Université de Strasbourg
UFR Math-Info

Licence 3 Informatique
Année 2019-2020

Algorithmes des réseaux

TP 1 et 2 - Programmation API socket

1 Communication UDP

1.1 Créations de sockets

1. Écrire un programme `senderUDP.c` qui crée un socket (primitive `socket`) de type datagramme sur le domaine `AF_INET`. En cas d'erreur, ce programme devra lancer un message d'erreur en donnant la cause de l'erreur (fonction `perror`). Quelles sont les différentes sources d'erreur (`man socket`) ?
2. Quelles sont les options de la commande `netstat` qui permettent de lister tous les socket udp (y compris les sockets d'écoute), et les processus (programme/pid) qui les ont créés ? Dans le manuel de `netstat`, lire plus particulièrement la rubrique `no option`.

1.2 Récepteur UDP

1. Copier le programme précédent dans un fichier `recvUDP.c`. Dans ce dernier, préparer une adresse locale (de type `struct sockaddr_in`) en lui donnant l'adresse IP de la machine locale (`INADDR_ANY`) et un numéro de port de votre choix supérieur à 2048.
2. Attacher le socket (primitive `bind`) à l'adresse créée ci-dessus. Une fois le socket attaché (vérifier qu'il n'y a pas d'erreur), on souhaite vérifier si le socket est effectivement attaché. Faire une pause de 20 secondes (fonction `sleep`) de façon à ce que le programme ne quitte pas immédiatement. Pendant ces 20 secondes, vérifier si ce socket est visible par la commande `netstat` vue à la question 2. Est-ce que le numéro de port affiché dans `netstat` est bien celui que vous aviez spécifié dans le programme ?
3. Avec la primitive `recvfrom` attendre l'arrivée d'un message. Dans votre programme, après le retour de la fonction `recvfrom`, afficher :
   — le contenu du message reçu
   — le nombre de caractères reçus
   — l'adresse IP de l'expéditeur (vous pouvez utiliser la fonction `inet_ntop` pour convertir l'adresse IP sous forme d'un entier long vers le format `A.B.C.D`)
   — le numéro de port de l'expéditeur

1.3 Émetteur UDP

1. Dans le fichier `senderUDP.c` préparer une adresse distante (de type `struct sockaddr_in`) en lui donnant l'adresse IP de la machine distante (celle sur laquelle tourne le récepteur) et le numéro de port où est attaché le récepteur.
2. Avec la primitive `sendto` envoyer un message au récepteur à l'adresse préparée ci-dessus. En lançant votre programme, vérifier que le message reçu par le récepteur est bien le même que celui envoyé par l'expéditeur, ainsi que l'adresse IP et le numéro de port.
3. Packaging de vos programmes :

```
./recvUDP <num_port_local>
./senderUDP <adresse_IP_dest> <num_port_distant> <message>
```

2 Communication TCP

2.1 Serveur TCP

1. Écrire un programme `serveurTCP.c` qui crée un socket (primitive `socket`) de type `SOCK_STREAM` sur le domaine `AF_INET`. En cas d'erreur ce programme devra afficher un diagnostique comme pour les programmes ci-dessus.
2. Comme pour le socket udp, attacher ce socket à un numéro de port que vous aurez choisi avec la primitive `bind`.

3. En faire un socket d'*écoute* avec la commande `listen`. Ce socket restera dédié à l'attente de demandes de connexion. Il ne servira donc pas à la communication proprement dite.
4. Faire de façon à ce que le serveur attente une connexion avec la primitive `accept`. Vérifier avec la commande `netstat` que le numéro de port a bien été attribué à votre processus. Lorsqu'il reçoit une telle demande de connexion, il devra afficher un message indiquant l'adresse IP et le numéro de port de l'hôte qui est à l'origine de cette demande de connexion. La valeur de retour de la primitive `accept` est le descripteur d'un nouveau socket qui servira à la communication avec le client.

2.2 Client TCP

1. Copier la partie du programme précédent qui s'occupe d'ouvrir un socket dans un fichier `clientTCP.c`. Préparer une adresse (`struct sockaddr_in`) distante désignant le serveur.
2. Avec la primitve `connect` faire une demande de connexion auprès du serveur. Vérifier que le serveur a bien reçu cette demande (`connect` se débloque du côté client et `accept` se débloque côté serveur).
3. Une fois la connexion établie, le serveur se met à l'écoute du client (primitive `recv`) et le client envoie un message (primitive `send`).
4. Que se passe-t-il sur le serveur lorsque le client ferme le socket (primitive `close`) ?

