IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Programmation réseau Unix/Linux

UE OSAP

Printemps 2024

Introduction

— Pour chaque exercice il y a un listing de programme à compléter. Prenez l'habitude de commenter le code de vos programme. Organiser vos listings de programme dans votre répertoire de travail. Chacun de vos listing aura un nom préfixé par le numéro de la question correspondante (p.ex. `1-client.c`), et débutera par un commentaire indiquant votre nom.

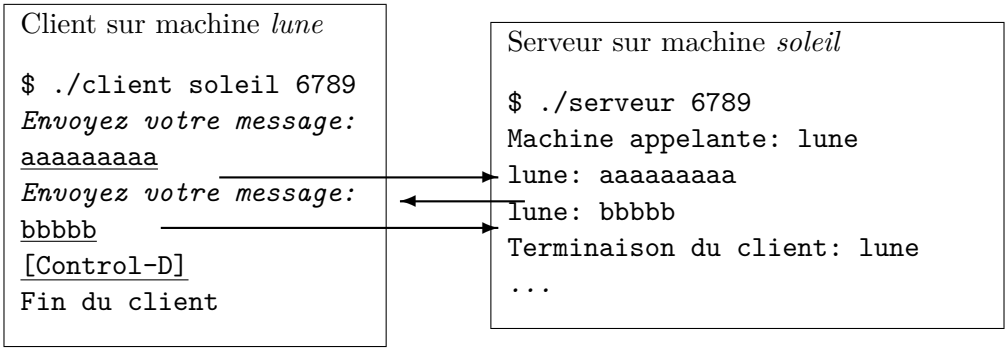
— Rappel pour compiler : `gcc -Wall -o programme programme.c`

Note : la personne qui va corriger vos programmes utilise également cette option `-Wall`...

— Avant toutes choses, récupérez sur Moodle le fichier `ProgReseau.tgz`, et desarchivez-le (commande `tar xvf ProgReseau.tgz`. Il contient les programmes à compléter au cours du TP.

1 Client/serveur simple en mode connecté TCP

Prennez les deux fichiers `client.c` et `serveur.c`. Ce sont des ébauches client/serveur mettant en œuvre les sockets. Vous devez les compléter. Le fonctionnement attendu est le suivant :



Le serveur est lancé avec un numéro de port en argument. Le client est lancé avec le nom du serveur et son numéro de port en argument. Dans le client ci-dessus, ce qui est en italique est envoyé par le serveur. Ce qui est souligné est entré au clavier par l'utilisateur.

Note : pour vous aider à développer séparément le client et le serveur, vous pouvez utiliser l'outil `netcat` (ou `nc`, `nc6`, etc. suivant les versions) comme *«bouchon»* : cet outil peut se comporter comme un client (`nc` *`nom_serveur port`*) ou comme serveur (`nc -l -p` *`port`*, suivant les versions l'option `-p` doit ou ne doit pas être précisée : lisez le man).

IPv6. Les vieilles fonctions `gethostbyname()` `gethostbyaddr()` ont certes l'avantage d'être assez simples à utiliser, mais ne sont valables que dans un contexte IPv4. D'ailleurs, la page de manuel de ces fonctions précise bien qu'elles ne devraient plus être employées dans des nouveaux programmes, et qu'il vaut mieux leur préférer les fonctions `getaddrinfo()` `getnameinfo()`.[1]

Si vous vous débrouillez bien, vos programmes peuvent fonctionner aussi bien en IPv4 qu'en IPv6. Consultez attentivement le `man getaddrinfo`.

1. La norme POSIX a déclaré ces fonctions obsolètes en 2001, et les a supprimé complètement en 2008.

La stratégie, pour le *client* est de laisser `getaddrinfo` faire sa recherche de l'adresse du serveur aussi bien en IPv4 qu'en IPv6 (i.e. utilisation du type `AF_UNSPEC`). Le client va ensuite utiliser directement le résultat de `getaddrinfo` pour créer sa socket (soit en IPv4, soit en IPv6), sans même se préoccuper faire un test sur le type de l'adresse trouvée.

La stratégie pour le *serveur* est légèrement différente. En effet, un serveur doit d'abord créer une socket d'écoute, puis attendre des connexions d'autres sockets (qui doivent donc être forcément du même type que la socket d'écoute). Lorsqu'il créé sa socket d'écoute, il ne sait pas a priori si les sockets de connexions qu'il recevra seront en IPv4 ou bien en IPv6. Et une fois qu'il a créé sa socket d'écoute, c'est fini on ne peut plus revenir en arrière. (Notons que `getaddrinfo` a tendance à retourner une liste où les adresses IPv4 sont avant les adresses IPv6, lorsque les deux sont disponibles.)

