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 Cinq fonctions (4 points)

Écrire cinq fonctions dont les profils sont :

```
int S_openAndBindSocket(int local_port);
int S_openSocket(void);
int S_distantAddress(char *IP_address, int port,
                     struct sockaddr **dist_addr);
int S_receiveMessage(int sock_fd, struct sockaddr *dist_addr,
                     char *msg, int length);
int S_sendMessage   (int sock_fd, struct sockaddr *dist_addr,
                     char *msg, int length);
```

La valeur retournée par ces fonctions devra être -1 en cas d'échec.

1. La fonction `S_openAndBindSocket` ouvre un socket de type datagramme et l'attache en local au port `local_port`.
2. La fonction `S_openSocket` ouvre un socket de type datagramme.
3. La fonction `S_distantAddress` prend en argument une chaîne de caractères représentant une adresse IP (de type "a.b.c.d" en IPv4), un numéro de port et l'adresse d'un pointeur `struct sockaddr*`. Cette fonction prépare ce pointeur pour qu'il puisse être utilisé par `sendto` ou `connect`.
4. La fonction `S_receiveMessage` permet d'attendre et recevoir un message `msg` sur le socket de descripteur `sock_fd`. Après l'arrivée du message, `dist_addr` devra désigner l'adresse de l'expéditeur du message et le tableau msg, de longueur `length` octets désignera le contenu du message arrivé. La valeur de retour de la fonction est le nombre d'octets effectivement reçus.
5. La fonction `S_sendMessage` utilise le socket `sock_fd` pour transmettre le message `msg` de longueur `length` octets à l'adresse `dist_addr`. La valeur de retour de la fonction désigne le nombre de caractères effectivement transmis.

Ces cinq fonctions devront, dans un premier temps, fonctionner avec des adresses IPv4. L'objectif de ces fonctions est de simplifier l'écriture du protocole. Grâce à elles, un serveur (par exemple 130.79.90.34 port 30000) et un client pourraient s'échanger un message "coucou" dans un sens et "hello" dans l'autre avec les programmes suivants (Je n'inclus pas les diagnostiques d'erreur pour ne pas surcharger ce document, mais je vous recommande de les utiliser).

```
// ---------- Serveur ----------------
#define BUFSIZE 1024
int main()
{
   struct sockaddr *dist_addr;
   char buffer[BUFSIZE];

   int sockfd = S_openAndBindSocket(30000);               // ouverture socket
   S_receiveMessage(sockfd, dist_addr, buffer, BUFSIZE);  // attente message
   printf("Message reçu=%s\n",buffer);                    // affichage message
   S_sendMessage(sockfd, dist_addr, "Hello", 6);          // envoi réponse
   return 0;
}
// ---------- Client----------------
#define BUFSIZE 1024
int main()
{
   struct sockaddr *dist_addr;
   char buffer[BUFSIZE];

   int sockfd = S_openSocket();                           // ouverture socket
   S_distantAddress("130.79.90.34",30000,&dist_addr);     // preparation adresse
   S_sendMessage(sockfd, dist_addr, "coucou", 7);         // envoi message
   S_receiveMessage(sockfd, dist_addr, buffer, BUFSIZE);  // attente réponse
   printf("Reponse reçue=%s\n",buffer);                   // affichage réponse
   return 0;
}
```

2 Transfert de fichiers (3 points)

Utiliser les cinq fonctions précédentes pour transférer un fichier d'une machine vers une autre et vérifier si le transfert s'est bien déroulé. Pour cela, il faudra écrire deux programmes dont les lignes de commande sont les suivantes :

```
receiver <filename> <local_port>
sender  <filename> <distant_host> <distant_port> [<local_port>]
```

