UNIVERSITÉ DE STRASBOURG

Licence 3 Semestre 5 Printemps
Projet de Réseaux et Protocoles

Fiabilisation et optimisation d'un canal de transmission

UDP (User Datagram Protocol) est un protocole de transport par datagrammes non fiable. Nous avons besoin d'un tel protocole (simple, rapide et à base de datagrammes) mais fiable. Pour ce faire, nous devons mettre en œuvre un protocole utilisant les services de transport d'*UDP*.

1 Transfert de fichiers à sens unique (3 points)

Écrire deux programmes *client* et *serveur*, qui réaliseront un transfert de fichier du client vers le serveur. Elles seront appelées de la façon suivante et fonctionneront comme dans le premier TP sur les sockets.

```
./serveur <fichier_recu> <port_local>
./client  <fichier_a_envoyer> <adr_IP_dist> <port_dist>
```

Le client lit le fichier `fichier_a_envoyer` par blocs et les envoie au serveur. Le serveur se connecte en local sur le port `port_local` et enregistre tout ce qu'il reçoit dans le fichier `fichier_recu`. Lorsque la transmission est terminée, les deux programmes ferment les sockets et les fichiers. Vous vérifierez que le fichier transmis est bien identique au fichier original (commandes *diff* ou *cmp*). Je demande à ce que les transferts se fassent par blocs de 1024 octets maximum. Vous veillerez à tester votre protocole sur des fichiers qui ont plus que 1024 caractères.

2 Transfert bidirectionnelle (4 points)

Ècrire deux nouveaux programmes permettant de faire un transfert dans les deux sens. Le profil des deux programmes change légèrement. Il restera le même dans toutes les sections à suivre.

```
./serveur <fichier_a_envoyer> <fichier_recu> <port_local>
./client  <fichier_a_envoyer> <fichier_recu> <adr_IP_dist> <port_dist> [<port_local>]
```

> **NB** Lorsque le programme client a 4 arguments, il peut connecter son socket local sur n'importe quel port. Lorsqu'il a 5 arguments, il doit interpréter ce cinquième argument comme le numéro de port local et attacher son socket sur ce port. Ceci s'avérera nécessaire lorsqu'on utilisera le programme *medium*.

Le client connaît l'adresse du serveur car elle lui est fournie dans ses arguments. On peut imaginer qu'avant tout transfert de données, le client envoie au serveur un datagramme vide (sans données), ce qui permet au serveur de noter l'adresse de ce client unique (il n'en attend pas d'autres). Après cette première phase de prise de contact, le protocole doit être le même des deux côtés. Et, hormis la première phase qu'on peut qualifier de "connexion", je vous recommende d'utiliser le même fichier source pour écrire ce protocole du côté serveur et du côté client.

Je vous laisse le soin d'organiser ces échanges. Je précise que vous n'avez pas le droit de faire un transfert complet de client vers serveur puis, lorsque l'échange est terminé, du serveur vers le client. Les transferts doivent être concomitants, en tout cas au départ. Si les fichiers transmis dans un sens et dans l'autre ne sont pas de même taille, le transfert sera terminé plus rapidement dans un sens. Le transfert dans l'autre sens devra continuer seul. Là aussi, veillez à ce que les fichiers transférés soient identiques aux fichiers originaux.

3 Fin de transmission (3 points)

Dans ces premiers programmes, comme dans les suivants, il vous est demandé d'élaborer une méthode rigoureuse pour la fin des transmissions. L'émetteur d'un fichier sait toujours quand la transmission est terminée, mais comment en informer le récepteur de façon fiable ? Cette méthode ne doit pas fonctionner seulement dans *la plupart des cas*, mais dans *tous les cas*, quelque soit le contenu, le type et la taille des fichiers transférés. En particulier elle doit fonctionner pour des fichiers texte, des fichiers image d'au moins 100Ko, des fichiers exécutables et même lorsque la taille des deux fichiers transmis dans chaque sens est sensiblement différente.

En aucun cas votre méthode ne doit arrêter le récepteur avant que l'émetteur n'ait cessé de transférer les données. Et en aucun cas elle ne doit laisser le récepteur en attente alors que l'émetteur a fini de transmettre.

En particulier, si votre méthode est fondée sur l'échange d'un caractère spécial, que se passe-t-il lorsque ce caractère se trouve dans les données ? Si votre méthode est fondée sur un datagramme non rempli, que se passe-t-il si, par hasard, le dernier datagramme est bien rempli ?

4 Bit alterné (5 points)

On suppose que le canal de transmission est caractérisé par un taux de pertes de $\tau \in [0, 1]$. Ce nombre est la probabilité pour qu'un paquet soit perdu pendant la transmission. Pour rendre ce canal fiable, vous devrez mettre en œuvre un protocole de type *bit alterné*. Dans un premier temps, vous pourrez tester votre protocole en transférant des fichiers directement entre les deux machines, comme à la section 2. Mais cela n'est pas très révélateur des capacités de votre protocole car *UDP* s'avère finalement assez fiable sur un réseau local comme *osiris*. Pour simuler un canal de tranmission avec des valeurs réglables de pertes, on peut faire transiter les informations par un troisième programme `medium` pouvant tourner sur une troisième machine. Plus concrètement, au lieu d'envoyer les informations directement d'une machine vers l'autre, chaque machine les envoie vers le programme `medium` qui, à son tour, les enverra (ou non) vers l'autre machine. Vous pourrez trouver un tel programme sur *Moodle* (cf appendice A pour le fonctionnement de ce programme).