La stratégie *dual stack* consiste à gérer en parallèle deux sockets d'écoute : l'une en IPv4, l'autre en IPv6. Une autre stratégie, la stratégie de *mapping*, consiste à ne créer qu'une socket d'écoute en IPv6, mais à demander au noyau de *mapper* les éventuelles sockets entrantes IPv4 dans des sockets IPv6, ce qui permet de ne manipuler que de l'IPv6 par la suite. C'est cette seconde stratégie, plus simple, que l'on va utiliser : on ajoute les flags `AI_V4MAPPED|AI_ALL` à `getaddrinfo`.

Pour vos tests vous pouvez utiliser l'adresse spécifique `ip6-localhost` (équivalent à l'adresse `::1`).

2 Serveur TCP concurrent

Modifiez le programme `serveur.c` afin qu'il devienne concurrent. Il devra pour cela créer un processus nouveau à chaque requête de connexion reçue.

Privilégiez une architecture père-fils claire en écrivant la partie communication dans une fonction spécifique : `communication()` par exemple.

Veillez bien à ce que les processus gérant les communications ne restent pas en *zombies* après leur terminaison.

Avec la commande `netstat -a -f inet` sous Solaris ou `netstat -atnp` sous Linux, indiquez quel est le port attribué à votre client une fois connecté. Donnez la ligne correspondante affichée en indiquant le nom des champs.

3 Un client java

Prendre l'ébauche `ClientTCP.java`. Compléter ce programme afin de réaliser un client qui fonctionne de le même manière que le client en langage C réalisé précédemment. Vous devez conserver le serveur concurrent précédent sans le modifier.

Spécifications : `http://java.sun.com/j2se/1.4.2/docs/api/`

4 Un serveur Web en langage C

Prendre l'ébauche `serv_web.c` et la compléter en y insérant le code d'un serveur TCP concurrent. Compléter aussi les fonctions suivantes qui permettent de répondre de manière satisfaisante à des requêtes émises par un client Web. Finalement, cela doit fonctionner avec tout navigateur web ; citons Firefox, Iceweasel, Epiphany, Chrome, Chromium, Konqueror, Rekonk, Internet Exploreur, Safari, Shiira, Opera, etc.

Note : ces navigateurs un peu trop intelligents ont tendance à vous cacher les choses lorsqu'il y a des problèmes. Aussi, pour débugger, n'hésitez pas à tester votre serveur avec des clients web en ligne de commande : `curl wget lynx w3m elinks links2` etc.

— `communication()`
— `envoiFichier()`
— `envoiRep()`

En d'autres termes, vous réaliserez ainsi un serveur Web et le client de test sera un navigateur standard. [2]

2. Éventuellement, configurez votre navigateur pour qu'il n'utilise plus le proxy web de l'école.

Quelques informations complémentaires : Supposons que votre serveur s'exécute sur la machine `uglas` et que le port TCP 7890 lui soit affecté. Voici ce que le navigateur transmet au serveur lors d'une requête avec l'URL : `http://uglas:7890/index.html`