Le programme `receiver` ouvre et attache un socket sur le port local `local_port` et enregistre tout ce qu'il reçoit dans le fichier `filename`. Le programme `sender` prépare l'adresse distante correspondant à l'hôte `distant_host` port numéro `distant_port`. Il ouvre le fichier `filename`, lit les données par blocs et envoie ces blocs à l'adresse préparée ci-dessus. S'il a 3 arguments, le programme `sender` n'attache

pas son socket en local. Le numéro de port sera donc choisi par le système. Par contre, si le programme a 4 arguments, il devra interpréter celui-ci comme un numéro de port local auquel attacher le socket local. Cette option s'avérera nécessaire lorsqu'on utilisera le programme `medium` pour simuler un canal bruité. Comment l'émetteur pourra-t-il faire comprendre au récepteur que la transmission est terminée ? Concevoir un protocole de fin de transmission. Assurez-vous que votre protocole ne pourra jamais interrompre une communication non-terminée ou, au contraire, laisser écouter le récepteur alors que l'émetteur a terminé. Si votre protocole est fondé sur l'échange d'un caractère spécial, que se passe-t-il lorsque ce caractère se trouve dans les données ? Si votre protocole est fondé sur un datagramme non rempli, que se passe-t-il si, par hasard, le dernier datagramme est bien rempli ?

Vos programmes doivent pouvoir échanger un fichier d'au moins 100 Ko, par exemple une grosse image. Les commandes `cmd` et `diff` permettent de savoir si deux fichiers sont identiques ou non.

3 Migration vers IPv6 (3 points)

Définir une variable globale `S_DOMAIN` qui régira le fonctionnement interne des cinq fonctions de la section 1 sans en modifier les services.

- Pour `S_DOMAIN=AF_INET` les cinq fonctions utiliseront des adresses IPv4 (comme jusqu'ici).
- Pour `S_DOMAIN=AF_INET6` les cinq fonctions utiliseront des adresses IPv6.

4 Bit alterné (5 points)

On suppose que le canal de transmission est caractérisé par un taux de pertes de τ. 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 un fichier directement depuis l'émetteur vers le récepteur. 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 vers le récepteur, l'émetteur les enverra au programme `medium` qui, à son tour, les enverra (ou non) au récepteur. 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, l'émetteur et le récepteur 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 $(\tau, \delta) = (0, 0s)$, $(0.5, 0s)$, $(0, 0.5s)$ et $(0.5, 0.5s)$.

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 mercredi 22 avril 2015 avant 23 :50 sur Moodle. Vous inclurez dans une archive :

- vos programmes avec un makefile ;
- un fichier texte `avancement.txt` indiquant les questions que vous avez traitées : (cinq fonctions, transfert de fichiers, ..., 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 au paragraphe 2.
- 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 22 avril 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. Par ailleurs, je vous demande également de vérifier que vos programmes fonctionnent sur des machines distinctes (et non pas seulement en local).

A Le programme `Medium`

Sur la plateforme *Moodle* vous trouverez un programme exécutable `mediumTuring` et un autre appelé `mediumT40`. Le premier peut tourner sur *Turing* et le second sur les machines de la salle T40. Vous pouvez également y trouver le code source de ce programme que vous pourrez compiler sur n'importe quelle machine. Par contre, pour tourner, ce programme a besoin, dans le fichier `socket.c`, des cinq fonctions de la section 1. Voici la ligne de commande :

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

`medium` écoute sur le port `local_port`. Il n'accepte que les datagrammes venant de `host1` port `port1` ou de `host2` port `port2`. Ces datagrammes auront une longueur maximale de `block_size`. 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`. 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 `original.jpg` d'une machine vers une autre avec un taux d'erreur de 0.5 et un délai de 0.1. L'émetteur tourne sur

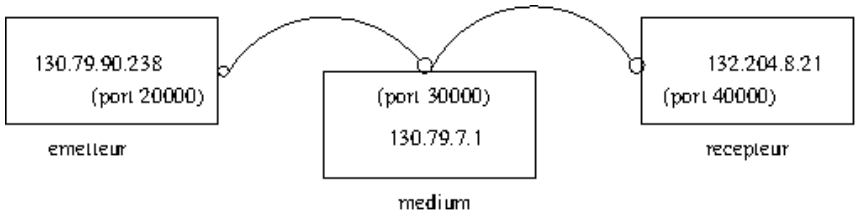

FIG. 1: Configuration décrite dans l'exemple ci-dessous.

130.79.90.238 port 20000, le récepteur tourne sur 132.204.8.21 port 40000 et que le medium tourne sur *Turing* 130.79.7.1 port 30000.

- commande lancé sur 132.204.8.21 : `./receiver copy.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 : `./sender original.jpg 130.79.7.1 30000 20000`