini, tracez la courbe $y = f(x)$ où $f(x)$ représente le temps de séjour du x-ème paquet.
7. À partir de maintenant, le buffer n'est plus de taille infinie. On s'intéresse à la perte des paquets. Étudiez la proportion de paquets perdus en fonction des valeurs de $lambda_c$, $lambda_s$ et B. On choisira n de l'ordre de 10000. B pourra être de l'ordre de 100. Faites des tests, et incluez dans votre rapport les graphiques qui vous semblent pertinents.
8. On suppose maintenant qu'il y a deux clients. Générez leurs distributions (on fusionne donc les deux listes des temps d'envoi).
9. Comparez le processus de Poisson de deux clients de paramètres $\lambda_c 1$ et $\lambda_c 2$ avec un seul processus de poisson de paramètre $\lambda_c 1 + \lambda_c 2$. Que remarquez-vous ?
10. On décide maintenant d'alléger la charge du serveur en installant d'autres serveurs (de même paramètre λ_s). Dans quelles proportion cela améliore-t-il le service : moins de paquets perdus ? Temps de séjour plus court (dans le cas buffer infini) ? On peut supposer que les paquets sont répartis équitablement sur les serveurs (1 paquet sur 2 sur chacun des deux serveurs), ou qu'ils ont un buffer commun (éventuellement deux fois plus gros) et que dès qu'un serveur est disponible, il prend le prochain paquet.

3. Ne nous remerciez pas pour ce rappel au jeu.
4. ou une autre structure