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Host: uglas:7890
Accept: image/gif, image/x-xbitmap, image/jpeg, ...
Accept-Encoding: gzip
Accept-Language: fr,en
Accept-Charset: iso-8859-1,*,utf-8
```

Votre serveur recevra une telle chaîne de caractères dans son tampon mémoire de réception. Il devra l'analyser et répondre à la requête (nous vous fournissons une solution d'analyse dans la fonction `communication()`). Cette requête peut correspondre à un «clic» sur un lien d'une page web ou bien à une chaîne de caractères directement entrée dans la barre d'adresse du navigateur. Elle signifie en quelque sorte : «*Envoyez-moi le fichier index.html. Je suis en protocole HTTP/1.0. Je sais traiter du gif, du jpeg, etc.*».

Nous vous demandons de limiter l'analyse de la requête à ce qui figure en gras ci-dessus. Si le nom de fichier n'est pas précisé dans l'URL demandée le serveur répond généralement en envoyant un fichier par défaut. Vous ferez de même pour votre serveur en envoyant le fichier `index.html` fourni. Les requêtes possibles dans la version 1.0 de HTTP (on dit encore *méthodes*) sont : `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE` et `CONNECT`. La plus utilisée est `GET`, vient ensuite `POST` pour envoyer des formulaires complétés. Vous vous limiterez au traitement de `GET` mais vous pourrez prévoir un aiguillage vers le traitement des autres méthodes sans implémenter ce traitement.

Si le fichier indiqué dans la requête est en réalité un répertoire vous enverrez le contenu de ce répertoire, un nom par ligne. Vous pourrez faire précéder chaque nom d'une petite icône indiquant le type du fichier : `icons/generic.gif` ou `icons/folder.gif` respectivement pour un fichier ordinaire et pour un répertoire.

Pour manipuler des chaînes de caractères vous utiliserez les fonctions décrites dans la page du manuel `string(3C)`, (faire `man string`). Vous pourrez ainsi utiliser :

- `strchr()` pour trouver l'adresse d'un caractère dans la chaîne
- `strncpy()` : pour copier des chaînes de caractères dans un buffer
- `strncat()` : pour ajouter des caractères à une chaîne existante
- `strtok()` pour délimiter des noms dans une chaîne de caractères
- etc.

Pour manipuler les fichiers et répertoires vous pourrez utiliser :

- `open()` : pour ouvrir un fichier
- `stat()` ou `fstat()` : pour obtenir des informations sur le fichier
- `access()` pour tester l'accessibilité à un fichier
- `opendir()` : pour ouvrir un répertoire
- `readdir()` : pour lire une entrée du répertoire

Vous serez amené à générer du code html directement, en particulier dans la fonction `envoiRep()`. Utilisez `sprintf()` pour formater vos chaînes de caractères puis `write()` pour les envoyer. Exemple :

```
sprintf(buf, "<html><title>Repertoire %s</title>",rep);
write(soc, buf, strlen(buf));
```

Entête HTTP : La réponse d'un serveur web débute toujours (enfin presque) par un entête. Cet entête contient au moins un code de statut (voir le RFC-2616 `http://www.ietf.org/rfc/rfc2616.txt?number=2616`). Concrètement, ce code est juste une ligne de texte. On connaît fameux «`404 Not found`» en cas d'URL erronée, mais en général on a plutôt des «`200 OK`» même si on ne le voit pas forcément à l'écran (le navigateur traite ce code, mais ne l'affiche pas à l'utilisateur). Cette ligne de statut est suivie de plusieurs informations optionnelles (nature du fichier, date, taille, etc.). L'entête se termine par une ligne vide, puis, vient ensuite le contenu du fichier ou du code HTML lui-même (par exemple pour expliciter le code d'erreur si besoin).

Typiquement, lorsque tout va bien, le serveur envoie au minimum la chaîne `HTTP/1.1 200 OK`, suivit d'une ligne vide, suivit du fichier ou document HTML demandé. Lorsque l'URL demandée est incorrecte, le serveur envoie `HTTP/1.1 404 Not Found`, suivit d'une ligne vide, suivit éventuellement d'un message HTML. Il y a ainsi quelques dizaines de codes de statut standardisés. Nous nous contenterons de n'en générer que deux ou trois.

Complément d'information sur le protocole HTTP : Dans la réalité, les serveurs Web sont plus complexes que ce que nous vous demandons au cours de ce TP. En particulier en ce qui concerne le protocole HTTP spécifié dans les documents IETF RFC-2616 et RFC-7540 (HTTP 2). Les réponses des serveurs contiennent des renseignements sur les informations renvoyées, le type de fichier par exemple (image, texte, etc.).

Dans notre exercice, pour simplifier l'en-tête, notre serveur ne génère pas d'information de type pour annoncer si ce qui suit est du HTML, une image, ou autre chose. La plupart des clients web acceptent cela avec plus ou moins de bonheur. Donc ne vous alarmez pas si, lors de vos tests, votre navigateur n'interprète pas bien le code HTML qu'il reçoit : c'est que tout simplement votre serveur ne lui a pas annoncé que c'est du HTML, et donc il l'affiche comme du texte... Mais si vous êtes courageux, faites-vous plaisir et générer une information de type dans l'entête.

Exemple :

```
HTTP/1.1 200 OK
Protocol Version: HTTP/1.1
Status Code: OK
Reason: OK
Date: Wed, 22 Mar 2000 10:58:31 GMT
Server: Apache/1.2.4 FrontPage/3.0.3
Content-Type: text/html
Set-Cookie: PHPSESSID=clp8ll3o6jovpf216i71t4nbp0; path=/