3 Transfert de fichiers par UDP

1. Écrire un programme qui ouvre en lecture un fichier (primitive `open`) donné en paramètre. Ce programme devra lire le fichier par blocs. Pour ce faire vous pouvez reprendre vos programmes du semestre passé en systèmes d'exploitation.
2. Écrire un second programme qui ouvre en écriture un fichier (primitive `open`) donné en paramètre. Ce programme devra écrire par blocs dans le fichier.
3. Reprendre le programme de la section 1 pour l'expéditeur. Au lieu d'envoyer un seul message, il doit envoyer, par blocs, le contenu de tout le fichier dont le nom est donné en paramètre.
4. Reprendre le programme de la section 1 pour le récepteur. Ce dernier devra enregistrer les données reçues par blocs dans un fichier dont le nom est donné en paramètre.
5. Tester les programmes avec des fichiers textes et binaires. Choisir des fichiers assez grand pour que la communication ne se limite pas à l'échange d'un seul bloc. Comparer le fichier source et le fichier destination avec la commande `diff`.
6. Packaging de vos programmes :

```
./recvTransferUDP <num_port_local> <nom_fichier_recu>
./sendeTransferUDP <adresse_IP_dest> <num_port_distant> <nom_fichier_a_envoyer>
```

4 IPv6

Réalisez le même programme que la section 2 mais avec un adressage IPv6. Pour cela il faut :
- Remplacer toutes les occurences de `AF_INET` par `AF_INET6`
- remplacer toutes les occurences de `struct sockaddr_in` par `struct sockaddr_in6`

Le structure `sockaddr_in6` est définie de la manière suivante :

```
struct sockaddr_in6 {
  sa_family_t     sin6_family;   /* AF_INET6        */
  in_port_t       sin6_port;     /* port number     */
  uint32_t        sin6_flowinfo; /* IPv6 flow label */
  struct in6_addr sin6_addr;     /* IPv6 address    */
  uint32_t        sin6_scope_id; /* scope ID        */
};

struct in6_addr {
  unsigned char s6_addr[16];     /* IPv6 address */
};
```

Il n'est pas nécessaire de compléter les champs `sin6_flowinfo` et `sin6_scope_id`. La fonction `inet_pton` et `inet_ntop` lisent et produisent des objets de type `struct in6_addr` et non des entiers longs comme en IPv4. Enfin, pour désigner l'adresse de la machine locale, on donne au champs `sin6_addr` la valeur `in6addr_any`.

5 Dialogue avec un serveur existant

On souhaite réaliser un petit navigateur web (très simplifié) en mode texte. Pour ce faire vous allez réaliser en langage C un client tcp. Ce client devra pouvoir se connecter en tcp sur le serveur web dont le nom (e.g. `google.fr`) sera donné en paramètre du programme. Une fois connecté, votre programme devra récupérer et afficher dans le terminal la page web qui aura été spécifiée sur la ligne de commande (e.g. `index.html`). Pour effectuer la résolution de nom (transformation du nom en adresse IP) vous devrez utiliser la fonction `getaddrinfo`.

On rappelle que les échanges de pages web se font à l'aide du protocole http (*hyper text transfer protocol*). Le port de communication par défaut d'un serveur web est le port numéro 80. La spécification du protocole http se trouve dans le rfc 2068, disponible à l'adresse `http://ietf.org/rfc.html`. Votre client de devra que supporter la commande `GET` du protocole http.

Dans le cadre de vos tests vous pouvez utiliser la commande `telnet` (man `telnet`).

6 Chat multi-utilisateurs

On souhaite réaliser un système client/serveur permettant de communiquer à plusieurs simultanément sur le réseau. Le programme client devra au préalable se connecter au serveur, puis récupérer les chaînes de caractères entrées par l'utilisateur sur l'entrée standard et les envoyer au serveur. À la réception d'une chaîne de caractères, le serveur doit prendre en charge l'envoi de cette chaîne à tous les clients connectés au serveur, sauf au client dont la chaîne est originaire. L'implémentation se fera en IPv6 à l'aide du protocole tcp.

1. Écrire un programme `clientTCP.c` qui se connecte à un serveur tcp.
2. Le client doit gérer deux événements après la phase de connexion : la réception d'un message depuis le serveur et son affichage sur la sortie standard, et la réception d'un message entré par l'utilisateur sur la ligne de commande qu'il faut transmettre au serveur. Ces deux événements doivent être gérés en parallèle à l'aide de la primitive `select`. À l'aide du manuel utilisateur, ajouter la primitive `select` à votre programme de sorte à attendre en lecture sur l'entrée standard et sur le socket tcp.
3. Pour finaliser le client, il faut ajouter le traitement que ce dernier doit réaliser lorsque qu'un événement se produit sur les descripteurs sur lequels la primitive `select` attend en lecture (i.e. lorsque `select` se débloque). Dans un cas, ce sera l'arrivée d'un message depuis le serveur qu'il faut simplement afficher sur la sortie standard. Dans l'autre cas il faudra récupérer le contenu de l'entrée standard et le transmettre au serveur.
4. Écrire un programme `serveurTCP.c` qui accepte la connexion d'un client.
5. Le serveur doit être capable d'accepter plusieurs connexions (une par client). Modifiez votre programme pour gérer la connexion de multiples clients. Vous devrez notamment définir une structure de données pour stocker les informations relatives à un client (e.g. le socket associé à ce client).
6. Maintenant que le serveur accepte la connexion de multiple clients, il faut qu'il puisse gérer la réception des messages issus de ces multiples clients. Pour ce faire, ajouter la primitive `select` à votre programme de sorte à attendre en lecture sur tous les sockets actuellement actifs.
7. Pour finaliser le serveur, ajouter le traitement qu'il doit réaliser lorsqu'un message arrive depuis un client (i.e. lorsque `select` se débloque). Identifiez l'expéditeur du message (pour éviter de lui retransmettre son propre message) et transmettez le contenu du message aux autres clients.
8. Tester vos programmes avec 1, 2 et 3 clients. Vérifier que la déconnexion (propre ou brutale) d'un client est bien gérée par le serveur.