5 GoBackN (5 points)

On suppose également que le canal de transmission est caractérisé par un délai de propagation de δ. Pour de grandes valeurs de δ, le client et le serveur passent beaucoup de temps à attendre les acquittements. Pour optimiser la transmission, vous devrez mettre en œuvre un protocole de type *GoBackN* (retransmission continue avec fenêtre d'anticipation à l'émission). Vous pourrez tester votre protocole avec le même programme `medium` qui peut également simuler un délai de propagation. Comparer les performances des deux protocoles (*bit alterné* et *GoBackN*) pour différentes valeurs de τ et de δ. Je vous demande en particulier les configurations suivantes pour $(\tau, \delta) = (0, 0s)$, $(0.5, 0s)$, $(0, 0.1s)$ et $(0.5, 0.1s)$.

6 Conditions

Je vous recommande de réaliser ce projet en binômes. Vous pouvez le réaliser seul, mais l'évaluation n'en sera pas plus favorable pour autant. Le deadline pour le rendu définitif du projet est le lundi 7 décembre 2015 à 0 :00 (avant dimanche minuit) sur Moodle. Vous inclurez dans une archive :

- vos programmes avec un makefile ;
- un fichier texte `avancement.txt` indiquant les questions que vous avez traitées : (transfert dans un sens, dans les deux sens, bit alterné, GoBackN) ;
- un fichier texte `protocole.txt` décrivant la structure de vos datagrammes, les types de trame et la description du protocole (indépendamment des questions d'implémentation) ; en particulier le protocole de fin de transmission évoqué à la section 3.
- un fichier texte `performances.txt` donnant les résultats que vous avez obtenus en étudiant les performances comparées des protocoles *bits alternés* et *goBackN* pour différentes valeurs de taux d'erreur et de délai.
- un fichier texte `procedure.txt` qui décrit la procédure que vous suivez pour réaliser au moins un transfert de fichier réussi. Cette procédure devra contenir toutes les lignes de commande instanciées (à entrer au clavier telles quelles) ainsi que l'ordre dans lequel vous réalisez ces opérations.
- le(s) fichier(s) que vous avez transférés pour tester votre programme.

Durant la séance de TP du mercredi 2 décembre 2015 en salle T40-GPI, je vous demanderai de me faire une démonstration de votre projet. Par ailleurs, quelque soit la plateforme que vous utilisez pour le développement de votre projet, je vous demande de le tester sur la machine *Turing* et sur les machines de la salle T40-GPI. C'est sur ces machines que vos programmes seront évalués. Enfin assurez-vous que vos programmes fonctionnent sur des machines distinctes (et non pas seulement en local). Pour ce faire vous pouvez utiliser les machines de la salle T02 ou T40-GPI.

A Le programme `Medium`

Sur la plateforme *Moodle* vous trouverez le code source d'un programme appelé `medium`. Voici la ligne de commande :

```
medium <versionIP> <local_port> <host1> <port1> <host2> <port2> <error> <delay>
```

`medium` s'attache en local sur le port `local_port`. Il n'accepte que les datagrammes venant de `host1` port `port1` ou de `host2` port `port2`. En principe il doit transmettre les données de `host1 :port1` vers `host2 :port2` et vice-versa. En pratique, à chaque datagramme reçu il décidera, de façon aléatoire, si celui-ci sera retransmis ou perdu. La probabilité que le datagramme soit perdu sera de `error`. Pour `error=0` il n'y a aucune perte. S'il n'est pas perdu, le datagramme sera envoyé vers son destinataire après une attente de `delay` secondes. La taille maximale des datagrammes transitant par `medium` est de 1024 caractères. Si `versionIP` vaut 4, alors les adresses devront être au format IPv4 et s'il vaut 6, elles devront être en IPv6.

Voici un exemple d'utilisation du programme `medium` pour transférer un fichier `toto.jpg` d'une machine vers une autre et un fichier `titi.jpg` dans l'autre direction, avec un taux d'erreur de 0.5 et un délai de 0.1 secondes. Le client tourne sur 130.79.90.238 port 20000, le serveur tourne sur 132.204.8.21 port 40000 et que le medium tourne sur *Turing* 130.79.7.1 port 30000.

- commande lancée sur 132.204.8.21 :
  ```
  ./serveur toto.jpg copy_toto.jpg 40000
  ```
- commande lancée sur 130.79.7.1 :
  ```
  ./medium 4 30000 130.79.90.238 20000 132.204.8.21 40000 0.5 0.1
  ```
- commande lancée sur 130.79.90.238 :
  ```
  ./client titi.jpg copy_titi.jpg 130.79.7.1 30000 20000
  ```

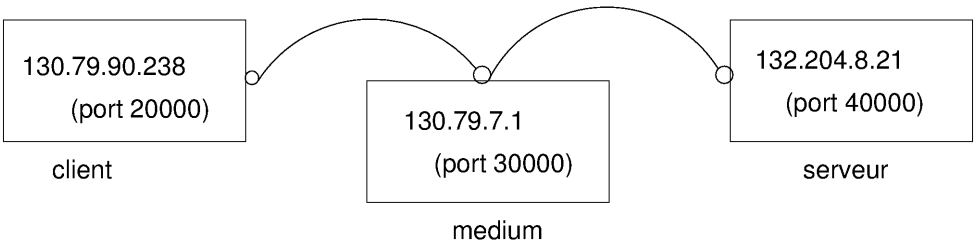

FIG. 1: Configuration décrite dans l'exemple ci-dessus.