<html>
...
</html>
```

Remarque sur la sécurité : Le serveur Web ainsi obtenu n'est pas du tout sécurisé car il permet de remonter dans la hiérarchie de fichiers de la machine serveur. Il faudrait vérifier les chemins des fichiers demandés (p.ex. avec des `realpath()` `dirname()` etc.). On pourrait l'obliger à situer sa racine dans un certain répertoire (avec `chroot()`) et ainsi le contraindre à la navigation dans une sous arborescence. On pourrait aussi lui donner les droits d'un utilisateur aux droits restreints (`setuid()`).

Une optimisation possible : La manière la plus naturelle d'envoyer le contenu d'un fichier sur une socket et de faire une boucle *read*/*write* (en faisant attention à faire un *write* de la taille retournée par le *read* et non pas sur la taille totale du buffer). Cela est très pédagogique, cela fonctionne bien, c'est très portable, cela fonctionne entre fichiers, sockets, etc. Par contre, cela nécessite des copie de données entre l'espace noyau et l'espace utilisateur. On peut faire mieux. Lisez le man de `sendfile()` et `splice()`.

5 Communication en mode datagramme UDP - Émetteur/Récepteur

Prenez les ébauches `emetteur.c` et `recepteur.c`. Complétez-les de telle manière que le récepteur puisse recevoir des messages depuis n'importe quel émetteur et que la source puisse envoyer vers n'importe quel récepteur.

Ce programme demande la taille du message à envoyer. Le message est composé à partir d'un tampon mémoire contenant la lettre «`a`».

Que constatez-vous si vous envoyez un message vers une adresse non existante (par exemple 192.168.100.170, port 5678) ? Obtenez-vous une erreur ? Expliquez le comportement.

Note : Encore une fois vous pouvez utiliser `netcat` comme bouchon, avec l'option `-u` (UDP) pour remplacer l'émetteur (`nc -u` *`recepteur port`*) ou pour remplacer le récepteur (`nc -u -l -p` *`port`*).

Par définition d'un protocol sans connexion (cas de UDP), l'émetteur n'a pas de retour d'information s'il y a eu une erreur sur le chemin (mauvaise adresse, pas de récepteur, etc.). Cependant, dans quelques cas de figure on peut (ou non) recevoir des messages ICMP de la part d'un routeur sur le chemin. C'est le système d'exploitation qui traite les cas d'erreur notifiés par ICMP. Cependant, une application peut demander à avoir une copie de ces messages en créant une socket raw (voir `man icmp`).

6 Taille des tampons mémoire de réception et taille des messages

Modifiez le programme récepteur afin que son tampon mémoire de réception soit de 80000 octets. Il faut agir sur une option de socket, donc niveau `SOL_SOCKET` (voir `man 7 socket`). (Notez que cela peut avoir un impact fort au niveau transport ; ce serait le cas pour TCP, voir `man 7 tcp`.)

Quelle est la taille maximale du message que vous pouvez envoyer (faites différents tests entre 65.500 et 65.535). En vous référant au cours sur UDP donnez une explication de cette limitation.

Réessayez en IPv6.

7 La diffusion restreinte ou le mode *multicast*

Avant tout, vérifiez que vos interfaces réseau ont bien le multicast d'activé : `ip link`.

C'est généralement le cas, sauf peut-être pour l'interface `lo` (*loopback*). Pour vos essais, il suffit d'avoir au moins une interface multicast. Si nécessaire, pour activer le multicast : `sudo ip link set lo multicast on` (Commande à adapter selon votre cas. Voir le man.)

Reprenez les programme `emetteur.c` et `recepteur.c` ci-dessus et modifiez-les pour fonctionner cette fois en multicast.

7.1 Émetteur multicast

Le programme émetteur est très simple : il envoie ses paquets UDP à destination d'une adresse multicast. On prendra par exemple 224.2.2.x où la valeur de x sera différente par poste de travail.

7.2 Récepteur multicast

Le programme récepteur est plus complexe. Il doit d'abord demander de se joindre au canal multicast (se joindre à la conférence en quelque sorte) avant de pouvoir recevoir sur ce canal. Cette opération permet de paramétrer l'interface locale de la machine en lui donnant une adresse multicast. Vous prendrez des indications dans le manuel en ligne en faisant `man 7 ip` ou `man 7 ipv6`. La fonction à utiliser est `setsockopt()`, le niveau est `IPPROTO_IP` ou `IPPROTO_IPV6`.

Aussi bien IPv4 qu'IPv6 : Pour que le récepteur puisse s'abonner aussi bien à un canal multicast IPv6 qu'un canal IPv4, il y a quelques précautions à prendre. Une première stratégie consiste à utiliser à chaque fois une socket IPv6 avec des adresses IPv4 mappées sur IPv6. Ça simplifie le code (puisqu'on ne manipule finalement que de l'IPv6), mais c'est assez inélégant et on peut faire mieux. La seconde stratégie consiste à décider si l'on va faire de l'IPv4 ou bien de l'IPv6 en fonction du canal multicast choisit par l'utilisateur. Concrètement cette décision peut se prendre assez facilement en fonction des structures d'adresses retournées par l'appel à `getaddrinfo()`.

IPv4 : La spécification de la structure à utiliser est la suivante (elle est fournie par `#include <netdb.h>` ou `#include <netinet/in.h>`, c'est à dire `/usr/include/netinet/in.h` ou `/usr/include/bits/in.h`).

```
struct ip_mreqn {
  struct in_addr imr_multiaddr; /* Adresse IP du groupe de diffusion multiple */
  struct in_addr imr_address;   /* Adresse IP de l'interface locale */
  int            imr_ifindex;   /* Numéro d'interface */
};
```

L'adresse multicast du «canal» auquel on se joindra sera passée en paramètre au programme et traduite via la fonction `getaddrinfo()`.

Pour des raisons de portabilité, l'adresse locale indiquée dans cette structure pourra être `INADDR_ANY`, et l'index d'interface [3] sera 0.

IPv6 : La spécification de la structure à utiliser est la suivante (elle est également fournie par `#include <netdb.h>` ou `#include <netinet/in.h>`).

```
struct ipv6_mreq {
  struct in6_addr ipv6mr_multiaddr; /* Adresse IPv6 du groupe de diffusion multiple */
  int             ipv6mr_ifindex;   /* Numéro d'interface */
};
```

7.3 Pour débugger

Notez que notre bouchon favori (netcat) a un peu de mal avec le multicast. On pourra lui préfer l'outil `multicat`, ou encore `socat`. Par exemple, un récepteur multicast IPv4 :
`socat UDP4-RECVFROM:`*port*`,ip-add-membership=224.2.2.`x`:0.0.0.0 -`

7.4 Raffinements

Quelques raffinements envisageables (éventuellement) pour un vrai récepteur multicast :

- Annoncer que l'on quitte le groupe multicast à la terminaison du programme (sur un `[Control-C]`) : l'option de socket `IP_DROP_MEMBERSHIP` au niveau `IPPROTO_IP` (ou `IPV6_DROP_MEMBERSHIP` au niveau `IPPROTO_IPV6`).
- Indiquer au système d'exploitation que l'on s'autorise à être plusieurs processus à écouter le même port : l'option `SO_REUSEADDR` de niveau `SOL_SOCKET` avant le `bind()`. (Note : l'option `SO_REUSEPORT`, bien qu'encore non standard, tend à être préférée, notemment pour les sockets unicast et même TCP.)
- Demander au système d'exploitation de donner une copie des messages qu'il émet sur ce canal à destination des processus locaux qui y sont abonnés : l'option `IP_MULTICAST_LOOP` de niveau `IPPROTO_IP` (ou `IPV6_MULTICAST_LOOP` de niveau `IPPROTO_IPV6`).
- Limiter la profondeur de la diffusion des messages multicast : option `IP_MULTICAST_TTL` de niveau `IPPROTO_IP` (ou `IPV6_MULTICAST_HOPS` de niveau `IPPROTO_IPV6`).

3. On peut connaître l'index d'une interface sous Linux avec `ip link show`.

— Gérer aussi bien le multicast IPv6 que IPv4 en faisant un test (`AF_INET` vs. `AF_INET6` sur la valeur du `rp->ai_family` retourné par `getaddrinfo()`. Les options de socket sont légèrement différentes et ont un nom addapté, par exemple `IPV6_ADD_MEMBERSHIP` de niveau `IPPROTO_IPV6`, avec une structure d'addresse de groupe `struct ipv6_mreq`. (Voir le `man 7 ipv6`)

8 Les données «*urgentes*» en TCP

Reprendre les premiers programmes client et serveur TCP. Modifier le client afin qu'il puisse émettre une donnée urgente sur réception du signal QUIT (Ctrl-\). Modifier le serveur en conséquence afin qu'il puisse lire cette donnée urgente. La donnée urgente sera le caractère «`z`».

Au vu du fonctionnement que pouvez-vous dire de ce qu'est vraiment la donnée urgente dans TCP ?